Generating Motion Sequences for AIBO ERS-7

Kim Kemilä

Examensarbete för 20 p, Institutionen för datavetenskap,
Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 20 credit points,
Department of Computer Science,
Faculty of Science, Lund University

Generating Motion Sequences for AIBO ERS-7

Abstract

This thesis provides a useful open source system for generating motion sequences for the AIBO ERS-7 robot. For this purpose a virtual ERS-7 robot and a corresponding virtual environment have been implemented. All motions are simulated by means of a physics engine, and several useful features have been implemented, in order to allow development of motion sequences as well as exporting of them to the real ERS-7 robot. Finally, good correspondence between the generated motion sequences and the exported motion sequences is achieved. This work, as a motion sequence generator for the ERS-7 robot, provides a useful system for educational and research purposes.

Generering av Rörelsesekvenser för AIBO ERS-7

Sammanfattning

Denna rapport berör uppgiften att förmedla ett användbart öppen källa-system för att generera rörelsesekvenser för AIBO ERS-7-roboten. För detta syfte har en virtuell ERS-7-robot och en motsvarande virtuell miljö implementerats. Alla rörelser simuleras med hjälp av en fysikmotor, och flera användbara funktioner har implementerats för att tillåta utveckling av rörelsesekvenser såväl som exporterande av dem till den riktiga ERS-7-roboten. Slutligen har god korrespondens mellan de genererade rörelseskvenserna och de exporterade rörelsesekvenserna uppnåts. Detta arbete, som en rörelsesekvensgenerator för ERS-7-roboten, förmedlar ett användbart system för utbildnings- och forskningssyfte.

Contents

Preface

First and foremost, I would like to thank my supervisor Jacek Malec for valuable guidance throughout the thesis. I would also like to thank my friend Björn Samuelsson for his understanding and suggestions for how to improve simulation accuracy and to tackle some design issues.

Kim Kemilä
Lund, January 19, 2006

Chapter 1

Introduction

This work provides a useful open source system for generating motion sequences for the AIBO ERS-7 robot. The only work that we are aware of that generates motion sequences for the ERS-7 robot is Webots [1]. Webots is a commercial simulator that provides a rapid prototyping environment for modeling, programming and simulating mobile robots. Webots also enables transfer of simulated motion sequences to several commercially available real mobile robots (among others the ERS-7 robot). A good description of Webots is avaliable at Michel's Webots: Professional Mobile Robot Simulation [2].

There are three suitable approaches, each with its benefits and disadvantages, to create such motion sequences. The simplest (but least attractive) approach is guessing all positions regarding the motions that are to be generated, and then hoping that everything works out well. Despite its simplicity, this approach would most certainly not provide many accurate results nor any kind of feedback from the motion sequences that are generated.

Another approach is to calculate all motions and the resulting interactions with the robot surroundings (the world). This approach offers great freedom in designing any kind of movement, but requires profound knowledge about details of the physical objects involved and their interactions, both kinematic and dynamic, and would therefore be prohibitively complex.

The third approach is to simulate all the motions by means of a physics engine, which will take care for all the calculations concerning physics and mathematics. In this way, we only have to define a virtual ERS-7 robot and a corresponding virtual environment. The accuracy of the simulations depend heavily on our choice of physics engine and how well the virtual robot and the virtual environment are defined. This approach is most suitable for our needs and will therefore be used in this thesis.

In order to accomplish the goal of this project, a faithfully modeled virtual ERS-7 robot and a corresponding virtual environment need to be avaliable. There is also a need for following operations:

- Recording motion sequences and navigating the virtual robot.
- Displaying the virtual robot, the virtual environment and the generated motion sequences.
- Storing the motion sequences for continued future work.

- Allowing the characteristics of the virtual robot and the virtual environment to be (re)defined.
- Exporting the generated motion sequences to the actual ERS-7 robot.

Finally, this project demands that a good correspondence between the generated motion sequences and the exported motion sequences is achieved.

The rest of the thesis is organized as follows. Chapter 2 introduces the AIBO ERS-7 robot and the OPEN-R software system that is essential for this thesis. Chapter 3 describes the Open Dynamics Engine: the core engine that simulates all physical behavior of the ERS-7 robot. Chapter 4 provides a brief overview of the Open Graphics Library and how it can be used in order to visually display the simulations. Chapter 5 presents a solution (the simulator) that fulfills all the criteria presented above. Chapter 6 evaluates the solution presented in Chapter 5. Finally, Chapter 7 summarizes the entire thesis and outlines some possible further improvements.

Chapter 2

The AIBO ERS-7 Robot

In this chapter the AIBO ERS-7 robot is introduced, starting with an overview of the robot followed by a brief description of its hardware. Then an explanation of the software system of ERS-7 is provided. This chapter is concluded by a few comments on other programming environments for AIBO robots.

2.1 Overview

The ERS-7 robot is today the most widely used robot of Sony's AIBO robot series. Other AIBO robots are: ERS-210, ERS-220, ERS-311 and ERS-312, and new robots are very likely to be expected in the near future. The ERS-7 robot is, however, more complex and more powerful than the other robots and is today widely used among robotics and artificial intelligence researchers for various research projects. The Figures 2.1 and 2.2 show two pictures of the ERS-7 robot, the first taken from the front and the second from the back[1].

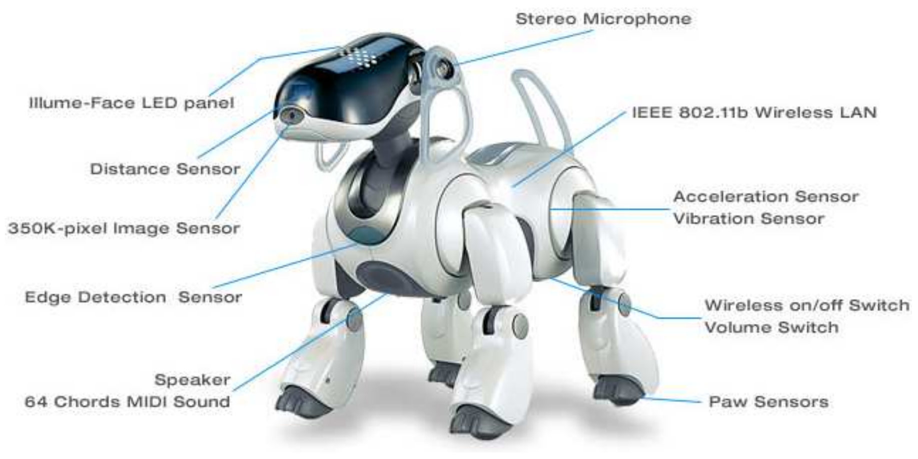


Figure 2.1: Front side of the ERS-7 robot.

