



UvA-DARE (Digital Academic Repository)

Validation of the dynamics of an humanoid robot in USARSim

van Noort, S.; Visser, A.

DOI

[10.6028/NIST.SP.1136](https://doi.org/10.6028/NIST.SP.1136)

Publication date

2012

Document Version

Submitted manuscript

Published in

NIST Special Publication

[Link to publication](#)

Citation for published version (APA):

van Noort, S., & Visser, A. (2012). Validation of the dynamics of an humanoid robot in USARSim. *NIST Special Publication, 1136*, 190-197. <https://doi.org/10.6028/NIST.SP.1136>

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Validation of the dynamics of an humanoid robot in USARSim

Sander van Noort
Intelligent Systems Lab Amsterdam
Science Park 904
1098 XH Amsterdam, NL
+31205257460
Alexander.vanNoort@student.uva.nl

Arnoud Visser
Intelligent Systems Lab Amsterdam
P.O. Box 94323
1090 GH Amsterdam, NL
+31205257532
A.Visser@uva.nl

ABSTRACT

This paper 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. To guarantee a fast frame rate several approximations in this calculation have to be tried, and the performance (both in dynamics and computational effort) is evaluated in a number of experiments. This extension is made and validated for the humanoid robot Nao. On this basis many other applications become possible. A validated simulation allows us to develop and to experiment with typical robotic tasks before they are tested on a real robot.

Categories and Subject Descriptors

I.2.9 [Artificial Intelligence]: Robotics—*Kinematics and dynamics*; I.3.5 [Artificial Intelligence]: Computational Geometry and Object Modeling—*Physically based modeling*; I.6.4 [Simulation and Modeling]: Model Validation and Analysis

General Terms

Design, Verification, Performance

Keywords

simulation, NAO, dynamics, collisions

1. INTRODUCTION

Robotic simulation is essential in developing control and perception algorithms for robotics applications. Simulation creates the environment with known circumstances, which

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PerMIS'12 March 20-22, 2012, College Park, MD, USA.
Copyright 2012 ACM ACM 978-1-4503-1126-7-3/22/12 ...\$10.00.

allows rapid prototyping of applications, behaviors, scenarios, and many other high-level tasks. Robot simulators have been always used in developing complex applications, and the choice of a simulator depends on the specific tasks we are interested in simulating. Yet, the level of realism of a simulator is also important in this choice.

A 3D simulator for mobile robots must also correctly simulate the dynamics of the robots and of the objects in the environment, thus allowing for a correct evaluation of robot behaviors in the environment. Moreover, real-time simulation is important in order to correctly model interactions among the robots and between the robots and the environment. Since simulation accuracy is computationally demanding, it is often necessarily an approximation to obtain real-time performance.

In this paper the focus is on the humanoid Nao robot, which is selected by the RoboCup organization as the standard platform for the Soccer competition. This robot (see Fig. 1) is widely used in many research institutes around the globe.

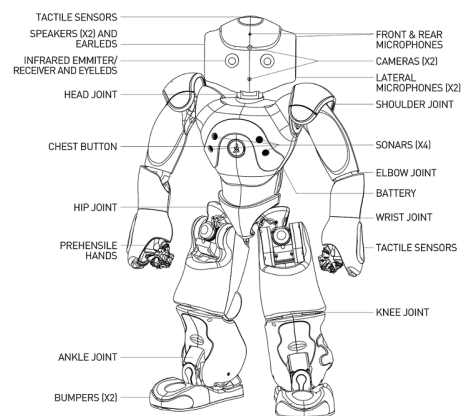


Figure 1: Schematic overview of the Nao (Courtesy of Aldebaran Robotics).

A model is described to replicate the dynamics of the Nao robot in USARSim[2]; an existing 3D simulator based on the Unreal Engine. Inside USARSim robots are simulated on the sensor and actuator level, making a transparent migration of code between real robots and their simulated counterparts possible. USARSim is an open source project, available on

sourceforge¹. It includes a powerful editor to create worlds and allows experiments, benchmarks and competition scenarios to be set up easily.

