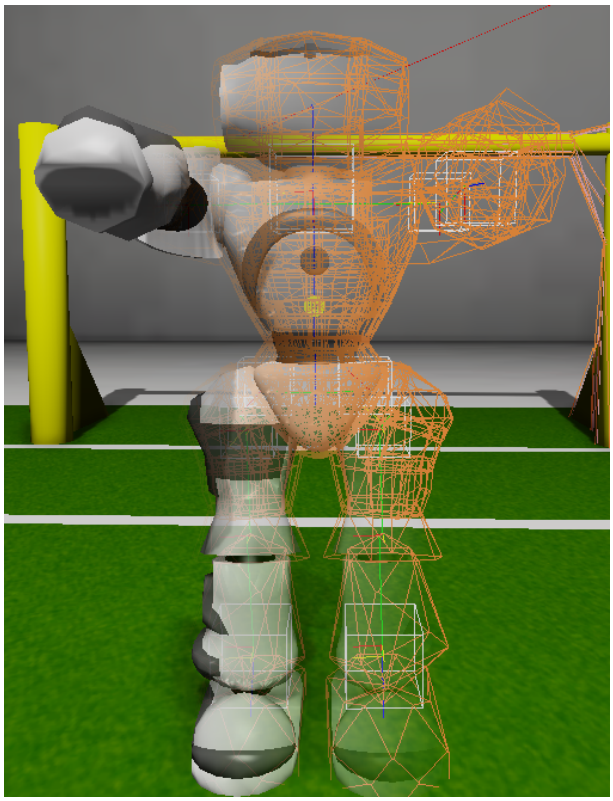
UNIVERSITY OF AMSTERDAM

Validation of the dynamics of a humanoid robot in USARSim

Sander van Noort



Validation of the dynamics of a humanoid robot in USARSim

Sander van Noort
5635667

Master thesis
Credits: 42 EC

Master Artificial Intelligence

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor
dr. Arnoud Visser

Intelligent Systems Laboratory
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

May 8, 2012

Abstract

This thesis describes a model to replicate the dynamics of a walking robot inside USARSim. USARSim is an existing 3D simulator based on the Unreal Engine, which provides facilities for good quality rendering, physics simulation, networking, a highly versatile scripting language, and a powerful visual editor. To model the dynamics of a walking robot the balance of the robot in relation with the contact points of the body with the environment has to be calculated. The extension of the simulator with a walking robot model is validated on the basis of the humanoid robot Nao. On this basis many other bio-inspired robots become possible. A validated simulation allows development and experiments with typical robotic tasks before they are tested on a real robot.

Contents

1 Introduction

Robotic 3D simulators are of growing importance in the development of robots. The purpose of robotic simulators vary from rapid prototyping, parameter optimization and maybe most importantly debugging and analyzing behaviors and algorithms. The usage of a simulator has the advantage that it is easier to reliable collect and analyze results using tools and visualization, which is otherwise impossible or difficult to setup. The downside is the simulator only being an approximation of reality.

One of such robotic 3D simulators is USARSim. USARSim (Unified System for Automation and Robot Simulation)[1] is a robot and environments simulator based on the Unreal Engine[1]. This simulator is designed as a high fidelity simulation of urban search and rescue (USAR) robots and environments intended as a research tool for the study of human-robot interaction (HRI) and multi-robot coordination. USARSim is used as the official simulator for the RoboCup Rescue Simulation league. One purpose of this league is to develop simulators that form the infrastructure of the simulation system and emulate realistic phenomena predominant in disasters. Another purpose is to develop intelligent agents and robots that are given the capabilities of the main actors in a disaster response scenario.

USARSim is also used in the Virtual Manufacturing Automation Competition (VMAC). This simulation-based competition is designed to stimulate research in robotics dealing with problems related to mixed-palletizing and intra-factory package delivery and logistics.

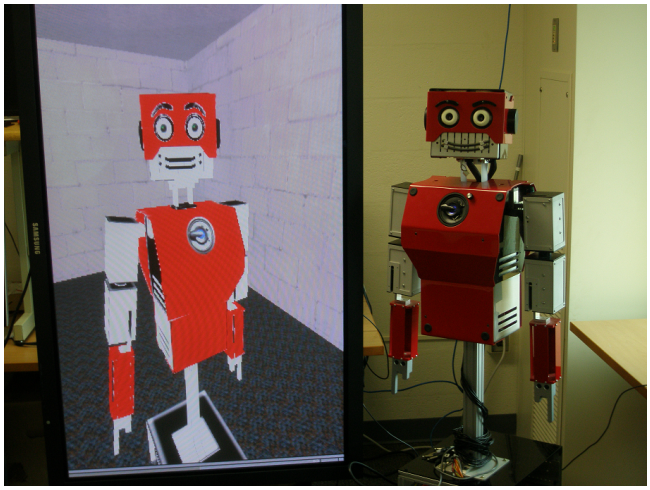


Figure 1: Kramer robot[2]

The first version of USARSim was based on the Unreal Engine 2. This version was created as a *modification* for the game Unreal Tournament 2004 (UT2004). As a consequence the game must be installed to use the simulator. The usage of game engines for research purposes is not too uncommon, for

[1] http://www.unrealengine.com/

example *CaveUT* [3] uses the Unreal Engine as a low cost solution for making projection-based virtual reality affordable and accessible. The UT2004 version supports a wide range of different robots varying from wheeled robots (ski-skeered, ackerman-steered), aerial robots (flying robots) and legged robots (i.e. robots that are able to walk using legs).

In case of simulating robots an important aspect is the underlying physics engine. The physics engine is responsible for the correct simulation of the objects (i.e. robots) in the world and the interaction between the objects and environment. This physics simulation of Unreal Engine 2 is provided by the Karma physics engine.

The next version of USARSim moved to Unreal Engine 3 and was created as a *modification* for the game Unreal Tournament 3 (UT3). This engine uses a different physics engine, NVidia PhysX. Compared to the Karma physics engine, PhysX is more focused on parallelization to make optimal usage of modern cpu's and gpu's. This USARSim version supports a less wide range of robots and is mainly limited to wheeled robots.

The current version of USARSim moved to the Unreal Development Kit (UDK). UDK uses a newer version of Unreal Engine 3 and is monthly updated. One advantage of this version is that the simulator is no longer a modification for a game, which simplifies the requirements for running the simulation. The active update cycle of UDK ensures it stays up to date with the most recent developments and results in a wide range of supported systems.

The intention of this thesis is to add support for legged robots in USARSim on the Unreal Engine 3 and validate the results through experiments. This thesis is inspired by the previous work of legged robots for USARSim. First is the work of Zarati for USARSim on Unreal Engine 2 [4]. This work extends USARSim with support for legged robots, specifically Sony's AIBO ERS-7 (four-legged) and QRIO (two-legged) robots. Second is the work of Greggi et al. [5]. They extended USARSim on Unreal Engine 2 with support for the Vstone Robovie-M robot. Due several changes in the Unreal Engine 3 version of USARSim it lacked legged robots and was limited to wheeled robots.

Figure 2: Aldebaran Nao robot

The focus in this thesis is on one particular two-legged robot: the Aldebaran Nao robot (Figure 2). This medium-sized humanoid robot is developed by Aldebaran Robotics, a French company headquartered in Paris.

The Nao is 58cm tall and weights about 5kg. It is autonomous and programmable in various programming languages. It lasts about 60 to 90 minutes on batteries depending on the usage of the robot.

Aldebaran made several different versions and models of the Nao, which differ in the degrees of freedom and hardware. The degrees of freedom (DOF) determine the number of parts a robot is able to move independently.

The H25 model, referred to as the Academics Edition, has as the name indicates 25 DOF. The H21 model, referred to as the RoboCup edition, has 21 DOF. This model differs from the H25 model by having two less DOF in each hand. Figure 3 shows a schematic overview of this model. Finally there are two more models, the T14 and T2. These models are stripped down versions of the Nao and cannot walk. The T14 model consists of the chest, head and arms. The T2 model consists of only a head and chest.

Version 3.X of the Nao runs on OpenNao, an embedded GNU/Linux distribution based on Gentoo[2]. The Nao version 3.X is equipped with a x86 AMD Geode 500MHz CPU, 256MB SDRAM and 2GB flash memory. The Nao version 4.0 is equipped with a Atom Z530 1.6GHz CPU, 1GB RAM, 2GB flash memory and 4 to 8 GB flash memory dedicated to user purposes.

The Nao robot is equipped with a wide range of sensors. The Nao can either be connected through Ethernet or Wi-Fi (IEE 802.11g). The head contains two cameras, of which one is pointed straight forward and the second one slightly downwards. The cameras support a resolution of up to 640x480px with diagonal field of view of 58 degrees. In version 4.0 this was increased to 960p with a diagonal field of view of 72.6 degrees. The framerate of the cameras depend on the resolution, cpu utilization and whether Wi-Fi or Ethernet is being used. Each feet contains two force sensitive resistors (FSR), which allows the Nao to detect when a feet bumps into an obstacle. The front of the chest is equipped with four sonar sensors (two transmitters and two receivers), which allows the Nao to determine distances with a maximum detection range of $0.25m$ to $2.55m$. The ears, eyes, chest and feet contain 51 leds, allowing the Nao to indicates its status or can be used for other purposes (e.g. displaying emotions [6]). All Nao robot versions also have an inertial measurement unit (IMS), which is used for pose detection and stability.

Since 2007 the Nao is used in the Soccer Standard Platform League. This league is the successor of the Four-Legged League, based on Sony's AIBO dog robots. Due the wide usage of this robot in the RoboCup competition the Nao robot is an interesting choice to simulate.

The Nao is widely used in many research institutes around the globe. Examples of such research vary from localization[7], behavior simulation[8], detecting bumps and touches[9] and making the Nao learn by imitating a human through interactive coaching[10].

In this thesis, the Aldebaran Nao H21 model (version 3.3), RoboCup edition, is modeled and used for various experiments.

[2]http://www.gentoo.org/

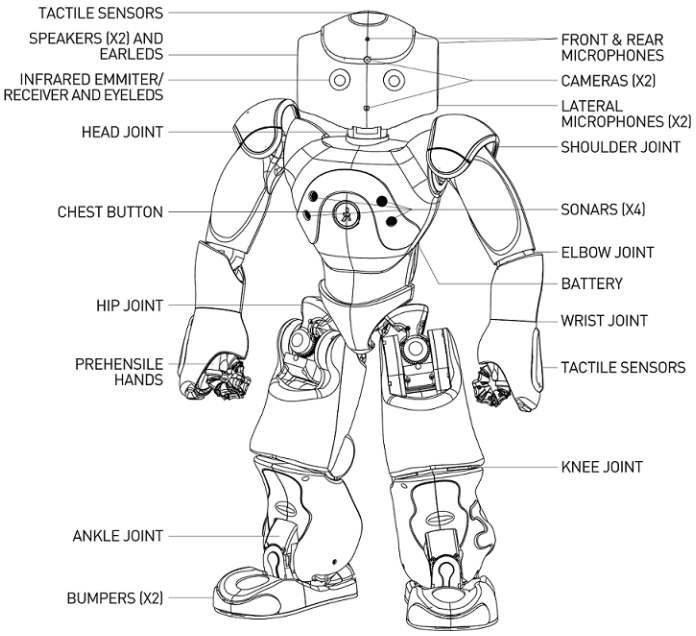


Figure 3: Schematic overview of the Nao (Courtesy of Aldebaran Robotics)

1.1 Research Questions

There are already robotic simulators available which support the type of robots we are describing, including the Nao itself. However, because each simulator is build for a certain purpose, it is never the intention to make a perfect model covering all aspects realistically.

This thesis aims to answer the following research questions:

1. What is lacking in the existing range of available robotic simulators?
2. What aspects of our proposed model is relevant for all robots?
3. For these aspects, how far do they approximate a real robot?

The first question is because there are already several simulators that support the simulation of legged robots. Why would we need another one? USARSim contributes due it's extensive support for creating environments. Section 2 discusses several of these other simulators in more detail.

The second question relates to what extent this model is applicable to other type of robots with movable limbs. This thesis limits the validation to the humanoid Nao robot. However, there are more robots that work in a similar way, for example spider like robots (Figure 4(b)) and industrial arm robots used for manufacturing (Figure 4(a)). Simulated industrial arm robots are very relevant for USARSim, because USARSim is used in the Virtual Manufacturing Automation Competition (VMAC).

The last question is due the continues need for validation research performed in this area. By comparing the real and simulated Nao, specifically focusing on

(a) Frida

(b) Spider like robot

Figure 4:

the walking behavior of the robot, intends to give a better answer on how the behavior of the simulated robot compares to the real robot.

Experiments in a simulator also have the advantage of being easy to reproduce. Inside the simulator it is easier to control the environment conditions, for example lighting. In reality this is not always the case.

The thesis is organized as follows: Section 2 discusses related simulators that support legged robots (in particular the Nao). Section 3 explains the basics of rigid body simulation and covers several parts of the physics engine PhysX and the Unreal Engine. Section 4 discusses the architecture of USARSim and the design of the legged robot. Section 5 includes the validation experiments and results. Section 6 discusses the results and future work. Finally, section 7 concludes this thesis.

2 Related Work

There are many robotic simulators available and it would take too much space to describe them all. For this reason the simulators listed here all provide support for the Nao robot.

The simulators are compared on the following points:

1. How are the robots controlled and what are the limitations?
2. Does it support controlling multiple robots in a single simulation?
3. How easy is it to create custom environments? One of the strong points of USARSim is creating custom environments using Unreal Editor.
4. How modificable is the simulator? Is it easy to make custom modifications to the simulator or add new custom robots?

Additionally there might be other strong points of the simulator worth to mention. The discussed simulators are shown in Figure 5.

2.1 NaoSim

NaoSim is the official 3D simulator supported by Aldebaran. The simulator is based on the (game development) framework Unity[1] and is developed by Cogmation Robotics[2]. NaoSim is closed source and uses Nvidia PhysX as a Physics Engine. Section 3.2 discusses PhysX in more detail.

The Nao is controlled using the NaoQi framework. NaoQi is a Multi-platform framework provided by Aldebaran and allows the user to control the Nao. An example of software using NaoQi is Choregraphe, which allows the user to create and edit movements and interactive behaviors in a simplified way using a user interface. An advantage of using NaoQi is that you can control the simulated and real Nao with the same code. Furthermore, you can manipulate the Nao (move, rotate, etc) and add basic primitives (e.g. cubes, spheres, triangles) to the simulated scene as obstacles.

A downside is that, as of writing, the possibilities to create custom environments are limited and simulation of more than one Nao is not possible. Another potential downside is that the simulator is specifically developed for the Nao robot and as a result does not support other robots and can neither be extended by the user to support new robots.

2.2 Simspark

Simspark[3] is the official simulator in the RoboCup 3D Soccer Simulation League and is primarily made for this purpose [11]. The simulator is open source and freely available. It uses a client-server architecture, where agents (i.e. robot controllers) are clients that communicate with the simulation server. Several robots (including the Nao) are supported and SimSpark makes it easy to add new robots through *rsg* files. These files describe the physical representation of a robot, which includes the 3D representation and collision model.

[1] `http://unity3d.com/`
[2] `http://www.cogmation.com/naosim.html`
[3] `http://simspark.sourceforge.net`

By default, Simspark starts a football simulation, including a soccer field, game states and referee. The starting environment can be changed at startup through *rsg* files. The robots are controlled using a custom protocol and does not provide support for NaoQi.

Noteworthy is the abstraction of the physics layer, which is supposed to make it easy to switch between different physics engines [12]. Currently SimSpark only supports the *Open Dynamics Engine* (ODE).

2.3 Webots

Webots[4] is a commercial closed source robot simulator for educational purposes [13]. This simulator uses the ODE physics engine for the simulation of the dynamics of the robots.

A Webots simulation is composed of a world, one or several controllers and optional physics plugins to modify the regular physics behavior of Webots. A world describes the environment and the properties of the robots. Using the included world editor new environments can be made.

Controllers are programs to control the robots in those worlds. These controllers are started as separate processes and have limited privileges in terms of interacting with the simulation. Multiple robots and controllers can be used at the same time in Webots.

Webots also includes a controller that allows the user to connect with the simulated Nao robot in Webots using the NaoQi framework.

2.4 SimRobot

SimRobot is a free open source general robot simulation and uses ODE as a physics engine[5]. SimRobot consists of several modules linked to a single application, which differs from the commonly used client/server based approach. This approach offers the possibility of halting or stepwise executing the whole simulation without any concurrency.

The specification of the robots and the environment (*simulation scene*) is modeled via an external XML file and loaded at run-time. This XML file uses the specification language *RoSiML* (Robot Simulation Markup Language), which was developed in an effort to create a common interface for robot simulations.

Controllers allow the user to command the robots and implement a sense-think-act cycle. This sense-think-act cycle is executed each step by the core component of the simulation to read the commands for the robot it controls.

SimRobot is actively used (and developed) by the B-Human team (a RoboCup team in the Soccer Standard Platform League) and they provide more information in B-Human Team Report and Code Release 2011 [14].

[4] `http://www.cyberbotics.com/`

[5] `http://www.informatik.uni-bremen.de/simrobot/`

Figure 5: Screenshots of the different simulators in action: NaoSim (top left), Simspark (top right), Webots (bottom left), SimRobot[15] (bottom right)

3 Background

The most important aspect when simulating a legged robot is the robot kinematics. Kinematics describes the motion of (rigid) bodies. Typical robots are characterized by a sequence of joints. Joints can be seen as constraints between two rigid bodies that limit the movement of those two bodies in some way.

This section is organized as follows: Section 3.1 describes the kinematics of rigid body motion in a simulation, including the required equations. Section 3.2 describes the general outline of the physics engine NVidia PhysX, which is used in USARSim. Section 3.3 describes joints more formally and the convention used in this thesis to describe a set of joints of a robot. Section 3.4 describes how to specify the orientation as Swing and Twist and why this parameterization is preferred above other parameterization like the Euler angles. Section 3.5 describes the relevant parts of the Unreal Engine for the simulation, including an outline of the engine and how the engine maps to the physics engine and networking.

3.1 Kinematics

The goal of this section is to provide a description of simulating rigid body motion, including the terms, concepts and equations. The description in this section is largely based on the SIGGRAPH tutorial about *Rigid Body Simulation* by Baraff [16].