[1]These pictures are taken from www.sonystyle.com.

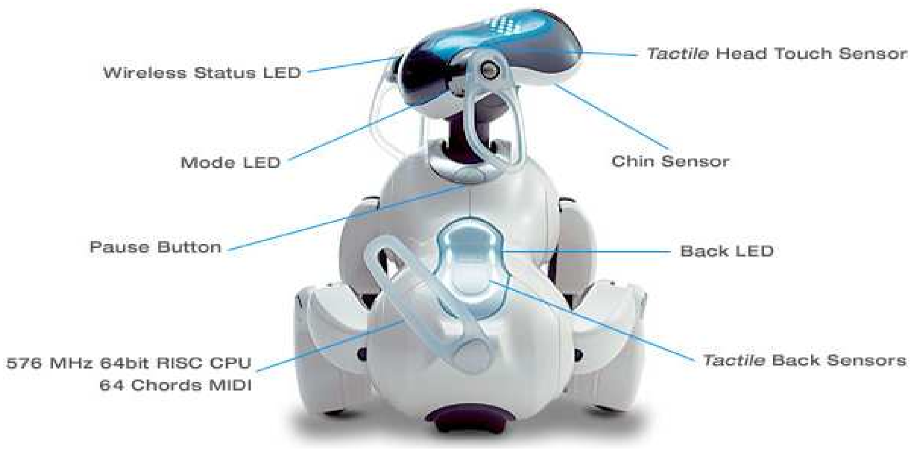


Figure 2.2: Back side of the ERS-7 robot.

2.2 Hardware and Peripherals

The hardware of the ERS-7 robot can be divided into three categories. Most of the information presented here is shown in the Figures 2.1 and 2.2. However, details that are left out can be found in the hardware documentation from Sony [3].

Computer System. The ERS-7 robot uses a 64-bit MIPS processor that operates at 576 MHZ. It is equipped with 64 MB RAM and a wireless LAN module, allowing it to communicate with an external computer network.

Hardware. As it is shown in Figures 2.1 and 2.2, the ERS-7 robot has four legs, each consisting of one upper part and one lower part. Paws are connected to the lowest end of each leg and a tail is connected to the back end of the torso. A head and a neck are connected to the front of the torso. Two ears are connected to the sides of the head and jaws are connected to the lower end of the head. The entire structure is joined together with joints. Adding all joints together, the ERS-7 robot has twenty degrees of freedom and is therefore very flexible. Every motion of a movable part is determined by its hardware limitations and the force applied on the joint. Complex motions are accomplished by creating concurrent motions of several joints simultaneously.

Devices. The ERS-7 robot supports several kinds of input and output devices. The input devices are: color camera, stereo microphones, head touch sensor, back touch sensor, chin touch sensor, paw touch sensor, distance sensors, acceleration sensor and vibration sensor. The output devices are: light emitting diodes (LEDs) and speakers. Common to them all is that they try to resemble real-life sensors of animals and human beings and that they provide a rich variety of information about the environment and about the robot itself.

2.3 The Software System of ERS-7

Software on ERS-7 robots is running under control of Aperios. Aperios is an object-oriented, distributed real-time operating system. It is very small and especially designed for small integrated systems with limited resources, like the AIBO robot series. The operating system functions as a network of objects as its only constituents. The objects, which usually define their own properties and semantics, are monitored and reconfigured on the fly as the demands changes.

The OPEN-R software system provides an interface for the entire AIBO robot series. An OPEN-R program consists of a set of objects that are executed concurrently on the robot. Objects are single-tasked, e.g. a left legged kick or a nodding with the head, and their lifetime spans throughout the program. They can easily be replaced by other objects and may have multiple entries depending on the complexity of the program. They can communicate with each other through inter-object communication, i.e. through predefined communication channels. However, all interaction between objects must be specified in advance using a description file. OPEN-R also supports hardware modularization. For instance, it is possible to do auto-detection of the robot hardware configuration, and to adapt software modules to hardware configuration. Other characteristics of OPEN-R is the very useful wireless LAN and TCP/IP network protocol support.

OPEN-R SDK (Software Development Kit) is a cross development environment based on GNU Tools and is free of charge. It discloses the OPEN-R interface and allows the programmer to develop software that works on the ERS-7 robot and several other robots from the AIBO robot series. OPEN-R SDK allows one to utilize following functions; make joints move, get information from sensors, get images from camera, use wireless LAN (TCP/IP), etc. OPEN-R SDK has been used in the project described in this thesis. For further details about the OPEN-R software system, please consult Serra and Baillie's Aibo programming using OPEN-R SDK [4] and OPEN-R SDK: Programmer's Guide [5].

2.4 Other Programming Environments

Apart from using OPEN-R to program the ERS-7 robot, other higher level programming environments exist. The description of the programming environments presented here originates from Téllez's Introduction to the Aibo programming environment [9].

The **Tekkotsu** framework is a huge system build on top of OPEN-R. It is well-documented and allows easy access and control of any of the sensors and actuators. It also allows powerful processing tools combined with programs like Matlab for image processing.

The **URBI** scripting language is a language that offers remote control through wireless LAN. A C++ library is also provided in order to allow control inside C++ programs.

The **Pyro** programming environment is an environment build on top of Tekkotsu and allows control by using the Phyton programming language.

The **Yart/RCodePlus** programming environment is an environment that is built on top of RCode. It is easy to use and program but functionality is limited by its easiness. It does contain some interesting tools, like wireless consoles, remote control of AIBO, etc.

