

**SkillBuilder:**  
**A Motor Program Design Tool for Virtual Actors**

by

Swetlana Gaffron

Dipl.-Ing., Technical University Berlin (1991)

Submitted to the Department of Mechanical Engineering  
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

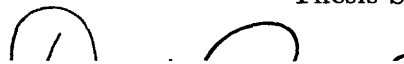
February 1994

© Massachusetts Institute of Technology 1994. All rights reserved.

Author.....  
Department of Mechanical Engineering  
November 4, 1993

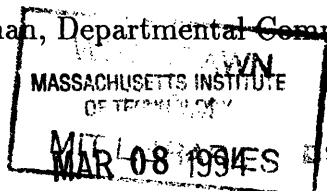


Certified by.....  
David L. Zeltzer  
Research Laboratory of Electronics  
Thesis Supervisor



Read by.....  
Derek Rowell  
Department of Mechanical Engineering  
Thesis Reader

Accepted by.....  
A.A Sonin  
Chairman, Departmental Committee on Graduate Students



LIBRARIAN

**SkillBuilder:  
A Motor Program Design Tool for Virtual Actors**  
by  
Swetlana Gaffron

Submitted to the Department of Mechanical Engineering  
on November 4, 1993, in partial fulfillment of the  
requirements for the degree of  
Master of Science

**Abstract**

To make more complex interactive virtual environments, it is desirable to incorporate human figure models that are autonomous and capable of independent and adaptive behavior. In order to achieve task-level interaction with virtual actors, it is absolutely necessary to model elementary human motor skills. This thesis presents the development of the SkillBuilder, a software system for constructing a set of motor behaviors for a virtual actor by designing motor programs for arbitrarily complicated skills. It shows how to generalize the generation of motor programs using finite state machines. Visually guided reaching, grasping, and head/eye tracking motions, for a kinematically simulated actor have been implemented. All of these actions have been successfully demonstrated in real-time by permitting the user to interact with the virtual environment using a VPL data glove.

Thesis Supervisor: David L. Zeltzer  
Research Laboratory of Electronics

## Acknowledgments

I am especially thankful to my advisor David Zeltzer for how much I have learned from him. I have certainly enjoyed working with him. His encouragement was always very helpful and supportive.

I am grateful to have had such a nice officemate as Steven Drucker who was always happy to help out when it was needed and who generously made the command loop idea and the views menu available for me.

Special thanks go to Dave Chen who has provided 3d and given me some starting help on the use of it.

The Technical University Berlin has made it possible for me to come to MIT by granting a scholarship for my first year. I am very happy for the help I got from the people in the foreign exchange office at TUB.

Jake Fleisher deserves some special thanks for his comments on my writing. Thanks also to Phill Apley for reading some of my thesis but even more for providing a nice living environment at Alien Landing during my stay in Boston.

I cannot put in words how important it was and is to me to feel the love and support from my parents, my sister, and my friends. Diego has shared lots of my fears and happiness, not only about this work with me, for what I thank him very much.

A source of light during the hard time of writing up this thesis has been my friend Walter who has brought strength and balance in my life.

# Contents

<b>1</b>	<b>Introduction</b>	<b>10</b>
<b>2</b>	<b>Related Work</b>	<b>12</b>
<b>3</b>	<b>Architecture of the SkillBuilder</b>	<b>16</b>
3.1	Task Level Interaction with Virtual Actors . . . . .	16
3.2	WavesWorld and the Skill Network . . . . .	17
3.3	Motor Goal Parser . . . . .	18
3.4	Components of the SkillBuilder . . . . .	19
3.4.1	Model of the Virtual Actor . . . . .	20
3.4.2	Motor Program Design . . . . .	20
3.4.3	The Ultimate SkillBuilder Interface . . . . .	22
<b>4</b>	<b>Implementation</b>	<b>23</b>
4.1	Development Environment for the SkillBuilder . . . . .	23
4.1.1	The <i>3d</i> Virtual Environment Simulation System . . . . .	23
4.1.2	The <i>Tcl</i> Embeddable Command Language . . . . .	24
4.1.3	The Denavit-Hartenberg Notation . . . . .	25
4.2	Graphical User Interface . . . . .	26
4.2.1	The Extended 3d Menu . . . . .	27
4.2.2	Views Menu . . . . .	31
4.3	Generation of a Human Figure Model . . . . .	31
4.3.1	Generating the Body Parts . . . . .	33
4.3.2	Building a Kinematic Structure of the Body . . . . .	34
4.4	Making It Move . . . . .	36



	<b>5</b>
4.4.1	Velocities . . . . . 36
4.4.2	Local Motor Programs . . . . . 39
4.4.3	Concurrent Execution . . . . . 42
4.5	Inverse Kinematics Algorithms . . . . . 44
4.5.1	Calculation of the shoulder-elbow-vector and the elbow-wrist-vector . . . . . 45
4.5.2	Inverse Kinematics Algorithms for Orienting the Arm Links . 48
4.5.3	Inverse Kinematics Algorithms for Orienting the Hand Axes . 52
4.5.4	Inverse Kinematics Algorithm for Integrated Positioning and Orientation of the Hand Thumb Axis . . . . . 55
4.6	Collision Avoidance . . . . . 55
4.7	General Structure of Motor Programs . . . . . 60
4.7.1	Motor Programs for Atomic Forward Kinematics Skills . . . . . 60
4.7.2	Motor Programs for Atomic Inverse Kinematics Skills . . . . . 63
4.7.3	Motor Programs for Composite Skills . . . . . 64
4.8	Condition Procedures . . . . . 68
4.8.1	Classification of Condition Procedures . . . . . 68
4.8.2	Implemented Condition Procedures . . . . . 70
4.8.3	Linking several Condition Procedures . . . . . 74
<b>5</b>	<b>Visually Guided Motion</b> . . . . . <b>75</b>
5.1	Orienting a Body Part . . . . . 75
5.1.1	Orienting the Upper Arm . . . . . 75
5.1.2	Orienting the Lower Arm . . . . . 76
5.1.3	Orienting the Hand Axes . . . . . 76
5.2	Reaching . . . . . 77
5.2.1	Simple Reaching . . . . . 77
5.2.2	Reaching for an Object with Alignment of the Hand . . . . . 78
5.2.3	Reaching for an Object along a Path . . . . . 79
5.3	Articulating the Fingers . . . . . 81

	<b>6</b>
5.3.1 Grasping . . . . .	81
5.3.2 Making a fist . . . . .	82
5.3.3 Changing between Different Hand Postures . . . . .	82
5.4 Integrated Reaching and Grasping . . . . .	85
5.5 Putting the Arm on the Table . . . . .	85
5.6 Waving the Hand . . . . .	86
5.7 Head/ Eye Controller . . . . .	87
<b>6 Future Work</b>	<b>90</b>
<b>7 Conclusion</b>	<b>93</b>
<b>References</b>	<b>97</b>
<b>A General Structure of Inverse Kinematics Skills</b>	<b>101</b>
<b>B Source Code Examples</b>	<b>103</b>
<b>C User Manual</b>	<b>134</b>

# List of Figures

3-1	Block diagram of the SkillBuilder . . . . .	19
3-2	Example finite state machine to model an actor's skill . . . . .	21
4-1	Denavit-Hartenberg link parameters . . . . .	26
4-2	Extended <b>3d Menu</b> . . . . .	28
4-3	<b>Object Edit</b> popup menu . . . . .	29
4-4	<b>Limb Edit</b> popup menu . . . . .	30
4-5	<b>Views</b> menu developed by Steven Drucker . . . . .	32
4-6	Rotational object that can be generated using the <b>make-bone-data</b> procedure . . . . .	33
4-7	Layout of kinematic chains for a human figure model . . . . .	35
4-8	The kinematic chains of a) a leg and b) an arm . . . . .	36
4-9	<b>Command Loop</b> menu . . . . .	43
4-10	Vectors defining the arm links . . . . .	46
4-11	Definition of the hand axes shown on the left hand (inner palm) . . .	53
4-12	Division of the space to find a collision free path (vertical cross-section)	56
4-13	Collision free path starting from an end effector position in area 3 . .	57
4-14	Insertion of $P_{new}$ in case the end effector is located in area 5 . . . . .	58
4-15	Generation of equidistant points on the collision free path . . . . .	60
4-16	Model of a composite skill composed of parallel, independent atomic skills . . . . .	65
4-17	Model of a composite skill composed of parallel, dependent atomic skills	65
4-18	Model of a composite skill composed of a network of atomic skills . .	66

4-19	Option “z_plus_in” for the <code>joint_in_box</code> condition procedure . . . . .	71
5-1	Virtual actor performing the <code>reach_object_along_path</code> skill . . . . .	80
5-2	Composite skill <code>grasp_object</code> . . . . .	81
5-3	Hand of the virtual actor grasping a glass . . . . .	83
5-4	Hand of the virtual actor in point position . . . . .	84
5-5	Finite state machine for the atomic skill <code>put_arm_on_table</code> . . . . .	85
5-6	Composite skill <code>wave_hand</code> . . . . .	87
5-7	Virtual actor looking at a ball in its hand . . . . .	89
7-1	Drinking actor . . . . .	96

# List of Tables

4.1	Denavit-Hartenberg parameters for the left leg and foot . . . . .	34
4.2	Denavit-Hartenberg parameters for the spine . . . . .	37
4.3	Denavit-Hartenberg parameters for the left arm . . . . .	37
4.4	Denavit-Hartenberg parameters for the left index . . . . .	38
4.5	Denavit-Hartenberg parameters for the left thumb . . . . .	38
4.6	Transformation between dhangle and real joint angles and joint limits	41

# Chapter 1

## Introduction

As applications of virtual environments increase [1] it becomes more and more important to incorporate computer models of human figures that can move and function in such a computer simulated world. Especially for applications in the education and training domain, the benefit for the virtual environment participant of interacting with human figure models, often referred to as *virtual actors*, becomes obvious. There are many situations in which it would be, for example, too hazardous, too distant, or too expensive to train in the real environment. Moreover, virtual actors in a virtual environment can substitute for other crew members to simulate team training.

Two types of virtual actors can be distinguished: The *guided* and the *autonomous* actor. An actor is guided if it is slaved to the motions of a human virtual environment participant. This can be done, for example, by means of body tracking. In my thesis I concentrate on the second kind: the autonomous actor that operates under program control and is capable of independent and adaptive behavior.

Zeltzer [31] recognized the need for finding the right level of abstraction to control simulated objects and articulated figures in virtual environments. Considering that the human body consists of about 200 degrees of freedom, it would certainly be much too demanding to ask the user to specify values for all of them in order to control a movement. It is desirable to have a higher level of interaction such that the user need only to specify the goal of an action rather than all the necessary actions themselves. One big step towards *task level interaction* is to provide the virtual actor with a set of skills each of which is represented by an appropriate motor program.

This thesis presents a software system, the SkillBuilder, that can function as a design tool for motor programs. Motor programs are the source code of skills a virtual actor can perform. A motor goal parser will take care of translating a given task from a

natural language interface to a set of task primitives which can then be passed to the skill network. The skill network will be responsible for finding out which skills need to be executed in what order and for invoking the appropriate motor programs corresponding to those skills. This will allow the user to interact with the actor on a task level.

The objective of this thesis was to develop motor skills to model an actors behavior and to understand what is needed to build a visual programming environment for defining motor skills. With regard to the latter, finite state machines have been chosen to represent motor skills. In order to allow the user to build a finite state machine it is essential to characterize the necessary components of which a finite state machine consists. First, the desired action of each state has to be identified by assigning appropriate local motor programs and second, initial conditions as well as transition conditions between the states (also called ending conditions), have to be defined. Furthermore, it is necessary to deal with issues like inverse kinematics and collision avoidance.

The main contributions of this thesis are the generalization of the design of motor programs and the characterization of transition conditions. A motor program library for interactive reaching and grasping skills has been created. It has been demonstrated that the presented approach is capable of generating approximate human movements that look reasonably realistic. The validation of these movements with experimental data is left for future work.

After reviewing some of the important work related to the thesis topic in chapter 2, chapter 3 presents an overview of the larger project in which the SkillBuilder development was embedded.

The implementation of the SkillBuilder using the virtual environment system *3d* will be discussed in full detail in chapter 4. It encompasses the generation of a kinematically controlled human figure model (section 4.3) and explains how to make it move. Most basic movements are initiated by local motor programs (section 4.4.2) that articulate a single joint each. A lot of motor skills are based on inverse kinematics algorithms explained in section 4.5. In section 4.6, a simple collision avoidance feature will be presented for reaching an object placed on a table. The general design of motor programs for different kinds of skills will then be described in section 4.7. A discussion of condition procedures and their characterization is included in section 4.8.

The chapter about visually guided motion (chapter 5) introduces the implemented skills that mainly include reaching and grasping motions. The thesis will close with suggestions for future work in chapter 6 and a discussion of the results in the conclusion (chapter 7).

# Chapter 2

## Related Work

Some issues in related fields, i.e. robotics, motor behavior, and figure animation, have been reviewed during the development process of the SkillBuilder.

The control of a human figure is closely related to robotics given that the figure model can be viewed as a composition of several robot manipulators. Paul describes the Denavit-Hartenberg joint notation [19] that is used in the SkillBuilder to construct the kinematic chains for the extremities. Lozano-Perez [14] and Brooks [5] have developed motion planning algorithms that will be of very much use at a later state of the SkillBuilder.

Researchers in motor behavior have studied the underlying processes of movement control in humans and come to the conclusion that a sequence of movements can be coordinated in advance of their execution to form a single complex action [4] [25] [23]. They call this process motor planning or motor programming. Bizzi [4] studied visually triggered head and arm movements in monkeys under the assumption that the arm movement controlling motor commands are precomputed somewhere in the central nervous system prior to movement initiation and that muscles can be modeled by a mass/spring system including damping. He found that the motor program specifies an intended equilibrium point between sets of agonist and antagonist muscles that correctly positions the arm and the head in relation to a virtual target. However, motor programs in the SkillBuilder control the movements by specifying necessary joint angle changes.

In [25] Schmidt discusses the nature of motor programs and their interaction with peripheral feedback. According to him, motor responses can be characterized in two kinds of movements: 1) very short movements ( $< 200\text{msec}$ ) that are planned in advance and do not seem to be consciously controllable while executed and 2) longer movements with the opportunity for conscious control via feedback during the move-



ment. His considerations are important for the SkillBuilder to decide when to allow interaction while executing a motor program and how big a motor program unit should be. Schmidt uses the term motor program as an abstract memory structure that is prepared in advance of the movement causing muscle contractions and relaxations when executing. Furthermore, he compares the feedback during the execution of motor programs to if-statements in computer programs. This suggests that using finite state machines is a good choice to model motor programs in the SkillBuilder.

Shaffer analyses the basic properties of skilled performances [23]. He identifies fluency, expressiveness and flexibility as basic properties of a skilled piano performance.

Badler and his research group at the University of Pennsylvania have been working on figure animation for several years. They developed the Jack system [3] for the definition, manipulation, animation, and human factors performance analyses of simulated human figures which is most comparable to the SkillBuilder. Like in the current state of the SkillBuilder they were most concerned in capturing some of the global characteristics of human-like movements without modeling too deeply the physical laws of nature, so they describe motions kinematically. Their figures consist of rigid body segments connected by joints like the SkillBuilder actor. They spent a lot of effort in articulating the torso [20] by providing 18 vertebral joints whereas the SkillBuilder concentrates more on the articulation of the hand. Jack contains features for anthropometric figure generation, end effector positioning and orienting under constraints, real-time end effector dragging, rotation propagation when joint limits are exceeded, and strength guided motion. However, to animate a motion sequence they use keyframing of a series of postures and the subsequent interpolation of the joint angles for postures in between. The SkillBuilder in contrast, computes the joint angles for the posture to be rendered at the next time frame in real-time. The animation sequences of Jack are primarily scripted and not provoked by motor programs which is what the SkillBuilder focuses on. Jack does not provide any tool to define motor skills. Badlers et al ultimate goal is to provide a high level of task understanding and planning. Their approach includes research in natural language instructions for driving animated simulations. The SkillBuilder is part of a bigger project that also includes research on natural language to describe tasks, as will be discussed in chapter 3.

Dynamics simulations as Wilhelms proposes in [28] allow physically correct motions by using forces and torques to drive a figure. Finding realistic movement laws is a difficult undertaking which often leads to movements that seem too regular. Besides, solving the equations of motion is computationally very expensive and not necessarily needed at this state of the SkillBuilder since the concern was not to model motions most realistically, but find a means by what to describe motor behavior.

Girard also argues [12] that the torque functions needed at each joint to produce a desired limb motion are sufficient to capture uncoordinated limb motion but that goal-oriented behavior cannot be properly simulated without kinematic trajectory planning. He has been working on inverse kinematics and visual programming interfaces to describe motor behavior for a number of years and concentrated mostly on locomotion and dance. His figures can be controlled interactively by specifying a spline curve as path for an end effector and applying inverse kinematics to compute the desired joint angles of a limb. Rotations of joints can also be specified directly to achieve desired limb postures. Assembling several postures yields a working posture sequence. Yet a third method is to adjust a posture while keeping the end effector at its current location. The SkillBuilder incorporates all of the named possibilities in its inverse and forward kinematics skills. Posture changes for a fixed end effector position can be achieved by modifying a global posture parameter for a limb.

Investigations on knowledge-based human grasping have been done by Rijpkema and Girard [22]. Their hand model is, very similar to the SkillBuilder hand, kinematically articulated by using the Denavit-Hartenberg notation. The high level control they propose for the hand includes single-finger control, group control (closing and opening or spreading a group of fingers), and hand control (using a hand posture library). All the named methods are incorporated in the SkillBuilder.

Calvert describes the ideal human figure animation system [2] as a system that should accept natural language and graphical input at its highest level and generate a script with the details of movement specifications. The lowest level would comprise detailed movement instructions for each limb segment as a function of time. He acknowledges that they are still far away from its realization. He and his research group developed a prototype system, COMPOSE, for outlining a sequence for multiple animated dancers. Many different stance phases for the dancers can be defined for placement in a number of scenes much like storyboard sketches.

A motion control system, PINOCCHIO, is described by Maiocchi and Pernici [16]. They classify movements and their attributes using entries from natural language in a general movement dictionary. Motions from their motion database can be assembled in an animation script to display a movement sequence. Since the movements are recorded from real human motions, however, no real-time interaction with the system is possible.

Magenat-Thalmann and Thalmann present a body animation system based on parametric keyframing [15]. In very tedious sessions the animator has to specify parameters like joint angles of a stick figure model to describe frame by frame. Without the possibility of any further interaction, in between frames will be interpolated and the HUMAN FACTORY software will transform the character accordingly while mapping

surfaces onto the stick figure. Joint dependent local deformation operators control the evolution of surfaces to model soft tissue. Since the system generates relatively realistic looking images but does not allow user interaction (either in real-time or on a sufficiently high level), it is more suitable for making movies than for the purposes aimed for by the SkillBuilder.

Zeltzer, who is involved in the project that includes the SkillBuilder development, has been working for many years in the field of figure animation. He describes the layers of abstraction for representing and controlling simulated objects and agents at different interaction levels [31]. At Ohio State University he had developed the walking skeleton George that is based on a skeleton description language. George's different gaits and jumps are controlled by motor programs using finite state machines [29]. In recent years, he and Johnson have concentrated on the process of motor planning [32]. Zeltzer [33] accounts for everyday human activities by introducing task primitives and reports motor goal parsing as a means to parse restricted natural language input into motor skills. The motor goal parser will be explained in section 3.3 of this thesis.

# Chapter 3

## Architecture of the SkillBuilder

### 3.1 Task Level Interaction with Virtual Actors

The development of the SkillBuilder is part of a project called *Virtual Sailor* which has as its goal the implementation of a virtual environment that includes autonomous and interactive virtual actors. The attempt is to provide a level of interaction with which the end-user is familiar from everyday life, i.e. through language and gesture in real time. To achieve this kind of interaction that Zeltzer calls *task level interaction* [31], elementary human motor skills have to be modeled and integrated into a behavior repertoire for an autonomous actor. Furthermore, a task-level, language-based interface has to be developed.

The presented research is particularly concerned with the kinds of mechanical operations on physical objects encountered in everyday activities, such as reaching and grasping, and manipulating tools and commonplace objects. The current effort is focussed on the simulation of one individual actor. However, the ultimate goal is to have multiple actors that are capable of coordinating their behavior with the actions of other actors or human participants in the virtual environment [35].

The SkillBuilder is a tool for constructing a set of behaviors for a virtual actor by designing motor programs for arbitrarily complicated skills. The coordination of the skills to perform a given task has to be undertaken at runtime by a separate mechanism, i.e. the motor planning system, which takes care of selecting and sequencing motor skills appropriate to the actor's behavioral goals and the state of objects. Zeltzer and Johnson call *motor planning* the process of linking perception of objects and events with action [32].

Finally, an interface for constrained natural language input is needed to “translate”

a task description, e.g. “Go to the desk and pick up the glass” into a set of task primitives [34]. The *motor goal parser* currently under development by Zeltzer takes care of this decomposition.

## 3.2 WavesWorld and the Skill Network

Johnson is working on the development of WavesWorld, a software system implemented for designing, building and debugging the object database, distributed virtual environments, and virtual actors, of which a detailed description can be found in [34].

The *motor planner* component of WavesWorld, a reactive planning algorithm, is based on a skill network [30], [32], [13] to model the behavior of virtual actors. A collection of motor skills can be assembled into the skill network and the output can be visualized by using WavesWorld. The execution of motor acts depends on sensory input and current behavioral goals.

A virtual actor’s skill network is comprised of several agents that are each realized as a set of processes distributed over a network of workstations. There are skill agents, sensor agents, goal agents and a registry/dispatcher.

The sensor agents handle the perception of a virtual actor by measuring signs and signals from the virtual world represented by proposition-value pairs, e.g. proposition: `a-door-is-nearby`, value: `TRUE`. Goals are handled by goal agents that are represented as desired states, i.e. a proposition-value pair that needs to be sensed true by a sensor agent to satisfy the goal. Finally, skill agents control the behavior of a virtual actor and attach a set of pre- and postconditions to a skill name. A skill can only be selected by the planning algorithm for execution if all its preconditions are fulfilled. Postconditions are predictions about the state of the virtual world after the execution of the skill. The central communication among all these agents is handled by the registry/dispatcher which maintains a shared database for all agents.

The skill network is characterized by:

- a comprehensive sensing structure
- high-level controls through time-varying sampling rates
- computational economics at the lowest level
- direct-manipulation interface to a large set of heterogeneous computing resources.

### 3.3 Motor Goal Parser

This section largely borrows from [34] in which Zeltzer describes the process of motor goal parsing.

In motor goal parsing (MGP), the task manager must decompose each task description into a set of well-defined motor units, i.e. task primitives, that can be named and simulated. A task description is a constrained natural language description of a specific movement sequence, e.g. “Go to the door and close it”. The set of task primitives that corresponds to the everyday activities to be simulated need to be specified. In the above example they might include “move\_to(door)”, “grasp(doorknob)”, and “rotate(doorknob)”.

For two reasons, however, the mapping from these task primitives to specific motor skills cannot be one-to-one a priori, and the task primitives must be effector-independent at the conceptual level. First, because of the problem of motor equivalence: a given goal can often be accomplished in various ways, e.g. the door can be closed by turning the doorknob with the right or left hand or by pushing the door with one of the feet. Second, the objects named in the task description may determine which skill or skills are necessary to accomplish the task. If, instead of “Go to the door”, the command were “Go to Mexico”, the task manager is expected to output a very different list of skills. Once the task primitives have been identified from the input task description, the task manager must consider the states of the virtual actor, the targeted objects, and ongoing events in the virtual world, and then determine the appropriate effector systems and motor skills to invoke. Therefore, the task primitives must constitute an intermediate, effector-independent representation which drives the selection of the underlying motor skills to be performed.

The task primitives Zeltzer uses are based on a set of primitive “ACTs” from Schank’s Conceptual Dependency (CD) theory [24] and on “A-1 action units” described by Schwartz et al [26]. Some of the task primitives represent physical actions, and some of them represent abstract “mental actions”.

Zeltzer defines atomic skills as such that serve as task primitives while composite skills represent learned behaviors. He proposes that affordances, (i.e. the properties of objects and their environment that are necessary predictions for executing particular acts), should be stored with the task primitives. He argues that the notion of affordances suggests the right approach for representing the common sense knowledge that makes it possible to perform everyday activities.

An example of a motor goal parsing process is given in [34].

### 3.4 Components of the SkillBuilder

The SkillBuilder consists of several components as shown in figure 3-1 [35]:

- A *skill template* that has to be filled in by the user with all necessary information for the motor planner: a set of pre- and postconditions, the end effector involved in the skill, and a pointer to the appropriate motor program.
- A *motor program builder* that generates motor programs: the source code for an actor's skill based on finite state machines, i.e. definite states in if-then-else clauses.
- A *motion checker* module that serves to display and visualize the execution of defined skills.

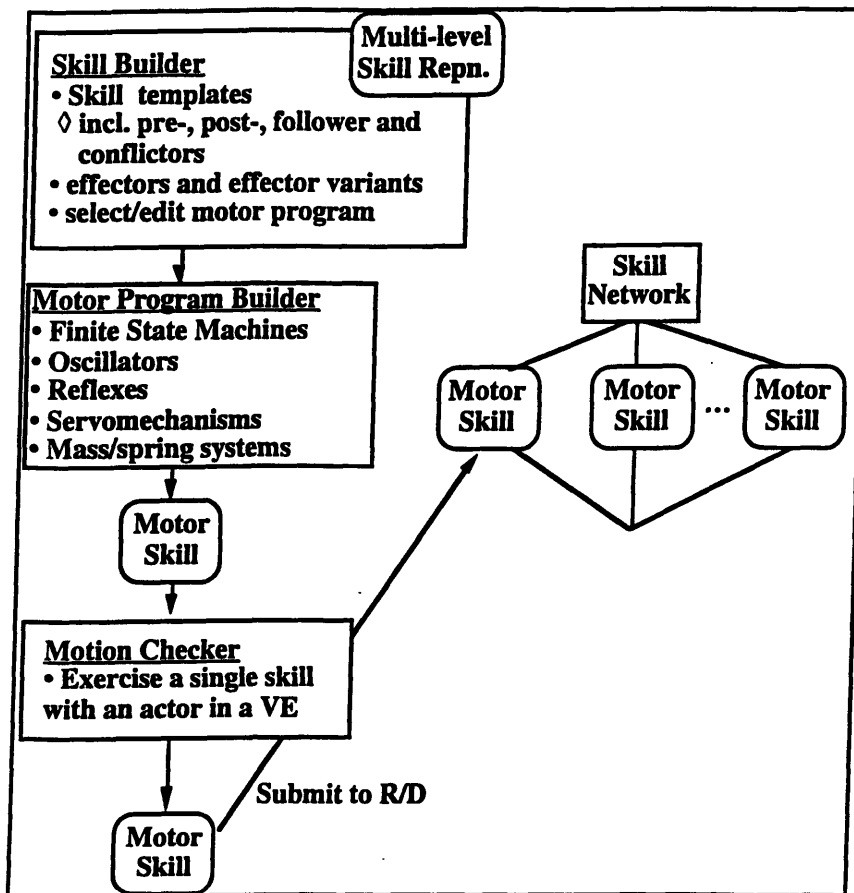


Figure 3-1: Block diagram of the SkillBuilder

The main focus of the presented work is put in the understanding of the design of motor programs and the elaboration of a general structure for motor programs by developing motor programs in particular for reaching and grasping. The development environment *3d* (to be explained in section 4.1) provides all necessary tools to display the defined skills which is the task of the motion checker module.

### **3.4.1 Model of the Virtual Actor**

The development of the SkillBuilder includes the generation of a virtual actor, a computer model of a human figure that consists of a compound of rigid body parts linked together by a kinematic structure. The model has to be simple enough to allow real-time interaction and facilitate the software development. Furthermore, it is not the intent to build the most realistic looking virtual actor but rather have an articulated model that suffices to show the execution of skills in order to learn how to generalize the design of motor programs. A rigid body model purely animated by kinematics will serve this purpose for the time being. As fidelity requirements increase for specific applications the kinematic model can later be extended to simulate dynamics.

### **3.4.2 Motor Program Design**

Every skill an actor can perform is based on a motor program that coordinates the execution of several local motor programs and is modeled by a finite state machine. A local motor programs is the mechanism that controls the rotation about a single joint axis. The use of finite state machines to control walking skills has been successfully demonstrated by Zeltzer [29], Raibert [21], and Brooks [6].

A finite state machine to model an actor's skill can consist of any number of branches and feedback loops and can be arbitrarily complicated (figure 3-2). Furthermore, oscillators, servos, reflexes, mass/spring systems and collision detection module all can be realized by finite state machines.

Most important is to decide which local motor programs should be involved in the performance of the skill. Often, a skill can be modeled in many different ways, involving different local motor programs. The next step is to decide which local motor programs make up each state and how the designed states should be connected. Finally, the initial conditions for the skill and the ending conditions of each state have to be defined.

Each state is executed until its ending conditions are fulfilled. Ending conditions can be geometric constraints ("Do this until the end effector is closer to a specified point



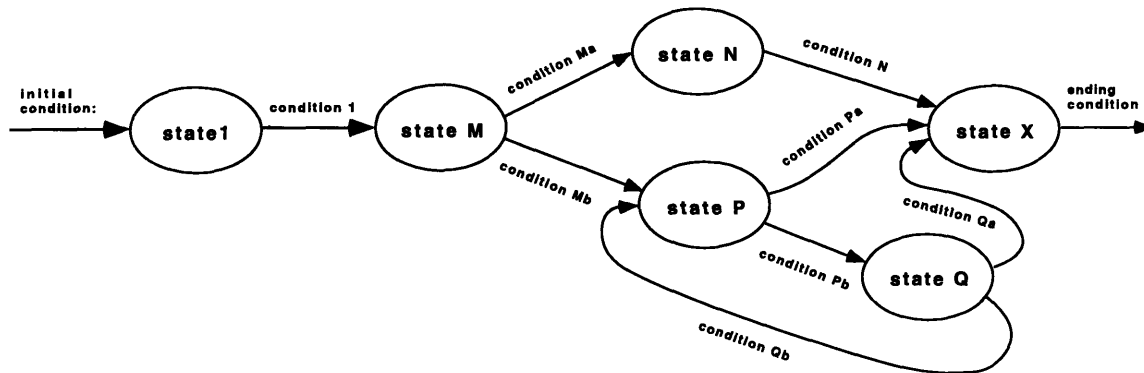


Figure 3-2: Example finite state machine to model an actor's skill

than a given *goal point precision*”), joint angle constraints (increase/decrease a joint angle until it reaches a specified target joint angle or its limits), or even external constraints (“Stop executing as soon as another motor program’s ending conditions are fulfilled”). Once the ending conditions for a motor program are satisfied, its end event will be evaluated. In defining another motor program as end event, several motor programs can be linked to execute sequentially. Motor programs for atomic skills will be classified as follows:

1. **Forward kinematics skills**

Forward kinematics skills are modeled by finite state machines with usually more than one state. Examples of implemented forward kinematics skills are `put_arm_on_table` and `reach_ceiling`.

2. **Inverse kinematics skills**

Inverse kinematics skills usually have just one state and contain an *inverse kinematics calculation procedure* to synchronize the angle step size for all local motor programs involved. Examples of inverse kinematics skills are `move_arm_to_goal` and `direct_hand`.

Motor programs for composite skills are composed of several atomic skills that will execute in parallel, sequentially, or in any possible combination of those two. This will be explained in more detail in section 4.7.3.

#### 3.4.3 The Ultimate SkillBuilder Interface

The ultimate SkillBuilder should provide a visual programming language for describing motor skills. Through the use of a graphical interface the user should be able to point and click on selections of initial conditions, as well as selections of local motor programs, defining each state of a finite state machine for an atomic skill. He should then be able to click and drag on symbols for each state to position them in the order of his choice, and then draw pointers between those states to define their connections.

A second graphical module should provide a menu of all pre-defined atomic skills, from which the user could select as many as he wants and then assemble them in an arbitrarily complicated network, thereby building a composite skill.

For atomic, as well as for composite skills, there should be an appropriate motor program builder to automate the generation of the motor program procedures that make up a skill once it has been graphically set up by the user.

A simple click on a motion checker button should execute the newly defined skill to see whether the simulation satisfies the expected performance.

# Chapter 4

## Implementation

### 4.1 Development Environment for the SkillBuilder

#### 4.1.1 The *3d* Virtual Environment Simulation System

The SkillBuilder was developed using the virtual environment system *3d*, an interpretive toolkit for creating custom VE applications.