Up to and including section 3.1.8 describe the motion of a rigid body excluding collisions with other rigid bodies, the environment and other constraints

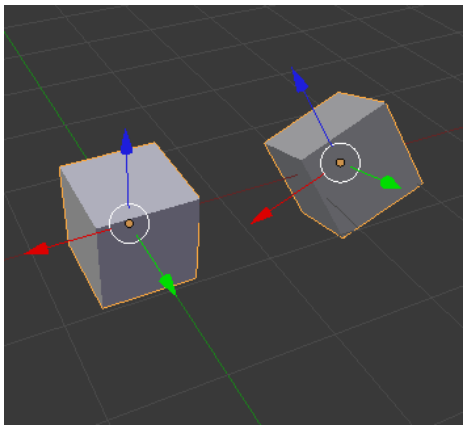


Figure 6: On the left a cube shown in body space. The body space origin of the cube is at the geometric center. On the right the cube is translated and rotated into world space.

imposed between rigid bodies. In section 3.1.8 everything is put together and the rigid body motion is defined by the function $Y(t)$, which computes the state vector of all rigid bodies in the simulation at time t. This state vector contains the spatial information (position and orientation) and momentum of those rigid bodies, which is all the information needed to represent the state of a rigid body. From the spatial and momentum variables other properties like the velocity and forces can be computed.

The transformation matrices in the next sections are notated as homogeneous matrices [17].

3.1.1 Position and Orientation

Rigid bodies (usually) occupy a volume of space in the world. The shape of this volume is defined in an unchangeable space called the *body space*. To know the position and orientation of the rigid body in *world space* (i.e. the environment) the rigid body must be translated and rotated from *body space* to *world space* (as shown in Figure 6). To simplify calculations it will be assumed that the *center of mass* of the rigid body is placed at the origin in body space.

The translation and rotation of the rigid body can be described as a single homogeneous transformation matrix:

$$T(t) = \begin{bmatrix} r_{xx} & r_{yx} & r_{zx} & t_x \\ r_{xy} & r_{yy} & r_{zy} & t_y \\ r_{xz} & r_{yz} & r_{zz} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

An arbitrary point p_0 can then be transformed into world space by:

$$p(t) = T(t)p_0 \tag{2}$$

Where p_0 is a four element position vector.

3.1.2 Linear and Angular Velocity

The next step is to define how the position and orientation of the rigid body changes over time. This change can be decomposed into two components: linear and angular velocity. The linear velocity causes the position of the center of mass to change over time in world space. Now to understand angular velocity assume the position of center of mass in world space does not change over a timestep (i.e. $x(t) = x(t+1)$), but the remaining points on the rigid body do change their position. In this case the points on the rigid body are spinning around one or more axes of the rigid body. This change is described by the angular velocity $\omega(t)$.

The linear and angular velocity can be computed by differentiating $T(t)$, resulting in the single transformation matrix $V(t)$.

$$V(t) = \frac{d}{dt}T(t) = \begin{bmatrix} \omega(t) \times r_{xx} & \omega(t) \times r_{yx} & \omega(t) \times r_{zx} & v_x \\ \omega(t) \times r_{xy} & \omega(t) \times r_{yy} & \omega(t) \times r_{zy} & v_y \\ \omega(t) \times r_{xz} & \omega(t) \times r_{yz} & \omega(t) \times r_{zz} & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

3.1.3 Rigid body composed from particles

To define the remaining equations, first the volume of a rigid body must be defined. This will be done by assuming a rigid body is conceptually made up from a large number of particles, indexed from 1 to N. The mass of the i^{th} particle is described by m_i and the position by r_{0i} in *body space*. The particle in *world space* r_i is then defined by $r_i(t) = T(t)r_{0i}$ (similar to equation 2).

Then the total mass M of a rigid body is defined by:

$$M = \sum_{i=1}^{N} m_i \tag{4}$$

Note that in an actual implementation the rigid body is not composed of particles. For example m_i would rather be represented by a density function based on the geometry shape of the object.

3.1.4 Center of Mass

The center of mass of a rigid body is defined by averaging the position of all its mass. The definition of the volume of a rigid body by a number of particles allows the definition of the center of mass of the rigid body in world space:

$$\frac{\sum m_i r_i(t)}{M} \tag{5}$$

Note that earlier the center of mass was already defined in body space as being the origin in that space. This means that in body space the center of mass of a rigid body is defined by:

$$\frac{\sum m_i r_{0i}(t)}{M} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{6}$$

The center of mass position serves as a convenient reference point for calculations.

3.1.5 Force and Torque

Forces can act on a rigid body and as a result cause a change in position or rotation of the rigid body. The origin of forces will here be defined as external influences (e.g. gravity, wind, contacts, etc). $F_i(t)$ denotes the total force due external influences acting on particle i at time t. The total external force acting on the rigid body then becomes:

$$F(t) = \sum F_i(t) \tag{7}$$

In case of rigid bodies there is also another force acting on it, named torque. External torque $\tau_i(t)$ differs from force $F_i(t)$ in the sense that it depends on the relative position to the center of mass of the rigid body. Intuitively, the direction between the position of the particle $r_i(t)$ and the center of mass $x_i(t)$ (remember the body space origin is placed at the center of mass) in world space can be thought of as being the axis around which the body would spin due force $F_i(t)$. This means $\tau_i(t)$ is defined by:

$$\tau_i(t) = (r_i(t) - x(t)) \times F_i(t) \tag{8}$$

And the total external torque $\tau(t)$ by:

$$\tau(t) = \sum \tau_i(t) = \sum (r_i(t) - x(t)) \times F_i(t) \tag{9}$$

Note that $\tau(t)$ tells something about the distribution of the forces $F_i(t)$ over the rigid body.

3.1.6 Linear and Angular Momentum

Linear momentum is a measure of an object's translational motion and is defined by $p = mv$, in other words the product between the mass and velocity of an object (in this case a particle).

The total linear momentum $P(t)$ of a rigid body is the sum of products between the mass and velocities of each particle i of the rigid body:

$$P(t) = \sum m_i v(t) = (\sum m_i) v(t) = M(t)v(t) \tag{10}$$

Note that the linear momentum is a conserved quantity: it does not change if there are no external forces acting on the rigid body.

Angular momentum measures an object's tendency to continue to spin. The definition of angular momentum is a bit less obvious, however is important because of the same property as linear momentum, namely the quantity is conserved if there are no external forces acting on the rigid body. Imagine an object floating in space without external forces: in this case the linear and angular velocity might be variable, but the angular momentum is constant.

The angular momentum L of a particle in the rigid body is defined as $L = r_{0i} \times m_i v_i$. The total angular momentum acting on the rigid body is then, similar to the definition of linear momentum, defined by the sum of angular momenta of all particles in the rigid body. The angular momentum $L(t)$ of a rigid body can be expressed as:

$$L(t) = I(t)\omega(t) \tag{11}$$

Here $I(t)$ is a 3×3 matrix that represents the inertia tensor (described in the next section) and $\omega(t)$ the angular velocity. Similar to Linear momentum $\tau(t)$ represents the rate of change of Angular momentum:

$$\dot{L}(t) = \tau(t) \tag{12}$$

3.1.7 Inertia Tensors

The inertia Tensor $I(t)$ (also called *Moment of inertia* or *Angular Mass*) describes the mass distribution of the rigid body, in other words the scaling factor between angular momentum $L(t)$ and angular velocity $\omega(t)$.

The inertia matrix in world space can be expressed by:

$$I(t) = \sum m_i((r\text{`}_i^T r\text{`}_i 1 - r\text{`}_i r\text{`})i^T \tag{13}$$

Where $r\text{`}$ is defined by $r\text{`}_i = r_i(t) - x(t)$. In body space the inertia tensor can be computed by:

$$I_{body} = \sum m_i((r_{0i}^T r_{0i} 1 - r_{0i} r)0i^T \tag{14}$$

The definition of I_{body} allows $I(t)$ to be rewritten to a much simpler definition:

$$I(t) = R(t) I_{body} R(t)^T \tag{15}$$

For a full explanation of these derivations see Baraff [16]. Because I_{body} is defined in body space, it can be precomputed before the simulation is started. This way only equation 15 needs to be evaluated while the simulation is running, which is much less expensive than having to evaluate equation 13 each time.

3.1.8 Rigid Body equations of Motion

The final part of rigid body motion, excluding constraints (e.g. contacts, joints), is to put the previous sections together by defining the state vector $Y(t)$:

$$Y(t) = \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} \tag{16}$$

Note the choice of the variables in the state vector. $x(t)$ and $R(t)$ represent the spatial information of the rigid body (translation). The other two variables are linear and angular momentum, $P(t)$ and $L(t)$. The total body mass M and precomputed body space inertia tensor I_{body} are already known. As a result the remaining variables are also known.

The derivative of $Y(t)$, $\frac{d}{dt}Y(t)$, is given by:

$$\frac{d}{dt}Y(t) = \frac{d}{dt}\begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ \omega(t)^* R(t) \\ F(t) \\ \tau(t) \end{bmatrix} \tag{17}$$

From the previous sections the remaining variables can be computed by $v(t) = \frac{P(t)}{M}$, $I(t) = R(t)I_{body}R(t)^T$ and $\omega(t) = I(t)L(t)$. Within this definition $F(t)$ and $\tau(t)$ are external forces and given by a function. Using this information it is possible to compute the state of a rigid body at any time t given the initial state $Y(0)$.

The following pseudo code describes an example simulation algorithm:

Algorithm 1 Example Simulation Algorithm

```
t = 0;
Y = INITIAL STATE;
while simulating do
  Y = Y(t);
  t = t + TIMESTEP;
end while
```

First it initializes the state vector Y to the initial state (free choice). Next the simulation is being executed by advancing t by a discrete time step. The size of the timestep is free choice and influences the precision of the simulation (also see section 5.2.2).

3.1.9 Contacts

The final step in this model of kinematics is to add contacts to the model. In the previous sections the rigid body motion was described by assuming the motion was unconstrained, excluding collisions between rigid bodies and constraints imposed by joints. When the rigid body encounters a contact, it imposes a constraint on the movement. Note that not only contacts impose constraints, but also joints between rigid bodies.

There are two types of contacts: *colliding* and *resting* contacts. A difference is made because in case of resting contacts the simulation does not need to do anything special, in contrast to the second case.

In case of a *resting contact* there is zero velocity change between the two rigid bodies (or one rigid body and the environment). The simulation only needs to compute the force that keeps the objects in contact in that state. For example in case of a *resting contact* between a particle and the floor only the upward force needs to be calculated (to compensate the gravity force).

A *colliding contact* causes a direct change in velocity between two rigid bodies. In this case the simulation of a rigid body using equation 17 must be halted because the collision might occur somewhere between two timesteps. This means that when the collision is detected the normal simulation must be stopped and the velocity change must be computed.

The actual detection of a collision usually consists out of two steps. The first step is to determine which pairs of rigid bodies are potentially colliding. This is usually determined by comparing the bounding boxes of the rigid bodies (described in the next section). The second part of the collision detection involves more complex algorithms. An important theorem in this is the separating axis theorem (SAT)[18]. This theorem says that if two convex rigid bodies are not penetrating, there exists an axis for which the projection of the rigid bodies will not overlap.

For more details see the second part about *Rigid Body Simulation* by Baraff [19].

3.2 PhysX Dynamics

Nvidia PhysX is the underlying physics engine of the Unreal engine and USARSim. The task of a physics engine is to give an approximate simulation of rigid body dynamics (or any other physical related system). Physics engines can roughly be divided in two classes of engines: *high-precision* and *real-time*. PhysX belongs to the later class.

High-precision physics engines focus on delivering the most precise physics simulation at cost of performance. Running these kind of simulations usually takes much more time than it would take in real-time (for real complex simulations more than a month would not be too uncommon). In practice this type of physics engines are used by scientists and computer animated movies.

The focus of real-time physics engines on the other hand is, like the name suggests, delivering physics in real-time. To do so, simplified calculations are used at cost of less precision in the simulation. Typical applications of this type of physics engine are in games, interactive computing and learning tasks.

In PhysX a simulation is executed within a scene. A scene is basically a container for actors, joints and effectors. It allows the user to simulate multiple scenes in parallel without objects between scenes influencing each other.

The simulation of a scene is advanced one time step at a time. Advancing a time step means the properties of the objects in the simulation change (i.e. the position and velocity of the objects). The choices of the timestep settings are important for the stability of the simulation. In general longer timesteps lead to poor stability in the simulation, while shorter timesteps can lead to poor system performance in complex simulations (see section 5.2.2).

The motion of a rigid body can either be constraint by contacts (with the static world or other rigid bodies) or joints. The *Constraint Solver* of PhysX limits the motion of rigid bodies (and satisfies the constraints) by reiterating all constraints on a rigid body a number of times. Basically, the higher the number, the more accurate the results become. The number of iterations used by the solver determines the number of rigid bodies that can be jointed together. PhysX calls this the *solver iteration count*.

The following important aspects of PhysX are highlighted: actors, materials, joints and collision detection.

Actors Actors define objects that are capable of interacting with the world and other objects. In PhysX actors can have two roles: static objects (fixed in the world reference frame), or dynamic rigid objects. These actors (objects) can have a shape assigned, which is used for collision detection. Static objects (like the environment) always have a shape assigned, since they are only used for collision detection. Rigid objects on the other hand do not always need to have a shape. In this case they represent an abstract point mass (can serve as connections between joints) and the properties of the rigid body must be assigned manually.

An object is represented by an inertia tensor and by a point of mass located at the center of mass. The inertia tensor describes the rigid bodies mass distri-

bution. In other words it describes how hard it is to rotate the rigid body in different directions (equation 15).

Materials Materials describe the surface properties of actors. These properties are used when two actors collide. Examples of such properties include friction and restitution. Friction determines how much resistance to movement the object will have when touching or sliding along another surface. Restitution is the amount of *bounce* an object will have, which represents the amount of force that will be returned to an object on collision. The result of a collision will influence the simulation and result in the actors bouncing, sliding, etc.

Joints Joints connect two rigid bodies and limit the movement between those two rigid bodies. The way the movement is limited is determined by the type of joint. The *Constraint Solver* enforces the limited motion between the two connected rigid bodies. PhysX supports a large number of different joints including Revolute, Prismatic and 6 Degrees of Freedom Joint (D6 joint). The D6 joint can again be configured to any of the earlier joints. In section 3.3 the concept of joints is described in more detail.

Collision detection The first step in collision detection is to determine which pairs of objects (i.e. rigid bodies) could potentially be colliding. This stage is usually called the *Broad Phase*. In case of PhysX this is the *Sweep and Prune* algorithm. This algorithm detects potentially colliding pairs by comparing the bounding boxes of rigid bodies. A bounding box is a box of minimum size that encloses an object. The *starts* (lower bound) and *ends* (upper bound) of the bounding boxes are sorted along a number of arbitrary axes. When a rigid body moves the bounding box may overlap with another bounding box of a different rigid body (determined by comparing the *starts* and *ends* of the bounding boxes). If the *starts* and *ends* of two of such bounding boxes overlap in all axes it means a pair of potentially colliding rigid bodies is found. This algorithm is also known as *sort and sweep*, called by Baraff[20] and later called *Sweep and Prune* by Cohen *et al.*[21].

The *Sweep and Prune* algorithm is based on the assumption of *Temporal Coherence*. This means it assumes that within a simulation timestep a rigid body does not move a very large distance. In the case a rigid body, possible small of size, is moving with a very high velocity this means some collisions might not be detected. One object could move through another object during a single time step.

To solve this problem PhysX has an optional technique called *Continuous Collision Detection*. Instead of testing collision between the rigid bodies bounding boxes at discrete points, it tests an extruded volume based on the motion of the fast moving rigid body. This means the the bounding box is extended in the direction of the linear velocity of the rigid body, making the bounding box much larger.

In the case of simulating large scenes with a huge number of rigid bodies it is not feasible to check all possible pairs. If there are n number of objects and o number of overlapping pairs, this means this algorithm has a complexity of approximately $O(n+o)$[22]. Instead, PhysX divides the world in partitions and only checks pairs that are nearby each other.

Once nearby pairs of shapes are identified the collision detection can move on to the *Near Phase* algorithm. In the Near Phase the exact collisions are computed. Details about the PhysX Near Phase algorithms are not available because they are part of PhysX's intellectual property. These algorithms are most likely based on the Separating Axis Theorem (described in section 3.1.9).

Figure 7 shows an overview of the collision detection process and PhysX.

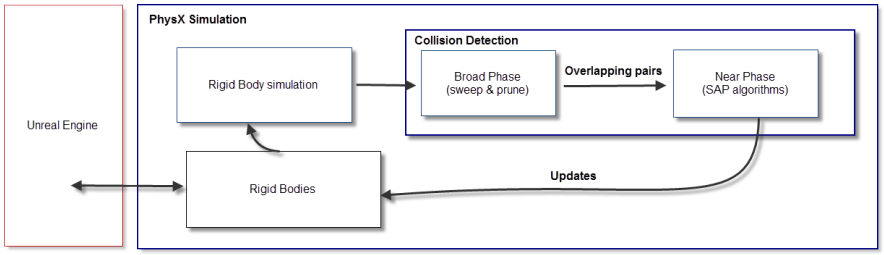


Figure 7: Overview of the collision detection in PhysX

3.3 Joint definition and convention

As said in the previous section, a joint connects two rigid bodies and limits the movement in some way. The type of movement limitation results in different types of joints, like a rotational joint, translational joint (also called prismatic joint), spherical joint, screw joint, etc.

The Nao only uses rotational joints, so this section will not describe in further detail about the other type joints. A rotational joint, also called *revolute joint*, is as the name suggests capable of rotating around an axis. Figure 8 shows a simple example of a rotational joint. This type of joint allows one degree of freedom (DOF) between the two rigid bodies, namely the range of motion around the specified axis. The motion of this type of joint is usually also limited to a specified range around the axis.

The positions of joints and rigid bodies (also called links) are described as a set of frames (coordinate systems). The first frame is usually the reference frame: (x_0, y_0, z_0). This frame is fixed into the world. The first link is fixed to this frame. The next frame (x_1, y_1, z_1) is attached to the next link 2. The same thing can then be applied to the next frames so that in general frame (x_i, y_i, z_i) is on link $i + 1$ and is affected by joint i.