2. RELATED WORK

There are many robotic simulator platforms available. The simulators listed here are selected because they provide support for the Nao robot (shown in Fig. 2).

2.1 NaoSim

NaoSim is the official 3D simulator supported by Aldebaran. The simulator is based on the game development framework Unity² and is developed by Cogmation Robotics³. NaoSim is closed source and uses Nvidia PhysX as a physics engine.

The Nao is controlled using the NaoQi framework, which is the native interface of the Nao. This means that the same code can be used for both the real and simulated Nao. Furthermore the user can manipulate the Nao (move, rotate, etc) and add basic primitives (cubes, spheres, triangles, etc).

A downside is that, currently, it is not possible to create custom environments or simulate more than one Nao in NaoSim. Another potential downside is that the simulator is specifically developed for the Nao robot and as a result no heterogeneous teams of robots can be simulated.

2.2 SimSpark

SimSpark⁴ is the official 3D RoboCup simulator and is primarily made for this goal. SimSpark is used as the official simulator in the RoboCup 3D Soccer Simulation League. The simulator is open source and freely available. It uses a client-server architecture, where agents (i.e. robot controllers) are the clients that communicate with the simulation server. Several robots (including the Nao) are supported and SimSpark makes it easy to add new robots with *rsg* files that describe the physical representation of a robot.

SimSpark always starts a football simulation, including a soccer field, game states and referee. The robots are controlled using a custom protocol, not the native interface of the Nao.

Noteworthy is the abstraction of the physics layer, which is supposed to make it easy to switch between different physic engines[5]. Currently SimSpark only supports *Open Dynamics Engine* (ODE) as physics engine.

2.3 Webots

Webots⁵ is a commercial closed source robot simulator for educational purposes[7]. It 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 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 us to connect with the simulated Nao robot using the NaoQi framework.

2.4 SimRobot

SimRobot is a free open source general robot simulation and uses ODE as physics engine⁶. SimRobot consist of several modules linked to a single application, which differs from the commonly chosen 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 runtime. 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 us to command the robots and implements a sense-think-act cycle and is called each step by the core component of the simulation to read the commands for the robot it controls.

SimRobot is an initiative of a team from the RoboCup Standard Platform League, B-Human, and they provide more information in their Team Report and Code Release[8].



Figure 2: Screenshots of the different simulators in action: NaoSim (top left), SimSpark (top right), Webots (bottom left), SimRobot (bottom right)

3. SIMULATION MODEL

The RoboCup version of the Nao (H21 model) has 21 joints, resulting in 21 degrees of freedom (DOF). There is also an academic version with 25 degrees of freedom, which has 2 additional DOF in each hand. See Fig. 1 for a complete schematic overview of the Nao robot.

The movement of each joint can be described by a rigid body equation[1]. The first step is to definition of unconstrained motion as described in equation (1). This equation

⁶<http://www.informatik.uni-bremen.de/simrobot/>

¹<http://usarsim.sourceforge.net>

²<http://unity3d.com/>

³<http://www.cogmation.com/naosim.html>

⁴<http://simspark.sourceforge.net>

⁵<http://www.cyberbotics.com/>

contains four vectors, it takes both the spatial information $x(t)$, $R(t)$ and the linear and angular momentum $P(t)$, $L(t)$ into account. $F(t)$ and $\tau(t)$ are external forces and the input to solve this equation. The linear and angular speed $v(t)$, $\omega(t)$ can be derived from the linear and angular momentum when the total mass M and the inertia tensor $I(t)$ of a rigid body is known.

$$\frac{d}{dt}Y(t) = \frac{d}{dt} \begin{bmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ \omega(t)^*R(t) \\ F(t) \\ \tau(t) \end{bmatrix} \quad (1)$$

The inertia tensor $I(t)$ is time dependent, but can be calculated from the inertia tensor I_{body} in body space, which is a fixed property, by taking the orientation of the body into account $I(t) = R(t)I_{body}R(t)^T$.

$$\left[v(t) = \frac{P(t)}{M}, \quad \omega(t) = I(t)L(t) \right]^T \quad (2)$$

The next step is to take contacts into account. When the rigid body encounters a contact it imposes a constraint on the movement.