*3d* is a “testbed” system designed to support the specification of behaviors for virtual worlds. It has been developed by the Computer Graphics and Animation Group of M.I.T.’s Media Lab. Systematic descriptions of *3d* can be found in [8] and [7]. The system provides an interpreted command language that has many special purpose rendering, dynamics, numerical math and user-interface functions.

The syntax for *3d* is based on the “tool command language” (*tcl*) from U.C. Berkeley [17]. The *tcl* environment includes an interpreted command evaluator that allows rapid prototyping of virtual worlds and VE interfaces. The application-specific code for *3d* has been developed on top of the RenderMatic C library of graphics and rendering software [10]. The command interpreter has over 700 built-in and application-specific functions. There are primitives for scene description and rendering, math, matrix and vector operations, general data structure manipulation, Denavit-Hartenberg joint description, finite element dynamics, and X/Motif interface building. A list of all available *3d* commands can be found in the appendix of [7].

In *3d* it is very easy to build a collection of useful subroutines, because *tcl* allows procedure definition. In this way, the application software can be organized by creating interpreted layers of language, as has been done in the work presented.

### 4.1.2 The *Tcl* Embeddable Command Language

For a better understanding of the software examples in this thesis, a brief description of the *tcl* language syntax follows. This section borrows largely from [18].

The syntax of *tcl* is similar to the UNIX shell syntax: A command consists of one or more fields separated by blanks. The first field is the name of the command, which can be either a build-in or application-specific command, or a *tcl* proc constructed from a sequence of other commands. Subsequent fields are passed to the command as arguments. Newlines and semicolons separate commands. *Tcl* commands return a string according to their evaluation, or an empty string if no return value was specified.

Additional constructs give *tcl* a lisp-like feel. Comments are delineated by a pound sign (`#`). The backslash character (`\`) denotes line continuation, or escapes special characters for insertion as arguments. Curly braces (`{}`) can be used to group complex list arguments. For example, the command

```
set echs {molchi {hoch drei}}
```

has two arguments, “`echs`”, and “`molchi {hoch drei}`”. The `set` command sets the first argument to be the value of the second argument.

Command substitution is invoked by square brackets (`[]`). The content between those brackets is treated as a command and evaluated recursively by the interpreter. The result of the command is then substituted as the argument in place of the original square bracketed string so that

```
div 6 [minus 4 1]
```

for example returns 2 because “`minus 4 1`” returns 3.

Finally, the dollar sign (`$`) is used for variable substitution. If the dollar appears in an argument string, the subsequent characters are treated as a variable name, and the contents of the variable are then substituted as the argument, in the place of the dollar sign and variable name. For example,

```
set x [sqrt 4]
plus 5 $x
```

returns 7 because the variable `x` has the value 2.

### 4.1.3 The Denavit-Hartenberg Notation

To support the construction of, and interaction with, kinematic chains as needed for robotics applications, the Denavit-Hartenberg joint notation [19] has been implemented in *3d*. This feature has been used in the presented work to build the structure of a jointed figure by defining kinematic chains for each of the extremities.

The Denavit-Hartenberg notation is a systematic method of describing the kinematic relationship between a pair of adjacent links with a minimum number of parameters. The relative position and orientation of the two coordinate frames attached to those two links can be completely determined by the following 4 parameters (denoted as dhparameters, see figure 4-1) :

1.  $a_n$  : the length of the common normal
2.  $d_n$  : the distance between the origin of joint n and the common normal along  $z_{n+1}$
3.  $\alpha_n$  : the twist angle between the joint axes about  $x_n$
4.  $\theta_n$  : the angle between the  $x_{n-1}$  axis and the common normal measured about  $z_{n-1}$

In a human skeleton all degrees of freedom can be satisfactorily modeled by revolute joints. Changing the angle  $\theta_n$  will cause a link n to turn about the  $z_{n-1}$  axis. All other dhparameters will remain constant. The relationship between adjacent coordinate frames can be expressed in a 4x4 matrix:

$$\mathbf{A}_i = \begin{pmatrix} \cos \Theta_i & -\sin \Theta_i \cos \alpha_i & \sin \Theta_i \sin \alpha_i & a_i \cos \Theta_i \\ \sin \Theta_i & \cos \Theta_i \cos \alpha_i & -\cos \Theta_i \sin \alpha_i & a_i \sin \Theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.1)$$

Thus, the position and rotation matrix of joint n can be calculated by multiplying the position and rotation matrix of a joint m ( $n > m$ ) with all transformation matrices from each joint in between m and n to its adjacent joint:

$$\mathbf{M}_n = \mathbf{M}_m \cdot \mathbf{A}_{m+1} \cdot \mathbf{A}_{m+2} \cdot \dots \cdot \mathbf{A}_{n-1} \cdot \mathbf{A}_n \quad , \quad (n > m) \quad (4.2)$$

In *3d* every defined Denavit-Hartenberg joint n has a *dhmatrix* assigned to it which is the matrix  $M_n$  from equation (4.2). This way, *3d* internally takes care of providing

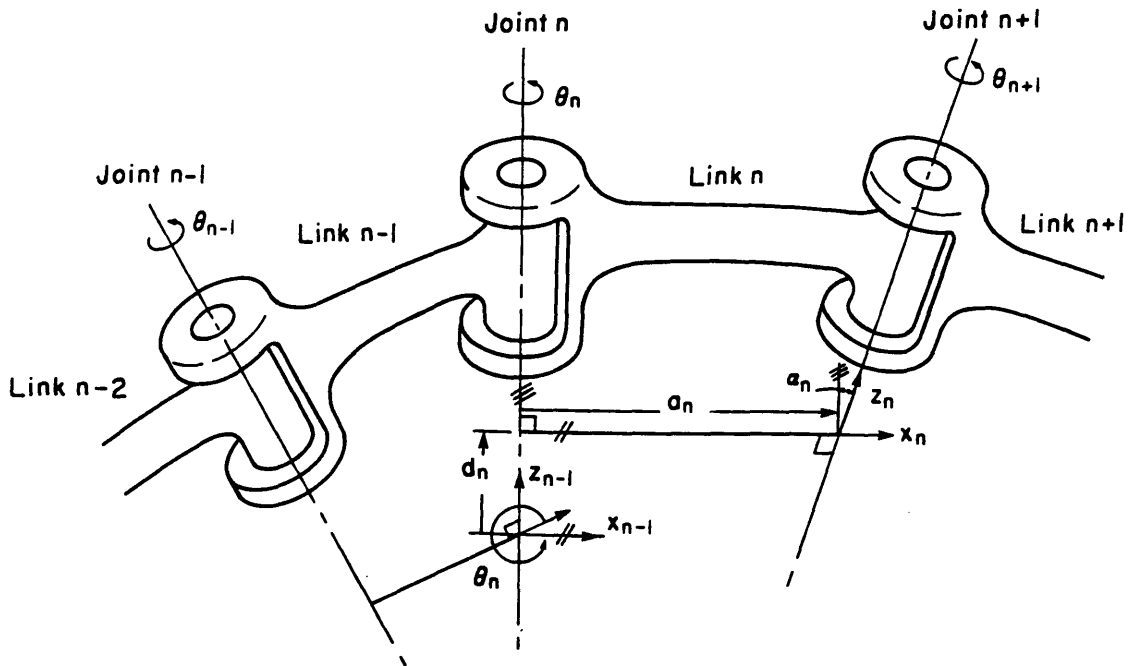


Figure 4-1: Denavit-Hartenberg link parameters

the absolute transformation matrices for all body parts, depending on the value of the dhparameters.

Spherical joints can be defined by a system of three coincident joints, where no two joint axes are parallel. When defining the kinematic chains for the extremities, a few “dummy” joints have to be introduced. This is caused by the fact that it is not always possible to rotate the subsequent coordinate frame to line up as desired, since the twist angle  $\alpha$  only produces a rotation about  $x_n$ . The “dummy” joints will not move but only serve to modify the coordinate frame orientation.

## 4.2 Graphical User Interface

The user interface of *3d* consists of four kinds of dialogs:

1. Keyboard input
2. Manipulators (such as *glovetool* to control a data glove)
3. Mouse manipulation of objects inside the graphics window

#### 4. Interactive X Dialog Boxes

*3d* supports building of graphical interfaces with application-specific commands [7] imported from OSF/Motif™ widgets [27].

It is desirable to control as much as possible through the use of a graphical interface in order to make the SkillBuilder easier to use. For the most part, however, the actor has to be controlled by using the keyboard, and typing commands directly in the *3d* window at the *3d>* prompt. The graphical user interface for the SkillBuilder is an important area that must be improved in the future.

In the following section the graphical user interface that has been used during the development of the SkillBuilder will be explained. The **3d Menu** provided by Dave Chen [8] has been extended to include a few more features, and the **Views** menu developed by Steven Drucker has been integrated. The **Command Loop** menu (see figure 4-9, page 43) is based on the idea of a command loop also provided by Steven Drucker that has been implemented as a graphical dialog for the SkillBuilder. It will be explained in section 4.4.3.

### 4.2.1 The Extended 3d Menu

Figure 4-2 shows the extended **3d Menu** that automatically pops up when calling the SkillBuilder.

Features that were not included in the basic **3d Menu** are:

1. **Current Object** as a button providing a pull down menu of all currently available objects.
2. The **Command** text field.

What follows is a review and explanation of the most important features used during the development of the SkillBuilder.

#### Changing the Current Object

Clicking on the **Current Object** button will cause a pull down menu listing all currently existing objects to appear. If one of those objects is chosen, it will appear in the text field right below the **Current Object** button. The current object can also be set by clicking on the object in question with the middle mouse button, or by typing it in the text field.

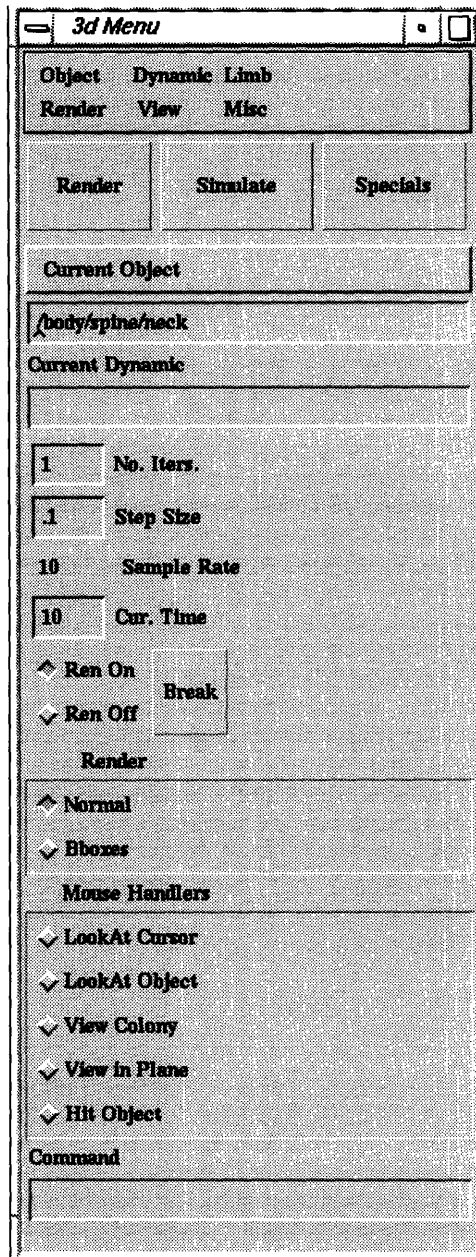


Figure 4-2: Extended 3d Menu



### Object Edit Popup Menu

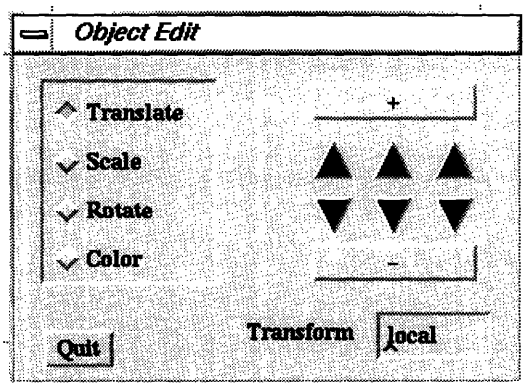


Figure 4-3: Object Edit popup menu

To make the **Object Edit** menu (see figure 4-3) pop up, click the **Object** label in the upper left corner of the **3d Menu**, and then choose **Edit** from the pull down menu that appears. Toggle buttons are provided to choose the action to be taken by the current object (whose name appears below the **Current Object** button in the **3d Menu**). The current object can be translated, rotated, scaled, or changed in color. For the first three actions, the arrow buttons in the menu correspond to the x, y, and z axis in this order, whereas when changing the color, those buttons control the r, g, and b values in the rgb code. The + and the - buttons change all three values at the same time by the same amount. Choosing “local” in the **Transform** field makes the centroid of the object the reference frame. A different reference frame can be chosen by typing its three coordinates in this text field instead of “local”.

### Posting and Unposting Objects

Objects can be easily posted or unposted by choosing the option **Post** or **Unpost** in the pull down menu which appears when clicking on **Object** in the upper left corner of the **3d Menu**. The object that will be posted or unposted is the current object.

### Limb Edit Popup Menu

The **Limb Edit** menu (see figure 4-4) can be invoked by clicking **Limb** in the upper right corner of the **3d Menu** and pulling down to **Edit**. The **Current Limb** has to be set to one of the limbs that is defined as a dhchain, i.e. l.arm, spine, l.leg and l.shoe. Clicking now on one of the arrows in the lower right corner of the **Limb Edit** menu

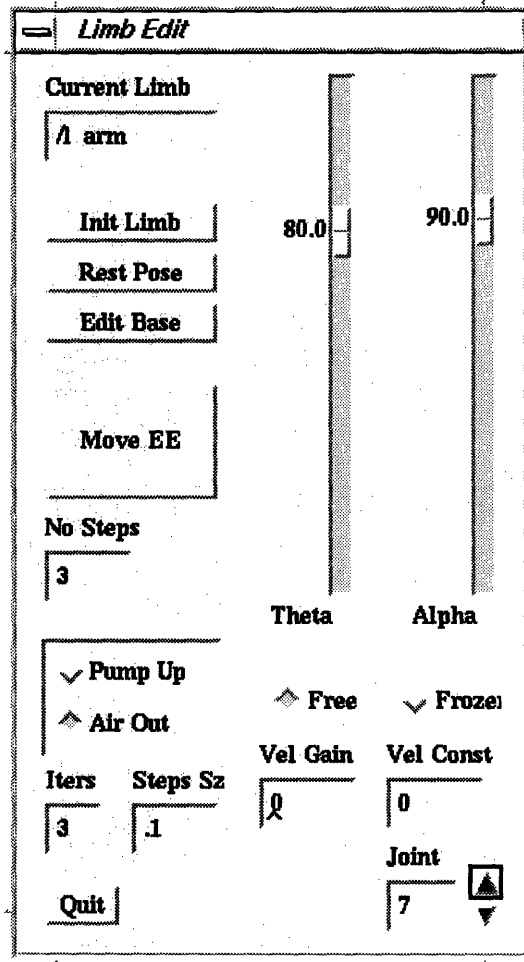


Figure 4-4: Limb Edit popup menu

will increase or decrease the current joint number. A coordinate frame displaying the position and orientation of the dhmatrix of the current joint will appear in the graphics window. To change the current joint angle, the slider above the label **Theta** can be changed. Since the dhparameter alpha would change the definition of the relationship between two links, its value should not be modified once the kinematic chain of a limb has been set up. Note that it is possible to move the joints to angles outside of their limits. That is the reason why the **Limb Edit** menu is meant only to be used for testing and development purposes.

### Command text field

At the bottom of the **3d Menu** there is a text field labeled **Command**. Any command that might be typed in at the **3d>** prompt in the 3d window can also be typed in here with the same result. Once the command loop is running (refer to section 4.4.3), the 3d window does not accept any input, but the menu windows do, because of an x flush event at the end of each loop step.

### Using the Mouse in the Graphics Window

Mouse use in the graphics window to change the view is explained in detail in the User Manual included in appendix C, page 134.

### 4.2.2 Views Menu

With the kind permission of Steven Drucker, the **Views** menu (see figure 4-5) developed by him is made available for use in the SkillBuilder. The views menu controls the synthetic camera which generates views of the 3d scene. A number of views stored in a file can be retrieved any time and new views generated by mouse interaction in the graphics window can be stored. The use of the **Views** menu is explained in the User Manual (appendix C, page 135).

## 4.3 Generation of a Human Figure Model

Before one can start to animate a human figure, a model appropriate to the given goal needs to be generated. Since this work is not concerned with soft tissue animation, a stick figure model is sufficient to represent its movements. The model of the actor consists of rigid 3d body parts that represent the limbs of the body. The limbs are

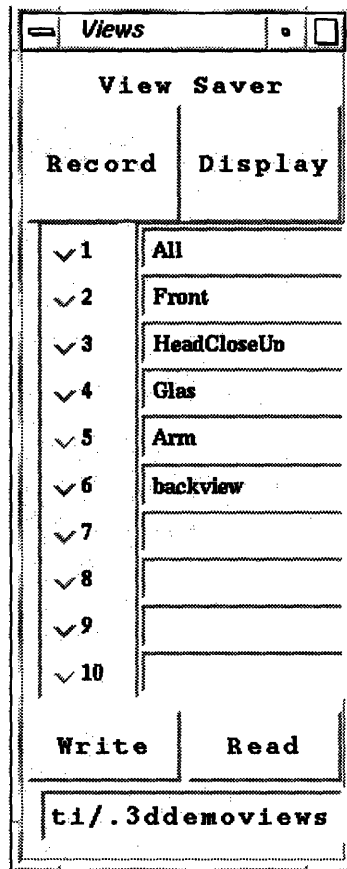


Figure 4-5: Views menu developed by Steven Drucker

joined together forming several kinematic chains. Each kinematic chain is defined using the Denavit-Hartenberg notation explained in section 4.1.3.

### 4.3.1 Generating the Body Parts

An object can be instantiated in *3d*, if a file with the object data in the so called OSU format exists [11]. To generate data files in OSU format for rotational symmetric objects, a *tcl* procedure named `make-bone-data` has been written (see appendix B, page 124). The parameters the procedure needs as input are as follows:

```
make-bone-data name precision nrrad r1 l1 r2 l2 ... rn
```

Alternately, the radii and the length between two radii have to be specified. To influence the shape of the cross-section, different precisions (number of points forming the cross-section) can be specified, e.g. a precision of 3 would correspond to a triangular cross-section, whereas 4 would yield a square cross-section and so forth. As an example, a data file `rot.asc` for the rotational object shown in figure 4-6 can be generated by the command:

```
make-bone-data rot 15 6 0 0 2 3 4 0 2 3 2 0 0
```

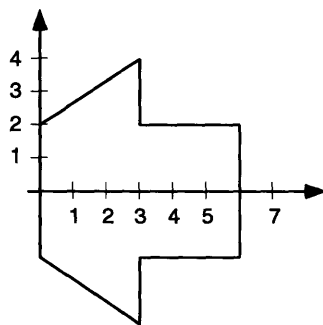


Figure 4-6: Rotational object that can be generated using the `make-bone-data` procedure

Most of the body parts have been constructed using the `make-bone-data` procedure, some of them with subsequent scaling in one direction (e.g. the upper body). Some of the conical body parts are extended by half a sphere of corresponding radius at each end that will overlap with the reversed sphere from the adjacent limb. This way, the transition between two limbs appears smooth while bending for example an elbow, knee, or finger joint without a gap opening in the middle .

Appendix B, includes all *tcl* procedures used to generate these kinds of objects. The two procedures, `left-circle` (page 124) and `right-circle` (page 129), serve to calculate the radii and length between every two radii needed to form a quarter of a circle. The `make-bone` procedure (page 124) automatically generates a conic object with half a sphere at each of its ends.

### 4.3.2 Building a Kinematic Structure of the Body

Figure 4-7 on page 35 demonstrates the design of the kinematic structure shown on top of a human skeleton with the outline of the human. The base of the actor is located in the pelvis. Attached to this center is a chain for each leg that goes along the ankle to the heel (as shown in figure 4-8 a) ). Starting at the ankle, there is another chain consisting of two links ending at the tip of the toes.

The *dhparameters* used to construct the two kinematic chains for the left leg are shown in table 4.1.

dhc	joint i	name	$\Theta_i$	$d_i$	$a_i$	$\alpha_i$	comment
l.leg	0	base: pelvis	90°	0	0	90°	
	1		90°	0	0	-90°	dummy
	2		-90°	0	3.24	90°	
	3	hip	0°	0	0	90°	
	4		90°	0	0	90°	
	5		0°	0	16.69	0°	
	6	knee	0°	-0.3	14.64	0°	
	7	ankle	0°	0	0	90°	
	8		90°	0	0	-90°	
	9		0°	0	0	0°	
10		90°	-3.25	2.21	0°	ee: heel tip	
l.foot	0	base: ankle	90°	-3.25	-3.66	0°	
	1		0°	0	0	-90°	dummy
	2	ball	0°	0	-3.66	0°	ee: toe tip

Table 4.1: Denavit-Hartenberg parameters for the left leg and foot

Also attached to the base is a kinematic chain for the spine all the way up to the head (see *dhparameters* in table 4.2, page 37). The chains for the arms originate in the center of the chest. Figure 4-8 b) shows the arm chain from joint 6 on. The *dhparameters* for the left arm are given in table 4.3 (page 37). Each finger is represented

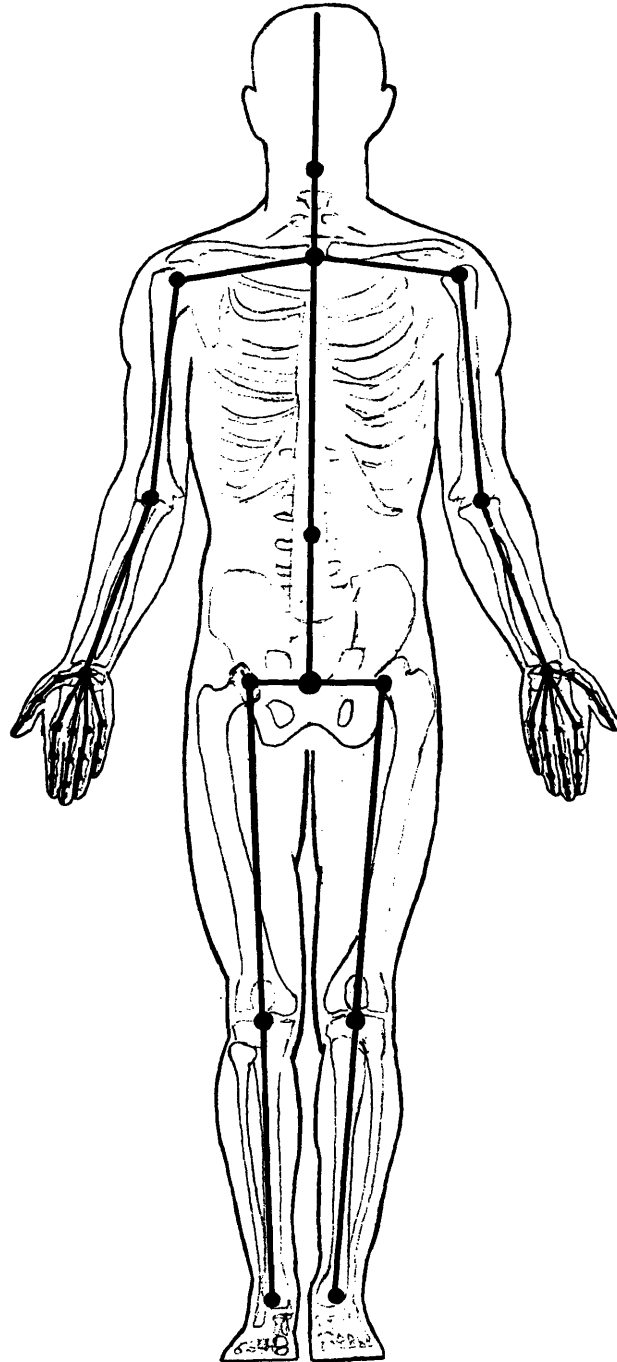


Figure 4-7: Layout of kinematic chains for a human figure model

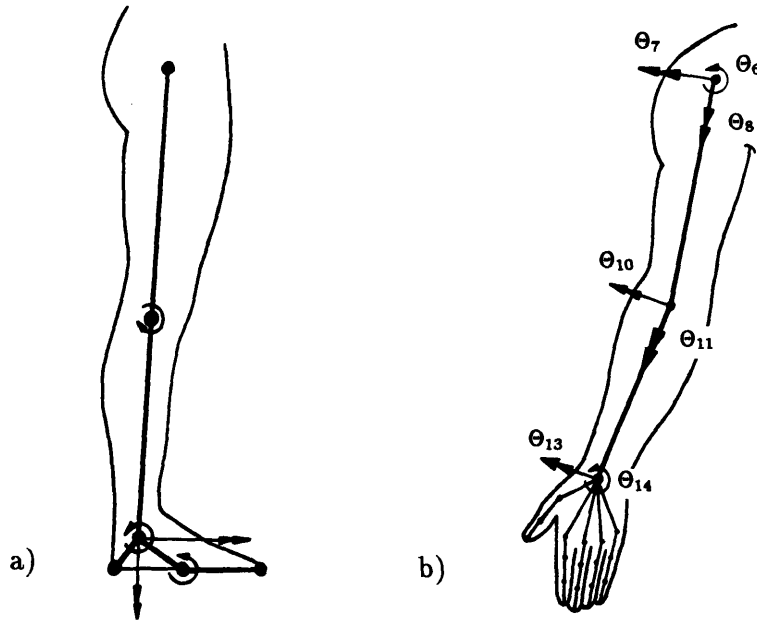


Figure 4-8: The kinematic chains of a) a leg and b) an arm

by a separate chain starting at the wrist. The dhparameters for the left index are given in table 4.4 (page 38).

## 4.4 Making It Move

No attempt has been made yet to validate the developed motor programs against clinical, experimental data. That will be left for future work. The movements are a visual approximation to the normal human velocity. This assumption is embodied in two global variables as will be explained in the following section.

### 4.4.1 Velocities

Velocities are controlled by two global variables depending on the kind of a skill. When dealing with inverse kinematics skills, the end effector velocity is controlled by the global variable `dist_per_timestep` that specifies the average distance the end effector should travel per timestep (see discussion in section 4.7.2). For all other skills, the joint velocities are controlled by the global variable `angle_per_timestep`. This variable specifies the average joint angle change per timestep. In case of a skill that brings a limb into a target configuration, the variable `angle_per_timestep` will be applied to the joint that has to overcome the maximum angle. The angle stepsizes for



dhc	joint i	name	$\Theta_i$	$d_i$	$a_i$	$\alpha_i$
spine	0	<b>base: pelvis</b>	$0^\circ$	0	0	$-90^\circ$
	1		$-90^\circ$	0	0	$90^\circ$
	2		$0^\circ$	0	7.1	$0^\circ$
	3	waist	$-90^\circ$	0	0	$-90^\circ$
	4		$90^\circ$	0	0	$90^\circ$
	5		$90^\circ$	0	12.78	$-90^\circ$
	6	neck	$-90^\circ$	0	0	$-90^\circ$
	7		$90^\circ$	0	0	$90^\circ$
	8		$90^\circ$	0	4.26	$-90^\circ$
	9	head	$-90^\circ$	0	0	$-90^\circ$
	10		$90^\circ$	0	0	$90^\circ$
11		$90^\circ$	0	0	$-90^\circ$	

Table 4.2: Denavit-Hartenberg parameters for the spine

dhc	joint i	name	$\Theta_i$	$d_i$	$a_i$	$\alpha_i$	local motor program
l.arm	0	<b>base: chest</b>	$0^\circ$	0	0	$90^\circ$	move_shoulder
	1		$10^\circ$	0	0	$90^\circ$	lift/drop shoulder
	2		$0^\circ$	0	5.3	$0^\circ$	(not used)
	3		$90^\circ$	0	0	$90^\circ$	(not used)
	4		$100^\circ$	0	0	$90^\circ$	(not used)
	5		$0^\circ$	0	2.3	$0^\circ$	(not used)
	6	shoulder	$0^\circ$	0	0	$-90^\circ$	lift_arm_sideward
	7		$80^\circ$	0	0	$90^\circ$	lift_arm_forward
	8		$50^\circ$	0	0	$90^\circ$	turn_arm
	9		$90^\circ$	0	10	$0^\circ$	(dummy)
	10	elbow	$100^\circ$	0	0	$90^\circ$	bend_elbow
	11		$110^\circ$	0	0	$90^\circ$	turn_hand
	12		$90^\circ$	0	8.6	$90^\circ$	(dummy)
	13	wrist	$0^\circ$	0	0	$90^\circ$	bend_hand
	14		$0^\circ$	0	0	$0^\circ$	twist_hand
15		$0^\circ$	-1.27	2.5	$0^\circ$	ee: hand palm	

Table 4.3: Denavit-Hartenberg parameters for the left arm

dhc	joint i	name	$\Theta_i$	$d_i$	$a_i$	$\alpha_i$	angle name
index	0	<b>base: wrist</b>	$-12^\circ$	0	2.95	$0^\circ$	(fixed)
	1		$12^\circ$	0	0	$-90^\circ$	spread_finger
	2		$10^\circ$	0	1.14	$0^\circ$	bend_finger1
	3		$20^\circ$	0	0.83	$0^\circ$	bend_finger2
	4		$13.33^\circ$	0	0.64	$0^\circ$	bend_finger3

Table 4.4: Denavit-Hartenberg parameters for the left index

dhc	joint i	name	$\Theta_i$	$d_i$	$a_i$	$\alpha_i$	angle name
thumb	0	<b>base: wrist</b>	$-30^\circ$	0	1.35	$0^\circ$	(fixed)
	1		$-10^\circ$	0	0	$-90^\circ$	spread_thumb
	2		$30^\circ$	0	1.12	$0^\circ$	thumb_down
	3		$90^\circ$	0	0	$90^\circ$	(dummy)
	4		$90^\circ$	0	0	$-90^\circ$	thumb_turn
	5		$-60^\circ$	0	0.81	$0^\circ$	bend_thumb1
	6		$15^\circ$	0	0.64	$0^\circ$	bend_thumb2

Table 4.5: Denavit-Hartenberg parameters for the left thumb

all other joints will be synchronized accordingly so that all joints reach their target angle at the same timestep.

#### 4.4.2 Local Motor Programs

The main focus of the presented work was put on skills that can be performed with the arm, hand, and/or fingers, i.e. reaching and grasping. Based on the structure of a skeleton arm [8], the kinematic chain for the arm provides joints for the clavicle and the scapula (joints 0-5). Only two of them, joint 0 and joint 1, are used to move the shoulder for- and backward as well as up and down. For reasons explained in section 4.1.3, there are two dummy joints in the arm, i.e. joint 9 and joint 12. The remaining 7 joints control the following joint motions: arm lifting sideward, arm lifting for- and backward, arm turning (about the upper arm), elbow bending, hand turning, hand bending, and hand twisting. Each of those joint motions can be invoked by a corresponding local motor program, e.g. `bend_elbow by 50 25` causes the elbow to bend by 50 degrees in 25 timesteps (see `proc bend_elbow` on page 107 in appendix B).

For the arm, there are the following local motor programs available:

```

move_shoulder      option angle timesteps renopt
lift/drop_shoulder option angle timesteps renopt
lift_arm_sideward  option angle timesteps renopt
lift_arm_forward   option angle timesteps renopt
turn_arm           option angle timesteps renopt
bend_elbow         option angle timesteps renopt
turn_hand          option angle timesteps renopt
bend_hand          option angle timesteps renopt
twist_hand         option angle timesteps renopt

```

The parameters to a local motor program are the `option`, which specifies whether the appropriate joint should be moved “by” the specified angle or “to” the specified joint position, the joint `angle` (be it the absolute or relative one), the number of `timesteps` in which the motion should be performed, and the `renopt` that can be set to “ren” or “noren” depending on whether the image shall be updated on the screen, i.e. rendered, with every timestep or not.

Bending and turning or twisting inwards are considered to change a joint angle in positive direction. A fully extended or untwisted limb corresponds to a neutral joint angle of 0°.

For not every joint is a change in the corresponding *dhtheta* value (also referred to as *dhangle*  $\Theta$ , *dhparameter*  $\Theta$ , or simply  $\Theta$ ) consistent with the above convention. Sometimes, the direction of *dhtheta* has to be reversed to convert into the real joint angle, denoted as  $\Phi$ . Table 4.6 shows a “-1” in the “dir” column if this is the case. Also, there might be an offset between  $\Theta$  and  $\Phi$  (see column “off” in the same table). Finally, the conversion from  $\Phi$  to  $\Theta$  can be calculated by

$$\Theta = \Phi * dir + off \quad (4.3)$$

and is noted in the column named *dhangle*  $\Theta$ .