It is important how the relative position and orientation of the frames is characterized. A commonly used convention to describe this is the *Denavit Hartenberg* (DH) notation. This convention uses homogeneous transformation matrices to describe the relative positions of the frames (coordinate systems). This convention is used in USARSim and in section 4.4 it is used to describe the frames of the simulated Nao.

The axes of the DH-Frames are determined based on three rules:

1. The z_{i-1} axis is along the axis of motion of the joint. In other words, this means in case of a rotational joint that it rotates around the z-axis.

2. The x_i axis is normal to the z_{i-1} axis and pointing away from it.

3. The y_i axis follows from x_i and z_i because all three form a right-handed coordinate frame.

Using these three rules the relative position of a frame can be described. To do so, for each frame four parameters are needed:

- θ_{i-1} - the joint angle from x_{i-1} to x_i about the z_{i-1} axis.
- d_i - the distance from the origin of frame $i-1$ to the intersection of z_{i-1} and x_i.
- a_i - the shortest distance from the z_{i-1} to z_i axes.
- α_i - the offset from z_{i-1} to z_i measured as the angle around the x_i axis.

In case of a revolute joint θ represents the degree of freedom.

Using the DH-representation the transformation from frame $i-1$ to frame i can then be described by the following general homogeneous transformation matrix ${}^{i-1}A_i$:

$$[{}^{i-1}A_i] = \begin{bmatrix} \cos\theta_i & -\cos\alpha_i \sin\theta_i & \sin\alpha_i \sin\theta_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\alpha_i \cos\theta_i & -\sin\alpha_i \cos\theta_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A more extensive description can found in the book Robotics, chapter 2.2.10, by K.S Fu *et al.*[23].

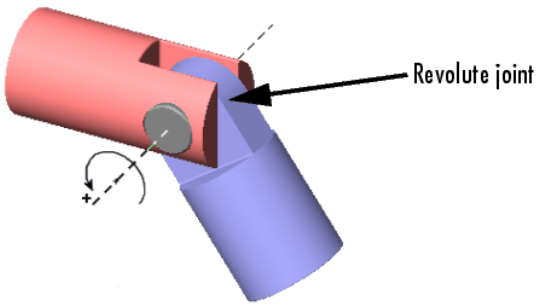


Figure 8: Example of a revolute Joint (Courtesy of Wikimedia Commons).

3.4 Representation of orientation as Swing and Twist

There are many parameterizations to represent an orientation and there is no parameterization that is the best choice for everything. For example, in case of some applications you want the parametrization to be intuitive to use by the end user of that application, while in other applications you want no singularities during computations. In case of a physics engine it must be intuitive to

specify the joint limits, but while running the actual simulation you want no singularities in the orientation parametrization.

Some well known parameterizations are the *Euler Angles*, *Unit Quaternion* and *Axis-Angle*. A less known parameterization is the *Swing and Twist* decomposition of an orientation. This parameterization is used by PhysX to specify the range of motion of a joint. Note that internally PhysX uses quaternions.

The *Swing and Twist* decomposition of an orientation is achieved by doing two rotations consecutively:

1. *twist* - rotation around an axis of choice.
2. *swing* - rotation around an axis in its perpendicular plane.

This results in three required parameters to represent the orientation. The first one is the angle for the twist rotation. The remaining two parameters are two Cartesian coordinates of its rotation vector in the equator plane.

One important requirement when specifying the range of motion is that it is intuitive to use. Unit quaternion and *Axis-Angle* are not considered intuitive. Euler angles on the other hand are intuitive and very well know.

Another important aspect is the number of singularities a representation contains. The unit quaternion representation has no singularities at cost of one more parameter. There are no known representations with less than four variables with no singularities. The *Axis-Angle* and *Swing and Twist* representations both have one singularity. Euler angles possess two singularities. The phenomenon *Gimbal Lock* can occur because of these singularities when two rotational axes of an object point in the same direction, which is discussed in many papers (for example in [24]).

Note that although *Swing and Twist* is similar to *Axis-Angle*, it is not the same. Both describe the direction of an axis followed by an angle of rotation around that axis, however the *Swing and Twist* axis is defined in a different way. In case of *Axis-Angle* the axis is the axis of the single shortest rotation, while the *Swing and Twist* axis is of free choice.

Baerlocher *et al.* [25] explains the rational why swing and twist is a good choice for specifying the range of motion and additional information can be found in the Nvidia PhysX documentation[1].

3.5 Unreal Engine

Unreal Engine is a game engine developed by Epic Games. A strong point of the engine is the tools that allow easy creation of complex environments (see section B.1). In this section *Actors*, *Unreal Collision engine*, *ticking* and *networking* are explained.

In Unreal Engine, *Actors* are objects that are world related. They have a position and orientation and are able to interact with the world and other actors. Actors can have a 3D representation using meshes. There are two types of meshes: *static meshes* and *skeletal meshes*. The first type has no animations as opposed to the second type and is cheaper to render. An animation is a pre-defined motion created for a mesh.

[1] `http://developer.nvidia.com/physx`

The Unreal Collision engine is responsible for the physics in the Unreal Engine. The PhysX physics engine is only one component of the Unreal Collision engine. There are actually various physics modes which allow actors to move around in the world, where PhysX is one of them. Most of the other physics modes involve simplified physics driven by game logic. For example *walking physics* are used by characters in a game and has advantages for gameplay and performance purposes. These modes are of no direct relevance for the actual physics simulation.

These alternative physics modes are implemented by the Unreal Engine and do not use the collision detection system of PhysX when moving an actor. For this reason each actor (with physics) has two collision representations. One collision representations is intended for the Unreal Engine and the other one for the physics engine (PhysX). Additionally there are two types of collision representations for Unreal Engine, resulting in a total of three collision representations.

The first collision representation is intended for *static meshes* in the Unreal Engine. Static meshes are a type of meshes that are not dynamic. This name does not imply they cannot move or interact with the world. There are two options to create this physics model. The first option is to create the (simplified) collision hull in a 3D modeling program, unreal editor or use the automatic generation. This collision hull is then converted into a set of convex hulls. The second option is to collide per polygon against the static mesh 3D model itself and is potentially an expensive operation.

The second collision representation is intended for *skeletal meshes* in the Unreal Engine. Skeletal meshes have as the name implies a skeleton (bones) and are used for animating the mesh. The collision model is built using PhAT (Physics Asset Tool) and allows attaching simple shapes (boxes, spheres) to bones of the skeletal mesh, which move with the animation. This type of meshes is not used in USARSim for the robots.

The last collision representation is intended for PhysX and is created in a similar way as the static mesh version. The PhysX collision model is used in the physics simulation.

In USARSim PhysX is used for the actual physics simulation of the robots; however, sensors will usually involve collision detection with one of the Unreal Engine physics representations. For example, a simulated sonar sensor uses Unreal Engine ray tracing to detect objects in the world, which uses the Unreal Engine collision model. Figure 9 shows the Unreal Collision and PhysX collision model side by side.

Actors are *ticked* once per frame. During such a *tick* they can update their logic. Actors can choose when they want to be ticked: before, during or after *asynchronous work*. Asynchronous work means other tasks are performed in parallel while executing a tick. This includes updating the physics, which means an actor cannot update physics settings when being ticked during asynchronous work.

For better control timers can be used. Timers execute at a fixed interval. Timers are updated during an actors tick. This means the precision of a timer is limited to the number of frames per second.

The task of the *networking* component of Unreal Engine is to keep the world state synchronized between the different users. For networking the Unreal Engine uses an approach called *generalized client-server model*. In case of *generalized client-server model* there is a server that is authoritative over the evolution

Figure 9: The left picture shows the PhysX collision model, the right picture the unreal engine collision model. Note both models are very similar.

of the world state and only the server knows the true state of the world. Clients maintain an accurate local subset of the world state and predict change of the world state by executing the same code as the server. Servers only need to send information about the world state to the client to correct the client world state, which is smaller than when the server would need to send full updates. The problem of approximating the world state between server and client is called *replication* in the Unreal Engine.

This networking model implies the physics simulation runs on both the server and client, where the physics simulation on the server represents the true state of the simulation. The server will send updates about the rigid body states to the client. In the case of the Unreal Engine such a state consists of the position, orientation as quaternion, linear velocity and angular velocity. When the server sends a new rigid body state to a client there are two types of updates. The first one is a small correction and adjusts the position by 20% (i.e. $clientPos = clientPos * 0.8 + serverPos * 0.2$) and the velocity by 80%. The second type is a full correction (or full update), which adjusts the position and velocity 100%.

4 Legged Robots in UsarSim

This section discusses the design of the legged robot in USARSim UDK. It also compares USARSim UDK to the previous versions (UT2004 and UT3) to highlight the changes that were made to USARSim.

Section 4.1 discusses in short the state of the different USARSim versions, based on Unreal Engine 2 and 3. Section 4.2 gives a general overview of the architecture in USARSim. Section 4.3 discusses the overview of changes made to USARSim and how the Legged Robot fits into the design. Section 4.5 discusses a tool that serves as a bridge between NaoQi and USARSim. This allows the user to control the simulated Nao using Aldebaran NaoQi SDK.

4.1 USARSim UT2004, UT3 and UDK

There are several differences between the three different versions of USARSim. Some of these differences are caused by migrating to newer versions of Unreal Engine. The UT2004 version of USARSim used Unreal Engine 2, which includes the Karma Physics engine. UT2004 was actively used and maintained for several years, and contained a wide range of type of robots, varying from ski-steered vehicles to flying robots.

The next version of USARSim moved to Unreal Engine 3. This engine offers much better rendering of lighting and improved tools. It also changed the physics engine to NVidia PhysX. The implementation of robots is centered on the wheeled vehicles and is not suitable for other types of robots like manipulators or bio inspired robots. All robots used a specific PhysX class optimized for wheeled vehicles. The robots would use a skeletal mesh for the robot 3D model definition. The main downsides from this approach is the very simplified collisions of the wheels (i.e. collision of a wheel is represented by a single line), the approach can only be used for wheeled robots and cannot be extended to other type of robots. Due this limitation this version was supported for a relatively short time and contains only a limited number of robots compared to the UT2004 version.

The UDK version of USARSim is based on a continuation of Unreal Engine 3, namely Unreal Engine 3.5. The implementation of robots was rewritten in this version to allow better collisions and other types of robots (as described in section 4.3).

One thing that did remain roughly the same is the general architecture of USARSim that allows creating and controlling the robot (e.g. message protocol), as described in the next section.

4.2 USARSim Robot Architecture

In USARSim a robot is controlled using an external program. This program sends commands to USARSim and is called a controller. The task of the robot controller is to send messages to the robot in USARSim based on the information it receives from the sensors of that robot. This robot controller runs in an external process. To communicate between the robot controller and USARSim *Gamebots* is used [26]. Gamebots is a bridge between USARSim and the robot controller. It opens a TCP/IP socket for communication purposes and uses a custom message protocol for exchanging messages.

Once a connection is established, the controller can either request basic information about the simulation environment (request starting positions, objects, etc) or send the INIT message to create the robot in the simulation environment. Once the robot is initialized in the world, the controller can start sending messages for controlling the robot. When the controller disconnects, the robot is removed from the simulation.

On the USARSim side the simulated robot consists out of multiple Unreal actors. These actors all have their own *Tick* function and basically act like separate threads (i.e. order execution is not guaranteed). The majority of these actors represent the rigid bodies and joints (described in the next section). Another part of these actors represent the sensors. Furthermore, there is an actor that represents the connection with the controller (*communication actor*) and finally there is one actor that serves as the *brain*. The *brain* actor is responsible for creating all robot parts, joints and sensors in the simulation. The *brain* actor timer function updates the current joint angles values and sends status messages to the external controller. The exact content of the status messages depend on the type of robot. In the case of a legged robot the status message contains information like the current joint angles and joint stiffness.

Sensor actors either send periodic status messages from their timer function or send information on request, depending on the type of sensor. In case information about a sensor is requested, the message goes through the *brain* actor, which subsequently requests the information from the sensor.

The simulated robot receives messages from the controller through the *TcpLink* class, which creates a TCP socket. The *TcpLink* class tick function checks for new data during *asynchronous work* work. This class triggers an event when new data is received. The new data is then parsed according to the *Gamebots* protocol and a corresponding action is taken (e.g. set a desired target joint angle).

4.3 USARSim Legged Robot

This thesis describes the introduction of the legged robot to the UDK version of USARSim. The version difference is stated here because the UDK version of USARSim models robots in a different way than the UT3 version.

In the new design, wheeled, legged and aerial robots are modeled using parts and joints. The robot models are defined in Unreal Script, one robot per file. Appendix A shows an example of a robot model. Sensors are added using configuration files, allowing more flexibility when changing sensors. The robot models itself are not defined in the configuration files because the syntax of Unreal Engine configuration files do not provide enough flexibility. A better configuration file format would be preferred in case configuration files would be used to model the robots (e.g. *RoSiML*).

Parts are always modeled by a *PhysicalItem*. These are static meshes with physics and represent a rigid body. These parts are connected by joints, which are modeled by different types. The type of joint depends on the desired the motion.

For legged and wheeled robots the most important joint type is the *RevoluteJoint*. In USARSim the motion of this joint is always around the z-axis, according to the DH convention. Internally the joint maps to the PhysX D6 type, which allows specifying the linear and angular degrees of freedom of the

joint. The angular degrees of freedom are specified using the Swing and Twist notation. Because the RevoluteJoint only needs one degree of freedom, all limits are set to zero except for the twist. The limit of the twist angle is specified using one value, which is assumed to be symmetrical. This introduces a problem because not all joint limits are symmetrically defined. To solve this problem the joint angle limits are normalized to be symmetrical and transformed back when needed. The *RevoluteJoint* is used by assigning a target orientation, which PhysX will try to satisfy.

Furthermore there is the *PrismaticJoint*. The PrismaticJoint allows a linear movement between the two connected rigid bodies. For example this type of joint could be used in the bumpers of a wheeled robot.

The *WheelJoint* is used by, as the name suggests, wheeled vehicles. The setup of this joint is very similar to the RevoluteJoint, except the motion range is never limited and instead of moving the joint to a target orientation it is driven by a target velocity.

A specific issue that arises when simulating a complex legged robot such as the Nao is the presence of *complex joints*. This type of joint constructions occur when multiple joints are located in almost the same location in the robot. An example of such a joint is the Nao shoulder joint with two degrees of freedom. In reality such joints consist of two revolute joints with a small intermediate limb (rigid body).

To overcome this issue a few simplifications are made to the model by disabling contact generation between parts that are located close to each other. This allows the collisions between the parts to be ignored. Additionally the masses and inertia tensors are manually determined to ensure improved behavior.

This solution introduced another problem because Unreal Script only allows disabling contact generation between rigid bodies that are connected through a joint. To overcome this problem a *PhysX proxy DLL* was introduced, which directly uses the PhysX API to modify these physics properties of the rigid bodies. Unreal Script then communicates with PhysX using *DLLBind* [3], which like the name implies allows binding C functions to Unreal Script code. DLLBind only allows calling C functions from Unreal Script.

The resulting robot model is shown in Figure 10. The robot is placed in front of the reference sheet of the real Nao. The Nao reference sheet texture is imported at real scale into Unreal Engine.

4.4 Denavit Hartenberg Chains

Section 3.3 described the general homogeneous transformation matrix for Revolute Joints according to the Denavit Hartenberg representation. Starting from the torso of the Nao, the robot contains five chains of transformations: the head, both arms and both legs. This leads to three unique chains of transformation matrices.

The Denavit Hartenberg representation is visualized in Figure 11. Red lines show the z axes (motion axis) of the joints, while the yellow and green lines show respectively the y and x axes of the joints. The middle blue line shows

[3] http://udn.epicgames.com/Three/DLLBind.html

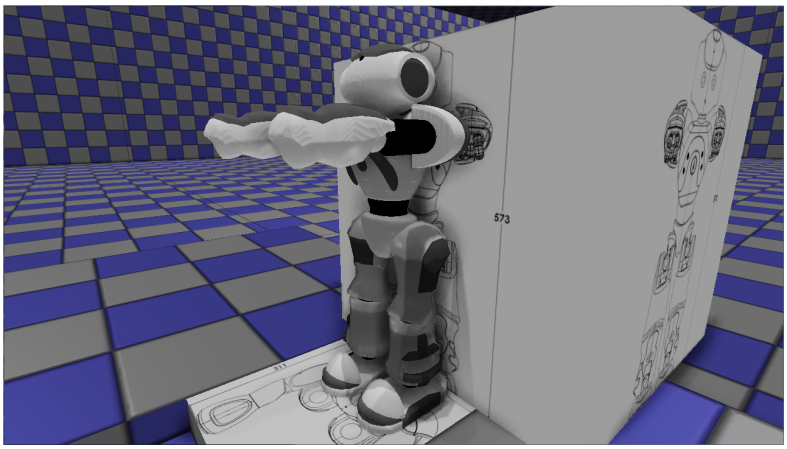


Figure 10: Simulated Nao in reference pose in front of the reference sheet.

the start z axis. The other blue lines represent the end points of the five joint chains. This transformation is represented by a translation matrix.

Recall the homogeneous transformation matrix describing the transformation from frame $i-1$ to frame i uses four parameters: θ_i (joint angle), d_i (distance from the origin of frame $i-1$ to the intersection of z_{i-1} and x_i), a_i (the shortest distance from the z_{i-1} to z_i axes) and α_i (the offset from z_{i-1} to z_i measured as the angle around the x_i axis).

Three of these parameters are known in advance: d_i, a_i and α_i. Filling in these parameters simplifies the transformation matrices. The joint angle θ is denoted using Phoenician symbols to avoid confusing between the different chains.