Chapter 3

The Open Dynamics Engine

In this chapter some of the basic concepts of the Open Dynamics Engine (ODE) are reviewed. First an overview of ODE is provided, followed by an explanation of the build-in collision detection, joint types and the simulation loop that is used. At the end, a few comments on the reliability of an ODE simulation and on other competing physics engines are given.

3.1 Overview

The Open Dynamics Engine (ODE) is a free quality library for simulating articulated rigid body structures. Examples of articulated structures are legged creatures, ground vehicles and moving objects, i.e. rigid bodies of various shapes that are connected together by joints of various kinds.

ODE is designed to be used in interactive or real-time simulation, where changes to the simulation can be done on the fly with complete freedom. The integrator (the core engine) that ODE uses is very stable, robust and fast. This means that the simulation errors of a simulated system hardly ever grow out of control. Note that ODE emphasizes stability and speed over physical accuracy. However, good physical accuracy can be achieved by reducing the simulation errors of the integrator (the speed factor).

ODE allows several parameters of its internal approximations to be tuned. Some of the internal approximations are: different friction models, extended support for rigid bodies, many kinds of joints (see section 3.3) and joint parameters, different integration functions and several global parameters such as gravity, simulation error, etc. More information about the internal approximations can be found in ODE User's Guide [6].

Another important and useful feature of ODE is the built-in collision detection. Collision detection is essential for a simulation system in order to find out whether two objects are potentially colliding with each other.

3.2 Collision Detection

In order to make collision detection possible, all dynamic bodies must be given geometric shapes. Often the shapes are approximated with simple geometric primitives, such as spheres, cylinders and boxes, but ODE allows any shape to be defined. Once a dynamic body has a geometric shape associated with it, it is called a geometric object or a collision object, and it becomes a part of the collision system. All collisions that take place between geometric objects or between geometric objects and the static environment are handled in ODE as follows:

1. The collision detection functions are called to determine what is touching what, and a list of contact points is returned. Each contact point specifies some properties about the collision, e.g. position in space, penetration depth, etc.

2. Each contact point creates a contact joint with additional information about the contact, e.g. the friction present at the contact surface, how soft or bouncy the contact is, etc.

3. All contact joints are put into a container called joint group. This allows them to be added to and removed from the system quickly.

4. A simulation step is taken with all contact joints affecting the simulation.

5. The joint group is emptied.

In order to reduce the workload of the collision system, built-in collision detection of ODE expands the algorithm given above to also include collision culling. Collision culling is the process of quickly excluding non potentially-colliding objects. This is done by encapsulating all geometric objects with bounding boxes. The bounding boxes, that are usually much simpler in structure, allow the system to quickly determine whether two bounding boxes are intersecting. A picture of a geometric object and associated bounding box can be found in Figure 3.1[1].

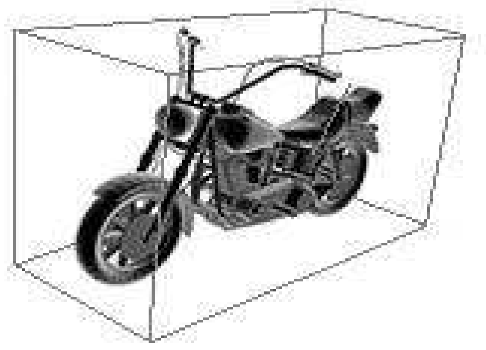


Figure 3.1: Bounding Box.

Another concept utilized by ODE to make collision detection go faster is the use of spaces. A space is a geometric object that contains other geometric objects or other spaces (creating a space hierarchy). The advantage of using spaces is that complex geometric objects, e.g. legged creatures or cars, can be

[1]This picture originates from www.ensight.com.

divided into several space hierarchies, and the collision detection function is recursively called for the space hierarchies. If the space hierarchies are properly defined, the workload of the collision system can be heavily reduced.

Note that the built-in collision detection functions do not have to be used as ODE allows other collision detection libraries to be used instead. A tutorial of how to define own collision detection functions can be found in ODE User's Guide [6].

3.3 Joint Types

ODE supports several kinds of joints, but only hinge joints and universal joints are used in this thesis and need further explanation.

An ODE hinge joint has an anchor defining its position and one axis describing its movement. A universal joint consists of two hinge joints that are put perpendiculars to each other. It has an anchor defining its position and two axes describing its movement. Figures 3.2 and 3.3 show a hinge joint and an universal joint, respectively[2].

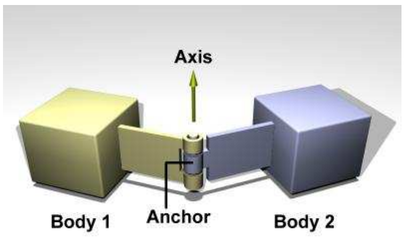


Figure 3.2: A hinge joint.

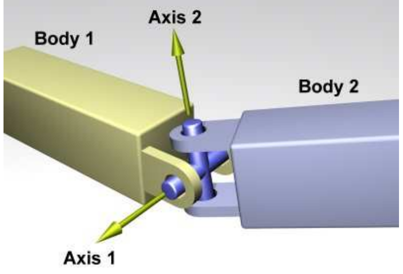


Figure 3.3: A universal joint.

Common to all joints is that they provide several parameters that can be tuned. This feature is very useful because it offers great freedom when defining the characteristics of a joint. More information about these two joints and other joints that are supported by ODE can be found in ODE User's Guide [6].

3.4 Simulation Loop

A typical simulation loop for an ODE application proceeds as follows:

1. Create a virtual world (a virtual environment where physics applies).
2. Create bodies in the world.
3. Set the states (position, mass, etc.) of each body.
4. Create joints and attach them to the bodies.
5. Set the parameters (sponginess, springiness etc) of each joint.

[2]These pictures originate from [6].

6. Create a collision world (collision system).
7. Create collision objects for each body.
8. Create a joint group.
9. Loop:
   (a) Apply forces to all bodies.
   (b) Adjust joint parameters.
   (c) Call collision detection.
   (d) Put all contact joints into the joint group.
   (e) Take a simulation step.
   (f) Clear the joint group.
10. Terminate the application.

ODE permits several virtual worlds to exist and to be simulated simultaneously, and everything that defines a world and a collision world can be read, changed or altered on the fly during simulation.