Two different types of contacts can be distinguished. The first is a contact caused by bumping into another rigid body or into the world. The other type of contact is caused by having a joint defined between two rigid bodies.

3.1 PhysX Dynamics

Nvidia PhysX is the underlying physics engine of Unreal and USARSim. A physics engine gives an approximate simulation of rigid body dynamics (or any other physical related system). 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 influencing each other over large distances.

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 choice of the time-step settings is important for the stability of the simulation. In general longer time steps lead to poor stability in the simulation, while shorter time steps can lead to poor system performance.

The motion of a rigid body can either be constraint by contacts (with the static world or other rigid bodies) or joints. The PhysX constraint solver limits the motion of rigid bodies (and satisfies the constraints) by reiterating the constraints a number of times.

The following three important aspects of PhysX are highlighted: actors, materials and joints. Collision detection is described in Section 3.3.

3.1.1 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. Importantly, actors 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 I_{body} and by a point of mass M located at the center of mass. The inertia tensor describes the rigid bodies' mass distribution. For our simulated Nao robot, care has been taken so that each body part has the actual mass as specified in Aldebaran's documentation⁷.

3.1.2 Materials

Materials describe the surface properties of actors. These properties are used when two actors collide. The result of a collision will influence the simulation and result in the actors bouncing, sliding, etc.

3.1.3 Joints

Joints connect two rigid bodies and limit the movement between those two bodies. How the movement is limited is specified by the type of joint. PhysX supports a large number of different joints including Revolute, Prismatic and 6 Degrees of Freedom Joint (which can again be configured to any of the earlier joints).

3.2 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.

A rotational joint, also called *revolute joint*, is as the name suggests capable of rotating around an axis. This type of joint allows one degree of freedom (DOF) between the two rigid bodies, namely the range of motion around the specified axis. In case of this type of joint the motion is usually also limited to a specified range around the axis.

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. A full description can be found in the book Robotics, chapter 2.2.10, by K.S Fu *et al.*[4].

3.3 Collision Detection

The Unreal Engine is designed to build multi-user games, which means that they apply an approach called the *generalized client-server model*. The task of networking is to keep the world state synchronized between the different users. In the case of *generalized client-server model* there is a server that is authoritative over the evolution 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 then 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* by 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

⁷http://users.aldebaran-robotics.com/docs/site_en/reddoc/hardware/masses_3.3.html

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, linear velocity and angular velocity.

Each client has its own scene, which contains actors, joints and effectors. Actors are world related objects which can interact with the world and other actors. Actors are *ticked* once per frame. During such a *tick* they can update their logic, including their physics.

The PhysX 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.

These alternative physics modes are implemented by the Unreal Engine and do not use the collision detection system of PhysX. 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).

The first collision representation is intended for *static meshes* in 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. The advanced option for static meshes is to check collisions per polygon against the static mesh 3D model itself and is potentially expensive to use. There is also a (simplified) collision hull option, but this option is not used for robots inside USARSim. Additionally there is a collision representation which is intended for *skeletal meshes* in the Unreal Engine. Skeletal meshes are used for game characters, not for USARSim robots.



Figure 3: The left picture shows the PhysX collision model, the right picture the Unreal Engine collision model.

The second collision representation is intended for PhysX and is created in the same way as the advanced static mesh version. The PhysX collision model is used in the physics simulation. However sensors will usually involve collision detection with the first representation. For example a simulated sonar sensor uses Unreal Engine tracing to detect objects in the world, which uses the Unreal Engine collision model. Care has been taken (as can be seen in Fig. 3) to keep both representations equivalent for the Nao robot.

PhysX collision detection algorithm.