The head chain matrices are as follows, with $\aleph$ (pronounced as *alpeh*) as the joint angle parameter:

$$HeadYaw = \begin{bmatrix} \cos\aleph_1 & \sin\aleph_1 & 0 & 0.0000\cos\aleph_1 \\ \sin\aleph_1 & -\cos\aleph_1 & 0 & 0.0000\cos\aleph_1 \\ 0 & 0 & -1 & 0.09 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$HeadPitch = \begin{bmatrix} \cos\aleph_2 & 0 & \sin\aleph_2 & 0 \\ \sin\aleph_2 & 0 & -\cos\aleph_2 & 0 \\ 0 & 1 & 0 & 0.00 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The left arm chain matrices are as follows, with $\beth$ (pronounced as *beth*) as the joint angle parameter:

$$LShoulderPitch = \begin{bmatrix} \cos\beth_1 & 0 & \sin\beth_1 & 0.0900\cos\beth_1 \\ \sin\beth_1 & 0 & -\cos\beth_1 & 0.0900\cos\beth_1 \\ 0 & 1 & 0 & 0.08 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

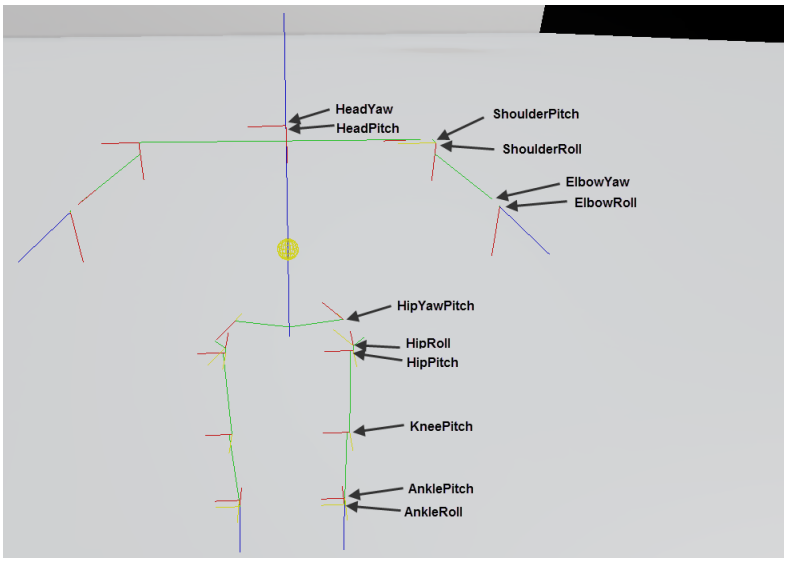


Figure 11: Visualization of joints according to the DenavitHartenberg convention. Red lines show the z axes, yellow the y axes and green the x axes of the joints.

$$LShoulderRoll = \begin{bmatrix} \cos ק_2 & 0 & \sin ק_2 & 0.0100 \cos ק_2 \\ \sin ק_2 & 0 & -\cos ק_2 & 0.0100 \cos ק_2 \\ 0 & 1 & 0 & 0.01 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$LElbowYaw = \begin{bmatrix} \cos ק_3 & 0 & \sin ק_3 & 0.1097 \cos ק_3 \\ \sin ק_3 & 0 & -\cos ק_3 & 0.1097 \cos ק_3 \\ 0 & 1 & 0 & 0.01 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$LElbowRoll = \begin{bmatrix} \cos ק_4 & 0 & \sin ק_4 & 0 \\ \sin ק_4 & 0 & -\cos ק_4 & 0 \\ 0 & 1 & 0 & 0.00 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The left leg chain matrices are as follows, with ת (pronounced as *gimel*) as the joint angle parameter:

$$LHipYawPitch = \begin{bmatrix} \cos ת_1 & -\frac{1}{4}\pi \sin ת_1 & \frac{1}{4}\pi \sin ת_1 & 0.0461 \cos ת_1 \\ \sin ת_1 & \frac{1}{4}\pi \cos ת_1 & -\frac{1}{4}\pi \cos ת_1 & 0.0461 \cos ת_1 \\ 0 & \frac{1}{4}\pi & \frac{1}{4}\pi & 0.07 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$LHipRoll = \begin{bmatrix} \cos ת_2 & 0 & \sin ת_2 & 0.0134 \cos ת_2 \\ \sin ת_2 & 0 & -\cos ת_2 & 0.0134 \cos ת_2 \\ 0 & 1 & 0 & 0.03 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$LHipPitch = \begin{bmatrix} \cos\gimel_3 & 0 & \sin\gimel_3 & 0.0050\cos\gimel_3 \\ \sin\gimel_3 & 0 & -\cos\gimel_3 & 0.0050\cos\gimel_3 \\ 0 & 1 & 0 & 0.00 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$LKneePitch = \begin{bmatrix} \cos\gimel_4 & -\sin\gimel_4 & 0 & 0.0880\cos\gimel_4 \\ \sin\gimel_4 & \cos\gimel_4 & 0 & 0.0880\cos\gimel_4 \\ 0 & 0 & 1 & 0.00 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$LAnklePitch = \begin{bmatrix} \cos\gimel_5 & -\sin\gimel_5 & 0 & 0.1001\cos\gimel_5 \\ \sin\gimel_5 & \cos\gimel_5 & 0 & 0.1001\cos\gimel_5 \\ 0 & 0 & 1 & 0.00 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$LAnkleRoll = \begin{bmatrix} \cos\gimel_6 & 0 & \sin\gimel_6 & 0.0100\cos\gimel_6 \\ \sin\gimel_6 & 0 & -\cos\gimel_6 & 0.0100\cos\gimel_6 \\ 0 & 1 & 0 & 0.00 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These transformation matrices can be used to calculate the position of a limb or joint given the joint angles.

4.5 UsarNaoQi

UsarNaoQi is a program to control the Simulated Nao in USARSim using NaoQi. This makes it possible to control both the real and simulated Nao robot with the same code. NaoQi is a multi-platform framework provided by Aldebaran and allows the user to control the Nao in various programming languages (C++, Python, C#, Urbi, Matlab and Java), although only programs written in C++ and Python can run directly on the OS of the Nao. Other languages can only be used to remotely control the robot.

In NaoQi modules can be seen as libraries. The robot is configured to load a list of modules. These modules vary from motion, audio, video and various other types of modules. As stated before, Python is the embedded interpreter in the NaoQi framework. The interpreter allows standard scripting without learning a new language and allow using numerous Python libraries (e.g. OpenCV, graphical extensions).

UsarNaoQi consists of three main components. An overview of the different components is given in Figure 12.

First is the USARSim message parser. This component receives messages from USARSim regarding the state of the Nao (joint angles, sensor values, etc), allows sending messages back and provides utilities for creating messages. This component runs in separate thread and each received parsed message is added to a queue.

The second component receives frames from the USARSim Image Server (Appendix B.2) and updates the camera images of NaoQi. Just like the message parser, this component runs in a separate thread. Each time step the component sends a request for a new frame to USARSim, waits for the data, then updates the current frame and finally deletes the old frame.

The last component is the NaoQi proxy and updates NaoQi using the information from the message parser and frame receiver. This information includes

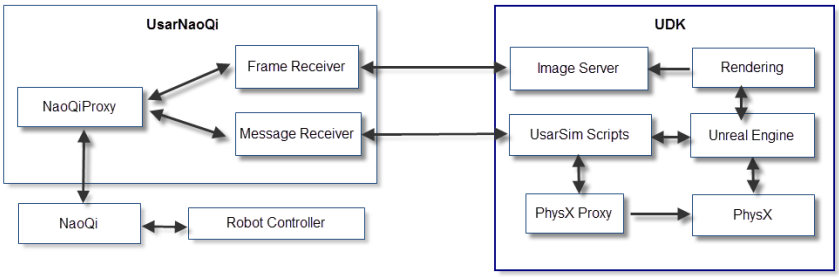


Figure 12: Overview of the different UsarNaoQi components

the joint angles, camera frames and sonar distances. For efficiency, this component only pushes updates to NaoQi when the state changes (i.e. joint angles, stiffness, etc).

The NaoQi proxy starts and updates NaoQi using the simulation interface provided by the NaoQi SDK of Aldebaran. It also allows other programs that use NaoQi, like Choregraphe, to identify the Nao as being simulated when connecting. A downside of using this interface is that not all NaoQi modules are supported. Examples of such modules are ALTextToSpeech (makes the Nao speak text, usually used to make the Nao tell what it is doing for debugging purposes) and ALAudioDevice (provides access to the sound data of the Nao robots microphones, and to send sound toward its loudspeakers). To solve this issue, wrapper modules could be written, with the functionality translated to the simulator.

5 Experiments

This section validates the behavior of the simulated Nao. Experiments are performed for both the real as simulated Nao. The results of these experiments are compared between the real and simulated robot.

The experiments are divided into three categories: basic, advanced and applied experiments. The first group are experiments that do not directly involve the Nao robot, but rather test the behavior of the simulation. The second group contains specific experiments with the simulated and real Nao. The third group are experiments which applies the simulator in a more real life scenario.

Section 5.1 describes the parameters affecting the simulation. In section 5.2 *basic* experiments without the simulated Nao are performed in USARSim. The first experiment is to validate the gravity settings in USARSim and the second experiment is to determine the precision of the physics at different simulation time steps. In section 5.3 *advanced* experiments with the real and simulated Nao are performed. The experiments vary from playing fixed motions to walking. The aim is to determine how similar the simulated Nao behaves like the real Nao. Finally in section 5.4 the simulator is applied in a real setting by simulating the RoboCup.

5.1 Parameters

When doing experiments it is important to understand which parameters affect the simulation of the Nao. These different parameters cover for example the global simulation settings, joints and rigid bodies. This results in a huge parameter space.

The following list describes the most important parameters for joints and rigid bodies in PhysX and describes how they affect joints and rigid bodies.

1. **Motor spring** - The amount of torque needed to move a joint to a target orientation. A higher value would result in the motor moving the joint with more force to the target orientation.

2. **Motor damping** - The damping constant resists the force applied by the spring setting. The result is that it smooths out oscillations.

3. **Solver Iteration Count** - PhysX solves the physics iteratively for each rigid body (as explained in section 3.2). This parameter determines the number of iterations used in the constraint solver. Due the high number of connected rigid bodies through joints this parameter should be carefully chosen. Setting this parameter too low results in incorrect physics (jointed bodies oscillating and behaving erratically), while setting it too high, possible combined with a shorter simulation timestep, might result in poor system performance (i.e. processing a frame takes too long).

4. **Linear and Angular Damping** - The amount of resistance the rigid body encounters when moving or rotating in the world. This can be seen as a force working opposite to the force on the rigid body.

5. **Center of Mass** - The center of mass controls the balance of the Nao. It affects the stability of the walking, whether standing up succeeds or not and more. By default the physics engine calculates a center of mass based

on the shape, which might not be the desired center of mass. One reason is that it only looks at the shape and assumes the rigid body consists out of one type of material. However in reality the rigid body might be composed out of different materials. Additionally even if the center of mass is correct, it might still not give the desired behavior in the simulation and in this case a slightly less "realistic" center of mass might be preferred (remember the simulation is only an approximation of the reality).

5.2 Basic Experiments

This first experiment section describes preliminary experiments that do not directly involve the Nao robot, but rather the physics engine and the used parameters in USARSim.

In section 5.2.1 the gravity of the simulation is verified. Gravity is one of the main factors influencing the balance of the robot. In section 5.2.2 the effects of the simulation timing on a single joint are tested. When the simulation timing is too low it can affect the precision of the simulation (resulting in a large joint error), while when it is too high the system performance might suffer.

5.2.1 Gravity

The first experiment verifies the gravity in USARSim. The reason for this initial experiment is that changing the gravity at a later point would affect the way the Nao behaves due the balance of the robot changing. A concept like the *Zero moment point* depends on the gravity force. This concept states that the point where the total of vertical inertia and gravity forces equals zero, the contact of the robot foot with the ground does not produce any moment in the horizontal direction. This concept is used in many walking techniques [27].

Another reason for doing this experiment is because prior USARSim versions were still using the default Unreal Engine gravity parameter, contradicting the gravity documentation[1] of USARSim.

USARSim uses *the International System of Units*[28] to specify lengths, sizes, masses, etc. Inside Unreal Engine USARSim converts the different measurement units to Unreal Units (uu). One meter is converted to Unreal Engine by multiplying the value 250 times (this value has no particular meaning). Additionally, Unreal Engine scales the gravity of rigid bodies by the *rigid body gravity scale*.

The experiment was performed by dropping a block from a high distance and measuring the fall distance after a number of different times. Then using the gravity formulas, the distance the block was supposed to fall was computed (*expected fall distance*). This *expected fall distance* assumes there is no force slowing down the falling block. Using the *expected fall distance* and measured fall distance the *correction* value can be computed. Results in this experiment were averaged over ten runs.

This default setting corresponds to a gravity constant of $4m/s$, which seems not sufficient for Earth (-520uu with the *rigid body gravity scale* set to 2.0). This setting results in a too slow descend with a factor of about 2.5, which slowly increases when the fall distances increases. The increasing correction is caused by the default linear damping setting of a rigid body (0.01). Linear damping

[1] `http://usarsim.sourceforge.net/wiki/index.php/Gravity_Documentation`

G (uu/s) / Dist (uu)	1024	2048	4096	8192	16384	32768
-520.0uu (rbs 2, ld 0.1)	2.44	2.46	2.50	2.56	2.65	2.78
-520.0uu (rbs 2, ld 0.0)	2.36	2.36	2.36	2.36	2.37	2.37
-2452.5uu (rbs 1, ld 0.1)	1.06	1.06	1.08	1.1	1.13	1.19
-2452.5uu (rbs 1, ld 0.0)	1.03	1.02	1.01	1.01	1.01	1.00

Table 1: Correction values for different gravity and fall distances. rbs is the rigid body scale and ld the linear damping.

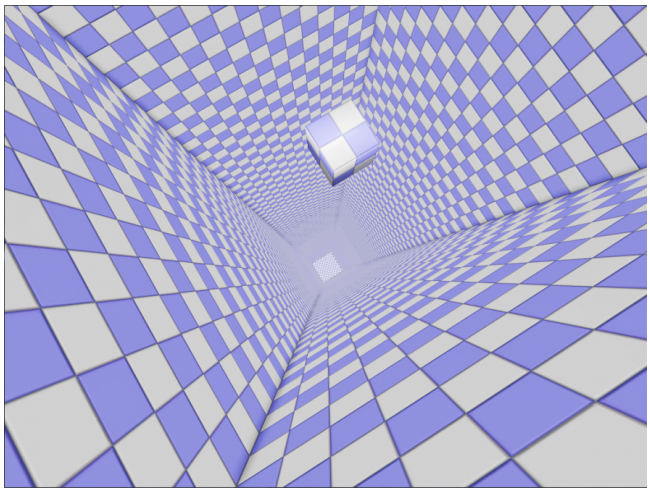


Figure 13: Experiment setup for testing gravity fall distances.

applies a small force to the rigid body to make it stop moving over time (if no other force is added). Setting the linear damping to zero results in the correction becoming slowly smaller for larger fall distances. This behavior is likely caused by the precision of the used timer in the measurement code.

Table 1 shows the correction values for different fall distances and gravity, including a more realistic gravity setting ($-250uu \times 9.8 = -2452.5uu$ and the *rigid body gravity scale* set to 1.0). Different linear damping values were used to show the effects of the linear damping. Without linear damping the correction starts very low and converges to 1.0. When using higher linear damping value the correction increases over time.

The result of this experiment shows -2452.5uu ($-2452.5uu/250 = -9.81m/s$) is realistic and correct gravity setting and the physics engine behaves as expected with regard to the gravity.

5.2.2 Simulation Timing

The second experiment is to investigate how the simulation timing parameters affects the simulation. Considering the complexity of the simulation, simulating a 21 DOF robot, the default simulation time step used in Unreal Engine ($\frac{1}{50}$ second) might not be sufficient for a correct simulation.

The following parameters control the simulation timing settings in PhysX:

1. **Time Step** - The PhysX simulation is updated by calling the simulation function with the 'elapsed time'. This function runs a number of TimeSteps to synchronize the physics behavior with the rendered frame rate. Longer time steps lead to poor stability in the simulation. The Unreal Engine uses a default time step of $\frac{1}{50}$.

2. **Fixed Time Step** - If not fixed, the *elapsed timed* is not divided into sub time steps. Using fixed time sub steps is very important to ensure a stable and reproducible simulation. Take for example a variable timestep of $\frac{1}{50}$ with the following framerates: 50, 25 and 100. At 50 fps the framerate equals the timestep and one timestep will be executed each frame. At 25 fps two timesteps fit into a frame, so two timesteps will be executed to synchronize the physics behavior with the framerate. At 100 fps only half a timestep fits into a frame. Because the timestep is variable, the executed timestep in that frame becomes $\frac{1}{100}$, which changes the actually timestep. In case the fixed timestep settting was used, the time would accumulate through the frames until it is able to execute one full timestep. In this case a higher framerate would change the precision of the simulation. The default setting in the Unreal Engine does not use fixed timesteps. In USARSim and the experiments in this thesis fixed time steps were used.

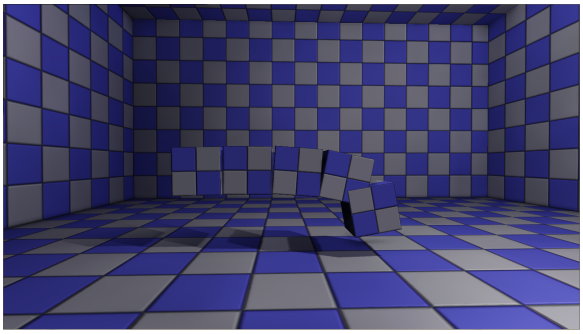
3. **MaxSubSteps** - Provides a cap to the number of time steps executed in the simulation function. If the elapsed time exceeds the $MaxTimeStep \times MaxSubSteps$, the remainder of the time is added to the next call of the simulation function. In other words, if the total time of $MaxTimeStep \times MaxSubSteps$ is lower than the framerate, the physics will never be correct. In this experiment this setting is set to a sufficient high number, so it does not influence the results.

For this experiment a test setup was made with several rigid bodies (simple block shaped objects) connected through joints. All rigid bodies are stacked on each other. The block on the bottom is fixed to the world frame. The bottom and next block in the stack are connected through a revolute joint. The remaining three blocks on top of that block are all connected through fixed unmovable joints (Figure 14). This results in a total of four rigid bodies and four joints. The motor spring of all joints is set to a very high value, to ensure the joints are not limited by the spring value while trying to satisfy their constraints.