3.5 Reliability

In order to achieve good correspondence between the simulations and the real world, all physical data (body weights, mass distribution, etc) must be well approximated. But this alone is not enough to guarantee accurate simulations.

The internal approximations also matter. In fact, too coarse approximations might lead to very inaccurate simulations. ODE does offer a very simple (but not very good) solution to this problem. Each integration step advances the simulation by a given time factor. The size of the errors of the internal approximations is controlled by the step size of the integrator. A small step size will result in small simulation errors, but will also make the simulation run much slower. Even if accuracy is gained by reducing the step size, the problem with internal approximations still exists. In fact, replacing the current internal approximations with exact models is very challenging and complex. However, the internal approximations that ODE uses are fairly accurate and as long as the step size of the integrator is kept very small and the simulated system is properly defined, ODE will simulate accurately.

Another issue that has to be taken care of is related to collision detection. The geometric objects that model the dynamic bodies should have the same shape as the bodies. If this is not met, the simulated bodies will bounce off either before or after they actually collide with each other. The step size of the integrator also plays an important role in collision detection. Two moving objects might, for instance, pass each other undetectedly if the step size is too large.

Despite the inaccuracies described in this section, the overall performance of ODE is very good. More information about the inaccuracies and the internal approximations that are being made by ODE can be found in ODE User's Guide [6].

3.6 Other Physics Engines

According to ODE homepage [7] and EuclideanSpace homepage [8], there is a number of good physics engines available today. This section does not attempt to describe nor compare, in any way, the different physics engines against each other, instead only a short list of well-known physics engines and a link to their homepage is given for reference. It is quite natural to divide the physics engines according to their commercial and non-commercial status, as it is done below:

Some commercial physics engine software:

- Havok (www.havok.com)
- Novodex (www.novodex.com)
- MathEngine Karma (www.mathengine.com)
- CM-Labs Vortex (www.cm-labs.com)

Some non-commercial physics engine software:

- DynaMechs (dynamechs.sourceforge.net)
- Multibody Dynamics Simulation (www.concurrent-dynamics.com)
- Aero (www.aero-simulation.de)
- Dynamo (home.iae.nl/users/starcat/dynamo)

Chapter 4

The Open Graphics Library

A brief introduction to the Open Graphics Library (OpenGL) and the extension called OpenGL Utility Toolkit (GLUT) is presented.

4.1 Overview

The Open Graphics Library (OpenGL) is the most widely adopted 3D and 2D graphics library in the industry. It is window-system and operating-system independent, and has been integrated with Microsoft Windows and the X Window System under Unix. The OpenGL interface allows developers to create high-performance, visually compelling graphics software applications. Some of the supported features are: modeling of graphical structures, color, lighting, flat and smooth shading, texture mapping, tessellation, fog, shadows, bitmaps, fonts, images, and much more. For the X Window System there is also a common extension that allows an OpenGL client on one vendor's platform to run across a network to another vendor's OpenGL server (so called network-transparency).

The rendering process (rendering pipeline) of OpenGL requires consecutive execution of the following four tasks:

1. **Modeling** usually resulting in a set of vertices, specifying the geometric objects that are supported by the system.

2. **Geometric processing** determining which geometric objects appear on the display and assigning shades or colors to these objects.

3. **Rasterization** using the remaining vertices and generating a set of pixel values that are to be drawn.

4. **Display** sending the image (containing all pixel values) to the output device. It is here the actual displaying takes place, once all pixels have their colors determined.

Figure 4.1 shows how these tasks are organized in a pipeline manner. A more precise description of the rendering pipeline and all its constituents can be found in Angel's Interactive Computer Graphics [10].

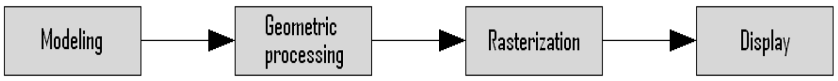


Figure 4.1: Rendering pipeline.

4.2 The OpenGL Utility Toolkit

One common extension to OpenGL is the OpenGL Utility Toolkit (GLUT) library. It is a free, cross-platform library that primarily performs system-level Input/Output with the host operating system. The main functions are: window definition, window control, creating pop-up windows and monitoring of keyboard and mouse input. It does also support routines for drawing some commonly used geometric primitives such as: cubes, spheres, etc.

All window manipulation in the software described in this thesis has been done using GLUT.

Chapter 5

The Simulator

This chapter presents a solution (the simulator) that meets all the criteria listed in the introduction. First an overview of the solution is given, followed by an in-depth explanation of each criterion that has to be met. An evaluation of the solution presented in this chapter is given in Chapter 6.

5.1 Putting Things Together

The simulator has been developed with modularity and generality in mind. To achieve this, the simulator is divided into four separate units and only limited interaction between the units is allowed. In this way, as long as the interactions between the units are kept persistent, each unit can be exchanged or replaced by another unit without damaging the overall functionality of the simulator. The four units are:

System. This unit serves as an entry point for the simulator. It makes sure that all other units are properly initialized.

ODE. This unit represents the physics engine (ODE) that is used. It holds all information about the virtual robot, the virtual environment and the generated motion sequences. This information can be accessed or altered by other units.

Visualization. This unit represents an OpenGL window for visualizing the virtual robot, the virtual environment and the generated motion sequences. This is made possible by letting this unit access all necessary information needed from the ODE unit. This unit also has a mouse listener associated with it, which provides good facilities for viewing the virtual robot and the generated motion sequences from any possible viewing angle.

UserRequests. This unit represents an OpenGL Window for displaying all user requests made to the simulator. Examples of such requests are: record of motion sequences or alter simulation settings. All requests made by the user require interaction with the ODE unit, and are supplied exclusively to this unit via the keyboard device.

The control of the simulator resides in the System unit. From here all other units are properly initialized. The core unit for all simulations is ODE. It holds all relevant information about the virtual robot, the virtual environment and the generated motion sequences, and is vital for the Visualization unit and the UserRequests unit. The System unit treats the Visualization and the UserRequests as separate threads, with listeners for mouse-interaction and keyboard-interaction respectively. In this way, instructions that concern both these units can be treated independently from one another. For instance, the user may navigate the camera with a mouse device and, at the same time, record a motion sequence by using a keyboard device. The overall structure of the simulator is shown in Figure 5.1.