The first step in collision detection is to find out which pairs of objects could collide. This stage is usually called

the *Broad Phase*. In case of PhysX this is the *Sweep and Prune* algorithm[3]. This algorithm detects potentially colliding pairs by comparing the bounding boxes of rigid bodies. 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 rigid body (done by comparing the starts and ends). If the starts and ends of two of such bounding boxes overlap in all axes it means a pair of possible colliding rigid bodies is found.

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 shapes it means this algorithm would roughly have a complexity of $O(n^2)$. 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.

4. EXPERIMENTS

The experiments are divided into two categories; experiments which check general properties for constrained rigid body motion and experiments that are directly related to the proposed Nao model.

4.1 Basic Experiments

This first experiment section describes preliminary experiments that do not directly involve the Nao robot. Yet, these experiments on fundamental properties of constrained body motion need to be performed before more advanced experiments are done, because they can have major influence on the dynamics of a robot that has to maintain its balance.

In section 4.1.1 the gravity of the simulation is verified. Gravity is one of the main factors influencing the balance of the robot. In section 4.1.3 the effects of the frame rate on the correctness of the simulation is tested.

4.1.1 Gravity

This first experiment is to verify 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. Another reason for doing this experiment is because prior USARSim versions were still using the default Unreal Engine gravity parameter, contradicting the gravity documentation⁸ of USARSim.

One real meter is converted to Unreal Engine units by multiplying the value 250 times. Additionally Unreal Engine scales the gravity of rigid bodies two times by default (*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 fall distance from the experiment the *correction* value can be computed. Results were averaged over ten runs.

⁸http://usarsim.sourceforge.net/wiki/index.php/Gravity_Documentation

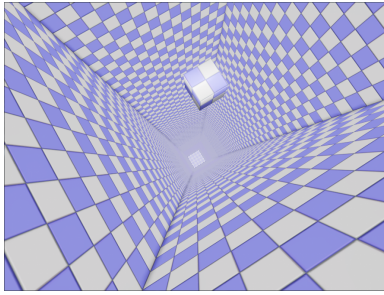


Figure 4: Experiment setup for testing gravity fall distances.

The default setting of the Unreal Engine is $-520uu$ with the *rigid body gravity scale* set to 2.0. This setting results in the block not falling far enough; the result has to be corrected with a factor of 2.5. Next we used a more realistic gravity setting based on g ; the standard acceleration due to free fall of an object in vacuum. Near the surface of the Earth this constant is $9.8m/s^2$ which corresponds for the Unreal Engine gravity parameter value of $-250uu \times 9.8 = -2450uu$ and the *rigid body gravity scale* set to 1.0. With those values the object falls the expected distance.

The result of this experiment shows $-2450uu$ is a realistic and correct gravity setting and the physics engine behaves as expected with regard to the gravity.

4.1.2 Simulation Timing

The second experiment is to investigate how the simulation timing settings affects the simulation. Considering the complexity of the simulation (21 DOF robot) the default simulation timing in the Unreal Engine might not be sufficient for a correct simulation.

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.

For this experiment a test setup was made with several rigid bodies connected through joints. Of these joints only one is movable. 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 we have a number of rigid bodies connected we also added 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. The setup of this experiment is similar to the rigid bodies chained in, for example, the leg of the Nao.

In figure 5 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 solver iteration count set to 8). Although such an error may seem small, the error accumulates through the chained joints. Making the timestep smaller and the solver iteration higher results in a lower av-

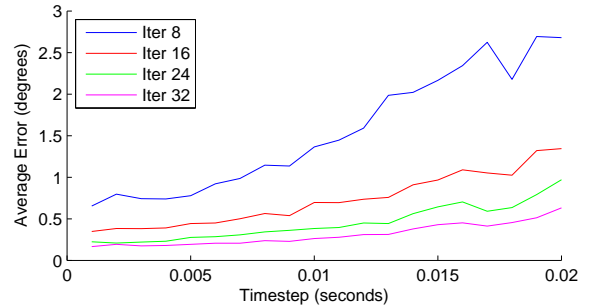


Figure 5: PhysX Time Step experiment results

erage error. Based on the results we used a default physics timestep of $\frac{1}{200}$ second, combined with a solver iteration count of 32.