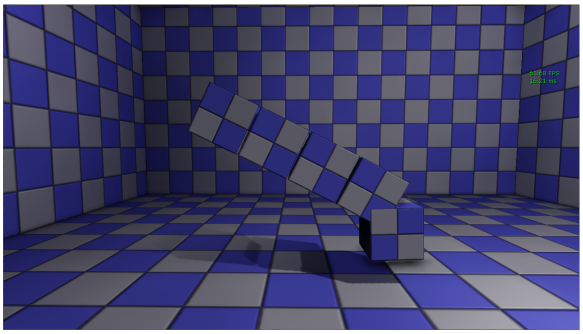
The experiment consists of setting the one movable joint to a specified angle and measuring the error between the desired target angle and measured angle. In this position the gravity will push the blocks down to the ground, while the joints will have to try to satisfy the constraints. This real angle is measured by taking the rotation between the bottom and next block in the chain.

The experiment was executed for twenty different time steps. Because a number of rigid bodies were connected, the experiment also used four different solver iteration count settings. For each timestep and solver iteration count setting the experiment was repeated five times. The measured error was averaged over these runs. The setup of this experiment is similar to the rigid bodies chained in, for example, the leg of the Nao. Executing this experiment shows how these settings affect the simulated Nao.

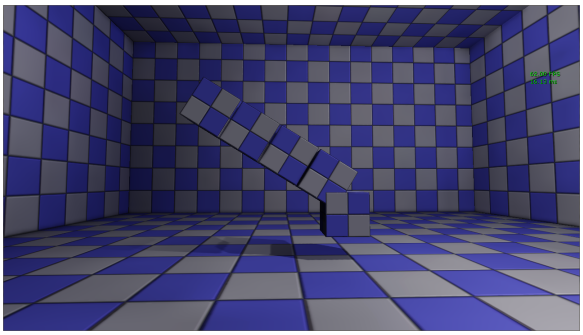
In Figure 15 the results are shown. The average errors for these tests vary between 2 and 3 degrees for the default timestep in UDK ($\frac{1}{50}$ second with the



(a) Time Step $\frac{1}{50}$



(b) Time Step $\frac{1}{100}$



(c) Time Step $\frac{1}{1000}$

Figure 14: Physic Timestep experiment under different simulation time steps. Solver iteration count was set to 8.

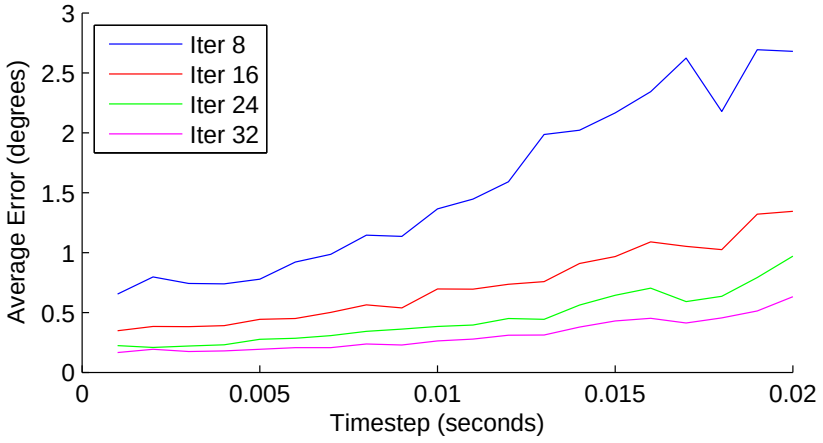


Figure 15: Physic Time Step experiment results

solver iteration count set to 8). Although such an error may seem small, the error accumulates through the chained joints. Figure 14(a) shows the experiment for timestep $\frac{1}{50}$) and as seen each joint has a relative large error (in contrast to fig 14(c)). Making the timestep smaller results in a lower average error and around $\frac{1}{200}$ the error does not decrease much more. Based on these results the remaining experiments used a default physics timestep of $\frac{1}{200}$ second , combined with a higher solver iteration count of 32.

5.3 Advanced Experiments

This section describes experiments with the simulated and real Nao. The results of these experiments are compared to see how close the simulated and real robot resemble each other.

In section 5.3.1 several fixed motions (kicking a ball and Tai Chi Chuan) are executed by both the real and simulated Nao. The center of mass is visualized and the joint angles are recorded for several runs, averaged and compared. Section 5.3.2 includes several walking experiments. The walking behavior of the real and simulated Nao are compared by analyzing the walk distances, joint angles and walk trajectories. Section 5.3.3 looks at how the framerate of the simulation affects the correctness of the simulation.

5.3.1 Fixed Motions

In this experiment the real and simulated Nao were set to perform two different fixed motions (i.e. play a sequential set of commands). Playing fixed motions is interesting for several reasons.

The first reason is to perform the animation correctly, the simulated Nao must maintain balance. The balance of the Nao is largely determined by the center of mass. An incorrect center of mass during movements can cause the Nao to be unable to maintain balance and as a result fall down to the ground.

To correctly perform this in the simulation, the center of mass must be above the supporting leg to ensure balance (visualized as the green sphere in Figure 19).

Due the limited precision of the physics engine and the small size of several parts of the robot, the simulated model will not perfectly match the masses from the real robot. An example of such a part is the pelvis, which is connected to the main body. When the mass ratio between such parts becomes too high, the solver will have more trouble due the precision of the physics engine.

The second reason is because the motion is a fixed animation, the experiment can be repeated for several runs. The results over these runs can be averaged, which not only gives the mean response but also the variance of the joint angles. The results of these recordings are compared between the simulated and real Nao.

The first motion is a motion to kick a ball with the right leg. The motion was performed ten times for both the real as simulated Nao robot. The recorded joint angles were averaged and the standard deviation was computed.

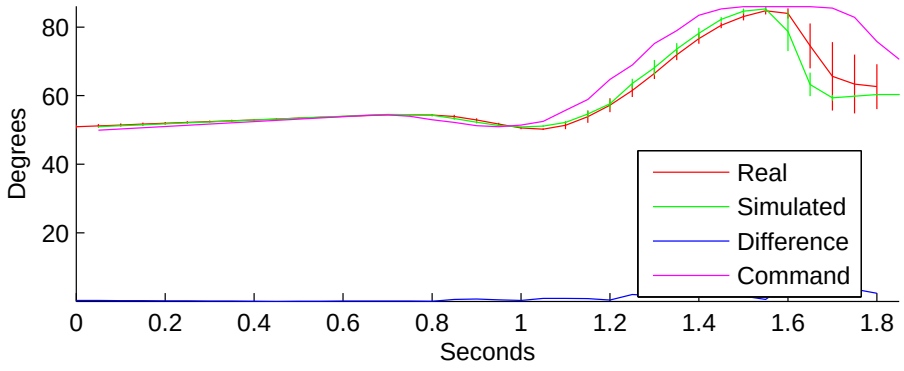


Figure 16: Joint angles and standard deviation of the RKneePitch joint while executing a kick motion. Results were averaged over ten runs. The red line shows the angles trajectory of the real Nao, while the green line shows the same for the simulated Nao. The blue line shows the difference between the two angle trajectories.

Figure 16 shows the average joint angles of the RKneePitch joint with the standard deviation and the difference between the average joint angles. The red line represents the real Nao and the green line the simulated Nao. The standard deviation of the real Nao is higher than the simulated Nao, indicating the joint angles vary more between the different runs. Similar behavior is seen for the standard deviation of other joints (see Appendix B.3). The blue line shows the difference of the RKneePitch joint angles between the real and simulated Nao and the purple line shows the command angles (i.e. desired angles). The simulated Nao shows a similar joint angles trajectory. Differences become larger when sudden motions occur due slight differences in the timing of the motion.

Figure 17 shows the average joint angles of the RAnkleRoll joint with the standard deviation and the difference between the average joint angles. This joint is interesting because the joint angles trajectory is not the same. During the kick motion the RAnkleRoll joint is told to move to 10 degrees in half a

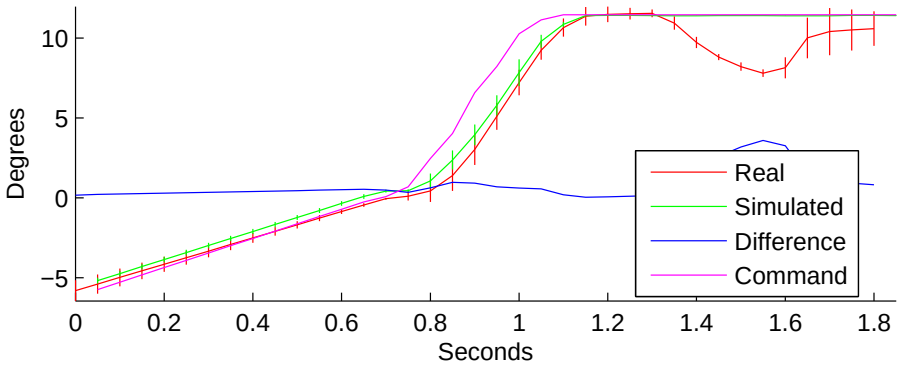


Figure 17: Joint angles and standard deviation of the RAnkleRoll joint while executing a kick motion. Results were averaged over ten runs. The red line shows the angles trajectory of the real Nao, while the green line shows the same for the simulated Nao. The blue line shows the difference between the two angle trajectories.

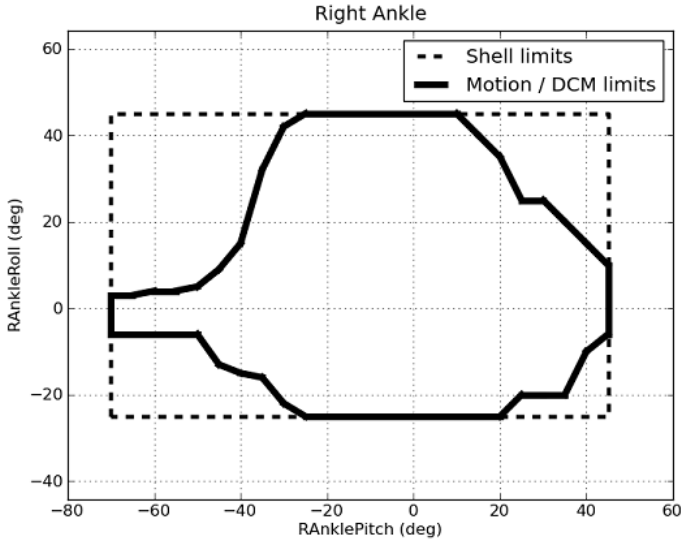


Figure 18: RAnkleJoint limits visualized in relation with the RAnklePitch joint.

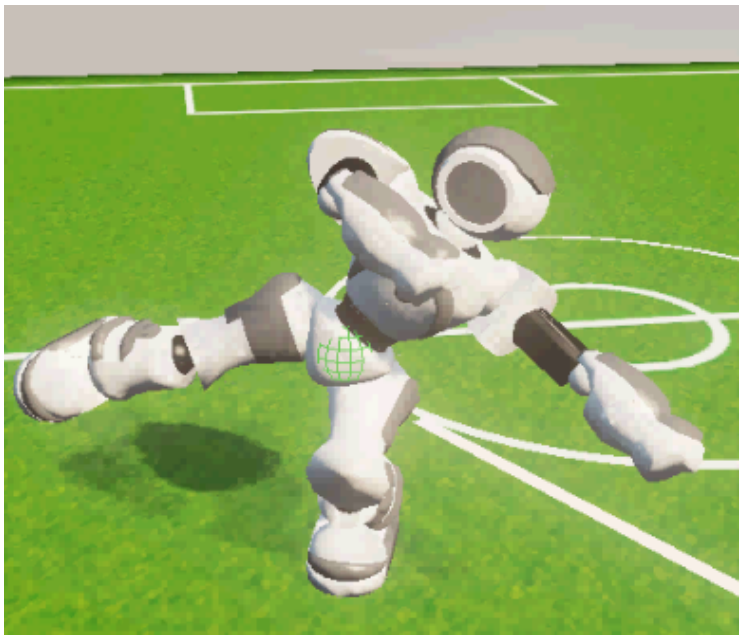


Figure 19: Nao performing the Tai Chi Chuan dance. The center of mass is visualized as a green sphere.

second and stay at 10 degrees for the remaining part of the motion.

The angles trajectory shows not much difference in moving towards 10 degrees, although again the standard deviation of the real Nao angles is higher than the simulated Nao angles. However when staying at 10 degrees the real Nao joint is not able to maintain this angle around 1.5 second. In this case the Nao fails to reproduce the behavior of the real joints because we did not include the restrictions of the collision hull of this particular joint in our model. The joint angle range of this joint is limited by the movements of the AnklePitch joint. Around 1.5 second the RAnklePitch joint moves from around -30 degrees to -60 degrees. At this joint angle the RAnkleRoll becomes limited to to a range of between -6 and 3 degrees. Figure 18 visualizes the limits of the RAnkleRoll joint for the different RAnklePitch joint angle values.

The second motion is the Tai Chi Chuan animation. This animation is a dance provided by Aldebaran. This animation is used by the manufacturer Aldebaran as diagnostic behavior; as long as a Nao is able to execute the Tai Chi Chuan, no major malfunction in the motors and gears is expected. During this animation the Nao first balances on one leg by stretching the other leg and the arms are positioned to maintain balance. The animation repeats this motion for the other leg (as shown in Figure 19).

For both the real as simulated Nao the animation was played ten times. For each run the joint angles of the Nao were recorded. The results were averaged and the standard deviation was again computed.

Figure 20 shows the average joint angles of the RHipYawPitch joint with standard deviation. This joint connects the upper body with the right leg. The

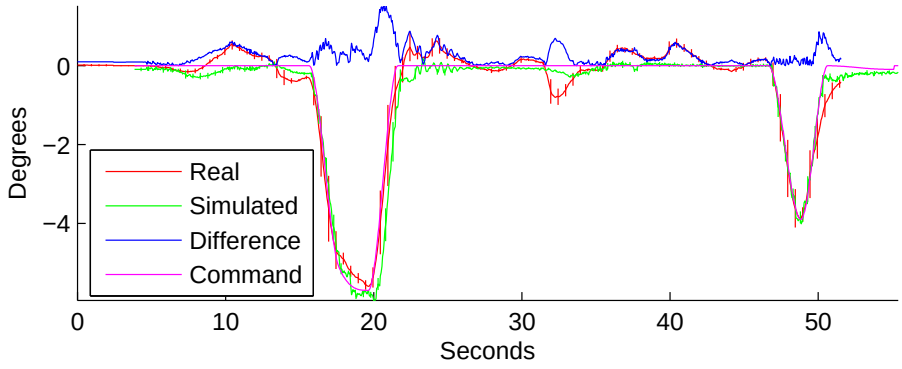


Figure 20: Joint angles and standard deviation of the RHipYawPitch joint while executing the Tai Chi Chuan dance. Results were averaged over ten runs. The red line shows the angles trajectory of the real Nao, while the green line shows the same for the simulated Nao. The blue line shows the difference between the two angle trajectories.

joint is, as the name suggests, relatively rotated 45 degrees in both the yaw and pitch direction relative to the parent limb (as opposed to all other joints which are either rotated in the pitch, yaw or roll direction). On top of that the joint shares the motor with the LHipYawPitch joint, which means the joint angle commands are shared between the two joints (i.e. they cannot have different joint angles). The red line represents the average angles of the real Nao with the red vertical lines visualizing the standard deviation. The green line similar represents the average angles of the simulated Nao with the green vertical lines visualizing the standard deviation. The purple line shows the command angles of the Nao robot (i.e. desired angles).

Similar to the standard deviation of the kick motion, the standard deviation is also higher for the real Nao. Similar behavior is seen for the standard deviation of the other joint angles.

The RHipYawPitch joint angle of the real Nao shows slight variations of max one degree around several times, like at $t = 10$ and $t = 33$. The simulated Nao shows less response around these times. Considering the desired joint angle is zero at these times, the expected behavior would be the Nao following these angles very closely. The command angles are all at zero around these times. This behavior is likely caused by the movement of other joints, like the RHipRoll and RHipYaw (see Appendix B.3.2).

Figure 21 shows the average joint angles for the RAnkleRoll joint. This joint is interesting because it shows, similar to the RAnkleRoll joint of the kick motion, a difference in the angle trajectories of the real and simulated Nao.

The command angle around 22 seconds is about 45 degrees. The angle trajectories show the same problem as the RAnkleRoll joint during the kick motion. The real Nao joint is unable to follow the command angles for the same reasons as during the kick motion.

Appendix B.3.1 and B.3.2 shows all recorded plots for both the kick as the Tai Chi Chuan motion.

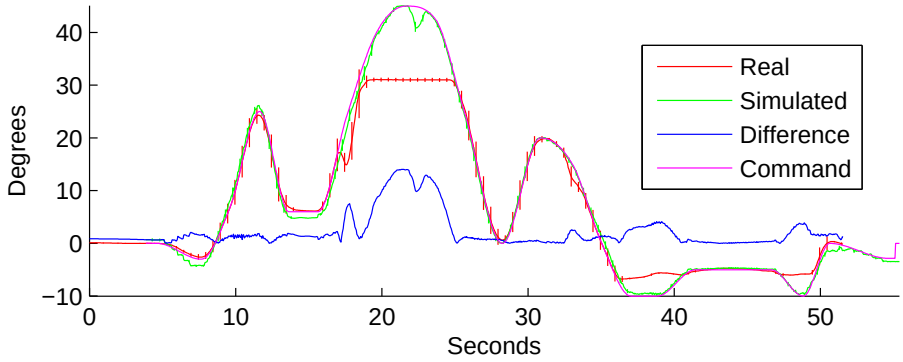


Figure 21: Joint angles and standard deviation of the RAnkleRoll joint while executing the Tai Chi Chuan dance. Results were averaged over ten runs. The red line shows the angles trajectory of the real Nao, while the green line shows the same for the simulated Nao. The blue line shows the difference between the two angle trajectories.

5.3.2 Walking

Realistic walking comparable to the walking behavior of the real Nao is crucial in a robot simulation. During a RoboCup match a robot walks a large part of the time. To investigate the walking behavior of the robot in the simulator, the experiments in this section are build up from doing a single step to walking a circle shaped path.

For this experiment several walking and turning tests were done for the simulated and real Nao using the included walk engine of the Nao provided by Aldebaran. This walk engine uses a simple dynamic model inspired by work of Kajita *et al.* [29] and is solved using Quadratic programming [30]. When walking at full speed it can reach a velocity of $9.52cm/s$ and $42deg/s$ when turning[4].