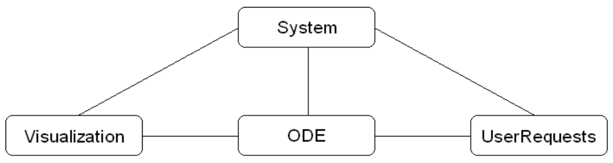


Figure 5.1: The simulator.

Until now we have only scratched the surface of the simulation system and presented a brief overview of the general structure. The sections that follow elaborate further on how the criteria that are set on the simulator are resolved and implemented.

5.2 Virtual Robot

The virtual robot is composed of eleven different body parts that are connected together with either hinge joints or universal joints. All body parts are approximated by geometric primitives. The geometric primitives in question are spheres and capped cylinders. A capped cylinder is a cylinder with half spheres at both ends. Working with geometric primitives has many advantages. One of them is that almost all of them have direct counterparts in OpenGL. This makes both creation and visualization of each body part much easier.

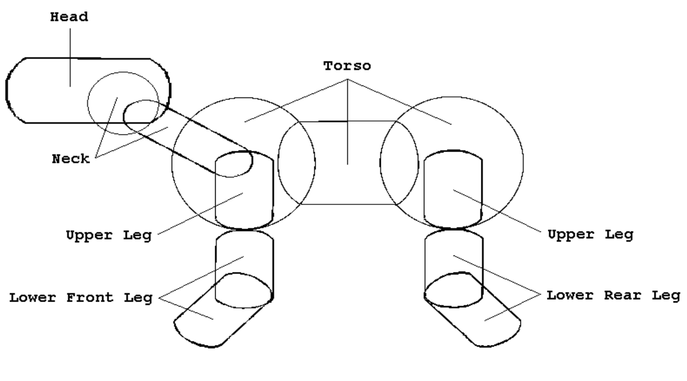


Figure 5.2: The virtual robot.

Figure 5.2 shows the composition of each body part and how they are put together to form the entire robot structure. The figure mustn't be taken literally. It only reveals the coarse structure of the virtual robot and how it may be visualized by OpenGL. The only body parts missing from the real robot are the ears, the tail and the jaws. They are excluded because of their low influence on the simulation. All external measurements and hardware limitations of the joint motions are taken directly from the hardware documentation of Sony [3]. However, since all body parts are approximated with geometric primitives, all of the external measurements had to be recalculated.

As already mentioned, the virtual robot is connected together with hinge joints and universal joints. Hinge joints are used for connecting the lower legs with the upper legs and the lower part of the neck with the torso. Universal joints are used for connecting the upper legs with the torso and the upper part of the neck with the head. In this way, the lower legs and the neck have one degree of freedom, while the upper legs and the head have two degrees of freedom each.

Another issue that must be taken care of concerns the weights of each body part and the strength of the joint motors. At the present moment, there is no information available about them and we must either guess or measure them somehow. In this thesis, all the values are roughly estimated. For instance, the weights of each body part, excluding the torso, are estimated to 100 grams. The torso is estimated to 600 grams. In this way, the total weight is 1600 grams, which is close to the real weight of the robot. Note, however, that in order to guarantee good simulation accuracy, all the weights and all the joint motors must be carefully measured before usage.

5.3 Virtual Environment

In order to create a realistic virtual environment, it is essential that obstacles can be created (and removed) from the environment and that the environment interacts well with the virtual robot. In this thesis, we have chosen to limit ourselves to only allow creation and removal of simple boxes. It is, however, an easy task to expand the simulator to also allow other geometric primitives to exist, such as spheres and capped cylinders.

The boxes that are created may be of any size, any weight and be positioned anywhere inside the virtual environment. This freedom does have its price. If the user accidently positions a box so that it intersects with something else, then they will violently repel each other and that part of the simulation might misbehave. There is no limit for the number of boxes that can be created. However, a greater number of boxes will slow down the simulation process.

5.4 Generation of Motion Sequences

General Facts About Motion Sequences

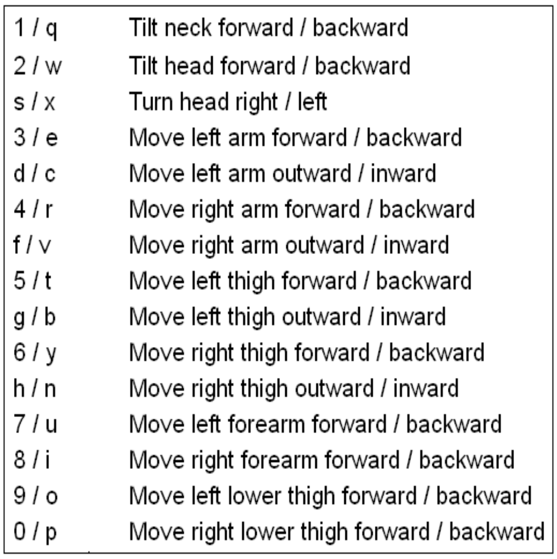
A motion sequence is represented by a discrete time-line, where each time-step holds precise information about the direction of movement and the movement velocity of each body part. In reality, the stored information has very little to do with the actual body parts. Instead, it deals with the joints and the joint velocities, since they are describing the movements of each body part. To talk about joint velocities in a context of moving body parts only complicates things. We shall therefore stay with the definition first presented. This way of representing information suits the recording and the monitoring of motion sequences very well, especially when it is done in a stepwise manner. Note however, that the ODE system often requires tens (sometimes hundreds) of simulation steps in order to reach the desired velocities.

Simulation Accuracy

Simulation accuracy is very important for any motion sequence we wish to generate. The meaning of simulation accuracy is that the motion sequences that we have generated correspond well to the exported motion sequences that run on the real robot. It lies in the nature of all simulation programs that the simulation accuracy can't be precisely calculated. However, decent estimations are easy to accomplish and one way of doing this is by measuring the stability of the virtual robot. By doing so, the user is always updated about any potential imbalances that the virtual robot might get exposed to. In this thesis, the stability check is performed by making an internal simulation of the current state. The internal simulation measures the maximum displacement of the virtual robot while being exposed to a large force in various directions. A more detailed discussion about simulation accuracy follows in Chapter 6.