Table 4.6 also shows the joint limits for the joint angles  $\Phi$  and their corresponding limits in the *dhparameter*  $\Theta$  according to the conversion equation 4.3. The joint limits are recorded in the procedures `get_max_angle` and `get_min_angle` included in appendix B, page 117.

To access a current joint angle  $\Phi$ , there is a procedure `get_jointname_angle` provided for every joint that reads the *dhtheta* for the joint in question and converts it back to the real joint angle  $\Phi$ . See procedure `get_elbow_angle` on page 117 of appendix B for an example. Every local motor program consists of three main parts:

1. Joint limit test:

The procedure

```
in_range limb joint_nr option angle min_angle max_angle restpos
      off direc part action
```

(see appendix B, page 123) tests whether the desired motion exceeds the joint’s limits. If it does, the procedure returns an appropriate reachable angle closest to the one specified. Otherwise it returns the angle itself.

2. Angle conversion:

Conversion from the desired joint angle  $\Phi$  to the *dhangle*  $\Theta$ , depending on how the `option` parameter is specified (equation 4.3).

3. Joint angle change:

The change of the corresponding *dhtheta* value is invoked by the procedure

```
move_a_joint limb joint_nr option dhangle timesteps range_angle renopt
```

(see appendix B, page 125) which is the core of every local motor program. The `option` parameter can be set to “to” or “by” which provokes either a movement to a specified absolute angle or by a specified relative angle.

dhangle	name	$\Phi$ limits	dir	off	dhangle $\Theta$	$\Theta$ limits
$\Theta_0$	shoulder_fb	-20 - 40	-1	0	$-\Phi$	-20 - 40
$\Theta_1$	shoulder_ld	-5 - 45	1	0	$\Phi$	-5 - 45
$\Theta_6$	arm_ls	-180 - 180	1	0	$\Phi$	-180 - 180
$\Theta_7$	arm_lf	-90 - 190	-1	90	$-\Phi + 90$	180 - -100
$\Theta_8$	arm_turn	-170 - 175	1	0	$\Phi$	-170 - 175
$\Theta_{10}$	elbow	0 - 150	1	90	$\Phi + 90$	90 - 240
$\Theta_{11}$	hand_turn	-40 - 190	1	90	$\Phi + 90$	50 - 280
$\Theta_{13}$	hand_bend	-80 - 80	1	0	$\Phi$	-80 - 80
$\Theta_{14}$	hand_twist	-35 - 35	-1	0	$-\Phi$	35 - -35

Table 4.6: Transformation between dhangle and real joint angles and joint limits

Local motor programs to control the finger and thumb joints are:

```

spread_finger      finger      option angle timesteps renopt
bend_a_finger_joint finger joint option angle timesteps renopt
move_thumb_outwards      option angle timesteps renopt
turn_thumb          option angle timesteps renopt
move_thumb_down     option angle timesteps renopt
bend_thumb_joint    joint option angle timesteps renopt

```

These local motor programs are controlled by the same parameters as the local motor programs for the arm (explained earlier in the text). The `spread_finger` and `bend_a_finger_joint` procedures have an additional parameter to specify the finger in question, including the thumb.

When specifying “thumb” as input for the `finger` parameter, the procedure `spread_finger` calls the `move_thumb_outwards` procedure and is therefore in this case equivalent to the latter one. In the same way, the procedure `bend_a_finger_joint` with “thumb” as input for the `finger` parameter calls `bend_thumb_joint`, see the source code on page 107 in appendix B.

To control the remaining 2 dofs of the thumb, the procedures `move_thumb_down` and `turn_thumb` are provided. Refer to tables 4.4 and 4.5 (page 38) to find out which `dththeta` correspond to which local motor program.

Should all finger bending angles of one finger be bent by the same amount, the procedure

```

bend_one_finger finger option angle timesteps renopt

```

provided for convenience can be called.

As Rijpkema and Girard mention in [22], it is almost impossible for a human to move the last link (joint 4) of a finger without moving the next to last joint (joint 3) and vice-versa, without forcing one of the two to move in some unnatural way.

Experimental data taken by Rijpkema and Girard have led them to the assumption that the dependency of those last two joint angles could be reasonably approximated by a linear relationship:

$$\Theta_4 \approx 2/3 * \Theta_3 \quad (4.4)$$

The `bend_a_finger_joint` procedure therefore only accepts “1” or “2” as input for its parameter `joint` that specifies the number of the bending joint being moved. According to table 4.4 the first two bending joints of a finger correspond to the dhparameter  $\Theta_2$  and  $\Theta_3$ .  $\Theta_4$ , the dhparameter corresponding to the third bending joint will automatically be changed according to equation 4.4 with a change in  $\Theta_3$ . The last two links of the thumb are coupled in the same way.

### 4.4.3 Concurrent Execution

Performing composite skills often means that several atomic skills have to be executed concurrently (see section 4.7.3). Also, the actor might have different tasks at the same time, e.g. reach for a glass (which is a composite skill itself) and move his head and eyes towards the glass. A *command loop* is introduced to realize the concurrent execution of several procedures.

The idea of the command loop is to keep a list, the *command list*, of all commands that should execute concurrently and to loop constantly through the list. Every procedure that should execute at a particular time has an entry in the command list at that time. Procedures on the command list can be control procedures for specific motor programs, local motor programs, or procedures that provoke a change in the environment of the virtual actor, such as procedures to move an object.

The procedure controlling the command loop is the `server` that can be looked at on page 130 (in appendix B). After executing each of the commands on the command list once, the image of the screen is updated by calling the rendering routine. Processing several different local motor programs by small steps gives the impression of concurrent movement. The command loop can be started and run uninterruptedly or can be progressed step by step.

For example, bending the elbow by 50 degrees in 25 timesteps is realized by adding the command `bend_elbow 2 1` to the command list and removing it after 25 steps

through the command loop.

A procedure that returns either “limit”, “target”, or “stop” removes itself from the command list. Beyond that, motor program control procedures possess the ability to add or remove other procedures from the command list. The **server** takes into account that new procedures might have been added to the list, and that they should be executed before the end of the current step through the loop.

Access procedures that are able to add, insert, or remove commands from the command list are `add_c`, `insert_c`, and `rem_c` (included in appendix B pages 103, 123, and 129 respectively).

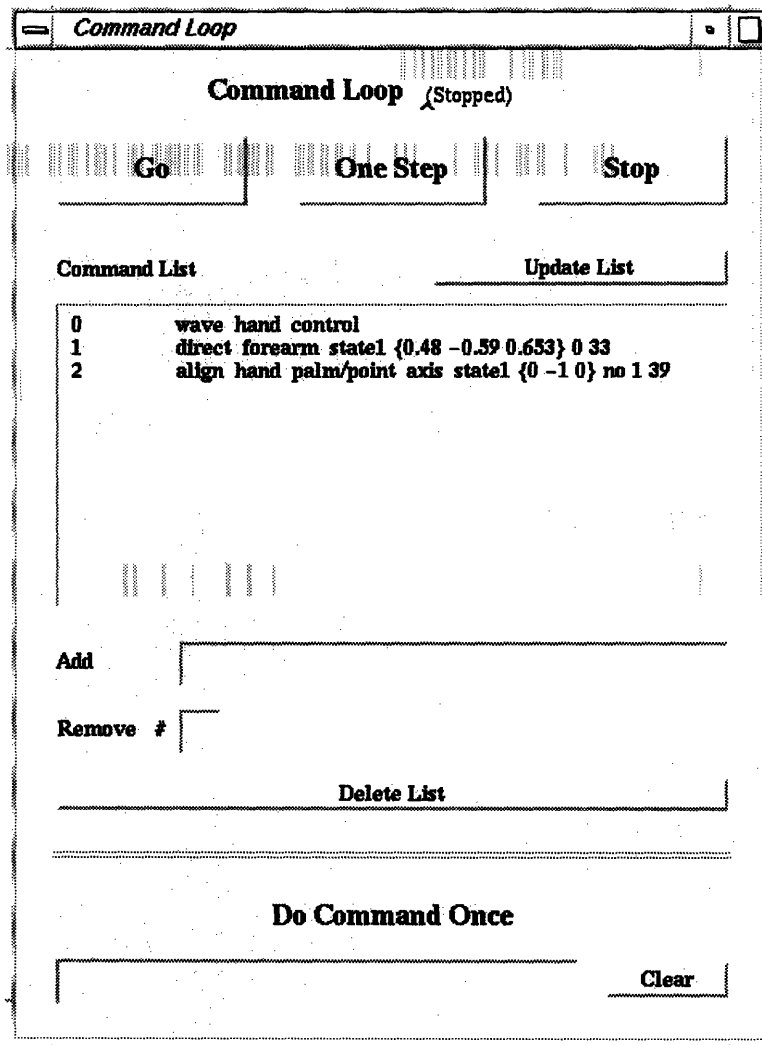


Figure 4-9: Command Loop menu

For convenience, an X menu is provided, i.e. the **Command Loop** menu, that shows the current command list and has buttons to start, stop or step through the list (see figure 4-9). At the end of every step through the list, all new x events are flushed. This way, the actor can react to any command typed in the **Do Command Once** text field at any time. Once the command loop is started, the actor can only be controlled by using the **Command Loop** menu.

The system has an internal clock implemented as a global variable that is increased by one timestep every step through the command loop.

## 4.5 Inverse Kinematics Algorithms

The basic inverse kinematics algorithm for the arm, explained in this chapter, is used to position either the wrist or the end effector of the hand at a specific point in space. The end effector is generally located in the center of the palm. This algorithm will be later extended in order to obtain appropriate joint angles for all joints involved to synchronously reach a target position and orientation of the end effector.

Four joint angles are taken into account by the basic inverse kinematics algorithm: the two angles of the spherical joint at the shoulder that determine the upper arm orientation, (i.e. the *arm-lifting-forward angle* (or *arm-lf angle*) and the *arm-lifting-sideward angle* (or *arm-ls angle*)), the *elbow-bending angle* (or *elbow angle*), and the *arm-turn angle*.

If the wrist is to be positioned, the hand angles (turning, bending, and twisting) do not play a role. For the positioning of the end effector of the hand, constant hand angles are assumed for now.

Having to determine 4 joint angles out of a 3 dimensional coordinate point in space leaves 1 degree of freedom. This is used to influence the rotation of the arm about the vector from the shoulder to the goal point. A parameter, called `arm_pose_parameter` is introduced in order to specify the additional degree of freedom. The parameter can take any value between 0 and 1, where 0 will result in a position with the elbow at the lowest possible point and 1 in an elbow position at the highest possible point. The elbow is always assumed to be on the outside of the plane defined by the upper arm, and the forearm in the lowest elbow position, as viewed from the center of the body. As a global variable, the `arm_pose_parameter` can be changed any time during the execution of inverse kinematics routines by the procedure `app` (included in appendix B on page 106).

The basic inverse kinematics algorithm is implemented in a procedure called `goal_arm_angles_abs` (see appendix B, page 118) that takes two parameters as input: the



`goal_point`, a three dimensional coordinate point in world space coordinates, and `gp_opt`, the *goal point option* that can be either set to “wrist” or “ee” to specify whether the wrist or the end effector is to be positioned at the given goal point. The procedure returns the absolute angles, i.e. the arm-ls angle, the arm-lf angle, the arm-turn angle, and the elbow angle, necessary to reach the goal point. The algorithm is divided into two major parts that will be explained in more detail in section 4.5.1 and 4.5.2. These are:

1. Calculation of the shoulder-elbow-vector and, in case `gp_opt` is set to “wrist”, calculation of the elbow-wrist-vector.
2. Inverse kinematics calculation to determine the joint angles.

Adaptation to a moving body, shoulder movements and/or changes in the hand angles will take place by recomputing the inverse kinematics algorithm every timestep.

Often it is desired not only to place the end effector or wrist at a specific point in space but also to reach a specific orientation of the hand. Imagine washing windows. The end effector should follow a looplike trajectory in the plane of the window, while the hand plane should stay parallel to the plane of the window all the time. Another example is grasping an object. If the actor grasps a glass, it should bring its end effector together with a specific point on the glass and simultaneously align its *thumb axis* (see figure 4-11 on page 53) with the longitudinal axis of the glass.

In section 4.5.3, the inverse kinematics calculations for aligning the hand axes will be described. Section 4.5.4 then shows how to integrate reaching for a goal point and the alignment of the hand.

### 4.5.1 Calculation of the shoulder-elbow-vector and the elbow-wrist-vector

Figure 4-10 shows the arm positioned such that the end effector (ee) coincides with a given goal point.

#### Calculation of the shoulder-elbow-vector

The vector from the shoulder to the elbow is calculated by the procedure `calc_shoulder_elbow_vector` (see appendix B, page 110). From all possible solutions for the shoulder-elbow-vector ( $s\vec{e}v$ ) one is chosen by taking the `arm_pose_parameter` into

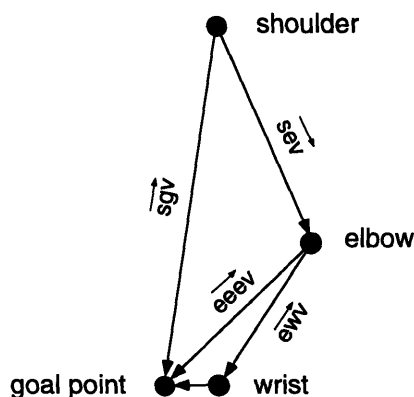


Figure 4-10: Vectors defining the arm links

account. In order to calculate  $\vec{s}e\vec{v}$ , the angle  $\gamma$  enclosed by  $\vec{s}e\vec{v}$  and the shoulder-goal-vector ( $\vec{s}g\vec{v}$ ) has to be computed first.  $\gamma$  can be computed out of the lengths of the vectors  $\vec{s}g\vec{v}$ ,  $\vec{s}e\vec{v}$ , and  $\vec{e}e\vec{v}$  (the vector from the elbow to the end effector):

$$\gamma = \arccos \left( \frac{|\vec{s}e\vec{v}|^2 + |\vec{s}g\vec{v}|^2 - |\vec{e}e\vec{v}|^2}{2 |\vec{s}e\vec{v}| |\vec{s}g\vec{v}|} \right) \quad (4.5)$$

Equation (4.5) is implemented in procedure `gamma` included in appendix B, page 117.  $|\vec{s}g\vec{v}|$ , i.e. the distance from the shoulder to the goal point, is known since the shoulder is assumed to remain at its current position,  $|\vec{s}e\vec{v}|$  is the length of the upper arm, and  $|\vec{e}e\vec{v}|$  is the distance between the current elbow position and the current end effector position that will not change if the hand angles stay the same.

`3d` provides a command that returns the end effector position (`dhgetee`), whereas the current positions of all joints can be accessed through the corresponding dhmatrices (see e.g. procedure `get_shoulder_position`, appendix B, page 118).

In case not the end effector, but the wrist, is to be positioned at the goal point,  $|\vec{e}e\vec{v}|$  has to be substituted by the length of the elbow-wrist-vector ( $|\vec{e}w\vec{v}|$ ), i.e. the forearm length.

All possible solutions for the shoulder-elbow-vector are located on a circle about  $|\vec{s}g\vec{v}|$ . Supposed  $z_{sev}$ , the z coordinate of  $|\vec{s}e\vec{v}|$  is known, (it will later be shown how to determine  $z_{sev}$ ), the other two coordinates can be computed using the following two equations:

$$|\vec{s}e\vec{v}|^2 = x_{sev}^2 + y_{sev}^2 + z_{sev}^2 \quad (4.6)$$

$$|\vec{s}e\vec{v}| |\vec{s}g\vec{v}| \cos \gamma = x_{sev} x_{sgv} + y_{sev} y_{sgv} + z_{sev} z_{sgv} \quad (4.7)$$

where equation (4.7) is nothing else but the definition of the dot product of two vectors.

Solving equations (4.6) and (4.7) yields two solutions for  $x_{sev}$  :

$$x_{sev1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q} \quad (4.8)$$

with

$$p = \frac{2 x_{sgv} (z_{sev} z_{sgv} - |s\vec{e}v| |s\vec{g}v| \cos \gamma)}{x_{sgv}^2 + y_{sgv}^2} \quad (4.9)$$

$$q = \frac{|s\vec{e}v| |s\vec{g}v| \cos \gamma (|s\vec{e}v| |s\vec{g}v| \cos \gamma - 2 z_{sev} z_{sgv}) + z_{sev}^2 (y_{sgv}^2 + z_{sgv}^2) - |s\vec{e}v|^2 y_{sgv}^2}{x_{sgv}^2 + y_{sgv}^2} \quad (4.10)$$

Equation (4.8) is implemented in procedure `calc_x2` included in appendix B, page 112. For the left arm, the solution with the greater  $x_{sev}$  will always be chosen so to keep the elbow turned “outside”, which is the more natural of the two solutions.  $y_{sev}$  will then be calculated using equation (4.6) (see procedure `calc_third_component`, appendix B, page 112). Whether the positive or negative  $y_{sev}$  is the valid solution will be verified by means of procedure `vangle2` (see appendix B, page 132) that calculates the angle between the resulting  $s\vec{e}v$  and  $s\vec{g}v$ . The resulting angle will be compared with the angle  $\gamma$  computed by equation (4.5).

A valid range for  $z_{sev}$  can be computed from the condition that the root argument in equation (4.8) has to be greater than or equal to zero (see procedure `calc_z2_range` in appendix A, page 112). The  $z_{sev}$  range will be mapped onto the `arm_pose_parameter` (app) mentioned earlier on page 106, where an `app=0` refers to  $z_{sev_{min}}$  and an `app=1` refers to  $z_{sev_{max}}$ .

The `calc_shoulder_elbow_vector` procedure also checks whether the computed solution would end up in an elbow position inside or too close to the body, in which case it would recompute the algorithm to find a better solution. The upper arm orientation results directly from the shoulder-elbow-vector.

### Calculation of the elbow-wrist-vector

If it is the wrist that is to be positioned at the goal point, the elbow-wrist-vector can be obtained easily. Subtracting the shoulder-goal-vector from the just calculated shoulder-elbow-vector yields the vector from the elbow to the wrist ( $e\vec{w}v$ ) :

$$e\vec{w}v = s\vec{e}v - s\vec{g}v \quad (4.11)$$

The elbow-wrist-vector determines the forearm orientation, i.e. the target axis of the forearm.

### 4.5.2 Inverse Kinematics Algorithms for Orienting the Arm Links

Inverse kinematics has to be applied in order to calculate the arm-ls and arm-lf angles that result in the desired orientation of the upper arm (procedure `upper_arm_angles_abs`). In case the `gp_opt` is set to “wrist” (refer to page 45), inverse kinematics must again be applied to calculate the elbow and arm-turn angle, given the orientation of the forearm (procedure `forearm_angles_abs`).

In case of a `gp_opt` “ee”, another inverse kinematics algorithm is used to calculate the elbow and arm-turn angle, given the resulting coordinate frame at the elbow joint, i.e. the dhmatrix  $\mathbf{M}_7$ .  $\mathbf{M}_7$  depends on the new arm-ls and arm-lf angles as well as on the goal point itself (procedure `calc_at_e`).

Knowing the dhmatrix  $\mathbf{M}_m$  of a joint  $m$  that is temporarily fixed in space and knowing parts of the target dhmatrix  $\mathbf{M}_n$  for a joint  $n$  located farther along the same *dhchain*, equation (4.2) (page 25) makes it possible to calculate joint angles for all joints in between  $m$  and  $n$ . After premultiplying equation (4.2) with the inverse of  $\mathbf{M}_m$

$$\mathbf{A}_{m+1} \cdot \mathbf{A}_{m+2} \cdot \dots \cdot \mathbf{A}_{n-1} \cdot \mathbf{A}_n = \mathbf{M}_m^{-1} \cdot \mathbf{M}_n \quad , \quad (n > m) \quad (4.12)$$

a set of equations can be formulated containing trigonometric functions of the joint angles in question. By skillful transformation of the resulting equation system, it is possible to eliminate the joint angles one by one.

#### Aligning the upper arm along a given axis: computing the upper arm angles

The upper arm angles arm-ls angle and arm-lf angle correspond to the dhangles  $\Theta_6$  and  $\Theta_7$ , which can be calculated according to matrix equation (4.12):

$$\mathbf{A}_6 \cdot \mathbf{A}_7 = \mathbf{M}_5^{-1} \cdot \mathbf{M}_7 \quad (4.13)$$

Taking the dhparameters for joints 6 and 7 from table 4.3 and putting them into the transformation matrix (4.1) (on page 25), yields

$$\mathbf{A}_6 = \begin{pmatrix} c_6 & 0 & -s_6 & 0 \\ s_6 & 0 & c_6 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{A}_7 = \begin{pmatrix} c_7 & 0 & s_7 & 0 \\ s_7 & 0 & -c_7 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.14)$$

where e.g.  $c_6$  denotes  $\cos \Theta_6$  etc.

The target axis of the upper arm that is given by the shoulder-elbow-vector is the third vector  $\vec{b}$  defining the coordinate frame at joint 7 and is therefore contained in dhmatrix  $\mathbf{M}_7$  as follows:

$$\mathbf{M}_7 = \begin{pmatrix} * & * & b_0 & * \\ * & * & b_1 & * \\ * & * & b_2 & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.15)$$

Multiplying  $\mathbf{A}_6$  and  $\mathbf{A}_7$  from equations (4.14) and defining

$$\mathbf{M}_5^{-1} \cdot \vec{b} = \vec{B} \quad (4.16)$$

the third column of equation (4.13) can be written as

$$\begin{pmatrix} c_6 s_7 \\ s_6 s_7 \\ c_7 \\ 0 \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ 0 \end{pmatrix} \quad (4.17)$$

In *tcl*, the calculation of vector  $\vec{B}$  is implemented as follows

```
Mset [dhmatrix 5 /l.arm/dhc]
Minvert
Mele 3 0 0
Mele 3 1 0
Mele 3 2 0
set B [Mxform $main_axis]
```

Solving equation (4.17) for  $\Theta_6$  and  $\Theta_7$ , results in

$$\Theta_{7a} = \arccos B_2 \quad \Theta_{7b} = -\arccos B_2 \quad (4.18)$$

$$\Theta_{6a} = \arccos\left(\frac{B_0}{\sin \Theta_7}\right) \quad \Theta_{6b} = -\arccos\left(\frac{B_0}{\sin \Theta_7}\right) \quad (4.19)$$

Depending on the joint limits (see table 4.6, page 41), valid solutions for  $\Theta_7$  and then for  $\Theta_6$  will be chosen. In case both solutions for  $\Theta_6$  or  $\Theta_7$  are valid, the one closer to the current joint position will be chosen. Sometimes both solutions are outside of the allowed joint angle range, in which case  $\Theta$  will be set to its closest limit and a warning message will be printed out saying that the target upper arm axis is not reachable.  $\Theta_6$  has to be verified using the second equation resulting from (4.17).

Finally,  $\Theta_6$  and  $\Theta_7$  are converted into the arm-ls and arm-lf angle, using equation (4.3). The described computation is implemented in procedure `upper_arm_angles_abs` (see appendix B, page 131).

#### Aligning the forearm along a given axis: computing the forearm angles

The following method, implemented in procedure `forearm_angles_abs` (appendix B, page 115), is applied in case the goal point is to be positioned at the wrist, (i.e. `gp_opt` is set to “wrist”). The computation of the forearm angles that correspond to dhangles  $\Theta_8$  and  $\Theta_{10}$  is very similar to the procedure described previously. The axis vector of the forearm, which is given by the elbow-wrist-vector, is contained in dhmatrix  $\mathbf{M}_{10}$  in the third column. Taking dhmatrix  $\mathbf{M}_7$  as input for the calculation, the application of equation (4.12) yields

$$\mathbf{A}_8 \cdot \mathbf{A}_9 \cdot \mathbf{A}_{10} = \mathbf{M}_7^{-1} \cdot \mathbf{M}_{10} \quad (4.20)$$

which results in

$$\begin{pmatrix} -s_{10}c_8 & s_8 & c_{10}c_8 & 0 \\ -s_{10}s_8 & -c_8 & c_{10}s_8 & 0 \\ c_{10} & 0 & s_{10} & 10 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \mathbf{M}_7^{-1} \begin{pmatrix} * & * & d_0 & * \\ * & * & d_1 & * \\ * & * & d_2 & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.21)$$

after multiplying the transformation matrices  $\mathbf{A}_8 \cdot \mathbf{A}_9 \cdot \mathbf{A}_{10}$ , where all known dhparameters are put in according to table 4.3.

With the definition

$$\mathbf{M}_7^{-1} \cdot \vec{d} = \vec{D} \quad (4.22)$$

equating the third column of matrix equation (4.21) yields

$$\Theta_{10a} = \arcsin D_2 \quad \Theta_{10b} = 180^\circ - \arcsin D_2 \quad (4.23)$$

$$\Theta_{8a} = \arccos\left(\frac{D_0}{\cos \Theta_{10}}\right) \quad \Theta_{8b} = -\arccos\left(\frac{D_0}{\sin \Theta_{10}}\right) \quad (4.24)$$

Again, a solution for  $\Theta_{10}$  is chosen depending on the limits for joint 10. In case of further ambiguity, the solution that is closer to the current joint angle is chosen. The solution for  $\Theta_8$  is verified using the second equation resulting from (4.21). After conversion of  $\Theta_8$  and  $\Theta_{10}$ , the procedure `forearm_angles_abs` returns the arm-turn angle and elbow angle.

Parameter  $M_7$  is assigned to the procedure `forearm_angles_abs`, which can either be set to the current dhmatrix  $M_7$ , in case the upper arm does not move, or to  $M_7$  calculated using new angles `arm-ls` and `arm-lf` as input (see procedure `calc_M7`, appendix B, page 109). For the inverse kinematics positioning of the wrist at a goal point,  $M_7$  is calculated using the `arm-ls` and `arm-lf` angles (returned by the procedure `upper_arm_angles_abs`).

#### Computing the forearm angles based on a given goal point for the end effector

This computation is similar to the previous ones. Since the hand angles are assumed to be constant and this time the goal point  $(g_0, g_1, g_2)$  that can be found in the fourth column of dhmatrix  $M_{15}$  is known, the problem can be reduced to the solution of

$$A_8 \cdot A_9 \cdot A_{10} \cdot A_{11} \cdot A_{12} \cdot A_{13} \cdot A_{14} \cdot A_{15} = M_7^{-1} \cdot M_{15} \quad (4.25)$$

After defining  $A_{\text{temp}} = A_{11} \cdot A_{12} \cdot A_{13} \cdot A_{14} \cdot A_{15}$ , equation (4.25) can be written as

$$\begin{pmatrix} -s_{10}c_8 & s_8 & c_{10}c_8 & 0 \\ -s_{10}s_8 & -c_8 & c_{10}s_8 & 0 \\ c_{10} & 0 & s_{10} & 10 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot A_{\text{temp}} = M_7^{-1} \cdot \begin{pmatrix} * & * & * & g_0 \\ * & * & * & g_1 \\ * & * & * & g_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.26)$$

where  $A_{\text{temp}}$  is known since it only depends on the hand angles. The elements of the matrix  $A_{\text{temp}}$  will be assigned to variables as follows

$$A_{\text{temp}} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.27)$$

With the definition

$$\mathbf{M}_7^{-1} \cdot \vec{g} = \vec{G} \quad (4.28)$$

the fourth column of matrix equation (4.26) looks as follows:

$$\begin{pmatrix} -a_{14}c_8s_{10} + a_{24}s_8 + a_{34}c_8c_{10} \\ a_{14}s_8s_{10} - a_{24}c_8 + a_{34}s_8c_{10} \\ a_{14}c_{10} - a_{34}s_{10} + 10 \\ 1 \end{pmatrix} = \begin{pmatrix} G_0 \\ G_1 \\ G_2 \\ 1 \end{pmatrix} \quad (4.29)$$

and yields

$$s_{10a,b} = \frac{-a_{23}(10 - G_2)}{a_{14}^2 + a_{34}^2} \pm \sqrt{\frac{a_{14}(a_{34}^2 + a_{14}(10 - G_2)^2 + a_{14}^2)}{(a_{14}^2 + a_{34}^2)^2}} \quad (4.30)$$

$$s_{8a,b} = \frac{2G_0a_{24}}{a_{24}^2 + (a_{14}s_{10} - a_{34}c_{10})^2} \pm \sqrt{\frac{(a_{24}^2 - G_0)(a_{14}s_{10} - a_{34}c_{10})^2 + (a_{14}s_{10} - a_{34}c_{10})^4}{(a_{24}^2 + (a_{14}s_{10} - a_{34}c_{10})^2)^2}} \quad (4.31)$$

In each case (solution *a* as well as solution *b* for each  $\sin \Theta$ ) there will be two solutions for  $\Theta$ , resulting from the arcsin function, yielding four solutions for each  $\Theta$ . Only two of the four solutions in each case are true when verifying them with one of the equations from 4.29. Of the two remaining solutions, one is picked that is within the joints limits or, if both are within the limits, the one closer to the current joint position is picked.

As in procedure `forearm_angles_abs`,  $\Theta_8$  and  $\Theta_{10}$  will be converted into the arm-turn angle and the elbow angle, which are the two return values of procedure `calc_at_e`.

### 4.5.3 Inverse Kinematics Algorithms for Orienting the Hand Axes

Figure 4-11 shows the definition of the hand axes on the left hand. There are two different inverse kinematics algorithms to calculate the necessary hand angles: one for alignment of the palm axis or palm and point axes with target vectors and another one for alignment of the thumb axis with the longitudinal axis of an object.



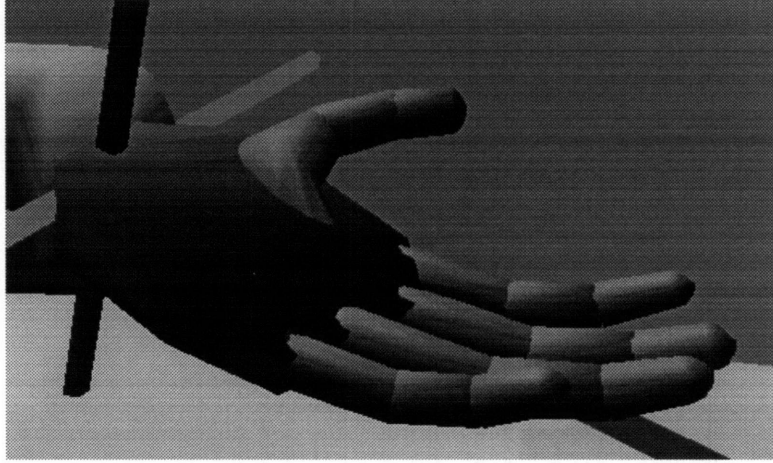


Figure 4-11: Definition of the hand axes shown on the left hand (inner palm)  
medium: point axis, light: thumb axis, dark: palm axis

### Computing the hand angles based on a target thumb axis

One way of grasping an object like a glass is to align the thumb axis with the longitudinal axis of the object, and then close the fingers around the object once approached. This kind of grasping is the only kind considered for this thesis.

The computation of the hand angles, given a thumb axis, is based on equation (4.12) so that the problem can be formulated as follows:

$$\mathbf{A}_{11} \cdot \mathbf{A}_{12} \cdot \mathbf{A}_{13} \cdot \mathbf{A}_{14} = \mathbf{M}_{10}^{-1} \cdot \mathbf{M}_{14} \quad (4.32)$$

The negative thumb axis is contained in dhmatrix  $\mathbf{M}_{14}$  in the second column

$$\mathbf{M}_{14} = \begin{pmatrix} * & -t_0 & * & * \\ * & -t_1 & * & * \\ * & -t_2 & * & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.33)$$

so that after further substitutions (4.32) yields

$$\begin{pmatrix} -s_{11} & s_{13} & s_{14} + c_{11} & c_{14} \\ c_{11} & s_{13} & s_{14} + c_{11} & c_{14} \\ & -c_{13} & s_{14} & \\ & & 0 & \end{pmatrix} = \mathbf{M}_{10}^{-1} \cdot \begin{pmatrix} -t_0 \\ -t_1 \\ -t_2 \\ 0 \end{pmatrix} = \begin{pmatrix} T_0 \\ T_1 \\ T_2 \\ 0 \end{pmatrix} \quad (4.34)$$

with the solutions

$$\Theta_{14a} = \arcsin \frac{T_2}{-\cos \Theta_{13}} \quad \Theta_{14b} = 180^\circ - \Theta_{14a} \quad (4.35)$$

$$\Theta_{11a} = \arcsin \left( \frac{T_1 \cos \Theta_{14} + T_0 T_2 \tan \Theta_{13}}{T_0^2 + T_1^2} \right) \quad \Theta_{11b} = 180^\circ - \Theta_{11a} \quad (4.36)$$

By checking the joint limits and considering the solutions closer to the current configuration,  $\Theta_{14}$  and  $\Theta_{11}$  are selected from (4.35) and (4.36).  $\Theta_{13}$  can be freely chosen and is therefore provided as an input parameter to the procedure `aligned_hand_angles_abs` (see appendix B, page 103), in form of the *hand-bend angle*. Another parameter to that procedure is  $\mathbf{M}_{10}$  that can either be set to the dhmatrix  $\mathbf{M}_{10}$  or to a matrix calculated depending on arm-ls, arm-lf, arm-turn and elbow angle (procedure `calc_M10`, appendix B, page 110).