In the first test the Nao was set to do a single full step with the left leg. The joint angles of the real and simulated Nao were recorded and compared.

Figure 22 shows the average joint angles of the LKneePitch joint (i.e. the left knee) with standard deviation calculated over ten recordings of the real and simulated Nao. In contrast to section 5.3.1 the standard deviation for the real Nao is lower than the simulated Nao. The same behavior is also seen for the standard deviations of the other joints. Appendix B.3.3 shows all recorded joint angle trajectory plots.

In the second test the Nao was set to walk straight forward for a duration of five seconds. During this test the position of the Nao needed to be recorded as ground truth. To record the position of the Nao different options were considered.

One option was to use the Microsoft Kinect to record both the position and orientation of the Nao. This method has already been applied as ground truth

[4]Experiments were done with NaoQi 1.10.52. In version 1.12 the walking engine was updated with increased speeds.

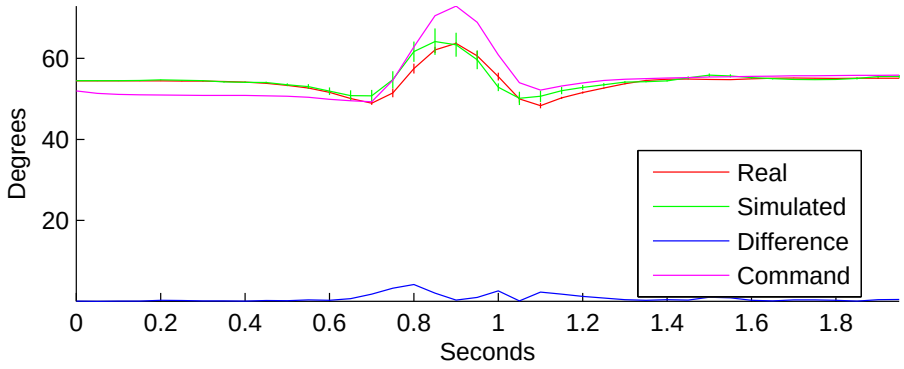


Figure 22: Average joint angles with standard deviation of the LKneePitch joint while executing a single step. Joint angles were averaged over ten runs for the real (red) and simulated (green) Nao. The blue and purple lines show the difference between the joint angle trajectories and desired angles respectively.

for the Nao by [31] et al. This sensor was introduced by Microsoft for the Xbox 360 gaming console, but quickly became popular in research due the low cost of the sensor. This sensor contains a low cost RGB-D camera, combined with information from a CMOS camera with an infrared based depth sensor.

Another option was to use a laser range finder to record the position of the Nao. The advantage of this method is that it is easy to setup, however you need to manually extract the position of the Nao from the laser Nao since it only records the changes. Marchant et al. [32] discusses such a setup and illustrate the results in a RoboCup SPL (RoboCup Standard Platform League) environment.

The final option, used in this experiment, that was considered was to record the position using a camera and extract the position of the Nao from the camera frames. This camera was attached to the ceiling. On the Nao robot head an infrared sensitive paper was attached. The position of the Nao was retrieved by applying a threshold filter to the captured camera frames.

For both the real and simulated Nao the forward walking was recorded ten times. Figure 23 shows the walking trajectories of the simulated and real Nao. The real Nao robots all walked around the expected distance (0.48 meter), while the simulated Nao robots only reached about 0.32 meter.

In the third test the Nao was set to turn at full speed for five seconds. This means the Nao should turn about 210 degrees. This test was executed ten times for the real and simulated Nao. During this test the real Nao reached the full 210 degrees turning, while the simulated Nao only reached about half.

In the last experiment the Nao was set to walk in a circle. Commands were generated by making one real Nao walk in a circle with a radius of 60cm. These commands were then replayed by the real and simulated Nao robots. Figure 24 shows the path trajectories of three different real Nao robots and one simulated Nao walking in a circle using the same walking commands. Each real Nao executed the walk five times, while the simulated Nao was set to repeat the walk ten times.

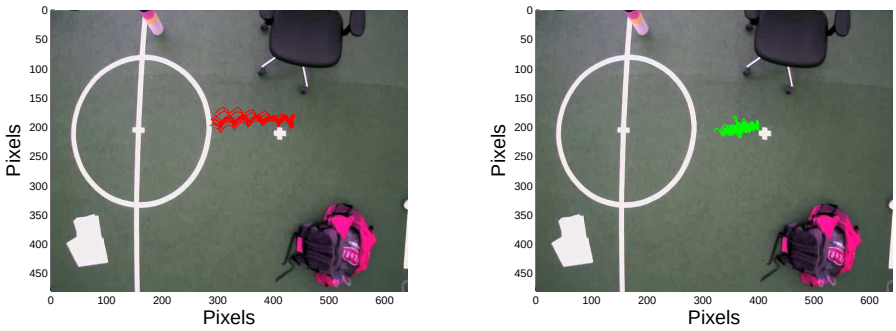


Figure 23: Forward walking (0.48 meter). The Nao started at the white cross. The red lines in the left figure represent the trajectories walked by the real Nao and the green lines in the right picture of the simulated Nao.

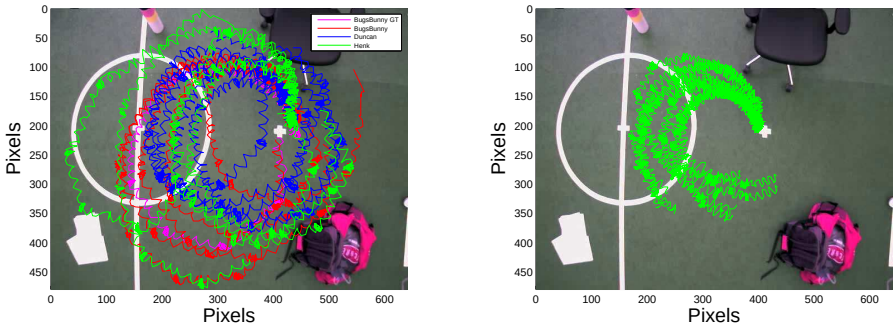


Figure 24: Left figure: trajectory of three different Nao's walking the white circle five times each. Right figure: trajectory of the simulated Nao walking the white circle ten times.

Most of the real Nao robots successfully walked a circle-like shaped path when replaying the commands, although there is a lot of variation in the paths.

On the other hand none of the simulated Nao robots were able to complete the circle. Considering the results of the forward walking and turning of the simulated Nao robots this is not totally unexpected.

One reason why the walking behavior of the simulated Nao fails to resemble the real Nao is possible due the selection of the parameters. The motor values of the simulated Nao were determined by making the simulated Nao walk 5 seconds forward for a range of different parameters. Then the parameters were chosen that were closest to the expected walk distance. Using better techniques (as discussed in section 6) could improve the selected parameters.

5.3.3 Framerate and simulation correctness

Another important aspect of a simulator is how well it runs on different machines. This might not seem so trivial because USARSim is based on UDK,

which is primarily intended as games development kit (although quickly becoming popular for other purposes as well). The main issue this choice causes is that the update logic is tied to the framerate at which the games engine is running. In other words, actors are ticked once per frame and during this tick they update their logic. The primarily logic that is affected by the framerate can be summed up as follows:

1. USARSim can only receive commands from the external control at most once per frame. These command updates include the updated joint parameters for the Nao, which must be sent at a high rate to execute the correct movement. Sending more than one command per frame will result in the commands to be processed all at once in a frame, making all commands except the last one received useless.

2. USARSim only sends status updates at most once per frame. These messages are sent in a timer function. The default behavior of USARSim is to send one status message per $\frac{1}{10}$ second. These status updates include the current joint angles for the Nao.

3. PhysX only simulates the physics at most once per frame. Although it always executes the same number of timesteps within a physics simulation call, it still means it is not possible to update the joint parameters between frames.

To find out the effects of the framerate on the correctness of the simulation a simple experiment was performed. The HeadYaw joint of the Nao performed a simple angle interpolation at different fixed framerates and the sensor HeadYaw angle values were measured by the controller. For reference the HeadYaw trajectory of a real Nao was also added. The results are plotted in Figure 25.

The blue line shows the desired HeadYaw joint angle sent to the Nao. The red line shows the angle trajectory of the HeadYaw joint for a real Nao. At 5 and 2 fps the effects of a low framerate become clearly visible. The trajectories become jagged and there is a delay between the desired and real angles. At 25 and 50 fps (the yellow and green line respectively) effects of a lower framerate are almost fully gone.

Another thing to keep in mind in case of higher framerates is the timestep setting of UsarNaoQi (interval at which UsarNaoQi processes messages and updates NaoQi) and the status message timer (interval at which new joint angle measurements are sent to UsarNaoQi). In Figure 25 the UsarNaoQi timestep was set to 1ms and the status timer interval to 100ms (default USARSim). Due this interval there is no difference between 25 and 50 fps in Figure 25. Figure 26 shows the same plot with the status timer timestep set to 10ms. Increasing the frequency of the status timer results in a smoother measured joint angle trajectory at higher framerates. A lower timestep is, just like a higher framerate, more demanding for the system.

5.4 Full Application Experiment

To test how well the performance is for real applications, the source code of the Dutch Nao Team[33] (DNT) has been tested with USARSim.

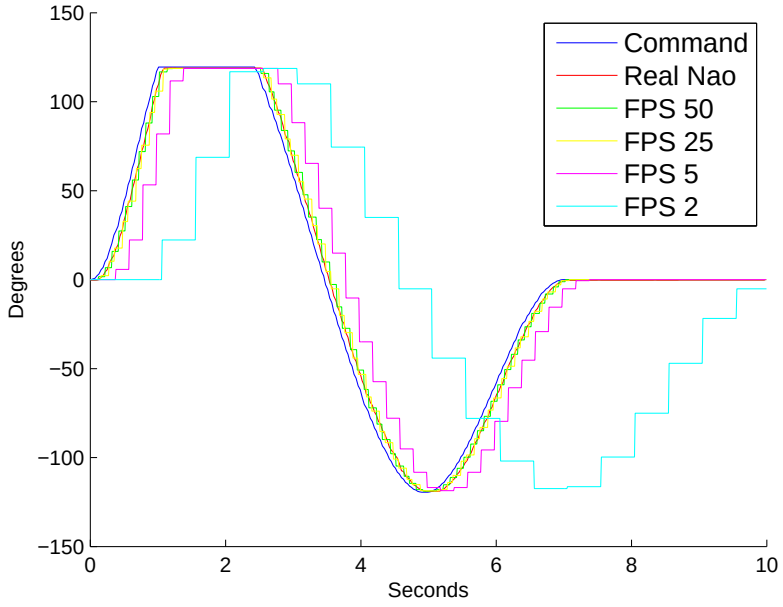
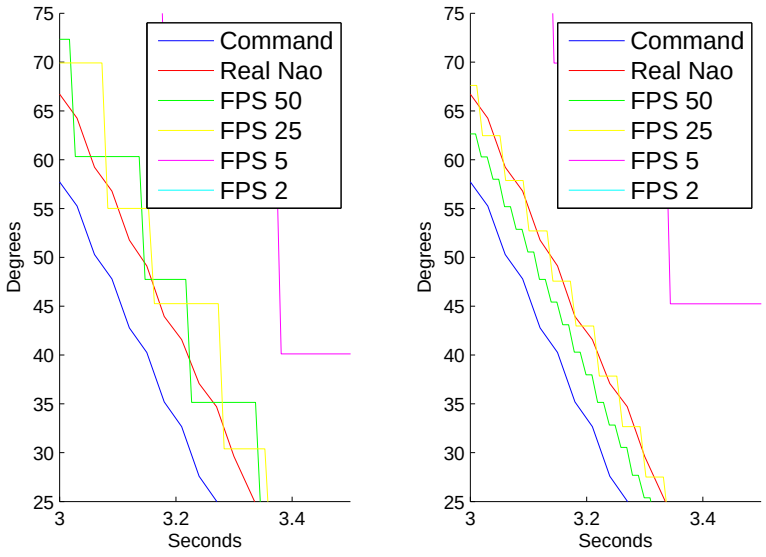


Figure 25: The effects of a lower framerate become visible as jagged lines and a delay between the desired trajectories appears.



(a) Status Message Timer 100ms (b) Status Message Timer 10ms

Figure 26: Shows the effects of the status message update rate at different intervals. UsarNaoQi timestep was set to 1ms.

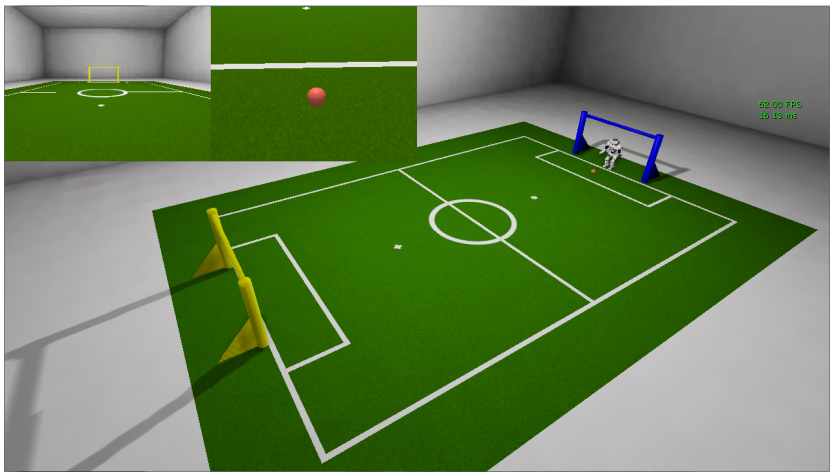


Figure 27: RoboCup football field 2011.

This experiment replicated a Standard Platform League scenario in USARSim. The Standard Platform League is a competition in the RoboCup league where identical robots play a football match. This means each team must use the same robots and no hardware modifications to the robots are allowed (with a few exceptions like using a custom flash card).

The rules for this competition change every year. The rules define the dimensions of the field, the number of players, game rules and illegal actions. The experiment was setup according to the RoboCup rules of 2011[5]. The football field is built on a total carpet area of length 7.4m and width 5.4m. Each team consists of four robots. All teams must use Nao humanoid robots manufactured by Aldebaran Robotics. One of the robots is the goal keeper and the remaining robots are field players. Field players are not allowed to move in their own penalty area.

The robot controller code must run directly on the Nao. The robots must play without human control and are only allowed to communicate between the robots on the fields and between the robots and the game controller. The game controller informs the robots about the state of the game (pre-match/ready, playing the game, post-match/finished, etc) and is provided by the organization of the RoboCup.

Figure 27 shows the simulated world containing a football field in USARSim, created using Unreal Editor according to the dimensions specified in the RoboCup rules of 2011.

The experiment was performed by putting a number of Nao robots in the simulated RoboCup environment. The average FPS was recorded for two different scenarios. In the first scenario, the Nao is simply standing and doing nothing. In the second scenario, the simulated Nao runs with the robot controller from the Dutch Nao Team [33]. Only minor modifications were needed to run the same code in the simulator. These modifications were mainly related to connection settings and the presence of absolute paths because normally the

[5]http://www.tzi.de/spl/pub/Website/Downloads/Rules2011.pdf

Number of Nao robots	FPS base	FPS DNT
0	320	320
1	120	110
2	100	55
3	65	30
4	50	10

Table 2: Simulation framerates with different numbers of Nao robots

code runs directly on the Nao robot. The controller was set in *play* mode. In this mode the Nao robots will walk around scanning for the ball.

The experiment was performed on an Intel iCore 7 920 with 6gb ram and an AMD Radeon 6850hd 1gb. USARSim was used in combination with the UDK December build 2011. UsarNaoQi was set to use a timestep of 10ms. The Nao robots in USARSim were set to send status updates at a rate of 10ms (joint angle updates).

The behavior of the simulated Nao robots running the Dutch Nao Team code is quite convincing. The Nao robots are able to stand up correctly when they fall on the ground. Using their cameras they are able to track the ball (as long as the right ball color is chosen). The main problem encountered is when the simulated Nao moves close to the ball to perform a kick motion; it does so by taking tiny steps. Due this it can take a long time to correctly position in front of the ball, longer than it would take the real Nao to position correctly. The problem is possible related to the results of the walking experiments (i.e. the simulated Nao only reaches about 65% of the real Nao walking distance).

Table 2 shows the framerate of the simulation with different numbers of Nao robots. The base FPS shows the framerate when the Nao robots are standing on the ground doing nothing, while FPS DNT shows the Nao robots in the *play* state of the game.

Without any Nao robots the scene rendered at 320 FPS. With one and two Nao robots the FPS drops to around 110 and 55 respectively, which is enough for running a decent simulation. With three Nao robots the FPS drops to 30, which is still enough for an acceptable simulation. With four Nao robots the simulation framerate drops to 10 FPS, resulting in incorrect movements.

To find the bottlenecks of the simulation profiler tools were used. These tools are provided by UDK. Using the *STATS* command various debugging information can be displayed on the screen. The most important one for the physics simulation is *STATS PHYSICS* and *STATS UNIT*. These commands respectively display the time per frame spent in the various physics components and the time spent drawing the frame and updating the logic code.

Using the information from the physics statistics tool the following results can be collected (Figure 28):

1. Fetch Results - The physics runs in a separate thread parallel to the game thread. When the asynchronous part of the game frame is finished it will make a call to the physics engine to fetch results. In other words, a high Fetch Results time indicates the game thread is doing nothing and waiting for the physics thread to complete the work of the current frame.

2. Total Dynamics Time - As the name indicates this is the total time spent

Figure 28: Four Nao robots in action with the physics statistics displayed

calculating the physics for the current frame and the time used for determining the physics calculation time for a frame.

Using the *gameplay profiler*, bottlenecks in the logic code can be found. This tool collects all kind of information about code execution in Unreal Script, including the time spent per frame in a function.

Using these tools reveals that when simulating four Nao robots half of the frame time is spent on the physics. The remaining part of the time goes to the sonar sensor (fires a high number of traces into the world), receiving and processing messages in the bot connection with the controller, sending the current status to the controller (joint angles) and updating the current joint angles by measuring the angle between the connected rigid bodies. Using these results several options can be considered to optimize the performance of the simulation with multiple Nao robots.