Recording Motion Sequences

The ability to record motion sequences is vital for this project. We have chosen to control the virtual robot by using a keyboard. Basically, each movement of a body part is controlled by a unique key. A list of supported commands is shown in Figure 5.3.



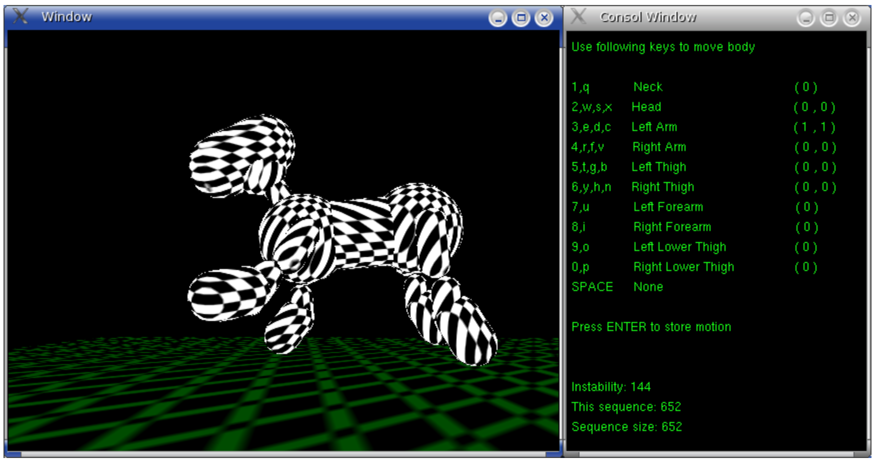
1 / q	Tilt neck forward / backward
2 / w	Tilt head forward / backward
s / x	Turn head right / left
3 / e	Move left arm forward / backward
d / c	Move left arm outward / inward
4 / r	Move right arm forward / backward
f / v	Move right arm outward / inward
5 / t	Move left thigh forward / backward
g / b	Move left thigh outward / inward
6 / y	Move right thigh forward / backward
h / n	Move right thigh outward / inward
7 / u	Move left forearm forward / backward
8 / i	Move right forearm forward / backward
9 / o	Move left lower thigh forward / backward
0 / p	Move right lower thigh forward / backward

Figure 5.3: List of commands.

Whenever a key gets pressed, the following actions are performed:

1. The movement velocity of the selected body part is incremented.
2. Information about the direction of movement and the movement velocity of each body part is saved.
3. A simulation step is taken.
4. The stability of the virtual robot is estimated.
5. The window is re-displayed.

Besides the keys for controlling the body parts, there is a special key that allows the user to continue the simulation without changing any of the velocities of the body parts. This feature might come handy if the user decides to add empty time into their simulations, or to keep recording an already initialized motion without changing any of the velocities. An example of the recording process is shown in Figure 5.4.



```
Use following keys to move body

1,q        Neck                 ( 0 )
2,w,s,x    Head                 ( 0 , 0 )
3,e,d,c    Left Arm             ( 1 , 1 )
4,r,f,v    Right Arm            ( 0 , 0 )
5,t,g,b    Left Thigh           ( 0 , 0 )
6,y,h,n    Right Thigh          ( 0 , 0 )
7,u        Left Forearm         ( 0 )
8,i        Right Forearm        ( 0 )
9,o        Left Lower Thigh     ( 0 )
0,p        Right Lower Thigh    ( 0 )
SPACE      None

Press ENTER to store motion


Instability: 144
This sequence: 652
Sequence size: 652
```

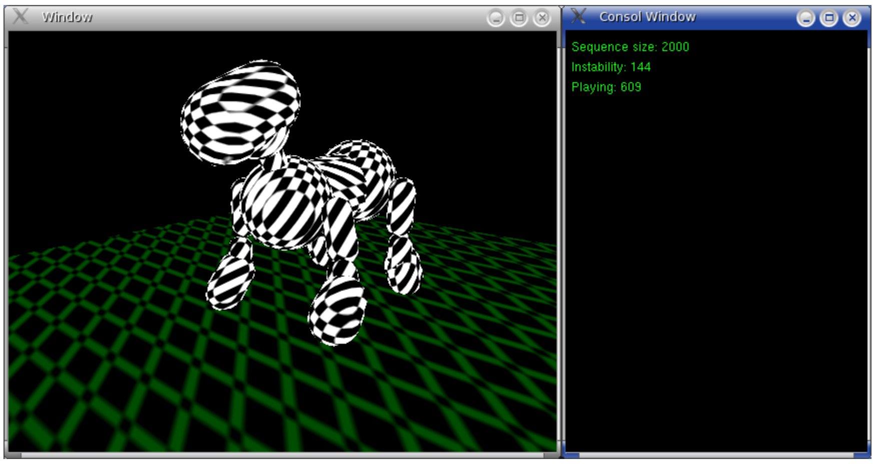
Figure 5.4: Recording of a motion sequence.

However, the algorithm described above has one obvious disadvantage: It doesn't allow the user to initialize two movements at the same time. This might sound inconvenient, but in reality motions only differ with a few milliseconds that are not visible for the human eye. This issue is discussed further in Chapter 6.

Displaying Motion Sequences

Once a motion sequence has been recorded, it is possible to display the whole motion sequence on the screen. This is achieved by first resetting the entire simulation and then by reading a corresponding line from the motion sequence for each simulation step that is taken, and displaying the virtual robot and the current stability of the virtual robot. When a motion sequence reaches the end, the next values to be used in the simulation will be the last values entered.

This feature resembles the special key feature when recording. Less formally, displaying a motion sequence is like recording, but this time without any user interaction. Figure 5.5 shows an example of the displaying process.



```
Sequence size: 2000
Instability: 144
Playing: 609
```

Figure 5.5: Displaying a recorded motion sequence.

This approach has one disadvantage. All motion sequences that are displayed must begin from start. There is no support to only display parts of a motion sequence, nor to display continuous changes made to the virtual robot and the virtual environment. Once the simulation is reset, the last settings made to the virtual robot and to the virtual environment will apply to the entire motion sequence that is displayed. It is not difficult to solve these issues, but it does require much more information to be stored for each time-step. Solving these issues are left for future work.