### Computing the hand angles based on target palm and point axes

In the same way as explained in the previous sections, the hand angles for given palm and possibly also pointing axes are computed.

Dhmatrix  $\mathbf{M}_{14}$  contains the palm axis, named  $\vec{p}$ , and the pointing axis, named  $\vec{n}$  as follows:

$$\mathbf{M}_{14} = \begin{pmatrix} p_0 & * & -n_0 & * \\ p_1 & * & -n_1 & * \\ p_2 & * & -n_2 & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.37)$$

The right side of equation (4.32) can then be defined as

$$\mathbf{M}_{10}^{-1} \cdot \mathbf{M}_{14} = \begin{pmatrix} P_0 & * & N_0 & * \\ P_1 & * & N_1 & * \\ P_2 & * & N_2 & * \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.38)$$

which yields

$$\begin{pmatrix} s_{11} & s_{13} & c_{14} + c_{11} & s_{14} \\ -c_{11} & s_{13} & c_{14} + s_{11} & s_{14} \\ & & c_{13} & c_{14} \\ & & & 0 \end{pmatrix} = \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ 0 \end{pmatrix} \quad (4.39)$$

and

$$\begin{pmatrix} -s_{11} & c_{13} \\ c_{11} & c_{13} \\ s_{13} \\ 0 \end{pmatrix} = \begin{pmatrix} N_0 \\ N_1 \\ N_2 \\ 0 \end{pmatrix} \quad (4.40)$$

From (4.40) it becomes obvious that the *hand-twist angle* (corresponding to  $\Theta_{14}$ ) does not influence the palm axis. Hence, it is possible to specify only a palm axis as input for the hand angles calculation procedure `hand_angles_rel` (see appendix B, page 119). In this case the procedure returns the new *hand-turn angle* and the new hand-bend angle, but the current hand-twist angle.  $\Theta_{13}$  and  $\Theta_{11}$  will be

$$\Theta_{13a} = \arcsin N_2 \quad \Theta_{13b} = 180^\circ - \arcsin N_2 \quad (4.41)$$

$$\Theta_{11a} = \arctan\left(\frac{-N_0}{N_1}\right) \quad \Theta_{11b} = 180^\circ + \arctan\left(\frac{-N_0}{N_1}\right) \quad (4.42)$$

Is a point axis for the hand also given, then  $\Theta_{14}$  can be computed considering the third row in (4.39)

$$\Theta_{14a} = \arccos\left(\frac{P_2}{\cos \Theta_{13}}\right) \quad \Theta_{14b} = -\arccos\left(\frac{P_2}{\cos \Theta_{13}}\right) \quad (4.43)$$

#### 4.5.4 Inverse Kinematics Algorithm for Integrated Positioning and Orientation of the Hand Thumb Axis

For synchronous alignment of the hand thumb axis with a target axis while reaching a goal point, it is necessary to compute the final hand angles depending on the dhmatrix  $\mathbf{M}_{10}$  that results when the end effector is positioned at the goal point. Procedure `reach_angles_abs` (see appendix A, page 129) combines the `goal_arm_angles_abs` procedure that calculates the arm-ls, arm-lf, arm-turn, and elbow angle with the `align_hand_angles_abs` procedure (in which the 3 hand angles are computed). The matrix  $\mathbf{M}_{10}$  is computed from the four goal-arm-angles by the procedure `calc.M10`, mentioned earlier, and used as input for the computation of the hand angles.

## 4.6 Collision Avoidance

Collision avoidance will be of great importance once the actor moves in a more complicated environment, performing ever more complex tasks. The collision avoidance feature implemented as part of this thesis is limited to the generation of a collision

free path for the end effector when performing a reaching motion towards an object on the table.

According to the average distance, the end effector is expected to travel per timestep (`dist_per_timestep`), equidistant goal points along the path are generated. The end effector is supposed to move sequentially through all these goal points in order to avoid collision.

The procedure that calculates points on a path around a *collision object* (`collobj` parameter), i.e. the table in this example , is

```
find_path_pos object collobj showpath
```

The `showpath` parameter can be set to “yes” which will cause green balls to be displayed at the calculated path points for demonstration purposes.

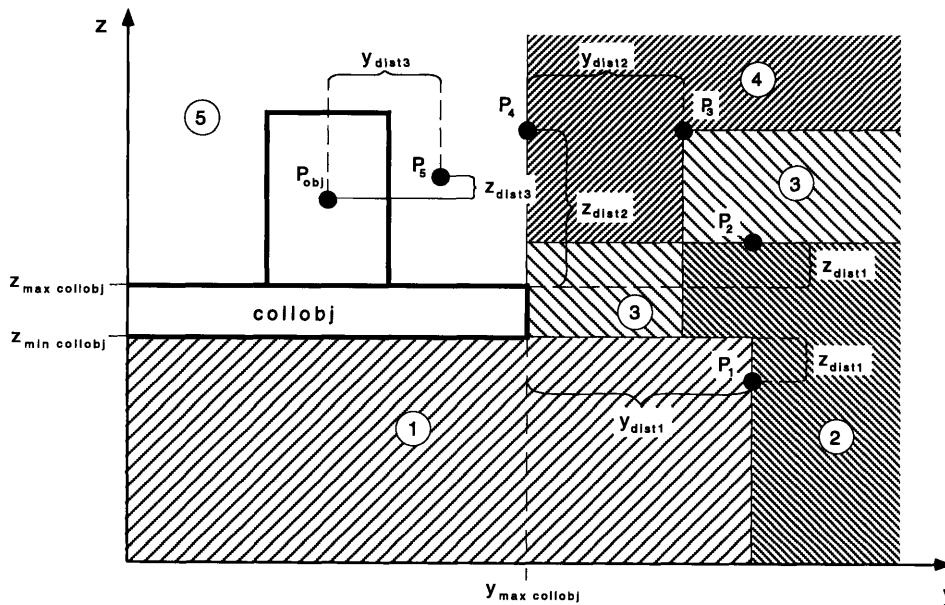


Figure 4-12: Division of the space to find a collision free path (vertical cross-section)

The space around the collision object viewed in the yz-plane is divided into several areas according to figure 4-12. The points  $P_1$  to  $P_5$  will be calculated depending on the position of the collision object and the object that is to be reached:

$$\begin{aligned}
 P_1 &= ( x_{p_1} , \quad y_{max\ collobj} + y_{dist1} , \quad z_{min\ collobj} - z_{dist1} ) \\
 P_2 &= ( x_{p_2} , \quad y_{max\ collobj} + y_{dist1} , \quad z_{max\ collobj} + z_{dist1} ) \\
 P_3 &= ( x_{p_3} , \quad y_{max\ collobj} + y_{dist2} , \quad z_{max\ collobj} + z_{dist2} )
 \end{aligned}$$

$$P_4 = ( x_{p_4}, \quad y_{max\ collobj}, \quad z_{max\ collobj} + z_{dist2} )$$

$$P_5 = ( x_{p_5}, \quad y_{obj} + y_{dist3}, \quad z_{obj} + z_{dist3} )$$

The following values have been chosen for:

$y_{dist1} = 5.2$  (accounting for the distance between the end effector and the tip of the middle finger plus a safety distance of 2, measured in inches).

$y_{dist2} = 3$

$y_{dist3} = 1$

$z_{dist1} = 1$

$z_{dist2} = 3.5$

$z_{dist3} = 0.4$

Depending on the current position of the end effector, the first point for the path will be chosen. If the end effector is located in area 3 for example, the first point on the path will be  $P_3$  followed by  $P_4$ ,  $P_5$ , and finally the object's centroid  $P_{obj}$ , see figure 4-13.

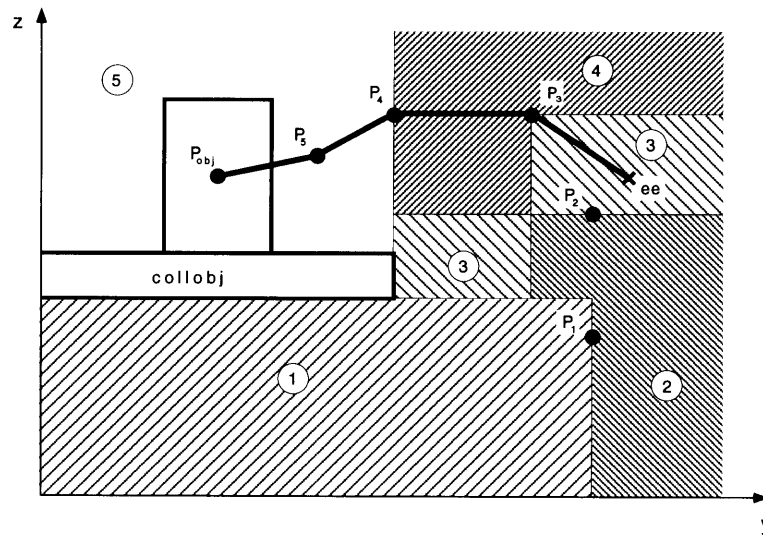


Figure 4-13: Collision free path starting from an end effector position in area 3

The  $x$  values of the path points depend on the  $x$  value of the current end effector position ( $x_{ee}$ ), and the position of the object on the table ( $x_{obj}$ ). If the body is very close to the table, and the end effector of the left hand is on the right side of the left shoulder joint, then the path should lead the end effector to move just slightly to the left of the shoulder joint at  $P_1$ :

```

if {($x_ee < $x_sp) & ($y_p1 > -5) & ($x_obj < $x_ee)} {
  set x_p1 [plus $x_sp 0.1]
} else {
  set x_p1 $x_eeep
}

```

In order to approach the object with the left hand from the left side,  $x_{p_5}$  will be set to

$$x_{p_5} = x_{obj} + 3$$

and the other x values

$$x_{p_2} = x_{p_1}, \quad x_{p_3} = x_{p_1} + \frac{x_{p_5} - x_{p_1}}{4}, \quad x_{p_4} = \frac{x_{p_1} + x_{p_5}}{2}$$

In case  $y_{p_5}$  is greater than  $y_{p_4}$ ,  $P_4$  will be skipped completely.

If the end effector is located in area 5, a new point  $P_{new}$  will be inserted between the end effector and  $P_5$  in case  $x_{ee}$  is less than  $x_{obj}$ , see figure 4-14. The coordinates of  $P_{new}$  will be

$$P_{new} = ( x_{obj}, y_{obj} + y_{dist1}, z_{obj} + z_{dist3} )$$

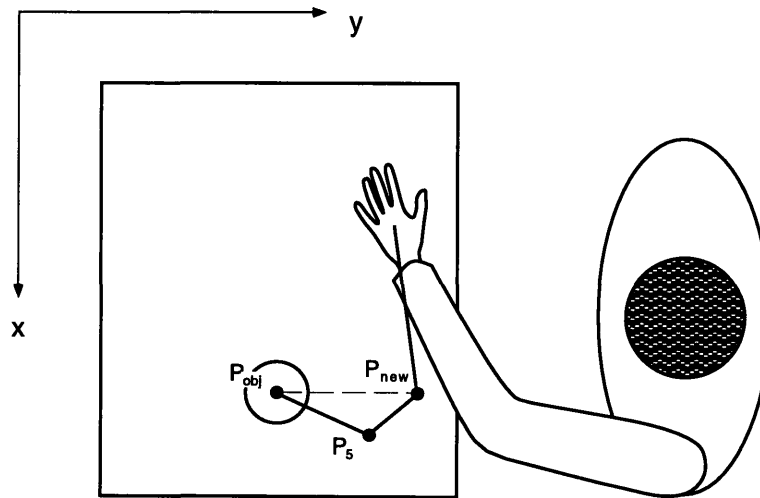


Figure 4-14: Insertion of  $P_{new}$  in case the end effector is located in area 5

In order to force the end effector to move along the collision free path smoothly, equidistant goal points have to be generated on the polygon that is defined by all path points included in the path. Procedure

```
make_equidistant_points point_list
```

implements the following algorithm:

1. Calculate the length  $l$  of the polygon through a point list ( $p_0$  to  $p_n$ ).
2. Parameterize the polygon, store the parameter  $t$  at each point of the point list ( $p_0 : t_0, \dots, p_n : t_n = l$ ).
3. Find the number of goal points by dividing the polygon length by the global variable `dist_per_timestep` and finding the closest integer ( $nr_{gp} = \text{int}(l/\text{dist\_per\_timestep})$ ).
4. Adapt the stepsize  $s$  accordingly ( $s = l/nr_{gp}$ ).
5. Set the first goal point to be the first point from the point list ( $gp_0 = p_0$ ).
6. Increase the parameter along the polygon by the stepsize ( $t_{gp_j} = t_{gp_{j-1}} + s$ ) and check between which points (from the point list) the new goal point  $gp_j$  is located ( $t_i < t_{gp_j} < t_{i+1}$ ).
7. Calculate the coordinates of the new goal point from the parameter  $t_{gp_j}$  and the linear equation formed by  $p_i$  and  $p_{i+1}$ .
8. Repeat steps 6 and 7 until the end of the point list.
9. Return a list with all goal points calculated.

Figure 4-15 shows the points of the original point list as black circles labeled  $P_1$  to  $P_5$  and the original polygon as a thin line. The new equidistant goal points are marked by crosses and the new polygon is drawn bold.

Finally, the procedure `equidistant_path_points` combines both previously explained procedures by providing the `point_list` returned by `find_path_pos` as input for `make_equidistant_points`.

The implemented collision avoidance algorithm just described is specific for the left arm and guarantees that the hand will move from its current position to the object without colliding with a specified collision object, i.e. the table. However, the algorithm does not yet guarantee collision free moves of the other arm links, i.e. the upper arm and the forearm. In order to avoid collisions of these arm links it is necessary to specify an appropriate `arm_pose_parameter`. Remember, changing this parameter will influence the position of the elbow, but not the end effector position.

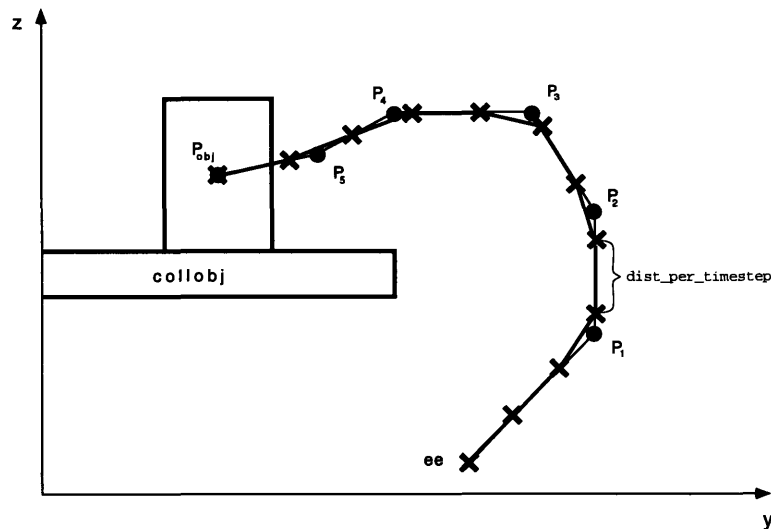


Figure 4-15: Generation of equidistant points on the collision free path

## 4.7 General Structure of Motor Programs

Motor programs are the source code for an actor's skills and implemented as finite state machines as explained in section 3.4.2.

### 4.7.1 Motor Programs for Atomic Forward Kinematics Skills

A motor program for an atomic skill can consist of several *tcl* procedures, but has at least two: the main motor program procedure and at least one state.

The name of the main motor program procedure is the skill's name. Therefore the execution of a skill is invoked by typing the skill's name in the **Do Command Once** text field of the **Command Loop** menu (see figure 4-9) if the command loop is already running. Otherwise, the skill's name can be typed at the `3d >` prompt and the command loop can be started thereafter by clicking the **Go** button in the **Command Loop** menu.

The task of the main motor program procedure is to check whether the initial conditions for a skill are fulfilled and to add the procedure for the first state to the command list. In case the initial conditions are not fulfilled, the procedure will quit and print out a warning message. The general structure of such a main motor program procedure looks like the following:



```

proc skill_name { args {end_event {}} } {
  #
  #   Initial condition
  #
  set condition0 [condition_procedure args_C0]
  if $condition0 then {
    global skill_name_end_event
    set skill_name_end_event $end_event
    add_c [list skill_name_state1 args_1]
  } else {
    echo Sorry, the initial conditions for executing this skill are not given!
  }
  return
}

```

“*args*” stands as a place holder for arbitrarily many arguments that the procedure might need, for instance as input to the condition procedures, or for one of the following states. The parameter *end\_event* is contained in any atomic skill.

For every state in a finite state machine (refer to figure 3-2 on page 21), there is a separate procedure named *skill\_name\_stateM*, where *M* is the number of the respective state. The following shows the general structure of such a procedure for an intermediate state:

```

proc skill_name_stateM {args_M} {
  #
  #   State ending condition
  #
  set conditionMa [condition_procedure_Ma args_CMa]
  set conditionMb [condition_procedure_Mb args_CMb]
  if $conditionMa then {
    add_c [list skill_name_stateN args_N]
    return stop
  }
  if $conditionMb then {
    add_c [list skill_name_stateP args_P]
    return stop
  }
  local_motor_programM1 by angle 1 noren
  local_motor_programM2 by angle 1 noren
  .....
  local_motor_programMn by angle 1 noren
  return
}

```

Every state is determined by several state ending conditions, and a number of local motor program procedures that cause different limbs to move concurrently while a state is being executed. As long as none of the ending conditions of a state is true, the state remains on the command list and all local motor program procedures belonging to that state are executed once every step through the command loop.

In the example procedure, the state M branched out to either state N or state P, depending on whether condition Ma or condition Mb becomes true. Accordingly, the state N or state P procedure will be added to the command list. In general, a state could have any number of ending conditions (which, by themselves, could consist of a compound of different subconditions). Any network of “and” and “or” connections between the subconditions can make up a state’s ending condition.

If a state procedure returns “stop”, the state is automatically removed from the command list. Refer to the source code of the `server` procedure that runs the command loop (page 130) to see how this is implemented.

The last state denoted here as state X will evaluate the end event upon satisfaction of its ending condition X, but might branch out again, if other conditions come true. This possible branching is not included in the example procedure for a *skill\_name\_stateX*:

```
proc skill_name_stateX {args_X} {
  #
  #   State ending condition
  #
  set conditionX [condition_procedure_X args_CX]
  if $conditionX then {
    global skill_name_end_event
    eval [set skill_name_end_event]
    return stop
  }
  local_motor_programX1 by angle 1 noren
  local_motor_programX2 by angle 1 noren
  .....
  local_motor_programXn by angle 1 noren
  return
}
```

Given as a parameter to the main motor program procedure was the `end_event`. This parameter is made accessible to the last state X by being defined as a global variable in the procedure *skill\_name*. As a default, the `end_event` is set to “{ }” so that the evaluation of a skill’s end event will have no effect. The procedure then goes on to

the command `return stop` which forces the skill to end, and causes the last state to be removed from the command list.

There are cases where a changing environment makes it necessary to perform the same skill over and over again, because once the skill is fulfilled, the initial conditions might come true again. By defining the *skill\_name* itself as `end_event`, the skill's first state will replace the skill's last state.

In case the skill consists of just one state and it is desired to execute that skill again, any time the initial conditions for the skill are satisfied, the `end_event` has to be set to "return". The procedure will then return before it reaches the command `return stop`, so it will not yet be removed from the command list. This way, the state1 for that particular skill will remain on the list until it will get removed e.g. when another skill's end event comes true.

With the explained method, an adaptive reaching algorithm for a moving object can be implemented. Once the object is reached, the ending condition will come true, but the reaching state would not be removed from the command list. Any time the object moves farther away in one timestep than a given *goal-point-precision*, the reaching algorithm would be activated again.

An example of a forward kinematics skill `put_arm_on_table` will be discussed in section 5.5.

#### 4.7.2 Motor Programs for Atomic Inverse Kinematics Skills

Inverse kinematics skills are determined by an inverse kinematics calculation routine. They consist of only one state containing all local motor programs involved in a specific movement. The inverse kinematics calculation routine synchronizes the angle steps of all these local motor programs, depending on how many timesteps the skill should take until its fulfillment.

An example of a main motor program procedure for an inverse kinematics skill is included in appendix A (page 101), and accounts for an initial condition composed of many single *condition\_procedures*.

Besides the parameter `end_event` which, as mentioned earlier, is contained in any atomic skill, the inverse kinematics skills will all contain the parameters `time_opt` and `timesteps`. If the parameter `time_opt` is set to "1", the time when the execution of the skill has to be finished, `end_time`, has to be calculated depending on the state of the internal clock. During the execution of the *IK\_skill\_name.state1* (see appendix A, page 102), the `end_time` will be compared with the internal clock `ic`, and the number of the remaining timesteps will be set accordingly. It will be guaranteed that

the number of timesteps never drops below one, even though the `end_time` for the skill might be exceeded.

Usually, and as a default, the `time_opt` parameter will be set to “0”, which means that the remaining number of timesteps will be calculated again every step by a *timesteps\_calculation\_procedure* to control the end effector or joint velocities.. For example, the procedure `calc_dist_timesteps` (see appendix A, page 109) calculates the timesteps necessary to overcome a certain distance depending on the value of the global variable `dist_per_timestep` that specifies the average distance an end effector should travel per timestep. This global variable can be changed any time, even during the execution of the skill, using the procedure `dps`.

Again, the parameter “args” accounts for any number of parameters specific to a skill (see page 61).

The *inverse\_kinematics\_calculation\_procedure* first calculates the final absolute joint angles for the involved limbs, then converts the absolute angles into relative joint angles necessary to reach the final goal (depending on the current state of the joints), and finally returns the relative angle steps every joint has to move during the next step. The local motor programs will only be activated if the corresponding angle step size is greater than 0.001.

### 4.7.3 Motor Programs for Composite Skills

Atomic skills can be combined to composite skills in different ways:

1. Parallel, independent:

Two or more atomic skills are added to the command list at the same time so that they start executing synchronously. Every atomic skill executes independently from the others. That means that it will be removed from the command list the moment it satisfies its ending conditions, independent of the duration of the other atomic skills that might be different. See figure 4-16 for the finite state machine model. The implementation is fairly easy:

```
proc composite_skill_name { args } {
    skill_name1 args_S1
    skill_name2 args_S2
    return
}
```

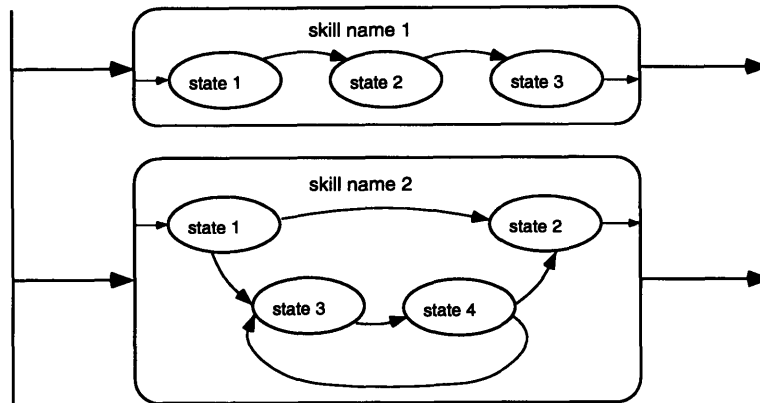


Figure 4-16: Model of a composite skill composed of parallel, independent atomic skills

2. Parallel, dependent:

As in the former case, all atomic skills will start executing synchronously, but in this case, their ending conditions are coupled. That is to say, all atomic skills corresponding to the same composite skill will be removed from the command list only when they all satisfy their corresponding ending conditions at the same time (see figure 4-17). To couple the ending conditions, all atomic skills but one will remain on command list, after satisfying their individual ending conditions. One of the atomic skills has to evaluate the combination of all individual ending conditions and, in case they are all satisfied, remove the atomic skills altogether from the command list.

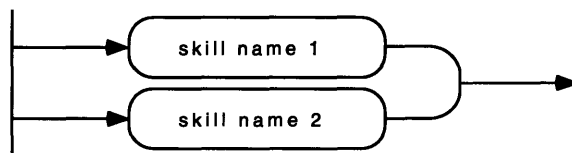


Figure 4-17: Model of a composite skill composed of parallel, dependent atomic skills

Keeping an atomic skill on the command list can be realized by defining the name of that particular atomic skill as its own `end_event` as explained earlier on page 63. This way, an atomic skill will add its first state again to the command list upon completion.

The evaluation of all ending conditions can be done in an extra procedure defined for that particular purpose. The skill, that has the `ending_condition_evaluation_procedure` as one of the commands of its `end_event` removes the other

skills directly from the command list by using the `rem_c` procedure and will be removed itself by returning “stop”.

```

proc composite_skill_name { args } {
  skill_name1 args_S1 {skill_name1 args_S1}
  skill_name2 args_S2 {skill_name2 args_S2}
  skill_name3 args_S3 {
    ending_condition_evaluation_procedure
    rem_c {skill_name1*}
    rem_c {skill_name2*}
  }
  return
}

```

### 3. Sequentially:

Feeding another *skill\_name2* as `end_event` to a *skill\_name1* will link those two skills to execute sequentially.

```

proc composite_skill_name { args } {
  skill_name1 args_S1 {
    skill_name2 args_S2 {}
  }
  return
}

```

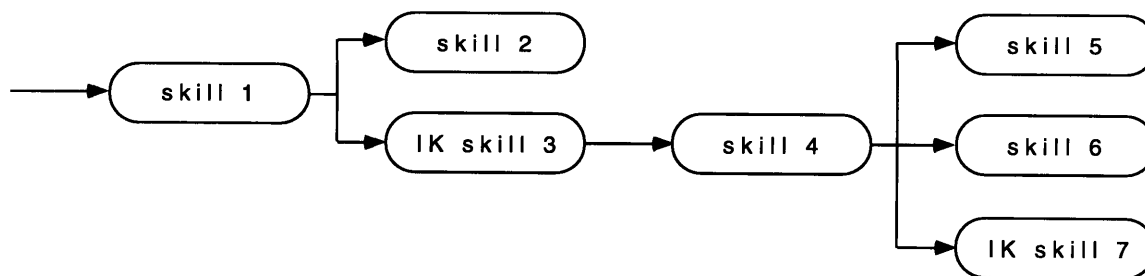


Figure 4-18: Model of a composite skill composed of a network of atomic skills

A more complex network of atomic skills making up a composite skill is shown in figure 4-18. The implementation follows:

```

proc composite_skill_name { args } {
    skill_name1 args_S1 {
        skill_name2 args_S2 {
            IK_skill_name3 args_S3 time_opt timesteps {
                skill_name4 args_S4 {
                    skill_name5 args_S5 {}
                    skill_name6 args_S6 {}
                    IK_skill_name7 args_S7 time_opt timesteps {}
                }
            }
        }
    }
    return
}

```

Note that the end event can consist of a list of commands. In this way it is possible to start several atomic skills parallel, upon completion of a previous skill. The skills that should execute sequentially are nested as each other's end events.

In this section it has been shown how to generalize the generation of motor programs using finite state machines. Every motor program consists of a compound of tcl procedures, i.e. one for each state and a main motor program procedure that checks the initial conditions. The definition of each state consists mainly of the identification of local motor programs and a definition of the ending conditions for that particular state. Local motor programs define the action of a state and the ending conditions define what conditions have to be satisfied in order to proceed to the next state.

A distinction has been made between forward and inverse kinematics skills. It has been demonstrated how to compose more complex skills from atomic skills in a parallel or sequential manner, or a combination of both.

During the development of the general structure of motor programs it became clear that there is a variety of arguments that might have to be passed to specific skills and a few arguments that are always the same, e.g. the `end_event` has to be specified for every skill. Other arguments might be the `goal_point`, i.e. the final position the end effector should reach in a skill `move_arm_to_goal` (as will be shown in section 5.2.1).

The next section will explain more about condition procedures that can check initial, as well as ending, conditions and introduce some of them.

Chapter 5 will discuss the implementation of different kinds of skills represented by motor programs according to the generalization worked out in this section.

## 4.8 Condition Procedures

Condition procedures are boolean procedures that check whether certain conditions are fulfilled or not and return either “1” or “0” accordingly.

Those procedures can be called to either check the initial conditions for a skill or to check the ending conditions of a state in a skill in order to decide when to go on to the next state and which state should be executed next (refer to the discussion of the motor program design in chapter 3.4.2). In general, the same procedures that function as ending conditions can also function as initial conditions. However, it has not yet been attempted to explore initial conditions specifically since this will later be the task of the skill network. A skill will then only be invoked by the skill network if the initial conditions specified in the skill template prove to be given. For now it will be taken for granted that the initial conditions for a skill are satisfied.

### 4.8.1 Classification of Condition Procedures

There is a great variety of ending conditions. Most of them can be classified in the following way, where conditions 1. - 8. represent geometric conditions and 9. and 10. conditions that do not imply a geometric calculation<sup>1</sup> :

#### 1. Geometric relationships between a joint and an object

Comparing the location of a joint with the bounding box of an object.

- (a)  $x_{joint}$  greater than  $x_{max\_obj}$ ,  $y_{joint}$  greater than  $y_{max\_obj}$ , or  $z_{joint}$  greater than  $z_{max\_obj}$ .
- (b)  $x_{joint}$  less than  $x_{min\_obj}$ ,  $y_{joint}$  less than  $y_{min\_obj}$ , or  $z_{joint}$  less than  $z_{min\_obj}$ .
- (c) Joint within specified distance from object.

E.g. “Is the wrist under the table, above the table or in front of it?”

#### 2. Geometric relationships between a body part and an object

Comparing the bounding boxes of a body part and an object.

---

<sup>1</sup>The following notion will be used:

bounding box of an object =  $(x_{min\_obj}, y_{min\_obj}, z_{min\_obj}, x_{max\_obj}, y_{max\_obj}, z_{max\_obj})$ ,

bounding box of a body part =  $(x_{min\_bp}, y_{min\_bp}, z_{min\_bp}, x_{max\_bp}, y_{max\_bp}, z_{max\_bp})$ ,

joint position =  $(x_{joint}, y_{joint}, z_{joint})$ , and

target position =  $(x_{target}, y_{target}, z_{target})$



- (a)  $x_{min\_bp}$  greater than  $x_{max\_obj}$ ,  $y_{min\_bp}$  greater than  $y_{max\_obj}$ , or  $z_{min\_bp}$  greater than  $z_{max\_obj}$ .
- (b)  $x_{max\_bp}$  less than  $x_{min\_obj}$ ,  $y_{max\_bp}$  less than  $y_{min\_obj}$ , or  $z_{max\_bp}$  less than  $z_{min\_obj}$ .

E.g. “Is the glass located to the right or to the left of the forearm?”

### 3. Geometric relationship between joints

Comparing the location of two joints.

- (a)  $x_{joint1}$  equal  $x_{joint2}$ ,  $y_{joint1}$  equal  $y_{joint2}$  or  $z_{joint1}$  equal  $z_{joint2}$ .
- (b)  $x_{joint1}$  less than  $x_{joint2}$ ,  $y_{joint1}$  less than  $y_{joint2}$ , or  $z_{joint1}$  less than  $z_{joint2}$ .
- (c)  $x_{joint1}$  greater than  $x_{joint2}$ ,  $y_{joint1}$  greater than  $y_{joint2}$ , or  $z_{joint1}$  greater than  $z_{joint2}$ .

E.g. “Is the wrist located below or above the elbow?” or “Is the elbow positioned to the right or to the left of the shoulder joint?”

### 4. Location of a joint

Comparing the location of a joint with a target position.