The first option is to lower the physics timestep and/or solver iteration count. By doing so the precision of the simulation becomes lower, but the performance higher. In this experiment lowering the physics timestep from $\frac{1}{200}$ to $\frac{1}{100}$ reduces the *Total Dynamics Time* with about half the time. The solver iteration count was kept the same.

The second option is to reduce the update timer of UsarNaoQi and status timer of the simulated Nao. This reduces the number of messages that need to be parsed. As a side effect the precision of the reported angles in NaoQi is reduced, resulting in effects as seen in Figure 26. A lower frequency of the command angles sent to UsarSim results in more abrupt movement of the joints. When this rate becomes too low, the motions of the simulated Nao will become incorrect.

The final option is to disable or simplify the most demanding sensors of the Nao, in particular the sonar sensors. The Nao robot is equipped with two sonar sensors. The simulation of this sensor is quite expensive because it fires a very high number of traces to determine the distance to an object. Turning the sonar sensors off drops the time spent in the game code, with four Nao robots, from

9 to 3 ms when idling and from 80 to 60 ms when running the DNT code.

6 Discussion and future work

In this thesis the design of a physical realistic walking robot model is described and a Humanoid Robot in USARSim was validated. The experiments show the simulated Nao mimics the behavior of the real Nao quite closely. Detailed analysis of the trajectories of the joint angles between the simulated and real joint angles of the Nao shows the difference between the angles is not very high, although there is no differences in the variance of the trajectories.. In case of a few joints the simulated Nao sometimes fails to reproduce the behavior of the real joints because we did not include the restrictions of the collision hull of this particular joint in our model. However, most plots of the joint angle trajectories show very similar behavior to the real Nao in our experiments. Considering a simulation is only an approximation of a reality the result is quite good, also because between real robots the behavior will never be the same. If we would redo the experiments with different robots and other environment conditions the results could be quite different.

Parameter optimization One reason why the joint angles of the simulated Nao robot do not always match the real joint angles is because the parameters (motor torque and damping, masses, simulation settings, etc) are determined using heuristic experiments (e.g. a linear search through a range of joint parameters). To improve the parameters better techniques should be used. For example [34] optimized the parameters of an AIBO robot, simulated in SimRobot, using an evolutionary algorithm.

Generic Interface In this thesis the focus was on the Nao Humanoid Robot. As a result an intermediate program, UsarNaoQi, was created which allows the user to access the simulated robot with its native NaoQi interface. The NaoQi interface allows to control the Nao using an extensive range of options (setting or interpolating joint angles, walk engine, etc). Since the intention of the model is to be also used for other type of legged robots, this is not ideal because the NaoQi interface cannot be used for other robots. Currently, the controls of USARSim are limited to setting the joint angles and stiffness. A more flexible interface should be created that allows more options. The challenge of such an interface is how to determine what options it should be given considering such options can quickly become too specific for generic legged robots. One example of a generic option that could be added is to interpolate towards a joint angle target. By adding this option the overhead of sending many messages between USARSim and UsarNaoQi could be potentially reduced, since NaoQi makes extensive usage of joint angle interpolation.

Virtual Nao To improve the compatibility of code written for the real Nao a virtual image of the simulated Nao could be created. Currently using the Dutch Nao Team code still requires several minor changes to make the code run in the simulator. These changes are required because the code is intended to run on the Linux distribution of the Nao (Gentoo[6]). Because of this several code changes are required, such as changing absolute paths and connection settings. To solve this problem a virtual image of the Nao could be created, equipped

[6] www.gentoo.org

with the same or similar OS and software as the real Nao (e.g. Gentoo and NaoQi). The only addition the virtual machine would require is to run the intermediate program, UsarNaoQi, configured with the connection settings of where USARSim runs.

Sensors The experiments are limited to the motion of the simulated Nao caused by the movement of the joints. However, the Nao is equipped with a wide range of sensors (as discussed in the introduction). The different sensors like the cameras, bumpers, sonars and inertial unit also contribute to the behavior of the Nao. More experiments are needed specifically aimed at these sensors. For example the Nao is equipped with two cameras and these cameras are actively used in the experiment of section 5.4. Although the camera sensors obviously function, it is not possible to say much about the correct working of these cameras without validation. Figure 29 shows an example of the problems you encounter when simulating a camera. The sensitivity of the camera of the different Nao versions results in a different camera image. Such differences would need to be modeled to simulate a camera properly. Although Unreal Engine already offers excellent rendering options, the current implementation in USARSim limits the simulation of the camera to simply capturing the image and sending it without modification.

Figure 29: Camera image of Nao 3.3 vs 4.0

Servo motor Another interesting research option is to extend the simulator with a more realistic servo motor and gears simulation, as used in the MoToFlex simulator [35]. In a physics engine a common way to control the joints of a robot is to set the desired joint angles and leave it to the physics engine to satisfy the constraints between the links (as done in this thesis). This approach is not the most realistic way to drive a joint and the method seen in the MoToFlex simulator could improve the behavior of the joints, which simulates the servo motors using a more realistic approach.

Previous work As said in the introduction, this thesis is inspired by the work of Zarati [4]. In this thesis the focus was moved from four legged robots (like the sony Aibo) to two legged Humanoid Robots and the robot was developed on a newer version of the Unreal Engine. Some of the issues mentioned by Zarati are

solved by newer versions of USARSim. For example, [4] mentions the simulation timestep is bound to the simulation framerate, while in Unreal Engine 3 this is controllable. Different kind of experiments were performed in this thesis. Zarati looked at collisions between different Aibos, while the experiments with the Nao robots in this thesis focused on walking.

Scaling issues Due several design choices it is not possible to scale up to high number of simulated Nao robots. Scaling up the number of Nao robots is important for simulating a RoboCup scenario. The goal of the RoboCup competition is to play a real football match with 22 Nao robots. Part of the reason why the simulation cannot scale up to this number of Nao robots is due the choice of the collision model, the physics timing step and possible overhead of message parsing and other, like the sonar sensor. Scaling of the number of Nao robots could be improved by simplifying the collision model by using more simple shapes (spheres, boxes) and lowering the physic timestep settings. The same could be applied to the message parsing (moving the code from Unreal Script to C for example) and the sonar sensor (using fewer traces to determine the sonar distance).

Existing Simulators As told in the introduction one of the questions is how the addition of legged robots contributes to the existing landscape of available simulators with support for this. The choice of the simulator depends on the needs of the user. The strong points of USARSim are the extensive tools, scripting, rendering and wide support of hardware. However, for example, SimRobot allows halting or stepwise executing the simulation, which is not seen in many other simulators.

Extending to other legged robots One of the goals was to make a generic model that could be applied to other limb typed robots, like the Frida or a spider like robot. The focus was solely on the Nao robot, because this robot demonstrates a wide range of possibilities. The implementation of the robot is very simple and only consists of two classes. One class represents the physical part (rigid body) and the other one the joint connecting the rigid bodies. The collision tools of Unreal Engine allows to quickly create a wide range of collision shapes, varying from simple models (boxes, spheres) to complex convex models, possible based on the visual shape of the robot.

This model can easily be applied to other robots as shown in Figure 30.

Validation What does the validation experiments tells about the simulated Nao robot? For one the experiments tell us how much the walking of the Nao (using the Aldebaran walk engine) behaves like the real Nao and how the system performance is while running the simulation. However for example it does not tell us much about collisions between different Nao robots. In other words the validation is limited to specific areas. This is not a bad thing, since the simulation is only an approximation of the reality.

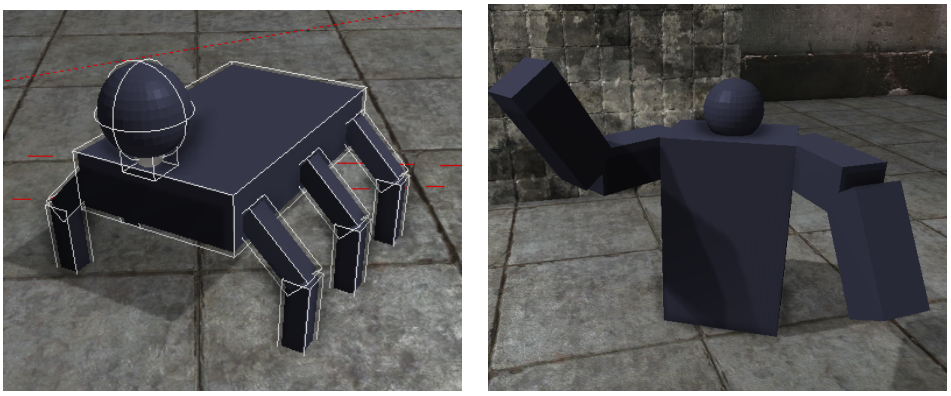


Figure 30: Example robots based on Figure 1.1

7 Conclusion

In this thesis is demonstrated that the simulation of the Nao in USARSim resembles reality quite closely. Our current model is usable in practice on the condition one keeps the flaws in mind like the walking behavior and scaling issues with the number of Nao robots. The combination UDK/USARSim provides several advantages to other robot simulators. It is a open source project, available on SourceForge[7]. It includes a powerful editor to create worlds and allows for easy setting up experiments, benchmarks and competition scenario's. USARSim includes all source code that is used to make the simulation. It is easy to make changes to the simulator and fit it to your needs. An intermediate program, UsarNaoQi, is written which allows to access the simulated robot with its native NaoQi interface. A disadvantage is that the underlying game engine is closed source and currently UDK does not support Linux and only partially OS X (excludes tools). The model developed for this humanoid robot demonstrates that robots with complex dynamics could be realistically modelled inside USARSim, which could be the basis of the introduction into USARSim of other models of complex robots like two arm manipulators and/or service robots.

[7] `http:\\usarsim.sourceforge.net`

8 Acknowledgments

I would like to thank my supervisor for his support thoughout my thesis. I would also like to thank the SimSpark project for their 3D model of the Nao robot.

A Nao Definition of Body and Head

The following code sample is the definition of the Nao main body and head in Unreal Script, including the joints.

```
// Create BodyItem part
Begin Object Class=Part Name=BodyItem
        Mesh=StaticMesh'Nao.MeshHi.naobody'
        Mass=1.03948
        SolverIterationCount=`NaoSolverIterationCount
End Object
Body=BodyItem
PartList.Add(BodyItem)

// Head + Joint
Begin Object Class=Part Name=Head
        Mesh=StaticMesh'Nao.MeshHi.naohead'
        Offset=(x=0,y=0,z=-0.155)
        Mass=0.52065
        SolverIterationCount=`NaoSolverIterationCount
End Object
PartList.Add(Head)

Begin Object Class=Part Name=Neck
        Mesh=StaticMesh'Nao.naohip'
        Offset=(x=0,y=0,z=-0.09)
        Mass=`MassSmallParts //0.05930
        SolverIterationCount=`NaoSolverIterationCount
End Object
PartList.Add(Neck)

DisableContacts.Add((Part1=BodyItem,Part2=Head))

Begin Object Class=RevoluteJoint Name=HeadYaw
        Parent=Neck
        Child=BodyItem
        Offset=(x=0,y=0,z=-0.09)
        LimitLow=-2.086 // -119.5
        LimitHigh=2.086 // 119.5
        Direction=(x=0,y=0,z=0)
        MaxForce=`MaxForceMotorType2
        Damping=`DampingMotorType2
End Object
Joints.Add(HeadYaw)

Begin Object Class=RevoluteJoint Name=HeadPitch
        Parent=Head
        Child=Neck
        Offset=(x=0,y=0,z=-0.09)
        LimitLow=-.672 // -38.5
        LimitHigh=.515 // 29.5
        Direction=(x=1.57,y=0,z=3.14)
        MaxForce=`MaxForceMotorType2
        Damping=`DampingMotorType2
End Object
Joints.Add(HeadPitch)
```

The other limbs are specified in a similar fashion.

B Tools

B.1 Unreal Editor

Unreal Editor is the level creation tool made by Epic Games for creating environments in Unreal Engine. A level could be seen as a collection of items that together forms a world or environment. In UsarSim *Unreal Editor* is used to create realistic scenarios. Using the content browser 3D definitions of models (of robots) and textures can easily be imported. Collision models of robots can either be created in external 3D modeling programs or inside Unreal Editor (including automatic generation based on the 3D definition). When creating the collision model in an external 3D modelling program, the collision model mesh must have a special name. This mesh is then automatically converted to a Unreal Collision model when importing the model in Unreal Editor.

The complete UDK package does not only contains the Unreal Editor, but also various other tools to help development. The gameplay profiler helps analyzing the performance of the UnrealScript code. The profiling data can be captured by manual starting or stopping the profiler or running it for a specified amount of time. The captured data can then be loaded in the tool to analyze time spent performing functions.

Extensive documentation for the *Unreal Editor* is available at UDN[1].

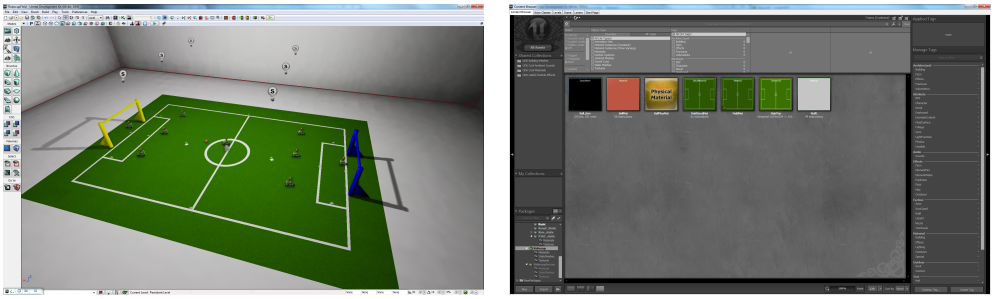


Figure 31: The left picture shows the main viewport of Unreal Editor with a RoboCup environment loaded. In this window the user can modify the environment (e.g. add, move or rotate objects). The right picture shows the content browser. In this window the user can import content (3D models, textures) and modify their properties.

B.2 Image Server

The task of the Image Server is to send camera frames of a simulated robot to the robot controller. Within UsarSim this is seen as a separate tool because of certain limitations when working with Unreal Script (due Unreal Engine not being open source). The main limitation in Unreal Script is that it is not possible to access the image buffer of a simulated camera. Due this limitation it is not possible to directly send the image data from unreal script to the robot controller. To come around this issue the Image Server draws the camera

[1] `http://udn.epicgames.com/`

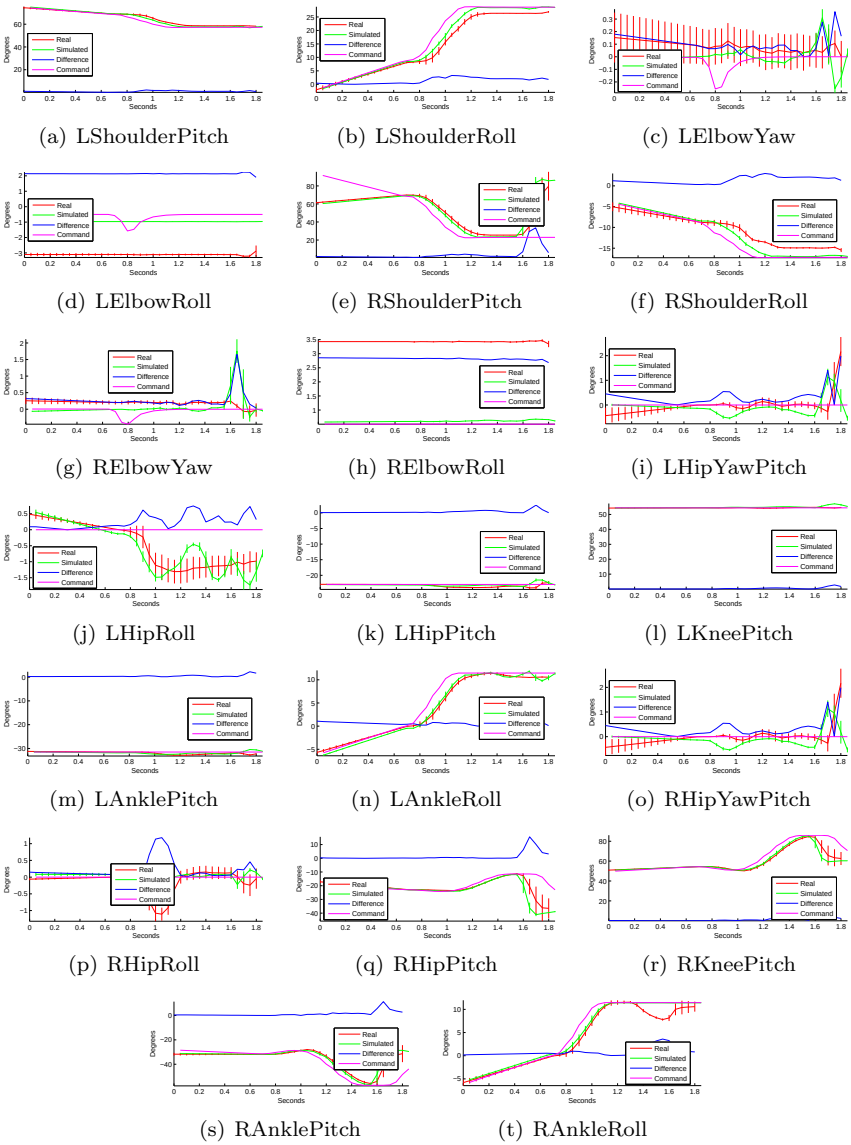
images on the screen. Then the function call of DirectX that presents the frame to the screen is hooked. *Hooking* is a term that refers to altering the behavior of software, which includes intercepting function calls. In the hooked function the specific part of the backbuffer containing the camera frame is copied and then finally send to the robot controller. The robot controller can then use the image for processing.

B.3 Graphs

This appendix contains the graphs of all joints for three motions: kicking a ball, Tai Chi Chuan dance and a single step.

B.3.1 Graphs Kick Experiment

Joint angles are averaged over 10 runs for both the real as simulated Nao. The red line shows the joint angles of the Real Nao, while the green line shows the same for the simulated Nao.