Saving and Loading Motion Sequences

The simulator supports very simple (but powerful) facilities to save and to load motion sequences. Once a motion sequence is generated and the user desires to save it, all present settings of the virtual robot, the virtual environment and the entire motion sequence are written to a file. In this way, all there is to know about the generated motion sequence is saved and can be re-loaded. Unfortunately, the simulator doesn't perform any checks on the filenames entered to the system, nor on the input data that is read from the files. These issues should be resolved in future.

5.5 Exporting Motion Sequences

In order to export a motion sequence into an ERS-7 robot, proper code must be generated for the OPEN-R software system and then compiled into an OPEN-R object. Once the OPEN-R program is compiled, it must be copied to the memory stick of the ERS-7 robot. Note that the description file, used for describing all inter-object communication between the OPEN-R objects, must be properly defined and also copied to the memory stick.

A comprehensive description of how to compile an OPEN-R program into an OPEN-R object and how to write a proper description file, can be found in Serra and Baillie's Aibo programming using OPEN-R SDK [4] and OPEN-R SDK: Programmer's Guide [5]. Providing all that information here is outside the scope of this thesis.

In order to translate a generated motion sequence into a properly written OPEN-R program, two things must be taken into account. The first is that all motions must be expressed in angles. This means that the simulator must not only store information about the velocities of each body part, it must also store the angles between them. The reason for this inconvenience comes from the facts that the ODE system prefers velocities while the OPEN-R software system prefers angles, and there is no apparent way to express one in terms of the other. The second is that the entire motion sequence must be divided into small packages, where each package holds information (angles) about the movements of each body part for a fixed amount of time. In other words, the entire motion sequence is sliced into small individual data packages that together describe all the movements that the robot will perform. A picture of the exporting process is shown in Figure 5.6.

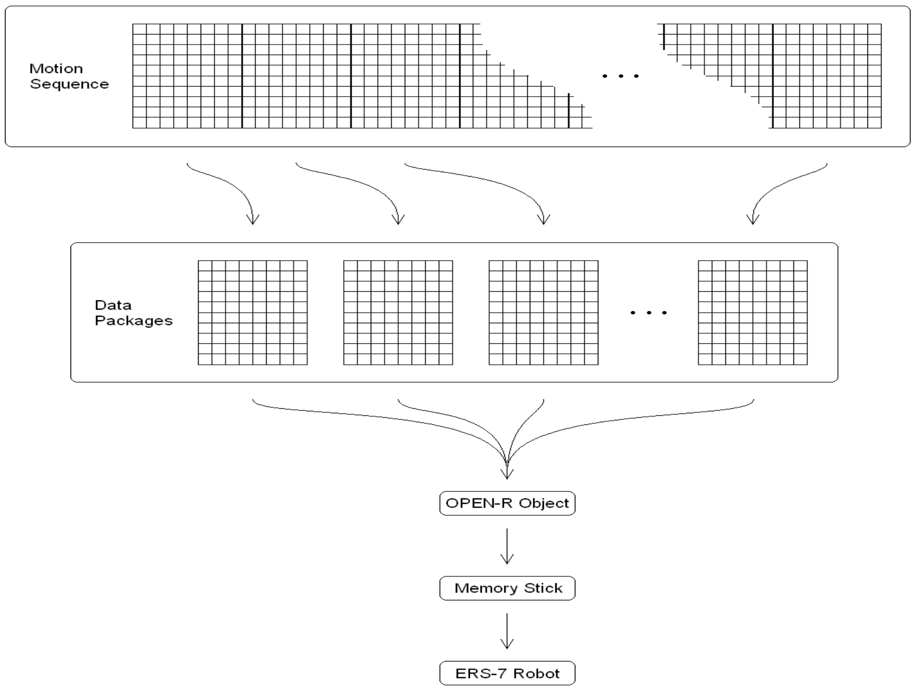


Figure 5.6: The exporting process.

Chapter 6

Evaluation

In this chapter the solution presented in Chapter 5 is evaluated. First the testings of the simulator is explained. Then the problems concerning the representation of the virtual robot are described and some improvements that can be made are suggested. After that, the difficulties that accompany generation of motion sequences are evaluated and some ideas are introduced for solving them. A few comments on the ODE system conclude this chapter.

6.1 Testing

In order to test the simulator, motion sequences of various lengths and various stability were recorded and then exported for comparison. Note that this is the only proper way to test the simulator, since we do not possess enough information about the ODE system to test it in another manner.

The total number of recorded motion sequences is 12, each varying between 5 and 30 seconds. All motion sequences have been divided into the categories: stable and unstable, depending on the results of stability checking. The results of tests are shown in Figure 6.1.

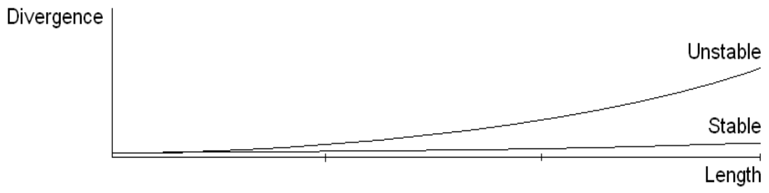


Figure 6.1: The tests of the simulator.

Despite all the coarse approximations made by the simulator, the overall performance is surprisingly good. The stability checker works satisfactorily for motions that are developed under steady conditions, and long motion sequences tend to have greater divergence than shorter ones.

6.2 Virtual Robot

Geometric Properties of The Body Parts

The geometric primitives that are used in order to approximate the shape of the virtual robot, only provide a coarse approximation of the shape. For some applications, a refinement of the shape is needed. A good way of doing this is by approximating the surface with a large collection of polygons. In this way, a larger number of polygons may be used for those parts that need refinement, and a lower number of polygons for those parts that do not. Even if the ODE system and the OpenGL system support triangular polygons, this approach does have a few drawbacks. One is that it is both difficult and time consuming to calculate all polygons and their normals for all body parts. Another drawback is that it decreases the simulation speed.