- (a)  $x_{joint}$  equal  $x_{target}$ ,  $y_{joint}$  equal  $y_{target}$ , or  $z_{joint}$  equal  $z_{target}$ .
- (b)  $x_{joint}$  less than  $x_{target}$ ,  $y_{joint}$  less than  $y_{target}$ , or  $z_{joint}$  less than  $z_{target}$ .
- (c)  $x_{joint}$  greater than  $x_{target}$ ,  $y_{joint}$  greater than  $y_{target}$ , or  $z_{joint}$  greater than  $z_{target}$ .
- (d) Joint position  $(x_{joint}, y_{joint}, z_{joint})$  equal target  $(x_{target}, y_{target}, z_{target})$ .
- (e) Joint position within specified distance of object.

E.g. “Is the wrist positioned at a height of  $z=75$  ?” or “Is the end effector positioned at a specified goal point?”

### 5. Orientation of a body part

Measuring the angle a body part axis includes with a target axis.

- (a) Body part orientation equal target orientation
- (b) Body part orientation aligned with a specified axis of an object

E.g. “Does the forearm point in the target direction?” or “Is the thumb vector aligned with the longitudinal axis of the glass?”

**6. Angular constraints**

Comparing a joint angle with a target angle.

- (a) Joint angle equal target angle
- (b) Joint angle greater than target angle
- (c) Joint angle less than target angle
- (d) Configuration of joint angles of a limb equal target configuration

E.g. “Is the elbow angle less than 130 degrees?” or “Is the arm positioned in the target configuration?”

**7. Contact between a body part and an object**

Checking for any collision between a body part and an object.

E.g. “Do the fingers touch the table?” or “Does the left toe touch the ground?”

**8. Contact between body parts**

E.g. “Does the palm touch the lower body?” or “Do the upper right and the upper left leg touch each other?”

**9. Change in one of the global variables of the SkillBuilder**

Checking, for example, whether the `arm_pose_parameter` changed since the last timestep.

**10. Ending condition of another skill currently being executed is satisfied**

The condition mentioned last can be used for composite skills composed of parallel dependent atomic skills (refer to section 4.7.3).

In many cases, a condition procedure is very general and is made suitable for a specific skill state by passing the appropriate arguments to the procedure. However, sometimes condition procedures have to be generated that are very specific for one skill. Those procedures can be of any of the above mentioned kinds and will have to be developed at an ad hoc basis when defining new skill states. An example is given for the 2. kind of condition procedures on page 71 that is specifically for the `reach_ceiling` skill.

**4.8.2 Implemented Condition Procedures**

So far, only condition procedures needed for the skills introduced in this thesis have been implemented.

**Condition 1:**

Very general is the procedure

```
joint_in_box joint object option
```

for which the `option` parameter can be specified as “in”, “x\_plus\_in”, “y\_plus\_in”, “z\_plus\_in”, “x\_minus”, “y\_minus”, “z\_minus”, “x\_minus\_in”, “y\_minus\_in”, “z\_minus\_in”, “x\_minus”, “y\_minus”, “z\_minus”. If, for example, the option is set to “z\_plus\_in” (see figure 4-19), the procedure checks, whether the joint is located above the object, but inside the horizontal bounding square of the object, so that a call `joint_in_box wrist table z_plus_in` is only true, if the wrist is located above the table plane. If it only plays a role, whether the wrist is located at a point higher than the table plane, option “z\_plus” should be used. Option “in” tests, whether a joint is located inside the bounding box of an object.

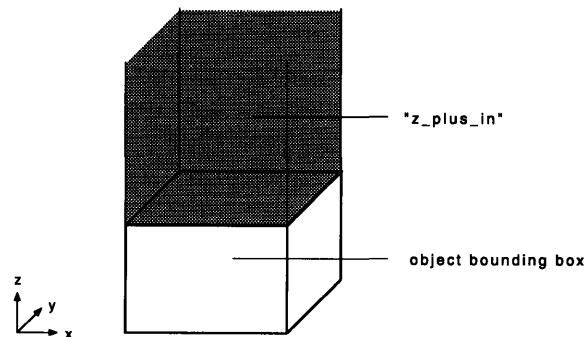


Figure 4-19: Option “z\_plus\_in” for the `joint_in_box` condition procedure

**Condition 2:**

The condition procedure

```
hand_on_ceiling ceiling_object
```

is true when the highest point of the hand is located above the lowest point of an object modeling the ceiling. This condition procedure is already quite specific since the body part in question, i.e. the hand, is hardcoded. In the case of the hand this is necessary because the compound of all bounding boxes of all fingers and the palm should be taken into account.

**Conditions 4:**

The following condition is true when the end effector (or in case `gp_opt` is set to “wrist” the wrist) coincides with a given goal point within the `goal_point_precision`:

```
goal_reached goal_point gp_opt
```

The `goal_point_precision` is a global variable that can be changed any time using procedure `gpp`.

If the distance between a goal point or the centroid of an object and the end effector (or in case `gp_opt` is set to “wrist” the wrist) is less than the length of the stretched out arm, the following conditions are satisfied:

```
goal_in_reaching_distance goal_point gp_opt
object_in_reaching_distance object gp_opt
```

**Conditions 5:**

Comparing the orientation of a body part with a target orientation is done by the procedures

```
test_hand_direction axis_opt target_axis
test_forearm_direction target_axis
test_upper_arm_direction axis_opt target_axis
```

For the hand, the possible `axis_opts` are “point”, “palm”, and “thumb”, specifying which of the hand axes is the one in question. The upper arm will generally be used with an `axis_opt` “main”, specifying the axis from the shoulder joint to the elbow. However, a possibility to specify “perp” is given, where “perp” denotes an axis perpendicular to the upper arm’s main axis in the plane defined by the upper arm’s and the forearm’s axes.

**Conditions 6:**

The following condition procedures check whether a specified joint angle configuration for the arm, a finger, or the thumb are reached by comparing the sum of the deviations of the current configuration from the target configuration with the global variable `angle_precision`:

```

arm_config_reached config
finger_config_reached finger config
thumb_config_reached config

```

The target configurations are passed to the procedures as n dimensional vectors, with n = 9 for the arm, n=3 for a finger and n=5 for the thumb. Procedure

```

arm_joint_angle_reached joint_name target_angle

```

serves to check whether a target angle for a specific arm joint is reached.

#### Conditions 7:

The contact between a body part and an object is detected when a collision takes place. Since there was no collision at the previous timestep and the stepsize by which the body parts move are very small, the collision depth is also very small. This way it is allowed to define the “collision” as contact between the body part and the object. The following procedures have been provided:

```

collision_fingers object
collision_lower_arm object
collision_fingertip finger object
collision_hand object

```

The procedure `collision_fingers` tests for collision of each link of all fingers with the object. The `collision_hand` procedure checks only for collision of the hand palm with an object.

#### Conditions 9:

Procedure

```

same_posture

```

checks whether the `arm_pose_parameter` changed since the last timestep and

```

reaching_problems

```

returns the boolean value of the global variable with the same name. This variable might have been set to “1” during the execution of inverse kinematics calculation procedures to indicate that the specified goal point cannot be reached.

### 4.8.3 Linking several Condition Procedures

An ending condition can be composed of any number of condition procedures by connecting them logically.

For example, if a state should only branch to state X, if condition 1 and condition 2 and (condition 3 or condition 4) are true, its ending condition might look as follows:

```
set condition_X [list [condition_procedure_1] & [condition_procedure_2] & \  
                    [list [condition_procedure_3] | [condition_procedure_4] ] ]
```

# Chapter 5

## Visually Guided Motion

### 5.1 Orienting a Body Part

All motor programs for skills to orient a body part, e.g. pointing with the forearm in a specified direction or aligning the thumb axis with an object axis are based on one of the *inverse-kinematics-calculation-procedures* introduced in chapter 4.5. They belong to the category of motor programs for atomic inverse kinematics skills (see section 4.7.2).

In how many timesteps the target orientation will be reached by a body part depends on the value of the global variable `angle_per_timestep` if the `time_opt` parameter to the *orient-body-part* skill is set to “0”. A `time_opt` parameter “0” means that the number of timesteps should be calculated by a *timesteps-calculation-procedure* such as `calc_angle_timesteps` (appendix A, page 108) in the case of *orient-body-part* skills. If the `time_opt` parameter is set to “1” the procedure will read the number of timesteps from the `timesteps` parameter instead of calculating it. The `angle_per_timestep` can be changed at any time by using the procedure `aps` (appendix A, page 106).

More complicated composite skills will often have one or more of the atomic *orient-body-part* skills as their elements. The name of *orient-body-part* skills often starts with `direct_***`. An example of how to incorporate the `direct_upper_arm`, `direct_forearm` and `direct_hand` skills in a skill called `wave_hand` will be shown in section 5.6.

#### 5.1.1 Orienting the Upper Arm

The atomic skill to orient the upper arm along a given vector is

```
direct_upper_arm main_axis perp_axis time_opt timesteps end_event
```

The `main_axis` has to be a 3 dimensional vector specifying a target orientation for the upper arm. It does not necessarily have to be a unit vector since the *inverse-kinematics-calculation-procedure* the motor program is based on (i.e. `upper_arm_angles_abs`) will make it a unit vector. The `perp_axis` parameter is usually set to “no”, meaning that no specific perpendicular axis has to be reached. With perpendicular axis is an axis denoted that is perpendicular to the main axis and forms together with the main axis the plane in which the forearm is located. In case a `perp_axis` is specified, the procedure `upper_arm_angles_abs` returns not only an arm-ls and arm-lf angle, but also an arm-turn angle. Those angles are the absolute angles needed to reach the specified orientation. A procedure `upper_arm_angles_rel` will calculate all differences between the current and the desired joint angles and divide them by the number of timesteps to find the appropriate stepsize for every local motor program called, i.e. the `lift_arm_sideward`, the `lift_arm_forward` and, in case a `perp_axis` is specified, the `turn_arm` procedures.

### 5.1.2 Orienting the Lower Arm

The orientation of the forearm can be changed by the atomic skill

```
direct_forearm main_axis time_opt timesteps end_event
```

which is based on the `forearm_angles_abs` procedure described in section 4.5.2 (page 50). The stepsizes for both local motor programs involved, i.e. `turn_arm` and `bend_hand`, are synchronized in procedure `forearm_angles_rel`.

### 5.1.3 Orienting the Hand Axes

For orienting the hand axes there are three different skills provided. They are:

```
direct_hand_palm/point_axes palm_axis point_axis time_opt timesteps
                             end_event
```

```
direct_hand_palm_towards object time_opt timesteps end_event
```

```
align_hand_thumb_axis_with_object object hand_bend_angle time_opt
                                 timesteps end_event
```

The `direct_hand_palm/point_axes` skill has an option to specify both a target palm axis and a target point axis, each consisting of three dimensional vectors. In case



both axes are given, all three local motor programs for the hand, i.e. `turn_hand`, `bend_hand`, and `twist_hand`, will be activated. In case there is no point axis specified (by setting the `point_axis` parameter to “no”), the current twist angle of the hand will remain the same, since it does not influence the palm axis (see section 4.5.3, page 55).

For orienting the palm axis towards an object the skill `direct_hand_palm_towards` is provided (see appendix, page 113). It is very similar to the previous one, but the palm axis will be computed by subtracting the current end effector position from the centroid of the specified object. No hand twisting will take place. This skill can e.g. be used in order to “look at a picture” if the picture is placed in the hand parallel to the palm, the palm is oriented towards the head and the head/eyes are oriented towards the picture.

As mentioned earlier, it is useful for some grips to have the thumb axis aligned with the longitudinal axis of an object. The skill `align_hand_thumb_axis_with_object` therefore first calls a procedure that returns a vector specifying the longitudinal axis of an object, given the object (see procedure `object_l_axis` included in appendix A, page 127) . The hand-turn and the hand-twist angle are calculated by the procedure `aligned_hand_angles_abs` discussed on page 53. The `hand_bend_angle` is provided as an input parameter to the skill.

## 5.2 Reaching

### 5.2.1 Simple Reaching

Like the *orient-body-part* skills, the simple reaching skills are atomic inverse kinematics skills. The core of the motor programs for simple reaching consists of a call to the `goal_arm_angles_abs` procedure, explained in detail in section 4.5.

By simple reaching it is meant that only positioning of the end effector (or wrist) takes place without orienting the hand in a specific direction, and there is no collision avoidance. The following motor program invokes simple reaching

```
move_arm_to_goal goal_point gp_opt time_opt timesteps end_event
```

(see appendix A, page 126), by changing the `arm-ls`, `arm-lf`, `arm-turn`, and `elbow` joint angles so as to position the end effector at the specified `goal_point`.

The procedure

```
move_arm_to_object object gp_opt time_opt timesteps end_event
```

is capable of adapting to a moving object. If the object is e.g. a ball, the orientation of the hand in respect to the ball does not play a role. Thus, applying a simple reaching procedure is sufficient. The end effector is moved to a fictitious point on the palm vector such that the distance between the new end effector position and the palm equals the radius of the object. This way, the centroid of the object is the point the end effector should reach for in order to achieve proper contact between the palm and the object. Every timestep, the `move_arm_to_object_state1` procedure sets the `goal_point` to the current centroid of the ball that might have changed meanwhile.

The `goal_arm_angles_abs` also recomputes for every timestep the final angles necessary to reach the current `goal_point`. If the object is moving faster than the distance per timestep the end effector is supposed to travel, the end effector would always stay behind the object. To avoid this effect, the global variable `dist_per_timestep` can be set to a greater distance by using the procedure `dps` (included in appendix A, page 115).

As for the *orient-body-part* skills, setting the `time_opt` to “0” will cause the number of timesteps to be calculated by the appropriate *timesteps-calculation-procedure* which is `calc_dist_timesteps` in that case.

### 5.2.2 Reaching for an Object with Alignment of the Hand

Theoretically, reaching for an object while aligning the hand along the objects longitudinal axis could be realized as a composite skill composed of the atomic skills `move_arm_to_object` and `align_hand_thumb_axis_with_object` executing in parallel. When aligning the thumb axis with an object, the absolute hand angles needed to bring the hand in the desired orientation are computed, based on the current coordinate frame of the forearm, located at the wrist. To orient the hand in the same way based on a different coordinate frame of the forearm, namely the one after the goal point is reached would yield different hand angles. Since the hand angles would be changed only by the amount of the current stepsizes and the inverse kinematics algorithms are recomputed every timestep, the composite skill would eventually bring the joints in the right position.

A better way to realize such a reaching skill is to integrate the calculation of the hand angles with the calculation of the arm angles, so that the hand angles calculation can be based on the final arm angles. As mentioned earlier in section 4.5.4, this is done in procedure `reach_angles_abs`.

The atomic skill based on the reach angles

```
reach_object object gp_opt hand_bend_angle time_opt timesteps end_event
```

controls seven joint motions: arm lifting sideward, arm lifting forward, arm turning, elbow bending, hand turning, hand bending and hand twisting, where the hand turning results from the target `hand_bend_angle` that was given as a parameter to the skill.

### 5.2.3 Reaching for an Object along a Path

In order to avoid collision when reaching for an object, a collision free path has to be generated using the procedure `equidistant_path_points` explained in section 4.6. The distance between one point on the path and the following point will always be the same and is determined by the value of the global variable `dist_per_timestep`.

The end effector will be guided through each of the path points except the last one, by linking several atomic `move_arm_to_goal` skills sequentially. One of the equidistant path points will be passed as input for the `goal_point` parameter to each of these skills in order. To force the end effector to reach the goal point in one timestep the `time_opt` has to be set to "1" and the number of `timesteps` to "1". So far, the hand will keep its original joint angles. To start reaching for the object, (involving the alignment of the hand thumb vector with the object), a dummy object is placed at the location of the last path point before the object is reached, and assigned the matrix of the object so that the dummy object ends up in the same orientation as the object to be reached.

Now, the atomic skill `reach_object dummy` will be defined as end event of the last `move_arm_to_goal` skill, and `reach_object object` as end event of `reach_object dummy` in order to execute all the named skills in sequence.

The sequential linking of the atomic skills will be done by the procedure `write_reach_file` that produces a file called `reach_command` at runtime which might look as follows:

```

move_arm_to_goal_new {7.87376884 -10.6772338 11.0305709} ee 1 1 {
  move_arm_to_goal_new {9.19729096 -9.05777862 11.0674169} ee 1 1 {
    move_arm_to_goal_new {10.5208131 -7.43832343 11.104263} ee 1 1 {
      move_arm_to_goal_new {11.8443352 -5.81886824 11.1411091} ee 1 1 {
        move_arm_to_goal_new {13.478763 -5.40810519 11.2134024} ee 1 1 {
          move_arm_to_goal_new {15.4184204 -6.18396817 11.3204958} ee 1 1 {
            gpp 2
            reach_object dummy ee -10 1 {
              gpp 0.4
              reach_object glas ee -10 1 {}
            }
          }
        }
      }
    }
  }
}

```

```
    }  
  }  
}
```

Finally, calling the procedure

```
reach_object_along_path object hand_bend_angle end_event
```

will cause the path points to be generated, write the file `reach_command` (see procedure `write_reach_file`, appendix A, page 133) and source the `reach_command` file so as to execute the composite skill. Figure 5-1 shows the hand while moving the end effector along the posted path towards a glass on the table. Note that the end effector is set to be a fictitious point 1 inch apart from the palm.

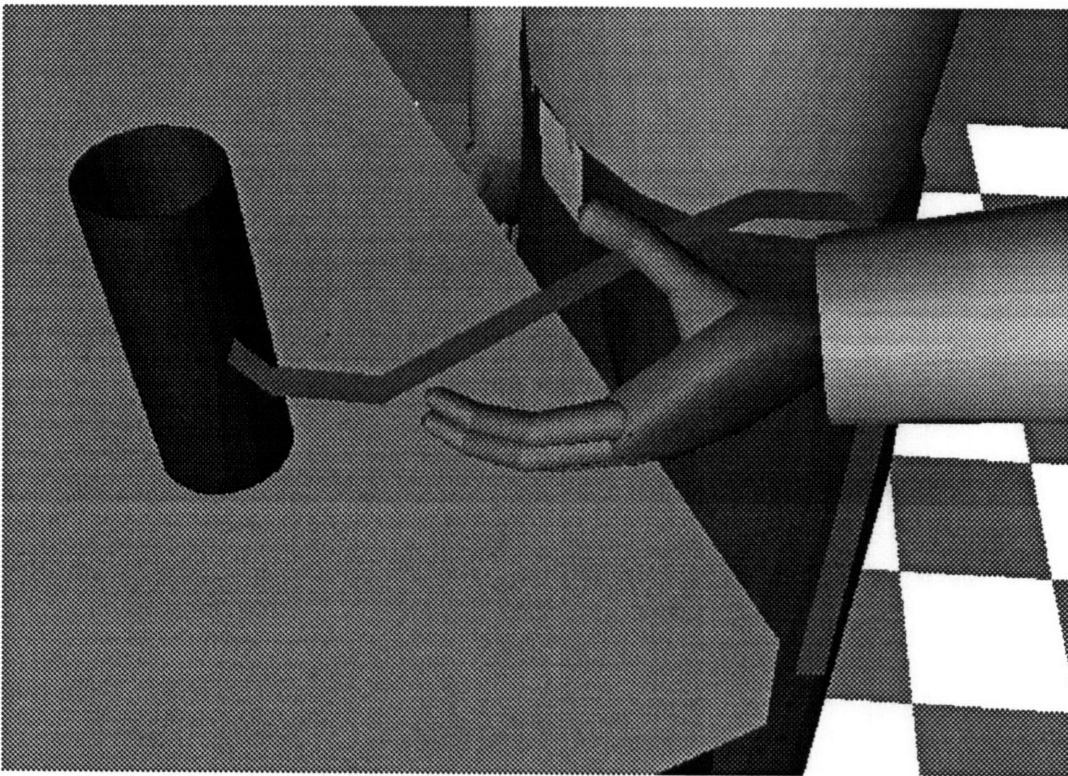


Figure 5-1: Virtual actor performing the `reach_object_along_path` skill

## 5.3 Articulating the Fingers

### 5.3.1 Grasping

Grasping should only be invoked when the object to be grasped is reached. The procedure

```
grasp_object object speed end_event
```

is implemented as a composite skill invoking the atomic skills

```
close_finger finger object bend1_step bend2_step end_event
close_thumb object bend1_step bend2_step end_event
```

for all fingers in parallel (see figure 5-2).

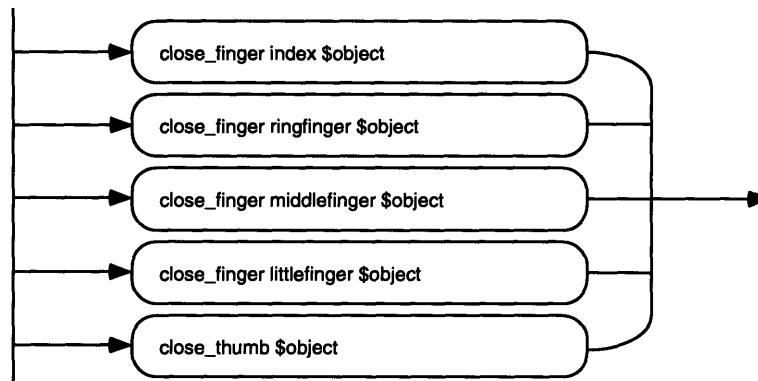


Figure 5-2: Composite skill grasp\_object

```
proc grasp_object { object {speed 1} {end_event {return stop}} } {
  set grasp_object_condition0 [object_reached $object]
  set bend1_thumb_step $speed
  set bend2_thumb_step [mult 2 $speed]
  set bend1_step [mult 3 $bend1_thumb_step]
  set bend2_step [mult 3 $bend2_thumb_step]
  if $grasp_object_condition0 then {
    close_finger index      $object $bend1_step $bend2_step {}
    close_finger middlefinger $object $bend1_step $bend2_step {}
    close_finger ringfinger  $object $bend1_step $bend2_step $end_event
    close_finger littlefinger $object $bend1_step $bend2_step {}
    close_thumb  $object $bend1_thumb_step $bend2_thumb_step {}
  }
}
```

```

    } else {
        echo Sorry, the initial conditions for executing this skill are not given!
    }
    return
}

```

The local motor program involved in the `close_finger` atomic skill is

```

    bend_one_finger_by finger bend_angle1 bend_angle2 timesteps renopt

```

and for the `close_thumb` atomic skill

```

    bend_thumb_joint joint option angle timesteps renopt

```

Figure 5-3 shows the actor's hand after grasping a glass.

### 5.3.2 Making a fist

Making a fist can be realized by defining the palm itself as the object “to be grasped” and invoking the same atomic skills as for the `grasp_object` skill. The procedure for that skill is

```

    make_fist speed end_event

```

### 5.3.3 Changing between Different Hand Postures

A hand posture is determined by a specific configuration of all finger and thumb joint angles. There are several composite skills

```

    bring_all_fingers_in_***_config timesteps end_event

```

where `***` can stand for “rest”, “point”, or “open\_grasp” that will change the hand-posture in a specified number of `timesteps` into the corresponding configuration. The point position is shown in figure 5-4.

The skills are composed of the atomic skills

```

    bring_finger_in_config finger configuration timesteps end_event
    bring_thumb_in_config configuration timesteps end_event

```

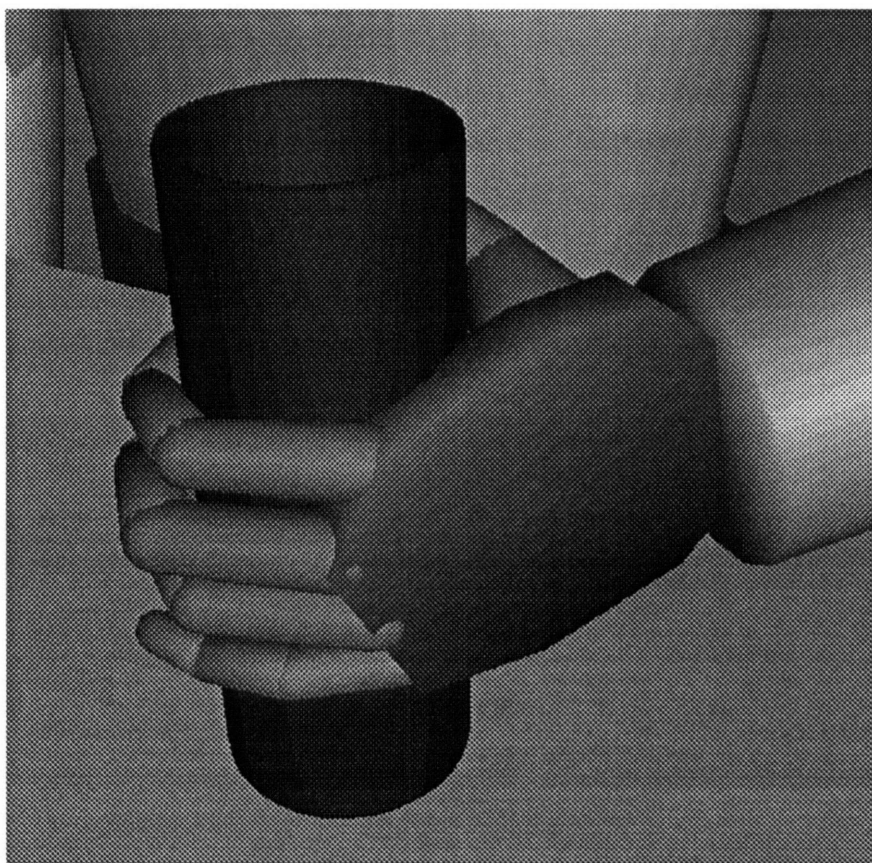


Figure 5-3: Hand of the virtual actor grasping a glass

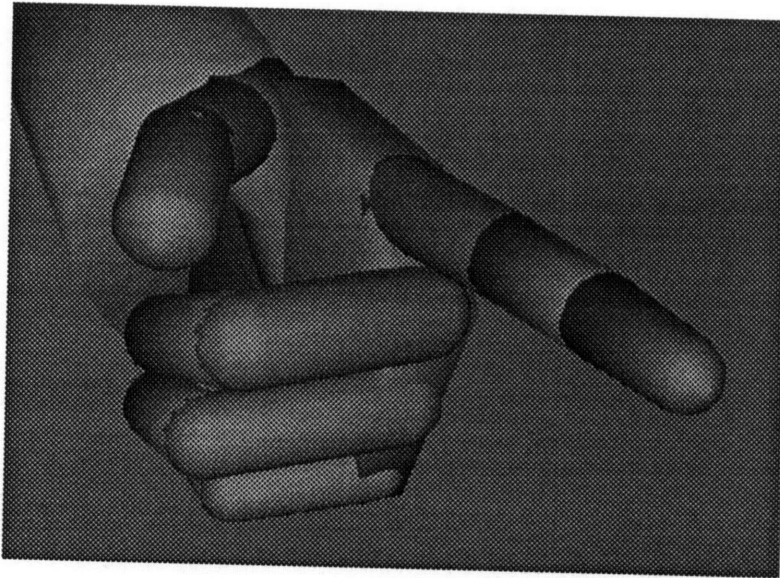


Figure 5-4: Hand of the virtual actor in point position

that are to be executed in parallel as in the following composite skill:

```

proc bring_all_fingers_in_point_config {{timesteps 4} {end_event {}}} {
  set bring_all_fingers_in_point_config_condition0 [hand_free]
  if $bring_all_fingers_in_point_config_condition0 then {
    bring_finger_in_config index      {0 2 4} $timesteps {}
    bring_finger_in_config middlefinger {10 80 90} $timesteps $end_event
    bring_finger_in_config ringfinger  {-5 90 90} $timesteps {}
    bring_finger_in_config littlefinger {0 96 90} $timesteps {}
    bring_thumb_in_config      {-20 40 90 20 7} $timesteps {}
  } else {
    echo Sorry, the initial conditions for executing this skill \
      are not given!
  }
  return
}

```

Since the number of timesteps for all atomic skills will be the same, the `end_event` only has to be evaluated by one of them.

The atomic skills `bring_finger_in_config` invoke the following local motor programs:

```

bend_one_finger_by finger bend_angle1 bend_angle2 timesteps renopt
spread_finger finger option spread_angle timesteps renopt

```



whereas the atomic skill `bring_thumb_in_config` invokes the local motor programs

```
spread_finger thumb option spread_angle timesteps renopt
move_thumb_down option angle timesteps renopt
turn_thumb option angle timesteps renopt
bend_thumb_joint 1 option angle timesteps renopt
bend_thumb_joint 2 option angle timesteps renopt
```

## 5.4 Integrated Reaching and Grasping

Grasping can be easily integrated in the composite skill `reach_object_along_path` (see section 5.2.3) by adding the atomic skill `bring_all_fingers_in_open_grasp_config`. The skill to open the fingers should start executing in parallel with one of the `move_arm_to_goal` skills depending on how many timesteps the finger opening should take. Defining the atomic skill `grasp_object` as end event will make the fingers grasp the object after it has been approached.

## 5.5 Putting the Arm on the Table

The following skill is presented as an example of an atomic forward kinematics skill modeled by a complex finite state machine, according to figure 5-5. If the wrist

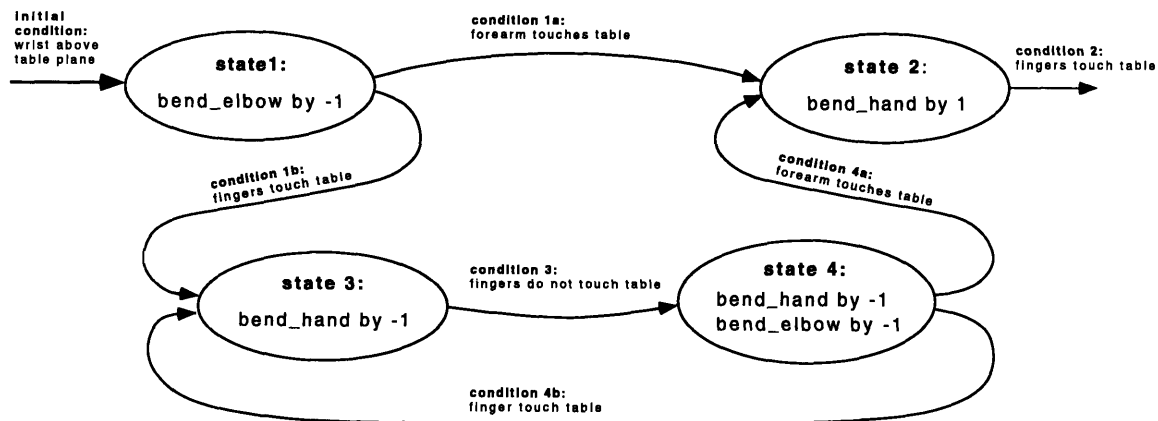


Figure 5-5: Finite state machine for the atomic skill `put_arm_on_table`

is located above the table plane (which is the initial condition for that skill), then

extending the elbow - as will be done in state 1 of the finite state machine - will make the hand approach the table plane. Depending on whether the forearm or one of the fingers will first touch the table, it will be branched to either state 2 or state 3 of the finite state machine. State 2 makes the hand bend until the fingers also touch the table (this is the ending condition of the skill). In case the fingers touched the table first, it will be branched to state 3, the hand will be bent outwards so as to take the fingers away from the table until they no longer have contact. The elbow will then extend further while the hand will still be bent outwards (state 4). If now the fingers touch the table first, states 3 and 4 will be repeated until the condition for branching to the final state 2 is fulfilled, namely that the forearm touches the table.

The `put_arm_on_table` skill does not include any collision avoidance. There is no visual controller that would guarantee that the elbow does not run into any other body link or object in the virtual environment while performing the skill. Also, the actor would not “realize” if there is an object placed on the table right where the hand approaches the table plane.

All procedures defining the motor programs of the `put_arm_on_table` skill are included in appendix A, page 127.

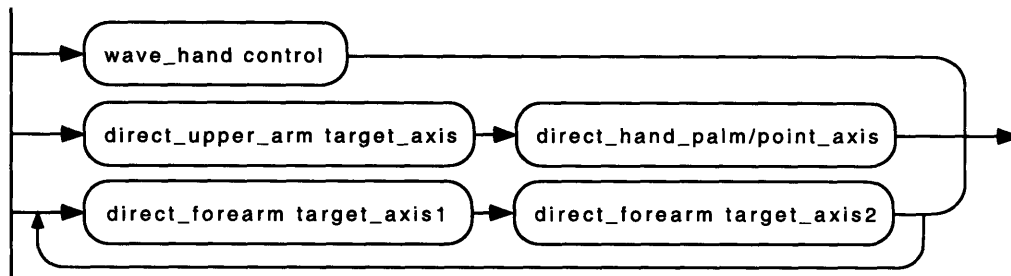
## 5.6 Waving the Hand

Another example of a composite skill is the

```
wave_hand end_event
```

skill, composed mainly of *orient-body-part* skills (see figure 5-6). In this skill, the arm first aims specific orientations for the upper arm and the forearm. Once the target orientation for the upper arm is reached, the hand is oriented so that the palm axis points forwards (viewed from the body). The target for the forearm orientation alters between two orientations while the hand angles are constantly adjusted so as to keep the specified palm orientation.