(a) LShoulderPitch (b) LShoulderRoll (c) LElbowYaw

(d) LElbowRoll (e) RShoulderPitch (f) RShoulderRoll

(g) RElbowYaw (h) RElbowRoll (i) LHipYawPitch

(j) LHipRoll (k) LHipPitch (l) LKneePitch

(m) LAnklePitch (n) LAnkleRoll (o) RHipYawPitch

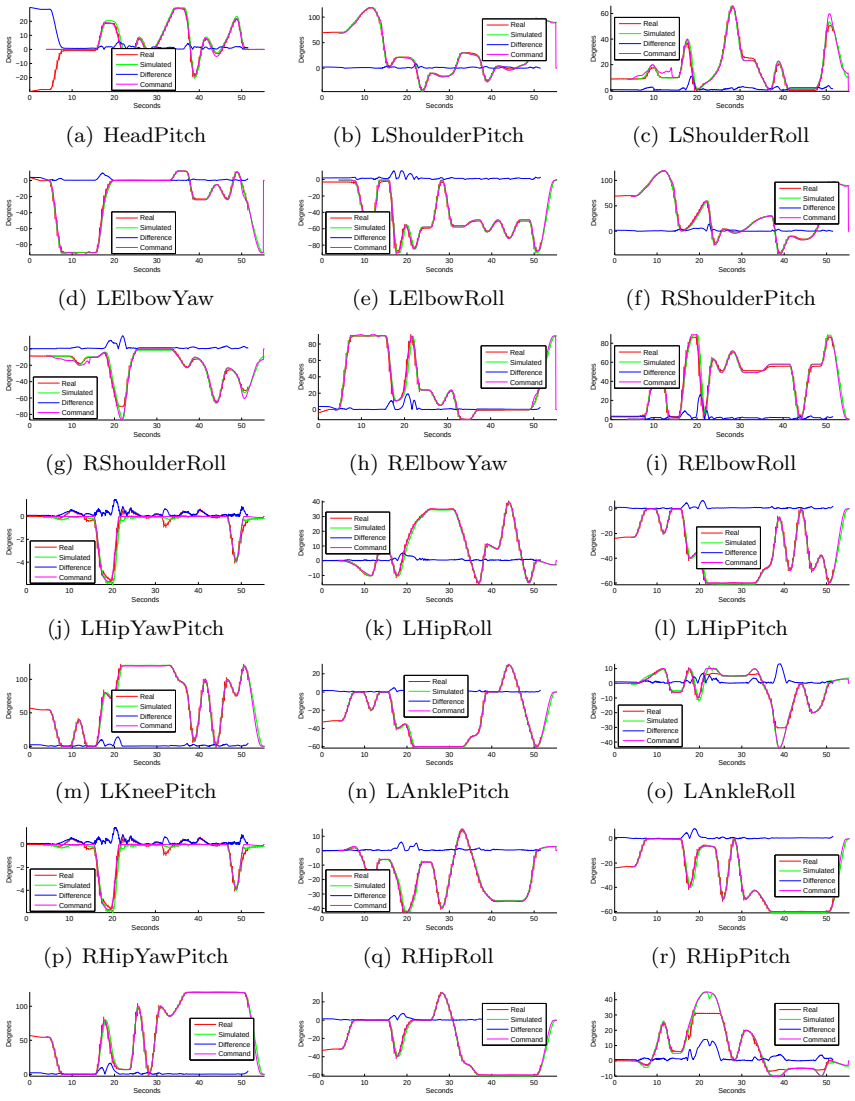
(p) RHipRoll (q) RHipPitch (r) RKneePitch

(s) RAnklePitch (t) RAnkleRoll

Figure 32: Graphs of joint angle trajectories with standard deviation for the Kick motion

B.3.2 Graphs Tai Chi Chuan Experiment

Joint angles are averaged over 10 runs for both the real as simulated Nao. The red line shows the joint angles of the Real Nao, while the green line shows the same for the simulated Nao.



(a) HeadPitch (b) LShoulderPitch (c) LShoulderRoll

(d) LElbowYaw (e) LElbowRoll (f) RShoulderPitch

(g) RShoulderRoll (h) RElbowYaw (i) RElbowRoll

(j) LHipYawPitch (k) LHipRoll (l) LHipPitch

(m) LKneePitch (n) LAnklePitch (o) LAnkleRoll

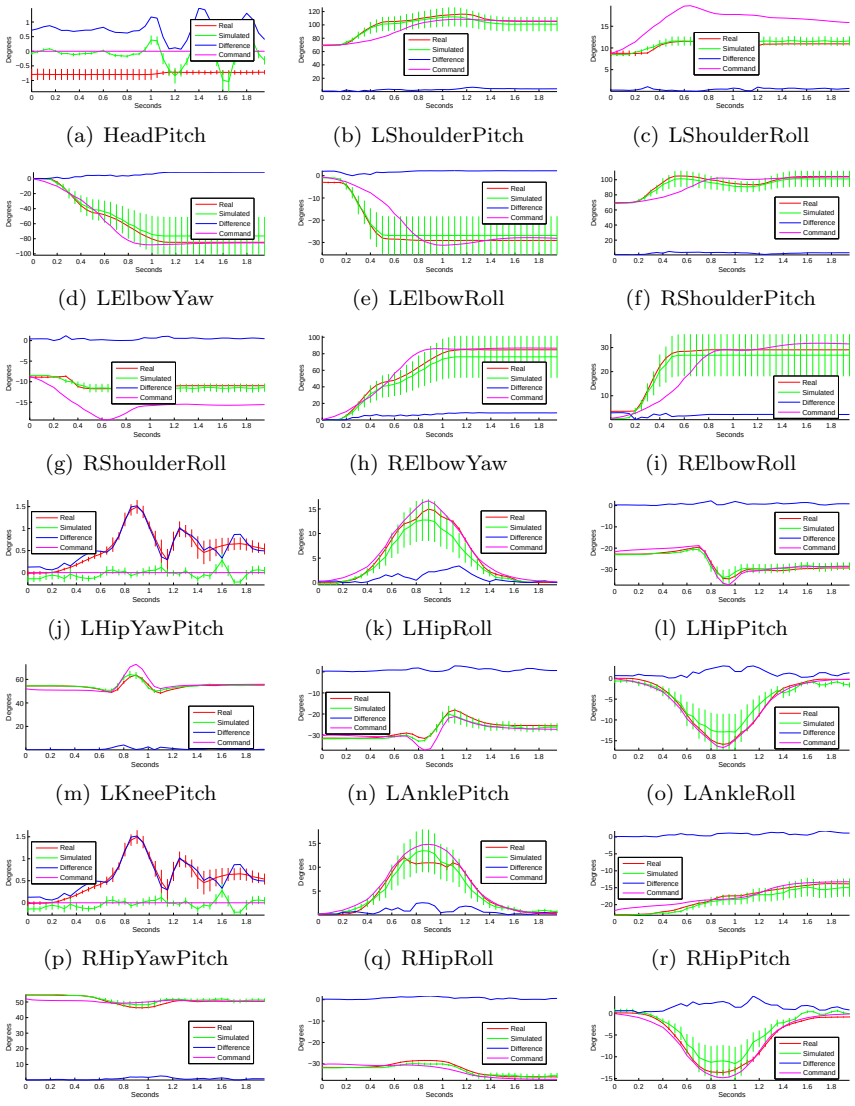
(p) RHipYawPitch (q) RHipRoll (r) RHipPitch

(s) RKneePitch (t) RAnklePitch (u) RAnkleRoll

Figure 33: Graphs of joint angle trajectories with standard deviation for the Tai Chi Chuan dance

B.3.3 Graphs Single Step Experiment

Joint angles are averaged over 10 runs for both the real as simulated Nao. The red line shows the joint angles of the Real Nao, while the green line shows the same for the simulated Nao.



(a) HeadPitch (b) LShoulderPitch (c) LShoulderRoll

(d) LElbowYaw (e) LElbowRoll (f) RShoulderPitch

(g) RShoulderRoll (h) RElbowYaw (i) RElbowRoll

(j) LHipYawPitch (k) LHipRoll (l) LHipPitch

(m) LKneePitch (n) LAnklePitch (o) LAnkleRoll

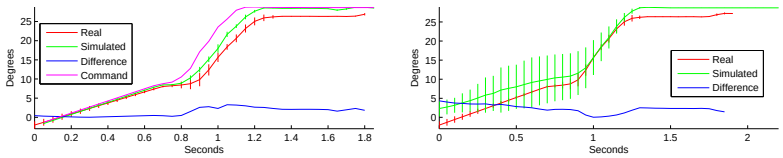
(p) RHipYawPitch (q) RHipRoll (r) RHipPitch

(s) RKneePitch (t) RAnklePitch (u) RAnkleRoll

Figure 34: Graphs of joint angle trajectories with standard deviation for the Single Step motion

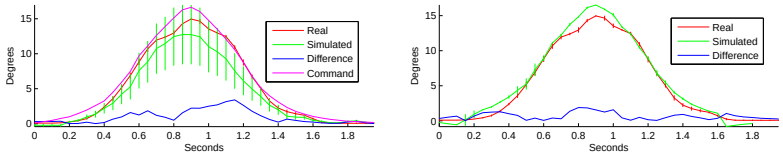
B.4 NaoSim

The experiments for the three motions were also executed for the simulated Nao in NaoSim[8]. These motions were recorded in NaoSim 1.12 and resulted in some interesting results. Overall NaoSim is able to replicate the behavior of the real Nao joint angles quite well. In the kick motion, NaoSim shows a higher standard deviation than the real Nao and UsarSim. On the contrary, the standard deviation for the single step motion is very similar to the real Nao, while UsarSim shows a much higher standard deviation for this motion. This is illustrated for two joints in Figure 35 (kick motion, LShoulderRoll) and Figure 36 (single step motion, LHipRoll).



(a) LShoulderRoll UsarSim (b) LShoulderRoll NaoSim

Figure 35: Comparison of the LShoulderRoll joint between the real Nao, UsarSim and NaoSim performing the kick motion. NaoSim shows a much higher standard deviation than the real Nao and UsarSim.



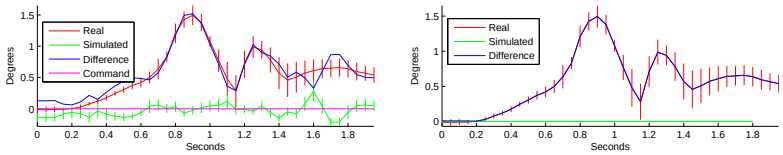
(a) LShoulderRoll UsarSim (b) LShoulderRoll NaoSim

Figure 36: Comparison of the LShoulderRoll joint between the real Nao, UsarSim and NaoSim performing the single step motion. UsarSim shows a much higher standard deviation than the real Nao and NaoSim.

Another interesting observed behavior is that few joints with hardly any movement stay perfectly at their target angle in NaoSim, while UsarSim and the real Nao show slight variation (Figure 37).

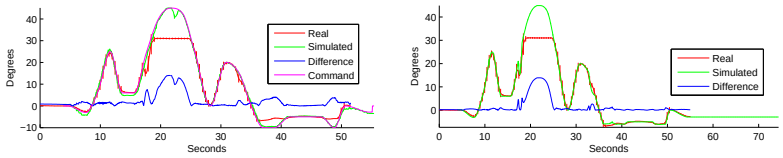
Finally NaoSim also showed the behavior of ignoring the collision hull limitations of the AnkleRoll joint, as shown in figure 38.

[8]For a full comparison of all graphs see: `http://nao.sandern.com/graphs/`



(a) LShoulderRoll UsarSim (b) LShoulderRoll NaoSim

Figure 37: Comparison of the LHipYawPitch joint between the real Nao, UsarSim and NaoSim performing the single step motion. The real Nao and UsarSim show slight variation, while NaoSim stays perfectly at the target angle.



(a) RAnkleRoll UsarSim (b) RAnkleRoll NaoSim

Figure 38: Comparison of the RAnkleRoll joint between the real Nao, UsarSim and NaoSim performing the Tai Chi Chuan motion. Both UsarSim and NaoSim ignore the AnkleRoll collision hull limitations around $t = 23$.

References

[1] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. Usarsim: a robot simulator for research and education. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1400–1405. IEEE, 2007.

[2] K. Carter, M. Scheutz, and P. Schermerhorn. A humanoidrobotic replica in usarsim for hri experiments. In *IROS Workshop on Robots, Games, and Research*, 2009.

[3] J. Jacobson and M. Lewis. Game engine virtual reality with caveut. *Computer*, 38(4):79–82, 2005.

[4] M. Zaratti, M. Fratarcangeli, and L. Iocchi. A 3D simulator of multiple legged robots based on USARSim. *Robocup 2006: Robot Soccer World Cup X*, pages 13–24, 2009.

[5] N. Greggio, G. Silvestri, S. Antonello, E. Menegatti, and E. Pagello. A 3d model of humanoid for usarsim simulator. In *First Workshop on Humanoid Soccer Robots*, pages 17–24, 2006.

[6] A. van der Mey, F. Smit, K.J. Droog, and A. Visser. Emotion expression of an affective state space; a humanoid robot displaying a dynamic emotional state during a soccer game. 2010.

[7] T. Whelan, S. Stüdli, J. McDonald, and R.H. Middleton. Line point registration: A technique for enhancing robot localization in a soccer environment. In *Proc. RoboCup Symposium*, 2011.

[8] A. Valtazanos and S Ramamoorthy. Naoisis: A 3-d behavioural simulator for the nao humanoid robot.

[9] C. Stanton, E. Ratanasena, S. Haider, and M.A. Williams. Perceiving forces, bumps and touches from proprioceptive expectations. In *RoboCup 2011 Symposium. Springer, Berlin/Heidelberg*, 2011.

[10] A. Bogdanovych, C. Stanton, X. Wang, and M.A. Williams. Real-time human-robot interactive coaching system with full-body control interface.

[11] O. Obst and M. Rollmann. Spark-a generic simulator for physical multi-agent simulations. *Computer Systems Science and Engineering*, 20(5):347, 2005.

[12] A. Held. Creating an abstract physics layer for simspark. 2007.

[13] O. Michel. Webots: Professional mobile robot simulation. *Int. J. Advanced Robotic Syst.*, 1(1):39, 2004.

[14] Thomas Röfer, Tim Laue, Judith Müller, Alexander Fabisch, Fynn Feldpausch, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Arne Humann, Daniel Honsel, Philipp Kastner, Tobias Kastner, Carsten Könemann, Benjamin Markowsky, Ole Jan Lars Riemann, and Felix Wenk. B-human team report and code release 2011, 2011. Only available online: `http://www.b-human.de/downloads/bhuman11_coderelease.pdf`.

[15] T. Röfer, T. Laue, J. Müller, A. Burchardt, E. Damrose, A. Fabisch, F. Feldpausch, K. Gillmann, C. Graf, T.J. de Haas, et al. B-human team report and code release 2010. 2010.

[16] D. Baraff. An introduction to physically based modeling: rigid body simulation iunconstrained rigid body dynamics. *SIGGRAPH Course Notes*, 1997.

[17] R.F. Riesenfeld. Homogeneous coordinates and projective planes in computer graphics. *IEEE Computer Graphics and Applications*, pages 50–55, 1981.

[18] S. Gottschalk. Separating axis theorem. *TR96*, 24, 1996.

[19] D. Baraff. An introduction to physically based modeling: Rigid body simulation iinonpenetration constraints. *SIGGRAPH Course Notes*, 1997.

[20] D. Baraff. Dynamic simulation of non-penetrating rigid bodies, 1992.

[21] J.D. Cohen, M.C. Lin, D. Manocha, and M. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff. ACM, 1995.

[22] D.J. Tracy, S.R. Buss, and B.M. Woods. Efficient large-scale sweep and prune methods with aabb insertion and removal. In *Virtual Reality Conference, 2009. VR 2009. IEEE*, pages 191–198. IEEE, 2009.

[23] K.S. Fu, R.C. Gonzalez, and C.S.G. Lee. *Robotics: control, sensing, vision, and intelligence*. McGraw-Hill, Inc., 1987.

[24] K. Shoemake. Animating rotation with quaternion curves. *ACM SIGGRAPH computer graphics*, 19(3):245–254, 1985.

[25] P. Baerlocher and R. Boulic. Parametrization and range of motion of the ball-and-socket joint. In *Deformable avatars: IFIP TC5/WG5. 10 DEFORM'2000 Workshop, November 29-30, 2000, Geneva, Switzerland and AVATARS'2000 Workshop, November 30-December 1, 2000, Lausanne, Switzerland*, volume 68, page 180. Springer Netherlands, 2001.

[26] R. Adobbati, A.N. Marshall, A. Scholer, S. Tejada, G.A. Kaminka, S. Schaffer, and C. Sollitto. Gamebots: A 3d virtual world test-bed for multi-agent research. In *Proceedings of the second international workshop on Infrastructure for Agents, MAS, and Scalable MAS*, pages 47–52. Montreal, Canada, 2001.

[27] M. Vukobratovic and B. Borovac. Zero-moment point-thirty five years of its life. *International Journal of Humanoid Robotics*, 1(1):157–173, 2004.

[28] International Bureau of Weights, Measures, B.N. Taylor, National Institute of Standards, and Technology (US). *The international system of units (SI)*. US Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 2001.

[29] S. Kajita and K. Tani. Experimental study of biped dynamic walking. *Control Systems Magazine, IEEE*, 16(1):13–19, 1996.

[30] P.B. Wieber. Trajectory free linear model predictive control for stable walking in the presence of strong perturbations. In *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, pages 137–142. IEEE, 2006.

[31] Piyush Khandelwal and Peter Stone. A low cost ground truth detection system using the kinect. In Thomas Roefer, Norbert Michael Mayer, Jesus Savage, and Uluc Saranli, editors, *RoboCup-2011: Robot Soccer World Cup XV*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2012. To appear.

[32] R. Marchant, P. Guerrero, and J. Ruiz-del Solar. A portable ground-truth system based on a laser sensor.

[33] A. Visser, S. Nugteren, H. van der Molen, M. van Bellen, E. Koster, S. Laan, B. Bodewes, T. Dingeman, S. Gieske, M. de Groot, et al. Dutch nao team: Team description for robocup 2011-istanbul.

[34] T. Laue and M. Hebbel. Automatic parameter optimization for a dynamic robot simulation. *RoboCup 2008: Robot Soccer World Cup XII*, pages 121–132, 2009.

[35] O. Urbann, S. Kerner, and S. Tasse. Rigid and soft body simulation featuring realistic walk behaviour. *RoboCup*, 2011.