Weights of Each Body Part

As it has already been mentioned, the weights of each body part need to be measured. If the weights are not properly determined, or if our estimations diverge too much from the real weights, the simulation will suffer from imprecision. Unfortunately, we do not know any easy way of measuring them without taking the robot apart.

Strength of Each Joint

As in the case of the weights of each body part, the strength (motor power) of each joint needs to be measured. If we do not measure them, or if the estimations done in this project diverge too much from the real values, then the simulations will be less accurate. Measuring the strength of each joint is not as difficult as measuring the weight of each body part. It is, however, out of scope of this thesis.

6.3 Generation of Motion Sequences

Recording Motion Sequences

In Chapter 5, a recording algorithm that incorporates pressing keys on a keyboard is described. This approach isn't, even if it works quite well, the best way to do motion recording. There are in fact several problems related to this approach. One of them is a feeling of insufficient control while actually recording. This problem originates from the simple reasons that the keyboard doesn't allow several keys (for recording) to be pressed simultaneously and that there is a delay between each key pressing. Another problem is the fact that two motions can not be initialized at the same time. This problem lies in the nature of the recording algorithm, i.e. due to the stepwise manner each motion is initialized, and cannot be solved unless another recording algorithm is defined.

There is, however, a possible solution to both problems described above, namely usage of a joystick to control movements of each body part. In such case, the recording algorithm must be rewritten to incorporate the joystick. The first change would be to allow continuous discrete-level recording, i.e. the recording process would proceed automatically from one time-step to the next.

The second change would be to collect all joystick originated (body part) movements performed during a short time interval into one time-step of the motion sequence. If these changes are implemented properly, one would then gain much better control of the recording process and there would no longer be any problems with initializing multiple motions at the same time. Note that with the new recording algorithm, no other changes to the simulator would be necessary.

Simulation Accuracy

Perhaps the most important problem in this work is related to the simulation accuracy. As it has already been mentioned in Chapter 5, there are many ways to tackle this problem, and the approach that is used in this work only estimates the stability of the virtual robot by performing an internal static simulation of the current state it is in. The basic idea behind this approach is that high stability results in more accurate simulations, while low stability results in less accurate simulations.

There are, however, a few problems concerning this approach that need to be taken care of. Perhaps the most important problem of them all is that none of the movements initialized during the recording process are taken into account by the stability checker. In other words: the stability checker only cares about the position and the posture of the virtual robot and totally ignores all motions that are performed by the virtual robot. The immediate consequence of this approach is that the stability checker only performs well in situations where the posture of the virtual robot is very steady.

A better approach would be to extend the current approach to also include initialized movements. If the new approach is given enough work space, then it will not suffer from the same problems as the original approach does. Note, however, that an implementation of this approach would be very resource demanding and would slow down the simulation time most considerably.

6.4 ODE System

In this work, ODE physics engine is used in order to simulate all physical behavior. This approach, however, has some potential problems that need to be considered. In Section 3.5, a discussion about the reliability of an ODE simulation and how well it performs in terms of simulation accuracy has been presented.

Continuing this, ODE system requires that all approximations of the virtual robot and the virtual environment are precise. This also means that the step size of the ODE integrator must be set very small and that the shape of each body part must be approximated with great detail.

If any of these criteria is not met, then the simulation accuracy would decrease dramatically. No matter how accurate the information provided to the ODE system is, the generated motion sequences will always diverge a little from the exported ones. The only thing that can be done is to minimize the amount of divergence and to keep the generated motion sequences as short as possible.

Chapter 7

Conclusions and Future Work

In this work, we have examined a possibility to generate motion sequences for the ERS-7 robot. The outcome is a faithful simulation of a virtual ERS-7 robot and a corresponding virtual environment, providing following necessary operations:

- Recording motion sequences and navigating the virtual robot.
- Displaying the virtual robot, the virtual environment and the generated motion sequences.
- Storing the motion sequences for continued future work.
- Allowing the characteristics of the virtual robot and the virtual environment to be defined.
- Exporting the generated motion sequences to the actual ERS-7 robot.

In this work, we have also achieved very good correspondence between the generated motion sequences and the exported motion sequences.

The work may be improved and continued in many directions. First of all, the weights of each body part and the strengths of each joint motor should be carefully measured and added to the simulation. These two improvements alone should increase the accuracy of the simulations most considerably.

For some applications, a refinement of the geometrical representation (the shape) of the virtual robot might be needed. An accurate way of doing this is by approximating the surface of each body part with polygons. This approach might be complex to implement, but usually yields good results.

An interesting and useful improvement would be to rewrite the entire recording algorithm to incorporate joystick navigation of the virtual robot. If this is done properly, then there is reason to believe that the new way of recording will be smoother and more intuitive than the current one.

Another interesting extension would be to incorporate Genetic Algorithms for auto-generating motion sequences. This basically means that, given two positions, the software would automatically generate the motion sequence that brings the robot from the first position to the other.

Bibliography

[1] Web site of Webots
http://www.cyberbotics.com
(Verified January 19, 2006)

[2] Olivier Michel:
WebotsTM: Professional Mobile Robot Simulation
Cyberbotics Ltd, 2004.

[3] OPEN-R SDK: Model Information for ERS-7,
Sony Corporation, 2004.

[4] Francois Serra and Jean-Christophe Baillie:
Aibo programming using OPEN-R SDK Tutorial,
ENSTA, 2003.

[5] OPEN-R SDK: Programmer's Guide,
Sony Corporation, 2004.

[6] Russell Smith:
Open Dynamics Engine: v0.5 User Guide, 2004.
http://www.ode.org/ode-latest-userguide.pdf
(Verified January 19, 2006)

[7] Web site of Open Dynamics Engine:
http://www.ode.org
(Verified January 19, 2006)

[8] Web site of EuclideanSpace:
http://www.euclideanspace.com
(Verified January 19, 2006)

[9] Ricardo A. Téllez:
Introduction to the Aibo programming environment, 2005.
http://www.ouroboros.org
(Verified January 19, 2006)

[10] Edward Angel:
Interactive Computer Graphics: A Top-Down Approach Using OpenGL
(3rd ed), Addison Wesley, 2003.