Hand waving is an oscillating skill. All oscillating skills need to add a control procedure to the command list (i.e. `wave_hand_control` in this case) that constantly checks, whether the skill should keep executing. The hand waving skill can be stopped by typing `stop_waving` (in the **Do Command once** textfield). This will set the global variable `stop_waving` to “1” so that the `wave_hand_control` procedure will remove all atomic skills invoked by `wave_hand` from the command list, including itself. Furthermore, it will evaluate the end event of `wave_hand`.

Figure 5-6: Composite skill `wave_hand`

This way it is possible, to bring the arm back in its original position by defining the skill

```
move_arm_to_config config time_opt timesteps end_event
```

as end event of `wave_hand`. The parameter `config` could be set to “`$org`” which is the global variable in which the original arm configuration is stored. The call would look as follows:

```

wave_hand {
    global org
    move_arm_to_config $org 0 0 {}
}

```

## 5.7 Head/ Eye Controller

A controller for the head and eye movements has been developed by Sunil Singh and his student Dan Popa at Dartmouth College and has been integrated in the SkillBuilder [9].

It is the task of the head-eye controller to track a target object moving in a 3-dimensional space. The controller has been implemented as a manipulator written in C, communicating with the tcl procedures of the SkillBuilder through a series of `ppread` and `ppwrite` commands.

The head, represented by a model of a skull, is placed on the neck at joint 9 of the `dhchain` of the spine and is capable of limited motion about three intersecting coordinate axes, like a spherical wrist in robot manipulators. The eyes are represented by two white balls. Two smaller blue balls inside the white balls represent the pupils.

Singh and Popa make use of the fact that a target can be characterized by its coordinates with respect to a fixed frame and by an angle  $\alpha$ , which represents the angular orientation of the object with respect to the rotated axis normal to the skull. They determine the rotation matrix of the skull relative to the fixed frame by using the Euler formalism, and solve the formulated inverse kinematics equations for the rotation angles around the fixed frame.

Since it is not intended to move the head towards a target at all times, but rather move only the eyes or both, they make the following considerations:

- The fovea of both eyes can be independently directed to the object of interest (*vergence eye movements*).
- If the object is “almost” in front of the eyes, the eyes move only in a quick, gaze-shifting response, called a *saccade*.
- Both the skull and the eyes have a relative rotation range which cannot be exceeded.
- If the object to be tracked is moving outside of the range for “comfortable” eye movements, the head-eye motions must be carefully distributed between the head and the eyes.

While the head control unit uses purely inverse kinematics, the eye control unit explicitly accounts for the dynamics of the object and the human system.

The SkillBuilder provides the following atomic skills to provoke head/eye motions:

```
head_track object end_event
head_straight time_opt timesteps end_event
head_move object time_opt timesteps end_event
```

Each of those skills will put the appropriate procedures on the command list and will be executed only if the command loop is running. The `head_move` skill will cause the head and eyes to move the focus towards the specified `object` in the given number of `timesteps` and evaluate its `end_event` upon completion of the skill. The skill will be able to adapt to a moving object.

Keeping the focus of the head and eyes on a moving object can be done by invoking the `head_track` skill. It is useful to define `head_track` as end event of `head_move`, in order to slowly move the focus towards an object before tracking it. `head_track`

is a continuous skill which means it will only stop executing when it is explicitly stopped by calling the procedure `stop_head_track`. Remember that while the command loop is running, interactions are only possible through the menus. This means `stop_head_track` has to be typed in the **Do Command Once** text field.

The skill `head_straight` is nothing else but a call to the `head_move` skill with a dummy object placed at the point the head should look at.

Figure 5-7 shows the virtual actor with the skull looking at a ball in its hand.

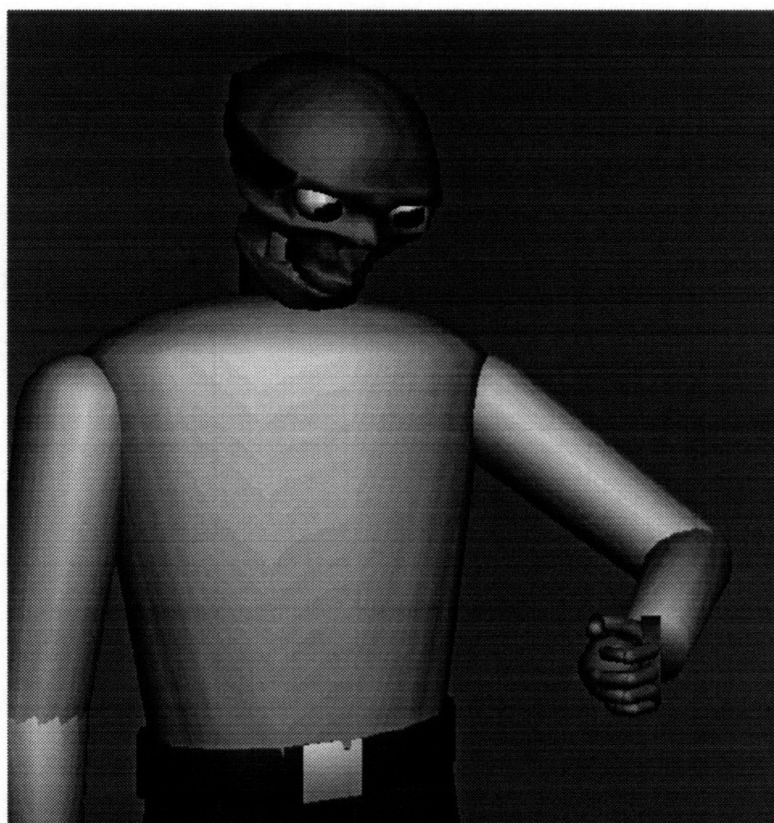


Figure 5-7: Virtual actor looking at a ball in its hand

# Chapter 6

## Future Work

The presented state of the SkillBuilder is prototypical. In this chapter there will be suggestions on how to extend the functionality of the SkillBuilder, improve the performance and make it more user-friendly.

Extensions of the functionality of the SkillBuilder could be made by:

- **Articulating the rest of the limbs**

So far, the virtual actor has kinematic chains defined for the spine, the left arm, all fingers and the thumb of the left hand, as well as for the left leg and its foot. For skills involving both arms and/or legs, however, the model needs to be extended to include kinematic chains for the right arm, all fingers and the thumb of the right hand, as well as for the right leg and foot. The process of finding the right Denavit-Hartenberg parameters might be a bit tricky, given that the notation does not allow “mirroring”. But looking at the parameters of the dhchain for the left limbs of the actor already provided, and using the limb edit menu will be very helpful.

- **Creating local motor programs for all articulated joints**

Local motor programs exist only for the left arm and all fingers and the thumb of the left hand. Creating local motor programs for all other joints will be trivial given the templates from the implemented ones (see section 4.4.2).

- **Extending the list of condition procedures**

The classification of condition procedures given in section 4.8.1 suggests several general conditions that have not yet been implemented. As the complexity and variety of skills increases more condition procedures appropriate for those skills will have to be generated.

- **Developing a range of representative motor skills**

The range of representative motor skills should be extended to match the task primitives mentioned in section 3.3.

- **Creating of a more complex virtual world for the actor**

The more complex the skills get the virtual actor should perform the more there will be a need for different kinds of objects in the environment, e.g. a chair he could sit on, a door he could open, different objects on the table he could pick up and lift, and so on.

Improvements to the performance of the motor skills are possible by:

- **Integrating a more general collision avoidance and collision detection feature**

There is much current research in finding the best collision free path through a cluttered environment. Especially in robotics, many algorithms for a manipulator with several links have been suggested. As the virtual environment for the actor gets more and more complex, it will be important to integrate a more general path finding algorithm and possibly extend it with a collision detection feature. It will be necessary not only to guarantee a collision free path for the end effector, i.e. the palm, but also all links of a moving limb.

- **Including dynamics**

As the fidelity requirements of the applications increase it will be necessary to include dynamics in the simulation to make it look more realistic.

- **Confirming the performance with experimental data**

The movements of the actor performing a skill could be compared with experimental data from human subjects performing the same tasks.

- **Speeding up**

By distributing computational processes over a network of machines the performance could be made faster and made look more realistic.

Improvements of the user friendliness of the SkillBuilder could be achieved by:

- **Developing an automated *motor program builder***

It would be nice if the skill designer would not have to write tcl source code. Because the general structure of motor programs has been defined in this thesis, it will easily be possible to generate a motor program builder for atomic as well as composite skills. A motor program builder would prompt the user for

all necessary information to create the source code, i.e. the motor programs, automatically.

- **Providing a graphical user interface**

This is certainly one of the most important issues to make the SkillBuilder easier to handle. Once the generation of motor programs has been automated, a visual programming language for defining motor skills should be implemented as described in section 3.4.3 (the ultimate SkillBuilder interface). Furthermore, all available skills should be able to invoked by choosing an item from a graphical list, taking its arguments from another graphical menu. The values of all global parameters should be displayed and able to be changed by graphical means.



# Chapter 7

## Conclusion

This thesis describes the development of the SkillBuilder, a software system that can function as a design tool for motor programs - the source code of skills a virtual actor can perform. The implementation was done in the *3d* virtual environment simulation system explained in the thesis.

A human figure model was generated consisting of rigid body parts that are linked together as several kinematic chains using the Denavit-Hartenberg notation. Every joint of the figure can be articulated by a corresponding local motor program.

A classification of skills into atomic and composite skills has been made where the atomic skills were further classified into forward and inverse kinematics skills. Atomic skills are modeled by finite state machines that define different states to be run through during the execution of a skill, as well as their transition conditions. The composition of more complex skills into composite skills has been demonstrated by linking atomic skills in a parallel or sequential manner, or in a combination of both.

It has been shown how to generalize the generation of motor programs. Every motor program consists of a compound of procedures, each defining one state of a finite state machine. A state mainly consists of the identification of local motor programs (i.e. the action to be taken during the execution of that particular state) and a definition of the ending conditions (i.e. conditions that have to be satisfied in order to proceed to the next state).

Ending conditions have been classified and a selection of condition procedures have been implemented. In many cases, a condition procedure is very general and is made suitable for a specific skill state by passing the appropriate arguments to the procedure. However, sometimes condition procedures have to be generated that are very specific for one skill. Those kind of procedures will have to be developed on an

ad hoc basis when defining new skills.

The problem of parallel execution of several motor programs has been addressed by introducing the command loop: an endless loop that constantly evaluates all procedures currently on the command list. Thus, every motor program currently executing has an entry of one of its states in the command list.

The focus of the work was on visually guided reaching and grasping skills using the left arm.

Reaching for an object involves inverse kinematics calculations. The algorithm for positioning the palm at a specified point in space takes four joint angles into account: the two angles of the spherical joint and the shoulder, the elbow angle and the arm-turn angle. The first two of these angles determine the upper arm direction. The arm-turn angle is the angle the forearm is turned about the upper arm axis. The additional degree of freedom is used to influence the relative elbow position which determines the rotation of the arm about the vector from the shoulder to the goal point. It is guaranteed that the elbow always stays outside of the plane defined by the upper arm and the forearm in the lowest possible elbow position, as viewed from the center of the body. The three angles of the hand are calculated from a target orientation the hand should finally reach. For the course of this thesis only a power grip has been considered for which the thumb axis (i.e. an axis in the palm plane pointing towards the thumb) needs to be aligned with the longitudinal axis of the object to be grasped.

Adaptation to a moving object or a changing shoulder position (when lifting the shoulder, when leaning forwards, or even when moving the whole body while reaching) is achieved by recomputing the inverse kinematics calculations every timestep.

The global variable `arm_pose_parameter` contains a measurement for the relative elbow position that can be changed at any time. Several other global parameters like the `dist_per_timestep`, `angle_per_timestep`, `goal_point_precision` and `angle_precision` have also been defined and can be changed any time during the execution of the skills in order to alter the performance.

A collision avoidance algorithm specific to the left arm has been implemented that guarantees that the hand will move from its current position to the object without colliding with a specified collision object, i.e. the table. However, the algorithm does not yet guarantee collision free moves of the other arm links, i.e. the upper arm and the forearm. In order to avoid collisions of these arm links the relative elbow position can be influenced appropriately by changing the `arm_pose_parameter`.

The reaching and grasping skills have been successfully demonstrated as real-time response to a direct interaction between the virtual environment participant and the

virtual actor. A glass positioned on a table located right in front of the virtual actor can be “picked up” and positioned anywhere else using a dataglove that has been hooked up to the SkillBuilder. When “asking” the actor to grasp the glass, an appropriate collision-free path is generated and, if desired, shown on the screen. The actor starts forming an open grasp position with the fingers while reaching for the glass along the collision free path. Once the glass has been reached the actor wraps the fingers around the glass until a contact has been detected between each of the fingers and the glass. When “picking up” the glass again, the actor may be “asked” to hang on to it. The actor always adjusts the angles of the hand so that it keeps the same orientation with respect to the glass. This way it is also possible to demonstrate a drinking actor (see figure 7-1).

The focus was deliberately put on visually guided reaching and grasping skills, rather than on the attempt to implement a range of behaviors. The exercise has led to a better understanding of the issues involved in designing the needed software tools that will serve to extend the SkillBuilder with a range of representative motor skills.

Validating the implemented motor programs with human clinical/experimental data is left for future work.

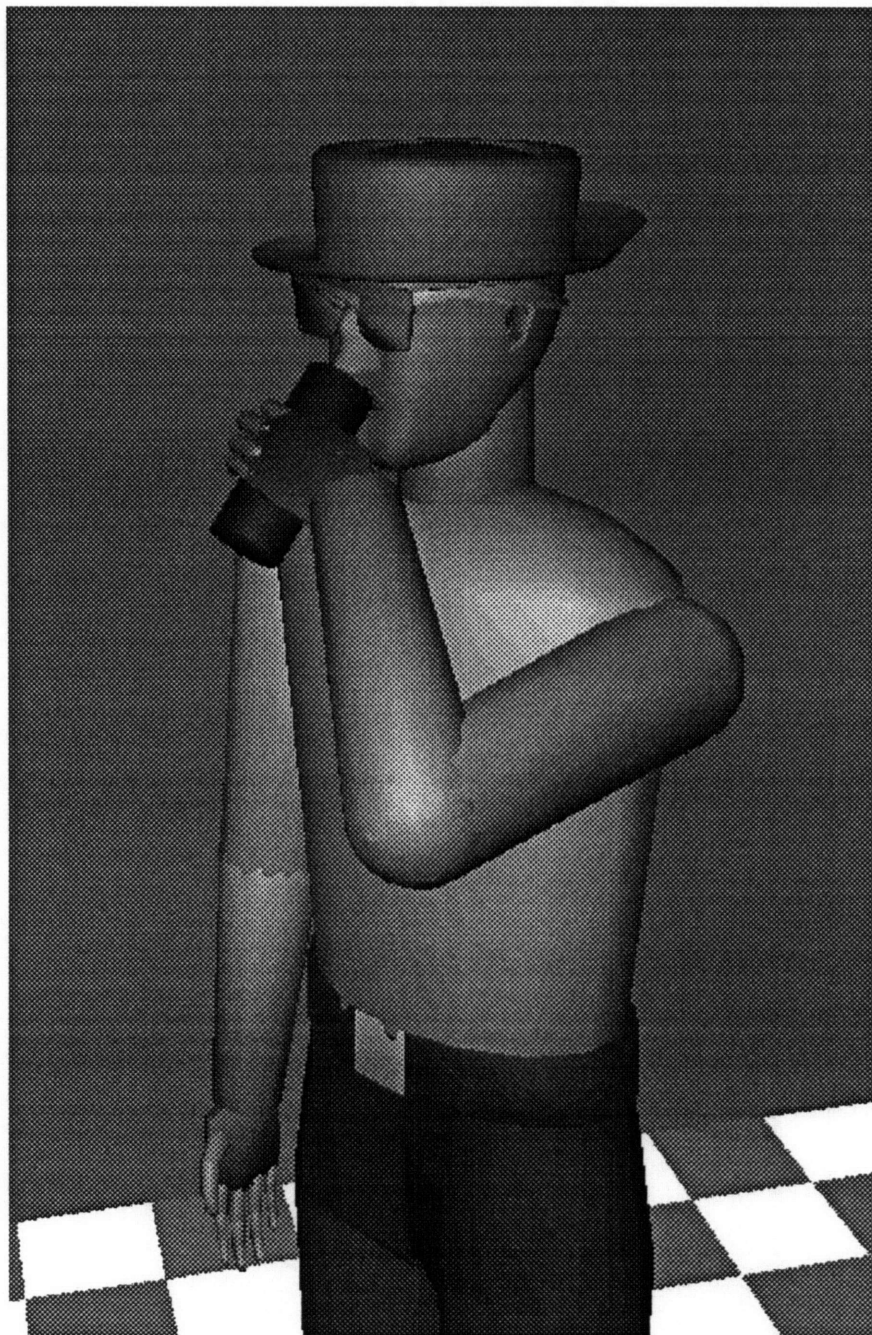


Figure 7-1: Drinking actor

# References

- [1] "Research Directions in Virtual Environments," Report of an NSF Invitational Workshop, *Computer Graphics*, Volume 26, Number 3, August 1992, pp.153-177.
- [2] Calvert T., "Composition of Realistic Animation Sequences for Multiple Human Figures," in *Making Them Move: Mechanics, Control and Animation of Articulated Figures*, N. Badler, B. Barsky, and D. Zeltzer, 1991, Morgan Kaufmann: San Mateo CA, pp. 35-50.
- [3] Badler N.I., "Human Modeling in Visualization," in *New Trends in Animation and Visualization*, N. Magnatnat Thalmann, D. Thalmann, 1991, John Wiley & Sons: Chichester, England, pp. 209-228.
- [4] Bizzi E., "Central and Peripheral Mechanisms in Motor Control," in *Tutorials in Motor Behavior*, G.E. Stelmach and J.Requin, 1980, North-Holland Publishing Company, pp. 131-143.
- [5] Brooks, R.A., "Planning Collision Free Motions for Pick-andPlace Operations," in *The International Journal of Robotics Research*, Vol.2, No.4, Winter 1983.
- [6] Brooks, R.A., "A Robot that Walks: Emergent Behavior from a Carefully Evolved Network", in *Making Them Move: Mechanics, Control and Animation of Articulated Figures*, N. Badler, B. Barsky, and D. Zeltzer, 1991, Morgan Kaufmann: San Mateo CA, pp. 99-108.
- [7] Chen, D. and D. Zeltzer, "The 3d Virtual Environment and Dynamic Simulation System," *Computer Graphics and Animation Group Technical Report*, Massachusetts Institute of Technology, Media Lab, August 1992.
- [8] Chen, D., "Pump It Up: Computer Animation of a Biomechanically Based Model of Muscle using the Finite Element Method," *Ph.D. Thesis*, Massachusetts Institute of Technology, 1991.

- [9] Chen, D., S. Pieper, S. Singh, J. Rosen, and D. Zeltzer, "The Virtual Sailor: An Implementation of Interactive Human Body Modeling," *Proc. 1993 Virtual Reality Annual International Symposium*, Seattle, WA, September, 1993.
- [10] Croll, B.M., "Rendermatic: An Implementation of the Rendering Pipeline." *Master's Thesis*, Massachusetts Institute of Technology, 1986.
- [11] Crow, F.C., "A More Flexible Image Generation Environment," in *Proc. ACM Siggraph 82*, Boston, July 1982, pp. 9-18 .
- [12] Girard, M., "Interactive design of 3D Computer-Animated Legged Animal Motion," *IEEE Computer Graphics and Applications*, June 1987, 7(6), pp. 39-51.
- [13] Johnson, M., "Build-a-Dude: Action Selection Networks for Computational Autonomous Agents", *M.S. Thesis*, Massachusetts Institute of Technology, February 1991.
- [14] Lozano-Perez, T., "Spatial Planning: a Configuration Approach," in *IEEE Transactions on Computers*, Vol. C-32, No.2, Februar 1982.
- [15] Magnenat-Thalmann, N. and D. Thalmann, "Synthetic Actors in Computer-Generated 3D Films", Springer-Verlag, Berlin, 1990.
- [16] Maiocchi, R. and B. Pernici, "Directing an Animated Scene with Autonomous Actors," in *Computer Animation '90*, N. Magnenat-Thalmann and D. Thalmann, Springer-Verlag, Tokyo, 1990, pp. 41-60.
- [17] Ousterhout, J.K., "Tcl and the Tk Toolkit" Addison-Wesley, in press.
- [18] Ousterhout, J.K., "Tcl: An embeddable command language," *1990 Winter USENIX Conference Proceedings*, 1990.
- [19] Paul, R., "Robot Manipulators: Mathematics, Programming, and Control," MIT Press, 1981.
- [20] Phillips, C.B. and N.I. Badler, "Interactive Behaviors for Bipedal Articulated Figures," in *Computer Graphics*, Volume 25, Number 4, July 1991, pp. 359-362.
- [21] Raibert, M., "Legged Robots that Balance", Cambridge MA: MIT Press, 1986.
- [22] Rijpkema, H. and M. Girard, "Computer Animation of Knowledge-Based Human Grasping" in *Computer Graphics*, Volume 25, Number 4, July 1991, pp. 339-348.

- [23] Schaffer, L.H., "Analysing Piano Performance: A Study of Concert Pianists," in *Tutorials in Motor Behavior*, G.E. Stelmach and J.Requin, 1980, North-Holland Publishing Company, pp. 443-455.
- [24] Schank, R.C., "Conceptual Information Processing," American Elsevier Publishing Company, New York, 1975.
- [25] Schmidt, R.A., "More on Motor Programs," in *Human Motor Behavior*, J.A. Kelso, Lawrence Erlbaum Associates Publisher, London, 1982.
- [26] Schwartz, M.F., E.S. Reed, M. Montgomery, C. Palmer, and N.H. Mayer, "The Quantitative Description of Action Disorganization after Brain Damage: A Case Study", *Cognitive Neuropsychology*, 1991, 8(5), pp. 381-414.
- [27] Young, D.A., "The X Window System Programming and Applications with Xt: OSF/Motiv Edition", Prentice-Hall, Inc., 1990.
- [28] Wilhelms, J. "Using Dynamic Analysis for Realistic Animation of Articulated Bodies", *IEEE Computer Graphics and Applications*, June 1987, 7(6), pp. 12-27.
- [29] Zeltzer, D., "Motor Control Techniques for Figure Animation," *IEEE Computer Graphics and Applications*, November 1982, 2(9), pp. 53-59.
- [30] Zeltzer, D., "Knowledge-based Animation" in *Proc. ACM SIGGRAPH/SIGART Workshop on Motion*, April 1983, pp. 187-192.
- [31] Zeltzer, D., "Task Level Graphical Simulation: Abstraction, Representation and Control," in *Making Them Move: Mechanics, Control and Animation of Articulated Figures*, N. Badler, B. Barsky, and D. Zeltzer, 1991, Morgan Kaufmann: San Mateo CA, pp. 3-33.
- [32] Zeltzer, D. and M. Johnson, "Motor Planning: An Architecture for Specifying and Controlling the Behavior of Virtual Actors," *Journal of visualization and Computer Animation*, April-June 1991, 2(2), pp. 74-80.
- [33] Zeltzer, D., "Virtual Actors and Virtual Environments: Defining, Modeling and Reasoning about Motor Skills," *Proc. Interacting with Images*, British Computer Society, London, February 16-18, 1993.
- [34] Zeltzer, D. and M. Johnson, "Virtual Actors and Virtual Environments: Defining, Modeling and Reasoning about Motor Skills," in *Interacting with Virtual Environments*, L. MacDonald and J. Vince, 1993, John Wiley & Sons: Chichester, England, in press.

- [35] Zeltzer, D., "The Virtual Sailor II: Realtime Task Level Interaction with Synthetic Humans," *A Proposal to the Office of Naval Research*, Massachusetts Institute of Technology, October 1993.



# Appendix A

## General Structure of Inverse Kinematics Skills

```
proc IK_skill_name { args {time_opt 0} {timesteps 1} {end_event {}} } {  
  #  
  #   Initial condition  
  #  
  set condition0 [list [condition_procedureA args] & \  
                   [condition_procedureB args] & \  
                   [condition_procedureN args]]  
  if $condition0 then {  
    global IK_skill_name_end_event  
    set IK_skill_name_end_event $end_event  
    global ic  
    set $end_time [plus $ic $timesteps]  
    add_c [list IK_skill_name_state1 args $time_opt $end_time]  
  } else {  
    echo Sorry, the initial conditions for executing this skill are not given!  
  }  
  return  
}
```

```

proc IK_skill_name_state1 {args time_opt end_time } {
  #
  # State ending condition
  #
  set condition1 [list [condition_procedureA args] & \
                    [condition_procedureB args] & \
                    [condition_procedureN args]]
  if $condition1 then {
    global IK_skill_name_end_event
    eval [set IK_skill_name_end_event]
    return stop
  }
  if { $time_opt == 0 } then {
    set timesteps [timesteps_calculation_procedure args]
  } else {
    global ic
    set timesteps [minus $end_time $ic]
    if [lesseq $timesteps 0] then {
      set timesteps 1
    }
  }
  set joint_angles [inverse_kinematics_calculation_procedure args $timesteps ]
  set joint_angle0 [lindex $joint_angles 0]
  ...
  set joint_angleN [lindex $joint_angles N]
  if [greater [abs $joint_angle0] 0.001] then {
    local_motor_program0 by angle 1 noren
  }
  ...
  if [greater [abs $joint_angleN] 0.001] then {
    local_motor_programN by angle 1 noren
  }
  return
}

```

# Appendix B

## Source Code Examples

```
*****
#           add_c
*****

proc add_c {comm} {
    global commandlist;
    lappend commandlist $comm
    return
}

```

```
*****
#           align_hand_angle_abs
*****

proc aligned_hand_angles_abs { thumb_axis hand_bend_angle {M10 dh10} } {

    set thumb_axis [div $thumb_axis [enorm $thumb_axis]]

    if {[string compare $M10 "dh10"] == 0} then {
        set M10 [dhmatrix 10 /1_arm/dhc]
    }
    Mset $M10
    Minvert
    Mele 3 0 0
    Mele 3 1 0
    Mele 3 2 0
    set T [Mxform [mult -1 $thumb_axis]]
    set T0 [lindex $T 0]
    set T1 [lindex $T 1]
    set T2 [lindex $T 2]

    ***** theta13 is given in form of hand_bend_angle *****

    set theta13 $hand_bend_angle
    set s13 [sin $theta13]
    set c13 [cos $theta13]

    ***** Calculation of theta14 *****

    set s14 [div [mult -1 $T2] $c13]
}

```

```

# asin yields in -90 <= theta14 <= 90
# The other solution doesn't have to be evaluated, since the
# allowed range for theta13 is -40 to 40 degrees.
set theta14 [asin $s14]
set c14 [cos $theta14]

#####      Calculation of theta11      #####

set s11 [expr {($T1*$c14 + $T0*$T2*$s13/$c13) / ($T0*$T0 + $T1*$T1)}]
# asin yields in -90 <= theta11 <= 90
set theta11a [asin $s11]
if {$theta11a <= -80} then {set theta11a [plus 360 $theta11a]}
# 90 <= theta11 <= 270
set theta11b [minus 180 $theta11a]

#
# Find out, which one of the two solutions for theta11 is the right one.
#
# case 1 (theta11) :
set M14 [calc_M14_from_M10_11_13_14 $M10 $theta11a $theta13 $theta14]
set case1_thumb_axis [mult -1 [list [lindex $M14 4] \
                                   [lindex $M14 5] [lindex $M14 6]]]
set angle_diff1 [abs [vangle2 $case1_thumb_axis $thumb_axis]]
if [less $angle_diff1 0.1] then {
  set theta11 $theta11a
} else {
  # case 2 (theta11) :
  set M14 [calc_M14_from_M10_11_13_14 $M10 $theta11b $theta13 $theta14]
  set case2_thumb_axis [mult -1 [list [lindex $M14 4] \
                                       [lindex $M14 5] [lindex $M14 6]]]
  set angle_diff2 [abs [vangle2 $case2_thumb_axis $thumb_axis]]
  if [less $angle_diff2 0.1] then {
    set theta11 $theta11b
  } else {
    #
    # case 3 (theta11) :
    # Take the best possible solution.
    #
    if {[less $angle_diff1 $angle_diff2]} then {
      set theta11 $theta11a
    } else {
      set theta11 $theta11b
    }
  }
}
}
set tla_angle [minus $theta11 90]
set bh_angle $theta13
set th_angle [mult -1 $theta14]

return [list $tla_angle $bh_angle $th_angle]
}

#####
#      align_hand_palm/point_axis
#####

proc align_hand_palm/point_axis {palm_axis {point_axis no} \
                                {time_opt 0} {timesteps 4} {end_event {}}} {
#
# Check whether the initial conditions are fulfilled

```

```

# and put the command on the list
#
set condition0a 1

if { $condition0a} then {
  global align_hand_palm/point_axis_end_event
  set align_hand_palm/point_axis_end_event $end_event

  global ic
  set end_time [plus $ic $timesteps]

  add_c [list align_hand_palm/point_axis_state1 $palm_axis $point_axis $time_opt $end_time]
} else {
  echo Sorry, the initial conditions for executing this skill are not given!
}
Com_Update_cb
return
}

#####
*          align_hand_palm/point_axis_state1
#####

proc align_hand_palm/point_axis_state1 {palm_axis point_axis time_opt end_time} {
  #
  # State ending condition
  #
  if {[llength $point_axis] == 3} then {
    set align_hand_palm/point_axis_condition1 \
      [list [test_hand_direction point $point_axis] \
            & [test_hand_direction palm $palm_axis]]
  } else {
    set align_hand_palm/point_axis_condition1 [test_hand_direction palm $palm_axis]
  }
  if ${align_hand_palm/point_axis_condition1} then {
    global align_hand_palm/point_axis_end_event
    eval ${align_hand_palm/point_axis_end_event}
    return stop
  }
  if {$time_opt == 0} then {
    if {[llength $point_axis] == 3} then {
      set ts1 [calc_angle_timesteps hand point $point_axis]
      set ts2 [calc_angle_timesteps hand palm $palm_axis]
      set timesteps [max $ts1 $ts2]
    } else {
      set timesteps [calc_angle_timesteps hand palm $palm_axis]
    }
  } else {
    global ic
    set timesteps [minus $end_time $ic]
    if {[lesseq $timesteps 0]} then {
      set timesteps 1
    }
  }
  set hand_angles [hand_angles_rel $palm_axis $point_axis $timesteps]
  if [equal $hand_angles 999] then {
    echo Sorry, cannot position the hand_palm axis in desired direction.
    return 999
  }
  set tla_angle [lindex $hand_angles 0]

```

```

set bh_angle [lindex $hand_angles 1]
set th_angle [lindex $hand_angles 2]

if [greater [abs $tla_angle] 0.0001] then {
    turn_lower_arm by $tla_angle 1 noren
}
if [greater [abs $bh_angle] 0.0001] then {
    bend_hand by $bh_angle 1 noren
}
if [greater [abs $th_angle] 0.0001] then {
    twist_hand by $th_angle 1 noren
}
return
}

#####
#          aps
#####

proc aps {{angle {}}} {

    global angle_per_timestep

    if {$angle == ""} then {
        echo angle_per_timestep: $angle_per_timestep
    } else {
        set angle_per_timestep $angle
    }
    return $angle_per_timestep
}

#####
#          app
#####

proc app {{para {}}} {

    global arm_pose_parameter
    global arm_pose_opt

    if {$para == ""} then {
        if {$arm_pose_opt == 0} then {
            global arm_pose_default
            echo arm_pose_parameter: $arm_pose_default (default value)
        } else {
            echo arm_pose_parameter: $arm_pose_parameter
        }
    } else {
        if {$para == "default"} then {
            global arm_pose_default
            set arm_pose_opt 0
            set arm_pose_parameter $arm_pose_default
            echo arm_pose_parameter: $arm_pose_parameter
        } else {
            set arm_pose_opt 1
            if {$para < 0} then {
                echo Attention: trying to set an arm_pose_parameter < 0
                set arm_pose_parameter 0
            } else {

```

```

        if {$para > 1} then {
            echo Attention: trying to set an arm_pose_parameter > 1
            set arm_pose_parameter 1
        } else {
            set arm_pose_parameter $para
        }
    }
}
return $arm_pose_parameter
}

#####
#           bend_a_finger_joint
#####

proc bend_a_finger_joint { {finger index} {joint 1} {option by} {angle 10} \
                          {timesteps 2} {renopt ren} } {
    if {$joint > 2} then {
        echo Please specify either joint 1 or 2.
        return
    }

    if {$finger == "thumb"} then {
        bend_thumb_joint $joint $option $angle $timesteps $renopt
    } else {
        set limb    /l_arm/$finger
        set joint_nr [plus $joint 1]
        move_a_joint $limb $joint_nr $option $angle $timesteps range_angle18 noren
        if {$joint == 2} then {
            incr joint_nr
            set angle [mult 0.6667 $angle]
            move_a_joint $limb $joint_nr $option $angle $timesteps range_angle18 noren
        }
        if {$renopt == "ren"} ren
    }
}

#####
#           bend_elbow
#####

proc bend_elbow { {option to} {angle 10} {timesteps 2} {renopt ren} } {

    set limb    /l_arm
    set joint_nr 10
    set min_angle [get_min_angle elbow]
    set max_angle [get_max_angle elbow]
    set restpos    10
    set off        90
    set direc      1
    #
    # Make sure given angle is inside allowed range
    # If it is not, find an appropriate angle
    #
    set new_angle [in_range $limb $joint_nr $option $angle $min_angle \
                        $max_angle $restpos $off $direc Elbow bent]

    if {($option == "by") & ($new_angle == 0)} {

```

```

    # The joint was already at it's limit
    echo already at limit
    return limit
}
#
# Transform angle into dhangle range ( 90 - 240 ), Restpos 100
#
#   set new_angle [mult $new_angle $direc]
if {($option == "to") | ($option == "rest")} {
    set dhangle [plus $new_angle $off]
} else {
    set dhangle $new_angle
}
move_a_joint $limb $joint_nr $option $dhangle $timesteps range_angle36 $renopt

m2ele 5 [get_elbow_angle] $limb/angle.cur

if {$new_angle != $angle} {
    # The joint has just reached it's limit
    echo limit just reached
    return limit
}
return
}

#####
#           bend_thumb_joint
#####

proc bend_thumb_joint { {joint 2} {option by} {angle 10} {timesteps 2} {renopt ren} } {

    set limb    /1_arm/thumb
    set joint_nr [plus $joint 4]

    if {($joint_nr == 5) & ($option == "to")} {
        set off -90
        set angle [plus $angle $off]
    }
    move_a_joint $limb $joint_nr $option $angle $timesteps range_angle18 $renopt
}

#####
#           calc_angle_timesteps
#####

proc calc_angle_timesteps {limb axis_opt target_axis} {

    set angle_diff [vangle2 [get_{$limb}_direction $axis_opt] $target_axis]
    global angle_per_timestep

    set timesteps [int [div $angle_diff $angle_per_timestep]]
    if {[less $timesteps 1]} then {
        set timesteps 1
    }
    return $timesteps
}

```



```
*****
#         calc_dist_timesteps
*****
```

```
proc calc_dist_timesteps {goal_point gp_opt} {
    if {$gp_opt == "wrist"} then {
        set wrist_position [get_wrist_position]
        set dist [enorm [minus $wrist_position $goal_point]]
    } else {
        if {$gp_opt == "ee"} then {
            set dist [enorm [minus [get_ee] $goal_point]]
        } else {
            echo Wrong gp_opt ($gp_opt) in proc goal_reached.
            echo Parameter has to be set to wrist or ee.
            return 999
        }
    }
    global dist_per_timestep

    set timesteps [int [div $dist $dist_per_timestep]]
    if {[less $timesteps 1]} then {
        set timesteps 1
    }
    return $timesteps
}
```

```
*****
#         calc_elbow_wrist_vector
*****
```

```
proc calc_elbow_wrist_vector {goal_position gp_opt sev} {
    set shoulder_goal_vector [shoulder_goal_vector $goal_position]
    if {$gp_opt == "wrist"} then {
        set ewv [minus $shoulder_goal_vector $sev]
    } else {
        if {$gp_opt == "ee"} then {
            set ewv [minus $shoulder_goal_vector [plus [wrist_ee_vector] $sev]]
        } else {
            echo Wrong gp_opt in proc calc_elbow_wrist_vector.
            echo Parameter has to be set to wrist or ee.
            return 999
        }
    }
    return $ewv
}
```

```
*****
#         calc_M7
*****
```

```
proc calc_M7 {als_angle alf_angle} {
    #
    # dhparameters 6 : {theta6 0 0 -90}
    # dhparameters 7 : {theta7 0 0 90}
    #
    set theta6 $als_angle
}
```

```

    set theta7 [minus 90 $alf_angle]

    Mset [dhmatrix 5 /l_arm/dhc]

    Mrot z $theta6
    Mrot x -90

    Mrot z $theta7
    Mrot x 90

    return [Mget]
}

#####
#           calc_M10
#####

proc calc_M10 {als_angle alf_angle at_angle e_angle} {
    #
    # dhparameters 6 : {theta6 0 0 -90}
    # dhparameters 7 : {theta7 0 0 90}
    # dhparameters 8 : {theta8 0 0 90}
    # dhparameters 9 : { 90 0 10 0}
    # dhparameters 10 : {theta10 0 0 90}
    #
    set theta6 $als_angle
    set theta7 [minus 90 $alf_angle]
    set theta8 $at_angle
    set theta10 [plus $e_angle 90]

    Mset [dhmatrix 5 /l_arm/dhc]

    Mrot z $theta6
    Mrot x -90

    Mrot z $theta7
    Mrot x 90

    Mrot z $theta8
    Mrot x 90

    Mrot z 90
    Mtrans 10 0 0

    Mrot z $theta10
    Mrot x 90

    return [Mget]
}

#####
#           calc_shoulder_elbow_vector
#####

proc calc_shoulder_elbow_vector { goal_position gp_opt} {

    set which_x max_x
    set sgv [shoulder_goal_vector $goal_position]
    set shoulder_goal_distance [enorm $sgv]

```

```

#
# Calculate the angle between the shoulder_goal_vector and the
# shoulder_elbow_vector
#
if {$gp_opt == "wrist"} then {
  set max_shoulder_goal_distance 18.6
  if {$shoulder_goal_distance > $max_shoulder_goal_distance} {
    echo Goal is not reachable without moving the body.
    global reaching_problems
    set reaching_problems 1
    return 999
  }
  set sgv_sev_angle [gamma 10 $shoulder_goal_distance 8.6]
  if {$sgv_sev_angle == 999} {return 999}
} else {
  if {$gp_opt == "ee"} then {
    set elbow_ee_distance [enorm [elbow_ee_vector]]
    set max_shoulder_goal_distance [plus 10 $elbow_ee_distance]
    if {$shoulder_goal_distance > $max_shoulder_goal_distance} {
      echo Goal is not reachable without moving the body.
      global reaching_problems
      set reaching_problems 1
      return 999
    }
    set sgv_sev_angle [gamma 10 $shoulder_goal_distance $elbow_ee_distance]
    if {$sgv_sev_angle == 999} {return 999}
  } else {
    echo Wrong gp_opt in proc calc_shoulder_elbow_vector.
    echo Parameter has to be set to wrist or ee.
    return 999
  }
}
}

#
# Calculate the minimal z_value possible for the shoulder_elbow_vector
#
set z_range [calc_z2_range $sgv 10 $sgv_sev_angle]
if {$z_range == 999} {return 999}
set z_min [lindex $z_range 0]
set z_max [lindex $z_range 1]
set sev_z $z_min

#
# Add a tiny amount to sev_z to make sure,
# calc_shoulder_elbow_vector_xy finds a solution for x.
#
set sev_z [plus $sev_z 0.0001]

global arm_pose_opt
global arm_pose_parameter
set new_z [plus $z_min [mult $arm_pose_parameter [minus $z_max $z_min]]]
set sev_z $new_z

global change_arm_posture_parameter
set change_arm_posture_parameter $arm_pose_parameter
set sev [calc_shoulder_elbow_vector_xy $sgv $sev_z $sgv_sev_angle $which_x]
if {$sev == 999} {return 999}
return $sev
}

```

```
*****
#       calc_third_component
*****
```

```
proc calc_third_component {vec_length y z} {
  set ls [mult $vec_length $vec_length]
  set ys [mult $y $y]
  set zs [mult $z $z]

  set T [minus $ls [plus $ys $zs]]
  return [sqrt $T]
}
```

```
*****
#       calc_x2
*****
```

```
proc calc_x2 {vector1 vec2_length z2 angle} {

  set vec1_length [enorm $vector1]
  set x1 [lindex $vector1 0]
  set y1 [lindex $vector1 1]
  set z1 [lindex $vector1 2]

  set T1 [mult [mult $vec1_length $vec2_length] [cos $angle]]
  set T2 [mult [mult -2 $T1] $x1]
  set T3 [mult $z1 $z2]
  set T4 [mult 2 [mult $T3 $x1]]
  set T5 [mult $y1 $y1]
  set T6 [plus $T5 [mult $x1 $x1]]
  set T7 [mult [mult -2 $T1] $T3]
  set T8 [mult [mult $vec2_length $vec2_length] $T5]
  set T9 [mult $T5 [mult $z2 $z2]]

  set p [div [plus $T2 $T4] $T6]
  set q [div [plus [minus [plus [plus [mult $T1 $T1] $T7] [mult $T3 $T3]] $T8] $T9] $T6]
  set root_arg [minus [div [mult $p $p] 4] $q]
  if {[less $root_arg 0]} {
    return 999
  }
  set x2_min [minus [div $p -2] [sqrt $root_arg]]
  set x2_max [plus [div $p -2] [sqrt $root_arg]]

  return [list $x2_min $x2_max]
}
```

```
*****
#       calc_z2_range
*****
```

```
proc calc_z2_range {vector1 vec2_length angle} {

  set x1 [lindex $vector1 0]
  set y1 [lindex $vector1 1]
  set z1 [lindex $vector1 2]
  set vec1_length [enorm $vector1]

  set noglob 1
```

```

set term1 [expr { $vec2_length*$vec2_length / ($vec1_length*$vec1_length)}]
set p [expr { -2 * $z1 * $vec2_length / $vec1_length * [cos $angle]}]
set q [expr { -1 * $term1 * ($y1*$y1 + $x1*$x1 - \
    ($vec1_length*$vec1_length) * [cos $angle]*[cos $angle])}]
set root_arg [expr { $p*$p / 4 - $q }]

if {[less $root_arg 0]} {
    return 999
}
set z2_min [minus [div $p -2] [sqrt $root_arg]]
set z2_max [plus [div $p -2] [sqrt $root_arg]]
return [list $z2_min $z2_max]
}

#####
#           direct_hand_palm_towards
#####

proc direct_hand_palm_towards {object {time_opt 0} {timesteps 10}
    {end_event {}}} {
    #
    # check whether the initial conditions are fulfilled
    # and put the command on the list
    #
    set condition0a 1

    if { $condition0a} then {

        global direct_hand_palm_towards_end_event
        set direct_hand_palm_towards_end_event $end_event

        global ic
        set end_time [plus $ic $timesteps]

        add_c [list direct_hand_palm_towards_state1 $object $time_opt $end_time]
    } else {
        echo Sorry, the initial conditions for executing this skill are not given!
    }
    Com_Update_cb
    return
}

#####
#           direct_hand_palm_towards_state1
#####

proc direct_hand_palm_towards_state1 {object time_opt end_time} {

    set point [centroid $object]
    set ee [get_ee]
    set palm_axis [minus $point $ee]

    #
    # State ending condition
    #
    set direct_hand_palm_towards_condition1 [test_hand_direction palm $palm_axis]

    if $direct_hand_palm_towards_condition1 then {
        global direct_hand_palm_towards_end_event

```

```

    eval $direct_hand_palm_towards_end_event
    return stop
}

if {$time_opt == 0} then {
    set timesteps [calc_angle_timesteps hand palm $palm_axis]
} else {
    global ic
    set timesteps [minus $end_time $ic]
    if {[lesseq $timesteps 0]} then {
        set timesteps 1
    }
}

set hand_angles [hand_angles_rel $palm_axis no $timesteps]
if [equal $hand_angles 999] then {
    echo Sorry, cannot position the hand_palm axis in desired direction.
    return 999
}

set tla_angle [lindex $hand_angles 0]
set bh_angle [lindex $hand_angles 1]
set th_angle [lindex $hand_angles 2]

if [greater [abs $tla_angle] 0.0001] then {
    turn_lower_arm by $tla_angle 1 noren
}
if [greater [abs $bh_angle] 0.0001] then {
    bend_hand by $bh_angle 1 noren
}
if [greater [abs $th_angle] 0.0001] then {
    twist_hand by $th_angle 1 noren
}
return
}

#####
#           direct_upper_arm
#####

proc direct_upper_arm {main_axis {perp_axis no} {time_opt 0} \
    {timesteps 10} {end_event {return stop}}} {

    set conditionI 1
    #
    # check which state needs to be put on the commandlist
    #
    set condition0a {[string compare $perp_axis "no"]}
    set condition0b {[equal [string compare $perp_axis "no"] 0]}

    if !$conditionI then {
        echo Sorry, the initial conditions for executing this skill are not given!
        return
    }

    global direct_upper_arm_end_event
    set direct_upper_arm_end_event $end_event

    global ic
    set end_time [plus $ic $timesteps]

```

```

if $condition0a then {
  add_c [list direct_upper_arm_state1 $main_axis $perp_axis $time_opt $end_time]
}
if $condition0b then {
  add_c [list direct_upper_arm_state2 $main_axis $time_opt $end_time]
}
Com_Update_cb
return
}

```

```

*****
#           dps
*****

```

```

proc dps {{dist {}}} {

  global dist_per_timestep

  if {$dist == ""} then {
    echo distance_per_timestep: $dist_per_timestep
  } else {
    set dist_per_timestep $dist
  }
  return $dist_per_timestep
}

```

```

*****
#           forearm_angles_abs
*****

```

```

proc forearm_angles_abs { dvec {M7 dh7}} {

  set current_theta10 [dhtheta 10 /l_arm/dhc]
  set current_theta8  [dhtheta 8 /l_arm/dhc]

  set dvec [div $dvec [enorm $dvec]]
  set d0 [lindex $dvec 0]
  set d1 [lindex $dvec 1]
  set d2 [lindex $dvec 2]

  if {[string compare $M7 "dh7"] == 0} then {
    set M7 [dhmatrix 7 /l_arm/dhc]
  }
  set a [lindex $M7 0]
  set b [lindex $M7 4]
  set d [lindex $M7 8]
  set e [lindex $M7 1]
  set f [lindex $M7 5]
  set g [lindex $M7 9]
  set h [lindex $M7 2]
  set i [lindex $M7 6]
  set k [lindex $M7 10]

  set h1 [expr { ($d1 * $a - $d0 * $e) * ($b * $h - $i * $a) }]
  set h2 [expr { ($d2 * $a - $d0 * $h) * ($f * $a - $b * $e) }]
  set h3 [expr { ($g * $a - $d * $e) * ($b * $h - $i * $a) }]
  set h4 [expr { ($k * $a - $d * $h) * ($f * $a - $b * $e) }]
}

```

```

if {[abs [plus $h3 $h4]] < 0.001} then {
    echo Sorry, cannot calculate forearm direction angles \
        because of the nature of matrix of the upper_arm (M7)
}
set s10 [expr { ($h1 + $h2) / ($h3 + $h4) }]
# asin yields -90 <= theta10 <= 90
set theta10 [asin $s10]

if {[equal $theta10 90]} then {
    set arm_turn_angle [get_arm_turn_angle]
    echo arm_turn_angle does not matter -> leave the current one
    set elbow_angle 0
    return [list $arm_turn_angle $elbow_angle ]
}
if {[less $theta10 90] | [greater $theta10 240]} then {
    set theta10 [minus 180 $theta10]
    if {[less $theta10 90] | [greater $theta10 240]} then {
        echo Sorry, the left lower arm cannot be positioned in the \
            specified direction. (Elbow needed to bend to [minus $theta10 90]).
        echo => Elbow will only be bent to 150 degrees.
        set theta10 240
    }
}
set c10 [cos $theta10]
set h5 [expr { ($d1 * $a - $d0 * $e) + ($d * $e - $g * $a) * $s10 }]
set h6 [expr { ($f * $a - $b * $e) * $c10 }]
if {[abs $h6] < 0.001} then {
    set h5 [expr { ($d2 * $a - $d0 * $h) + ($d * $h - $k * $a) * $s10 }]
    set h6 [expr { ($i * $a - $b * $h) * $c10 }]
}
set s8 [expr { $h5 / $h6 }]
set theta8a [asin $s8]
set theta8b [minus 180 $theta8a]

dhtheta 10 $theta10 /l_arm/dhc
#
# case1 (theta8a) :
#
dhtheta 8 $theta8a /l_arm/dhc
set M10 [calc_M10_from_M7_8_10 $M7 $theta8a $theta10]
set M10vec [list [lindex $M10 8] [lindex $M10 9] [lindex $M10 10]]
set diff1 [enorm [minus $dvec $M10vec]] ; myecho 4 diff1 $diff1
if {$diff1 < 0.01} then {
    set theta8 $theta8a
} else {
    #
    # case2 (theta8b) :
    #
    dhtheta 8 $theta8b /l_arm/dhc
    set M10 [calc_M10_from_M7_8_10 $M7 $theta8b $theta10]
    set M10vec [list [lindex $M10 8] [lindex $M10 9] [lindex $M10 10]]
    set diff2 [enorm [minus $dvec $M10vec]] ; myecho 4 diff2 $diff2
    if {$diff2 < 0.01} then {
        set theta8 $theta8b
    } else {
        #
        # case3 (theta8) :
        #
        # In case theta10 was artificially set to 240 (cause it cannot
        # reach any bigger angle) then no theta8 will satisfy the

```



```

        # equations exactly. Take the better one of the theta8's.
        if {[less $diff1 $diff2]} then {
            set theta8 $theta8a
        } else {
            set theta8 $theta8b
        }
    }
}
dhtheta 8 $current_theta8 /l_arm/dhc
dhtheta 10 $current_theta10 /l_arm/dhc

set arm_turn_angle $theta8
set elbow_angle    [minus $theta10 90]

return [list $arm_turn_angle $elbow_angle ]
}

#####
#           gamma
#####

proc gamma { a b c } {

    set argument [div [minus [plus [mult $a $a] [mult $b $b]] \
                        [mult $c $c]] [mult [mult 2.0 $a] $b] ]
    if [greater [abs $argument] 1] then {
        echo The absolut argument for acos in proc gamma is greater than 1.
        return 999
    }
    set gamma [acos $argument]
    return $gamma
}

#####
#           get_elbow_angle
#####

proc get_elbow_angle {} {

    set limb    /l_arm
    set dhc    $limb/dhc
    set joint_nr 10

    set off    90
    set direc  1

    set angle [mult [minus [dhtheta $joint_nr $dhc] $off] $direc]
    return $angle
}

#####
#           get_max_angle
#####

proc get_max_angle { jointmotion } {

    case $jointmotion in {

```

```

        shoulder_fb {return 40}
        shoulder_ld {return 45}
        arm_ls      {return 180}
        arm_lf      {return 190}
        arm_turn    {return 175}
        elbow       {return 150}
        hand_turn   {return 190}
        hand_bend   {return 80}
        hand_twist  {return 35}
    }
    echo Invalid parameter option in get_max_angle
    return
}

#####
#           get_min_angle
#####

proc get_min_angle { jointmotion } {

    case $jointmotion in {
        shoulder_fb {return -20}
        shoulder_ld {return -5}
        arm_ls      {return -180}
        arm_lf      {return -90}
        arm_turn    {return -170}
        elbow       {return 0}
        hand_turn   {return -40}
        hand_bend   {return -80}
        hand_twist  {return -35}
    }
    echo Invalid parameter option in get_max_angle
    return
}

#####
#           get_shoulder_position
#####

proc get_shoulder_position {} {

    set m [dhmatrix 5 /l_arm/dhc]
    return [list [lindex $m 12] [lindex $m 13] [lindex $m 14]]
}

#####
#           goal_arm_angles_abs
#####

proc goal_arm_angles_abs {goal_position gp_opt} {

    #
    # Calculate the jointangles for the arm_ls, arm_lf, arm_turn, and
    # elbow joint
    #
    set sev [calc_shoulder_elbow_vector $goal_position $gp_opt]
    if {$sev == 999} {return 999}
}

```

```

set upper_arm_angles [upper_arm_angles_abs $sev no]
if {$upper_arm_angles == 999} {return 999}
set new_arm_ls [lindex $upper_arm_angles 0]
set new_arm_lf [lindex $upper_arm_angles 1]

#
# Calculate the new 7. dhmatrix given the new upper_arm_angles
#
set M7 [calc_M7 $new_arm_ls $new_arm_lf]
if {$gp_opt == "ee"} then {
  set forearm_angles [calc_at_e $M7 $goal_position]
  if {$forearm_angles == 999} {return 999}
} else {
  set ewv [calc_elbow_wrist_vector $goal_position $gp_opt $sev]
  set forearm_angles [forearm_angles_abs $ewv $M7]
  if {$forearm_angles == 999} {return 999}
}
return [list $new_arm_ls $new_arm_lf [lindex $forearm_angles 0] [lindex $forearm_angles 1]]
}

```

```

#####
#           hand_angles_rel
#####

```

```

proc hand_angles_rel { palm_axis point_axis timesteps} {
  set current_hand_turn_angle [get_hand_turn_angle]
  set current_hand_bend_angle [get_hand_bend_angle]
  set current_hand_twist_angle [get_hand_twist_angle]

  if {[llength $point_axis] == 3} then {
    set point_axis [div $point_axis [enorm $point_axis]]
    set m0 [lindex $point_axis 0]
    set m1 [lindex $point_axis 1]
    set m2 [lindex $point_axis 2]
  }

  set palm_axis [div $palm_axis [enorm $palm_axis]]
  set n0 [lindex $palm_axis 0]
  set n1 [lindex $palm_axis 1]
  set n2 [lindex $palm_axis 2]

  set M10 [dhmatrix 10 /1_arm/dhc]
  set a [lindex $M10 0]
  set b [lindex $M10 4]
  set d [lindex $M10 8]
  set e [lindex $M10 1]
  set f [lindex $M10 5]
  set g [lindex $M10 9]
  set h [lindex $M10 2]
  set i [lindex $M10 6]
  set k [lindex $M10 10]

  #####          Calculation of theta13          #####

  set h1 [expr { ($e * $n0 - $a * $n1) * ($b * $h - $i * $a) }]
  set h2 [expr { ($h * $n0 - $a * $n2) * ($f * $a - $b * $e) }]
  set h3 [expr { ($g * $a - $d * $e) * ($b * $h - $i * $a) }]
  set h4 [expr { ($k * $a - $d * $h) * ($f * $a - $b * $e) }]

  if {[abs [plus $h3 $h4]] < 0.001} then {

```

```

        echo Sorry, cannot calculate hand direction angles \
            because of the nature of matrix of the hand (M14)
    }
    set s13 [expr { ($h1 + $h2) / ($h3 + $h4) }]
    set theta13 [asin $s13]
    # asin yields  -90 <= theta13 <= 90

    *
    * Check, whether theta13 is outside of it's allowed range.
    * If this is the case, set it to it's maximum/ minimum angle.
    *
    if {[less $theta13 -80] | [greater $theta13 80]} then {
        echo Sorry, the left hand cannot be positioned pointing in the
        echo specified direction. (Hand needed to bend to $theta13).
        # or [minus 180 $theta13]
        echo Therefore the hand will be bent as far as possible, to
        if {[less $theta13 -80]} then {
            echo -80 degrees.
            set theta13 -80
        } else {
            echo 80 degrees.
            set theta13 80
        }
    }
    dhtheta 13 $theta13 /l_arm/dhc

    *****      Calculation of theta11      *****

    set c13 [cos $theta13]
    set h5 [expr { ($e * $n0 - $a * $n1) + ($d * $e - $g * $a) * $s13 }]
    set h6 [expr { ($f * $a - $b * $e) * $c13 }]
    if {[abs $h6] < 0.001} then {
        set h5 [expr { ($h * $n0 - $a * $n2) - ($d * $h - $k * $a) * $s13 }]
        set h6 [expr { ($i * $a - $b * $h) * $c13 }]
    }
    set c11 [expr { $h5 / $h6 }]
    # acos yields in 0 <= theta11 <= 180
    set theta11a [acos $c11]
    # 180 <= theta11b <= 360
    set theta11b [minus 360 $theta11a]
    *
    * Find out, which one of the two solutions for theta11 is the right one.
    *
    * Note that the palm vector of the hand is not influenced
    * by theta14 (hand twisting angle), so that it does not
    * matter for the following test, what theta14 is.
    *
    # case 1 (theta11) :
    dhtheta 11 $theta11a /l_arm/dhc
    set angle_diff1 [abs [vangle2 [get_hand_direction palm] $palm_axis]]
    if [less $angle_diff1 0.1] then {
        set theta11 $theta11a
    } else {
        # case 2 (theta11) :
        dhtheta 11 $theta11b /l_arm/dhc
        set angle_diff2 [abs [vangle2 [get_hand_direction palm] $palm_axis]]
        if [less angle_diff2 0.1] then {
            set theta11 $theta11b
        } else {
            *

```

```

# case 3 (theta1) :
# Take the best possible solution.
#
if {[less $angle_diff1 $angle_diff2]} then {
  set theta1 $theta1a
} else {
  set theta1 $theta1b
}
}
}
#
# Now, check whether theta1 is inside it's allowed range
# If not, set it to it's maximal or minimal allowed angle.
#
# set theta1_min [plus [get_min_angle hand_turn] 90] => 50
# set theta1_max [plus [get_max_angle hand_turn] 90] => 280
#
if [less $theta1 50] then {
  echo Sorry, the left hand cannot be positioned pointing in the specified direction.
  echo The hand needed to be turned to [minus $theta1 90] \
    but can only be turned to -40 degrees for physical reasons.
  set theta1 50
}
if [greater $theta1 280] then {
  echo Sorry, the left hand cannot be positioned pointing in the specified direction.
  echo The hand needed to be turned to [minus $theta1 90] \
    but can only be turned to 190 degrees for physical reasons.
  set theta1 280
}
}
dhtheta 11 $theta1 /l_arm/dhc

##### Calculation of theta14 #####

if {[llength $point_axis] == 3} then {

  set h1 [expr { ($a * $m2 - $h * $m0) * ($f * $a - $b * $e) }]
  set h2 [expr { ($a * $m1 - $e * $m0) * ($b * $h - $i * $a) }]
  set h3 [expr { ($g * $a - $d * $e) * ($b * $h - $i * $a) }]
  set h4 [expr { ($k * $a - $d * $h) * ($f * $a - $b * $e) }]
  # Note: theta13 can never be 90 or -90, so c13 will never be 0

  if {[abs [plus $h3 $h4]] < 0.001} then {
    echo Sorry, cannot calculate hand direction angles \
      because of the nature of matrix of the hand (M14)
  }
  set c14 [expr { ($h1 + $h2) / (($h3 + $h4) * $c13) }]
  set theta14 [acos $c14]
  set theta14b [mult -1 $theta14]

# case 1 (theta14) :
dhtheta 14 $theta14 /l_arm/dhc
set angle_diff1 [abs [vangle2 [get_hand_direction point] $point_axis]]
if [less $angle_diff1 0.1] then {
} else {
# case 2 (theta14) :
dhtheta 14 $theta14b /l_arm/dhc
set angle_diff2 [abs [vangle2 [get_hand_direction point] $point_axis]]
if [less $angle_diff2 0.1] then {
  set theta14 $theta14b
} else {

```

```

    *
    * case 3 (theta14) :
    * Take the best possible solution
    *
    if [greater $angle_diff1 $angle_diff2] then {
        set theta14 $theta14b
    }
}
*
* Check, whether theta14 is inside it's allowed range.
* If it is not, set it to it's allowed maximum/ mininum.
*
if {[less $theta14 -35] | [greater $theta14 35]} then {
    echo {}
    echo Sorry, the left hand cannot be positioned pointing along \
        the specified point axis.
    echo (Hand needed to be twisted to [mult -1 $theta14])
    echo But I will do the best I can !
    echo Palm_axis might be as specified, but pointing_axis will not.
    echo You could try to change my arm_turn angle or      \
        elbow_angle to reach the point_axis you specified!
    echo (You have to find the direction of the change though.)
    echo {}
    if {[less $theta14 -35]} then {
        echo Hand is supposed to twist [plus $theta14 35] degrees more, but it cannot!
        set theta14 -35
    } else {
        * (meaning, if [greater $theta14 35])
        echo Hand is supposed to twist [minus $theta14 35] degrees more, but it cannot!
        set theta14 35
    }
}
} else {
    set theta14 0
}

* Set the joints back to the original angles
turn_lower_arm to $current_hand_turn_angle 1 noren
bend_hand      to $current_hand_bend_angle 1 noren
twist_hand     to $current_hand_twist_angle 1 noren

set tla_angle_diff [minus [minus $theta11 90] $current_hand_turn_angle]
set bh_angle_diff  [minus $theta13 $current_hand_bend_angle]
set th_angle_diff  [minus [mult -1 $theta14] $current_hand_twist_angle]

*
* angle step vector = angle difference vector / new_times
*
set tla_angle_step [div $tla_angle_diff $timesteps]
set bh_angle_step  [div $bh_angle_diff  $timesteps]
set th_angle_step  [div $th_angle_diff  $timesteps]

return [list $tla_angle_step $bh_angle_step $th_angle_step]
}

```

```

#####
#           in_range
#####

proc in_range { limb joint_nr option angle min_angle max_angle restpos off direc part action } {
#
# Make sure given angle is inside allowed range
#
case $option in {
  to {
    if {[less $angle $min_angle]} then {
      echo {}
      echo Given angle is outside of the allowed range.
      echo Therefore, the $part will be $action to the minimum
      echo position of $min_angle degrees.
      set angle $min_angle
    } else {
      if [greater $angle $max_angle] {
        echo {}
        echo Given angle is outside of the allowed range.
        echo Therefore, the $part will be $action to the maximum
        echo position of $max_angle degrees.
        set angle $max_angle
      }
    }
  }
}
by {
  set dhc $limb/dhc
  set current_angle [mult [minus [dtheta $joint_nr $dhc] $off] $direc ]
  set test_pos_angle [plus $angle $current_angle]
  if [less $test_pos_angle $min_angle] then {
    set angle [minus $min_angle $current_angle]
    echo $part would be $action to an angle outside of the allowed range.
    echo Therefore, the $part will be $action by maximal $angle degrees.
  } else {
    if [greater $test_pos_angle $max_angle] {
      set angle [minus $max_angle $current_angle]
      echo The $part would be $action to an angle outside of the allowed
      echo range. Therefore $part will be $action by maximal $angle degrees.
    }
  }
}
rest { set angle $restpos }
}
return $angle
}

```

```

#####
#           insert_c
#####

proc insert_c_list {list nr_next_command} {
  global commandlist;
  foreach comm $list {
    set commandlist [linsert $commandlist $nr_next_command $comm]
    incr nr_next_command
  }
  return
}

```

```

*****
#         left-circle
*****

proc left-circle {radius} {

    echo r1 = [set r1 0]
    echo {}
    echo l1 = [set l1 [mult $radius [minus 1 [cos 22.5]]]]
    echo r2 = [set r2 [mult $radius [sin 22.5]]]
    echo {}
    echo l2 = [set l2 [minus [mult $radius [minus 1 [cos 45]]] $l1]]
    echo r3 = [set r3 [mult $radius [sin 45]]]
    echo {}
    echo l3 = [set l3 [minus [minus [mult $radius [minus 1 [cos 67.5]]] $l2] $l1] ]
    echo r4 = [set r4 [mult $radius [sin 67.5]]]
    echo {}
    echo l4 = [set l4 [minus [minus [minus $radius $l3] $l2] $l1]]
    echo r5 = [set r5 $radius]
    echo {}
    return [list $r1 $l1 $r2 $l2 $r3 $l3 $r4 $l4 $r5 ]
}

```

```

*****
#         make-bone
*****

proc make-bone {name left_radius right_radius length {precision 15}} {

    set one [left-circle $left_radius]
    set two [right-circle $right_radius]

    make-bone-data $name $precision 10 \
        [lindex $one 0] [lindex $one 1] [lindex $one 2] \
        [lindex $one 3] [lindex $one 4] [lindex $one 5] \
        [lindex $one 6] [lindex $one 7] [lindex $one 8] \
    $length [lindex $two 0] [lindex $two 1] [lindex $two 2] \
        [lindex $two 3] [lindex $two 4] [lindex $two 5] \
        [lindex $two 6] [lindex $two 7] [lindex $two 8]
}

```

```

*****
#         make-bone-data
*****

proc make-bone-data {name precision nrrad r1 l1 r2 \
    {l2 1} {r3 1} {l3 1} {r4 1} {l4 1} {r5 1} {l5 1} \
    {r6 1} {l6 1} {r7 1} {l7 1} {r8 1} {l8 1} {r9 1} \
    {l9 1} {r10 1} {l10 1} {r11 1} {l11 1} {r12 1} {l12 1} \
    {r13 1} {l13 1} {r14 1} } {

    if {$name == ""} then {
        set filename ~/graphics/tcl/data/cylinder.asc
    } else {
        set filename ~/graphics/tcl/data/$name.asc
    }
    set file          [open $filename w]
    set nopts         [mult $precision $nrrad]
}

```



```

set nrmantelpatches [mult $precision [minus $nrrad 1]]
set nopolys          [plus $nrmantelpatches 2]

puts $file "$nopts $nopolys"

set step [expr 360.0/$precision]
set limit [expr 360.0-$step/2]
set count 0

set nrlength [minus $nrrad 1]
set length 0
for {set nrl 1} {$nrl <= $nrlength} {incr nrl} {
    set length [plus $length [set l$nrl]]
}
set x [mult $length -1]

for {set nr 1} {$nr <= $nrrad} {incr nr} {
    for {set angle 0.0} {$angle < $limit} {set angle [plus $angle $step]} {
        incr count
        set z [mult [set r$nr] [cos $angle]]
        set y [mult [set r$nr] [sin $angle]]

        if {[abs $z] < 0.0001} {set z 0.0}
        if {[abs $y] < 0.0001} {set y 0.0}
        puts $file "$x $y $z"
    }
    set x [plus $x [set l$nr]]
    if {[abs $x] < 0.0001} {set x 0.0}
}
for {set k 1} {$k < $nrmantelpatches} {set k [plus $k $precision]} {
    for {set i $k} {$i <= [plus $k [minus $precision 2]]} {incr i} {
        set next [plus $i $precision]
        puts $file "4 $i [plus $i 1] [plus $next 1] $next"
    }
    puts $file "4 $i $k [plus $k $precision] [plus $i $precision]"
}

set all {}
for {set n $precision} {$n >= 1} {set n [minus $n 1]} {
    append all " $n"
}
puts $file "$precision $all"

set all {}
for {set n [plus [minus $nopts $precision] 1]} {$n <= $nopts} {incr n} {
    append all " $n"
}
puts $file "$precision $all"
close $file
return "OSU file $filename successfully written"
}

```

```

#####
#           move_a_joint
#####

```

```

proc move_a_joint {limb joint_nr option dhangle {timesteps 2} \
    {range_angle range_angle18} {renopt ren}} {

    set dhc    $limb/dhc

```

```

*
* Find out if limb and joint exist
*
if (($option == "to") | ($option == "rest")) then {
    set dhtheta_current [dhtheta $joint_nr $dhc]
    *
    * Find out which angle the joint has to move to reach
    * the specified angle (joint position)
    *
    set dhangle [minus $dhangle $dhtheta_current]
} else {
    if {$option != "by"} {
        echo Wrong option: You can either choose
        echo "by" => bend joint by specified angle or
        echo "to" => bend joint to specified angle
        return
    }
}
if [less [abs $dhangle] 0.1] then {
    set timesteps 1
}
set step [div $dhangle $timesteps]
for {set i 1} {$i <= $timesteps} {incr i} {
    set dhtheta_current [dhtheta $joint_nr $dhc]
    *
    * Range the angles in case they get out of the range 0-360
    *
    set dhtheta_new [$range_angle [plus $dhtheta_current $step] ]
    dhtheta $joint_nr $dhtheta_new $dhc
    if {$renopt == "ren"} ren
}
return
}

#####
*      move_arm_to_goal
#####

proc move_arm_to_goal {goal_point {gp_opt ee} {time_opt 0} {timesteps 1} {end_event {}}} {
    *
    * check whether the initial conditions are fulfilled
    * and put the command on the list
    *
    * set condition0a [goal_in_reaching_distance]
    set condition0a 1

    if { $condition0a} then {
        global move_arm_to_goal_end_event
        set move_arm_to_goal_end_event $end_event
        global ic
        set end_time [plus $ic $timesteps]
        add_c [list move_arm_to_goal_state1 $goal_point $gp_opt $time_opt $end_time]
    } else {
        echo Sorry, the initial conditions for executing this skill are not given!
    }
    Com_Update_cb
    return
}
}

```

```

*****
#           object_l_axis
*****

proc object_l_axis {object} {
    set m [matrix $object]
    return [list [lindex $m 8] [lindex $m 9] [lindex $m 10] ]
}

*****
#           put_arm_on_table
*****

proc put_arm_on_table { {end_event {}} } {
    #
    # Check whether the initial conditions are fulfilled
    # and put the command on the list
    #
    set condition0 [joint_in_box wrist table z_plus_in]

    if $condition0 then {

        global put_arm_on_table_end_event
        set put_arm_on_table_end_event $end_event

        add_c {put_arm_on_table_state1}
        Com_Update_cb
    } else {
        echo Sorry, the initial conditions for executing this skill are not given!
    }
    return
}

*****
#           put_arm_on_table_state1
*****

proc put_arm_on_table_state1 {} {
    #
    # State ending condition
    #
    set condition1a [collision_lower_arm /table]
    set condition1b [collision_finger /table]

    if $condition1a then {
        add_c {put_arm_on_table_state2}
        return stop
    }
    if $condition1b then {
        add_c {put_arm_on_table_state3}
        return stop
    }
    bend_elbow by -1 1 noren
}

```

```

*****
#           put_arm_on_table_state2
*****

proc put_arm_on_table_state2 {} {
#
# State ending condition
#
set condition2 [collision_finger /table]

if {$condition2} then {
    global put_arm_on_table_end_event
    eval $put_arm_on_table_end_event
    return stop
}
bend_hand by 1 1 noren
}

*****
#           put_arm_on_table_state3
*****

proc put_arm_on_table_state3 {} {
#
# State ending condition
#
set condition3 [equal [collision_finger /table] 0]

if {$condition3} then {
    add_c {put_arm_on_table_state4}
    return stop
}
bend_hand by -1 1 noren
}

*****
#           put_arm_on_table_state4
*****

proc put_arm_on_table_state4 {} {
#
# State ending condition
#
set condition4a [collision_lower_arm /table]
set condition4b [collision_finger /table]

if {$condition4a} then {
    add_c {put_arm_on_table_state2}
    return stop
}
if {$condition4b} then {
    add_c {put_arm_on_table_state3}
    return stop
}
bend_elbow by -1 1 noren
bend_hand by -1 1 noren
}

```

```

*****
#       reach_angles_abs
*****

proc reach_angles_abs {goal_point gp_opt thumb_axis hand_bend_angle} {

    set arm_angles [goal_arm_angles_abs $goal_point $gp_opt]
    if {$arm_angles == 999} {return 999}

    set M10 [calc_M10 [lindex $arm_angles 0] [lindex $arm_angles 1] \
                  [lindex $arm_angles 2] [lindex $arm_angles 3]]
    set hand_angles [aligned_hand_angles_abs $thumb_axis $hand_bend_angle $M10 ]
    if {$hand_angles == 999} {return 999}

    return [concat [get_shoulder_fb_angle] [get_shoulder_ld_angle] $arm_angles $hand_angles]
}

*****
#       rem_c
*****

proc rem_c { command } {
    global commandlist;
    set nr [lsearch $commandlist $command]
    if {$nr != -1} {rem_nth $nr}
}

*****
#       rem_nth
*****

proc rem_nth {n} {
    global commandlist
    set newlist {}
    for {set i 0} {[less $i [llength $commandlist]]} {set i [plus $i 1]} {
        if {![equal $n $i]} {
            lappend newlist [lindex $commandlist $i]
        }
    }
    set commandlist $newlist
}

*****
#       right-circle
*****

proc right-circle {radius} {

    echo r6 = [set r6 $radius]
    echo {}
    echo l6 = [set l6 [mult $radius [cos 67.5]]]
    echo r7 = [set r7 [mult $radius [sin 67.5]]]
    echo {}
    echo l7 = [set l7 [minus [mult $radius [cos 45]] $l6]]
    echo r8 = [set r8 [mult $radius [sin 45]]]
    echo {}
    echo l8 = [set l8 [minus [minus [mult $radius [cos 22.5]] $l7] $l6] ]
    echo r9 = [set r9 [mult $radius [sin 22.5]]]
}

```

```

    echo {}
    echo r9 = [set r9 [minus [minus [minus $radius $r8] $r7] $r6]]
    echo r10 = [set r10 0]
    echo {}
    return [list $r6 $r7 $r8 $r9 $r10 ]
}

#####
#           server
#####

proc server { {nr 1000000} } {
    global commandlist;
    global loop_in loop_abort;
    global ic

    if {$loop_in} return
    set loop_in 1
    global topcommands
    set top [Toplevel]
    if {[nilp $topcommands]} {Toplevel $topcommands}

    for {set i 0} {[less $i $nr]} {set i [plus $i 1]} {

        if {$loop_abort} break;
        for {set c 0} {[length $commandlist] > $c} {set c [plus $c 1]} {

            set command [lindex $commandlist $c]
            set returnvalue [eval $command]
            if {($returnvalue == "limit") | ($returnvalue == "stop") | ($returnvalue == "target")} {
                rem_c $command
                if {[nilp $topcommands]} {Com_Update_cb}
                set c [minus $c 1]
            }
        }
        execjobs
        ren
        incr ic
    }
    set loop_in 0
    return
}

#####
#           twist_hand
#####

proc twist_hand { {option to} {angle 0} {timesteps 2} {renopt ren} } {

    set limb    /l_arm
    set joint_nr 14
    set min_angle [get_min_angle hand_twist]
    set max_angle [get_max_angle hand_twist]
    set restpos    0
    set off        0
    set direc     -1
    #
    # Make sure given angle is inside allowed range
    # If it is not, find an appropriate angle
}

```

```

*
set angle [in_range $limb $joint_nr $option $angle $min_angle \
          $max_angle $restpos $off $direc Hand twisted]
*
* Take into account that the original angle might be off
* (because of the formation of the dhc) and that the direction might
* be reversed
*
set angle [mult $angle $direc]
move_a_joint $limb $joint_nr $option $angle $timesteps range_angle18 $renopt

m2ele 8 [get_hand_twist_angle] $limb/angle.cur
return
}

```

```

*****
#           upper_arm_angles_abs
*****

proc upper_arm_angles_abs {main_axis perp_axis} {

  # It is important to make sure the specified axis is a unit vector
  set main_axis [div $main_axis [enorm $main_axis]]

  Mset [dhmatrix 5 /l_arm/dhc]
  Minvert
  Mele 3 0 0
  Mele 3 1 0
  Mele 3 2 0
  set B [Mxform $main_axis]

  set c7 [lindex $B 2]
  set theta7 [acos $c7]
  # acos yields 0 <= theta7 <= 180
  set theta7b [mult -1 $theta7]

  if {[greater $theta7 160]} then {
    # if 160 < theta7
    echo Sorry, the specified direction for the main axis
    echo of the upper_arm is not reachable.
    echo The upper_arm is supposed to be lifted forwards to $theta7a
    echo degrees, but it will only be lifted to 160 degrees
    set theta7 160

    set s7 [sin $theta7]
    set c6 [div [lindex $B 0] $s7]
    set theta6 [acos $c6]
    # do the best you can
    if {[greater [abs [minus [sin $theta6] [div [lindex $B 1] $s7]]] \
        [abs [minus [sin [mult -1 $theta6]] [div [lindex $B 1] $s7]]]]]
    } then {
      set theta6 [mult -1 $theta6]
    }
  } else {
    if {[greater $theta7 90]} then {
      # if 90 < theta7 < 160
      set s7 [sin $theta7]
      set c6 [div [lindex $B 0] $s7]
      set theta6 [acos $c6]
      if {[greater [abs [minus [sin $theta6] \

```

```

        [div [lindex $B 1] $s7]] 0.01] } then {
    set theta6 [mult -1 $theta6]
  }
} else {
  if {$theta7 == 0} then {
    # theta6 doesn't matter => leave current value
    set theta6 [dhtheta 6 /l_arm/dhc]
  } else {
    # if 0 < theta7 < 90
    # figure out whether theta7 or theta7b causes the
    # arm to move the least joint angle distance sum
    #
    set s7 [sin $theta7]
    set c6 [div [lindex $B 0] $s7]
    set theta6a [acos $c6]
    if {[greater [abs [minus [sin $theta6a] [div [lindex $B 1] $s7]]] 0.01] } then {
      set theta6a [mult -1 $theta6a]
    }
    set s7 [sin $theta7b]
    set c6 [div [lindex $B 0] $s7]
    set theta6b [acos $c6]
    if {[greater [abs [minus [sin $theta6b] [div [lindex $B 1] $s7]]] 0.01] } then {
      set theta6b [mult -1 $theta6b]
    }
    set theta6_current [dhtheta 6 /l_arm/dhc]
    set theta7_current [dhtheta 7 /l_arm/dhc]

    set suma [plus [abs [minus $theta6a $theta6_current]] \
                  [abs [minus $theta7 $theta7_current]]]
    set sumb [plus [abs [minus $theta6b $theta6_current]] \
                  [abs [minus $theta7 $theta7_current]]]
    if {[greater $suma $sumb]} then {
      set theta6 $theta6b
      set theta7 $theta7b
    } else {
      set theta6 $theta6a
    }
  }
}
}
# perp_axis not yet accounted for

set theta8 [dhtheta 8 /l_arm/dhc]
set off7 90

set als_angle $theta6
set alf_angle [minus $off7 $theta7]
set at_angle $theta8

return [list $als_angle $alf_angle $at_angle]
}

#####
#          vangle2
#####

proc vangle2 {p1 p2} {
  set dot [dot $p1 $p2]
  set arg [div $dot [mult [enorm $p1] [enorm $p2]]]
  #
}

```



```

# For the arc cos procedure the input needs to be 1 or less
# If arg happens to be just slightly greater than 1 set it to 1
# so the arc cos function will return 0.
#
if {[greater $arg 1] & [less $arg 1.001] | \
    [less $arg -1] & [greater $arg -1.001]} then {
    echo ***** arg was $arg, set to [sign $arg] *****
    set arg [sign $arg]
}
set angle [acos $arg]
return $angle
}

*****
# write_reach_file
*****
proc write_reach_file {sp {object glas} {hand_angle -10} {finger_timesteps 3} \
    {end_event {return stop}}} {

    set count [minus [llength $sp] 2]

    if {$count < 6} then {
        set finger_timesteps $count
    }
    set finger_count [minus $count $finger_timesteps]

    if ![slfile dummy isobj] {io {} dummy}
    matrix [matrix $object] dummy
    topol [lindex $sp $count] dummy

    set filename reach_command
    set file [open $filename w]

    echo finger_count $finger_count
    if {$finger_count == 0} {
        puts $file "bafiogp $finger_timesteps"
    }
    for {set i 1} {$i < [minus $count 1]} {incr i} {
        if {$i == $finger_count} {
            puts $file "bafiogp $finger_timesteps"
        }
        puts $file "move_arm_to_goal [list [lindex $sp $i]] ee 1 1 \{"
    }
    if {$i == $finger_count} {
        puts $file "bafiogp $finger_timesteps"
    }
    puts $file "gpp 2"
    puts $file "reach_object dummy ee $hand_angle 0 0 \{"
    puts $file "gpp 0.4"
    puts $file "reach_object glas ee $hand_angle 0 0 \{"
    puts $file "$end_event"

    for {set i 1} {$i <= $count} {incr i} {
        puts $file "\}"
    }
    close $file
    return "file $filename written"
}

```

# Appendix C

## User Manual

### 1 Running the SkillBuilder

Start up *3d*. Invoke the SkillBuilder by typing

```
source SkillBuilder
```

at the `3d>` prompt. A graphics window with the actor standing behind a high table with a glass on top will appear as well as the **3d Menu**, the **Command Loop** menu, and the **Views** menu. The actors coordinate system is located in the centroid of the pelvis. The x-axis points to the left side of the actor, the y-axis to the back, and the z-axis points upward.

Besides the SkillBuilder commands you can use any *3d* build-in command a list of which can be found in a technical report from the Computer Graphics and Animation Group about “The *3d* Virtual Environment and Dynamic Simulation System”, written by Chen and Zeltzer in August 1992. The *3d* build-in commands most often used in a SkillBuilder session will also be explained in this manual.

User interaction is possible through one of the menus or, as long as the command loop is not running, through typed input at the `3d>` prompt. While the command loop is running (see section 6.1 of this manual), commands can still be typed in the **Do Command Once** text field of the **Command Loop** menu.

### 2 Changing the View

#### 2.1 Using the Mouse inside the Graphics Window

In the **3d Menu**, click the **LookAt Cursor** toggle button right below the **Mouse Handlers** headline to enable the view changing by mouse click in the graphics win-

dow.

Depending on which mouse button you press, the following actions will take place:

1. **Right mouse button:** Zooming in and out.  
The higher you click in the graphics window above the centerline, the farther you will zoom in with every click. Accordingly, you will zoom out by clicking below the centerline. The lower you click inside the window, the more you will zoom out.
2. **Middle mouse button:** Toggling the coordinate frame on and off and changing the current object.  
The object the mouse pointer points at is made the current object. With every mouse click, the coordinate frame is posted and unposted alternately at the centroid of the current object.
3. **Left mouse button:** Changing the view.  
If you click above the center, you will look at the scene from more above. In contrast, clicking below the centerline will let you look from farther below. In the same way, you will “walk around” the current object to the right side by clicking in the right half of the window, or to the left side by clicking in the left half.

## 2.2 Using the Views Menu

The text field at the bottom of the **Views** menu contains the name of a file in which views are saved, or are to be saved. The views file name can be changed by typing another name in that text field.

Clicking the **Read** button causes the views saved in that file to be read in, i.e. made available to be called off. Their names will be displayed in as many fields next to the numbered buttons as there were views saved in the file. To display one of the views, the corresponding toggle button has to be clicked. The view in question is now the current view. Clicking the **Display** button displays the chosen view.

To record a new view, the current view has to be changed (you may use the mouse or the keyboard). Once the new view to be saved is found, one of the toggle buttons next to an empty text field has to be chosen and the **Record** button has to be clicked. The word “recorded” will appear in the text field and can be changed to name the view. To save the new view for use in the next SkillBuilder session, click the **Write** button so the current views file saves the new information.

## 3 Changing the Virtual World

### 3.1 The Virtual World after Starting the SkillBuilder

The virtual world of the actor contains, besides the actor itself, a few objects in the highest directory. These are: grid, table, glass, ball1, and ball2. The grid is the floor the actor stands on. The table is placed right in front of the actor and the glass is positioned on the table. The balls are not posted, i.e. displayed, but can be posted upon request.

You can get a list of available objects using the command

```
lo
```

### 3.2 Instancing Objects

To instance an object use the *3d* build-in command

```
hio filename object
```

For some objects convenience routines have been provided that include (besides instancing the object) coloring, scaling, positioning, and hardening of the object. Procedures of this kind are:

```
makeglas  
makecup  
makephone  
makeroom  
makegrid  
makeceiling  
maketable  
makesoccerballs
```

### 3.3 Defining the Current Object

Clicking on the **Current Object** button in the **3d Menu** will cause a pull down menu listing all currently existing objects to appear. If one of those objects is chosen, it will appear in the text field right below the **Current Object** button. The current object can also be set by clicking on the object in question with the middle mouse button, or by typing the name of the object in the text field.

### 3.4 Displaying Objects

Every object can be posted, i.e. displayed, or unposted upon request by one of the *3d* build-in commands

```
post object
unpost object
```

The next time the image is rendered, the specified object will appear on or disappear from the screen.

Objects can also be easily posted or unposted by choosing the option **Post** or **Unpost** in the pull down menu which appears when clicking on **Object** in the upper left corner of the **3d Menu**. The object that will be posted or unposted is the current object.

### 3.5 Finding the Location of an Object

An object's location can be requested by the *3d* build-in command

```
centroid object
```

### 3.6 Moving an Object

Objects can be move by the *3d* build-in commands:

```
to x y z object
ro axis angle object
```

or by using the **Object Edit** menu that pops up after clicking **Object** in the upper lefthand corner of the **3d Menu**.

In the **Object Edit** menu toggle buttons are provided to choose the action to be taken by the current object (whose name appears below the **Current Object** button in the **3d Menu**). The current object can be translated, rotated, scaled, or changed in color. For the first three actions, the arrow buttons in the menu correspond to the x, y, and z axis in this order, whereas when changing the color, those buttons control the r, g, and b value in the rgb code. The + and the - buttons change all three values at the same time by the same amount. Choosing "local" in the **Transform** field makes the centroid of the object the reference frame. A different reference frame can be chosen by typing its three coordinates in this text field instead of "local".

To position an object on the upper horizontal plane of another object, the procedure

```
position object plane nr_steps xyplane x y
```

can be called. *object* denotes the name of the object to be positioned, whereas *plane* denotes the object, the former one is to be positioned on (e.g. “table”). The parameter *nr\_step* determines in how many timesteps the animation (of the object moving from its current position towards the new one) should be performed. If “specified” is chosen for the *xyplane* parameter the values of the parameters *x* and *y* will be taken into account. The object will then be positioned *x* inches in x direction and *y* inches in y direction away from the centroid of the *plane* object at the maximum z value of the bounding box of *plane*. For any other value of the *xyplane* parameter, the object will be placed at the closest point on the *plane* from where it is currently located.

## 4 Getting Information about the Figure’s State

### 4.1 Checking the Current Joint Angles

The current joint angles of the 9 joints of the left arm can be retrieved one by one by the following commands:

```
get_shoulder_fb_angle
get_shoulder_ld_angle
get_arm_ls_angle
get_arm_lf_angle
get_arm_turn_angle
get_elbow_angle
get_hand_turn_angle
get_hand_bend_angle
get_hand_twist_angle
```

or all at once (in the order of the above procedures) by

```
get_all_arm_angles
```

The current joint angles of the finger joints can be retrieved by the following commands

```
get_finger_spread_angle finger
get_finger_bending_angle finger joint
get_thumb_down_angle
get_thumb_turn_angle
```

where the parameter *finger* can be specified as “thumb”, “index”, “middlefinger”, “ringfinger”, or “littlefinger”. The *joint* should be set to either “1”, “2”, or “3”. All

finger angles can be retrieved at once by

```
get_all_finger_angles
```

## 4.2 Checking the Current Joint Positions

Joint positions for the left arm can be inquired by the commands

```
get_shoulder_position  
get_elbow_position  
get_wrist_position
```

The end effector position of the left arm which is located on the palm vector will be returned by

```
get_ee
```

## 4.3 Checking the Current Orientation of Body Parts

To check the orientation of the arm links, one of the following commands has to be used:

```
get_upper_arm_direction  
get_forearm_direction  
get_hand_direction option
```

The *option* for the `get_hand_direction` procedure can be set to “palm”, “point”, or “thumb” specifying the axis of the hand that is in question.

## 5 Local Motor Programs

Local motor programs are procedures that cause a rotation about one of the joint axes of the figure. They are not implemented as skills, so they should be invoked outside of the command loop by typing the appropriate line at the 3d> prompt. For the arm there are

```
move_shoulder option angle timesteps renopt  
lift/drop_shoulder option angle timesteps renopt  
lift_arm_sideward option angle timesteps renopt  
lift_arm_forward option angle timesteps renopt  
turn_arm option angle timesteps renopt  
bend_elbow option angle timesteps renopt
```

*turn\_lower\_arm option angle timesteps renopt*  
*bend\_hand option angle timesteps renopt*  
*twist\_hand option angle timesteps renopt*

Local motor programs to control the finger and thumb joints are

*spread\_finger finger option angle timesteps renopt*  
*bend\_a\_finger\_joint finger joint option angle timesteps renopt*  
*move\_thumb\_outwards option angle timesteps renopt*  
*turn\_thumb option angle timesteps renopt*  
*move\_thumb\_down option angle timesteps renopt*  
*bend\_thumb\_joint joint option angle timesteps renopt*

The parameters to a local motor program are the *option*, which specifies whether the appropriate joint should be moved “by” the specified angle or “to” the specified joint position, the joint *angle* (be it the absolute or relative one), the number of *timesteps* in which the motion should be performed, and the *renopt* that can be set to “ren” or “noren” depending on whether the image shall be updated on the screen, i.e. rendered, with every timestep or not.

The *spread\_finger* and *bend\_a\_finger\_joint* procedures have an additional parameter, *finger*, to specify the finger in question, including the thumb. The *joint* parameter can take “1” or “2”. The third bending joint of a finger will automatically be bent with the second one by 2/3 of the specified angle.

## 6 Invoking Skills

### 6.1 The Command Loop

The execution of a skill can be invoked by typing the skill’s name in the **Do Command Once** text field of the **Command Loop** menu if the command loop is already running. Otherwise, the skill’s name can be typed at the **3d>** prompt and the command loop can be started thereafter by clicking the **Go** button in the **Command Loop** menu. All skills will add the necessary procedures for their execution to the command list. The **Command Loop** menu shows the current command list and has also buttons to stop the command loop and to step through the list. Access procedures that are able to add, insert, or remove commands from the command list are

*add\_c command*  
*insert\_c command nr\_next\_command*  
*rem\_c command*



## 6.2 Timing and End Events

Every skill has a parameter *end\_event* that is set to “{}” as a default. The end event specifies what should happen when the execution of the skill is completed. For example

```

wave_hand {
    global org
    move_arm_to_config $org 0 0 {}
}

```

will link the two skills (*wave\_hand* and *move\_arm\_to\_config*) sequentially which causes the arm to return into its original position after the hand waving skill stops. In order to keep the last state of a skill in the command list, *end\_event* has to be set to “return”.

Inverse kinematics skills all contain two parameters to control the velocity of the skill: *time\_opt* and *timesteps*. If the *time\_opt* parameter is set to “1”, the target number of timesteps is set to the number specified by the *timesteps* parameter. If the *time\_opt* parameter is set to “0”, the number of timesteps for the execution of the skill is automatically calculated and the *timesteps* parameter has no effect. The timesteps calculation for the inverse kinematics skills is based on the global parameters *dist\_per\_timestep*. The configuration skills (see section 6.3), as well as the skills to orient a body part (see section 6.4), rely on the global parameter *angle\_per\_timestep* which can be set by the procedure

```
aps angle
```

If no *angle* is specified, the procedure returns the current value of *angle\_per\_timestep* (in degrees). Similarly, the *dist\_per\_timestep* can be changed using the procedure

```
dps distance
```

and the value of the current *dist\_per\_timestep* (in inches) can be retrieved by calling the *dps* procedure without an argument.

## 6.3 Configuration Skills

Bringing a limb into a specific configuration can be achieved by one of the the commands

```

move_arm_to_config config time_opt timesteps end_event
bring_finger_in_config finger config timesteps end_event
bring_thumb_in_config config timesteps end_event

```

The `move_arm_to_config` skill expects a 9 dimensional vector as input for its `config` parameter specifying the following target angles: shoulder-fb angle, shoulder-ld angle, arm-ls angle, arm-lf angle, arm-turn angle, elbow angle, hand-turn angle, hand-bend angle, and hand-twist angle.

For the `bring_finger_in_config` skill, the parameter `config` has to be a three dimensional vector specifying target angles for the following angles of the `finger`: spread angle, bend1 angle, and bend2 angle.

The configuration `config` for the thumb has to be a 5 dimensional vector specifying the spread angle, down-angle, turn-angle, bend1 angle, and bend2 angle for the thumb.

The following skills will change the hand into a specific posture:

```
bring_all_fingers_in_rest_config timesteps end_event
bring_all_fingers_in_point_config timesteps end_event
bring_all_fingers_in_open_grasp_config timesteps end_event
```

## 6.4 Body Part Orientation Skills

In order to change the orientation of a body part, one of the following skills can be used for the upper arm, the forearm or the hand:

```
direct_upper_arm main_axis perp_axis time_opt timesteps end_event
direct_forearm main_axis time_opt timesteps end_event
direct_hand_palm/point_axes palm_axis point_axis time_opt timesteps end_event
direct_hand_palm_towards object time_opt timesteps end_event
align_hand_thumb_axis_with_object object hand_bend_angle time_opt timesteps
end_event
```

The `main_axis` in the `direct_upper_arm` and the `direct_forearm` skills has to be a 3 dimensional vector specifying a target orientation for the axis of the upper arm or the forearm respectively. It does not necessarily have to be a unit vector. The `perp_axis` parameter in the `direct_upper_arm` skill is usually set to “no”, meaning that no specific perpendicular axis has to be reached. With perpendicular axis is an axis denoted that is perpendicular to the main axis and forms together with the main axis the plane in which the forearm is located.

The `direct_hand_palm/point_axes` skill has an option to specify both a target palm axis and a target point axis, each consisting of three dimensional vectors. In case both axes are given, all three hand angles will be influenced. In case there is no point axis specified (by setting the `point_axis` parameter to “no”), the current twist angle of the hand will remain the same, since it does not influence the palm axis.

For orienting the palm axis towards an object the skill `direct_hand_palm_towards` is provided. It is very similar to the previous one when no point axis is specified: no hand twisting will take place. This skill can e.g. be used in order to “look at a picture” if the picture is placed in the hand parallel to the palm, the palm is oriented towards the head and the head/eyes are oriented towards the picture.

Finally, the skill `align_hand_thumb_axis_with_object` aligns the thumb axis with the longitudinal axis of an object by changing the hand-twist and hand-turn angles. The hand-bend angle does not play a role when aligning the thumb axis and can therefore freely be chosen.

## 6.5 Reaching and Grasping Skills

There are the following reaching skills:

```

move_arm_to_goal goal_point gp_opt time_opt timesteps end_event
move_arm_to_object object gp_opt time_opt timesteps end_event
reach_object object gp_opt hand_bend_angle time_opt timesteps end_event
reach_object_along_path object hand_bend_angle end_event

```

Positioning of the end effector at a specified *goal\_point* takes place when calling the procedure `move_arm_to_goal` with *gp\_opt* set to “ee”. If the wrist is to be positioned at the goal point, the parameter *gp\_opt* should be set to “wrist”. The same applies for the skill `move_arm_to_object`, but in this case, the actor will reach for the centroid of the *object*.

The skill `reach_object` incorporates an alignment of the hand along the longitudinal axis of the given *object*. The final *hand\_bend\_angle* can freely be chosen and is set to “-10”.

In none of the skills, `move_arm_to_goal`, `move_arm_to_object`, or `reach_object`, will possible collisions be avoided.

A collision free path is generated by the `reach_object_along_path` skill and the end effector of the actor is guided along that path. The skill includes forming an open grasp position with the hand (shortly before the object is reached) and wrapping the fingers around the object, once it is reached.

The grasping skill can also be invoked separately by the skill

```

grasp_object object speed end_event

```

where the *speed* should be a factor between 0.1 and 5. A fist can be made by

```

make_fist speed end_event

```

## 6.6 Oscillating Skills

An oscillating skill is

```
wave_hand end_event
```

The hand waving skill can be stopped by typing `stop_waving` in the **Do Command Once** textfield.

## 6.7 Head and Eye Control Skills

There are three skills to control the head and eye motions:

```
head_track object end_event  
head_straight time_opt timesteps end_event  
head_move object time_opt timesteps end_event
```

The `head_move` skill will cause the head and eyes to move the focus towards the specified *object* in the given number of *timesteps* and evaluate its *end\_event* upon completion of the skill. The skill will be able to adapt to a moving object.

Keeping the focus of the head and eyes on a moving object can be done by invoking the `head_track` skill. It is useful to define `head_track` as end event of `head_move`, in order to slowly move the focus towards an object before tracking it. `head_track` is a continuous skill which means it will only stop executing when it is explicitly stopped by calling the procedure `stop_head_track`. Remember that while the command loop is running, interactions are only possible through the menus. This means `stop_head_track` has to be typed in the **Do Command Once** text field.

The skill `head_straight` brings the actor's focus back to a point directly in front of him at the level of the eyes much like the `head_move` skill brings the focus towards an object.

## 7 Exiting the SkillBuilder

Use `exit` instead of `quit` when exiting the program. Otherwise, the manipulators for the head controller and/or the data glove might keep running.