4.1.3 Frame rate 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 uses UDK, which is primarily intended as games development kit. The main issue this choice causes is that the update logic is tied to the frame rate 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 frame rate 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 received one useless.
2. USARSim only sends status updates at most once per frame. 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 time steps 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 frame rate on the correctness of the simulation a simple experiment was performed. The HeadYaw joint of the Nao performed an angle interpolation at different fixed frame rates 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 Fig. 6.

The blue line shows the desired HeadYaw angle sent to the Nao. The red line shows the trajectory of the HeadYaw angle for a real Nao. At 5 and 2 frames per second (FPS) the effects of a low frame rate 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 frame rate are almost fully gone. When looking closer at both results it is

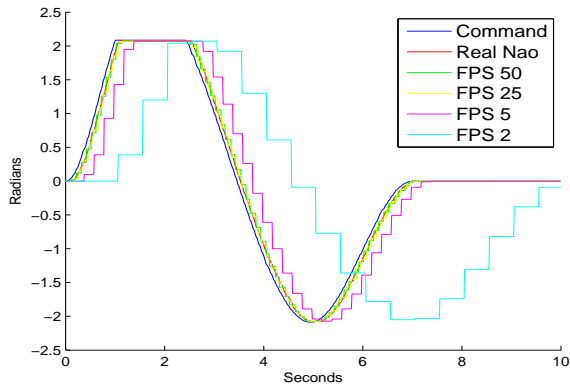


Figure 6: The effects of a lower frame rate become visible as jagged lines and a delay between the desired trajectories appears.

still possible to note differences between 50 and 25 FPS in terms of smoothness, although the measured angle is a very acceptable representation of the simulated angle.

4.2 Advanced Experiments

In this section experiments are done with the simulated and real Nao. The results of these experiments are compared to see how close they resemble each other. The experiments all consist of the combined movement of multiple joints. A more simple version of this experiment would be the movement of a single joint (for instance turning the head). Such simple experiments are performed and show close correspondence. The more advanced experiments are more interesting in the sense that they show sometimes unexpected results due to the interaction of the constraints in between joints. Alternative advanced experiments would be kicking the ball and collisions between two robots, as demonstrated by Zaratti *et al.*[11] for the four legged Aibo robot.

In section 4.2.1 a fixed motion is 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 4.2.2 includes several walking experiments. The walking behavior of the real and simulated Nao are compared by looking at the walk distances, joint angles and walk trajectories.

4.2.1 Tai Chi Chuan

In this experiment the real and simulated Nao were set to perform the Tai Chi Chuan dance. (i.e. play a sequential set of commands). During this animation the Nao first balances on one leg by stretching the other leg and keeping the arms in a specific position to keep balance. The animation repeats this motion for the other leg. Playing this dance is interesting for several reasons.

First 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 Fig. 7).

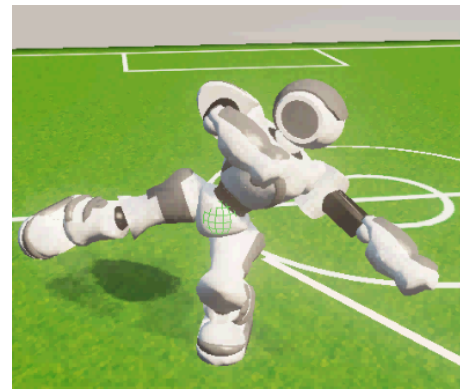


Figure 7: Nao performing the Tai Chi Chuan dance. The center of mass of the Nao is visualized as the green sphere.

Second because the motion is a fixed animation the experiment can be repeated for several runs, so the results over several runs can be averaged and compared against the joint angles between the simulated and real Nao. Finally, 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.

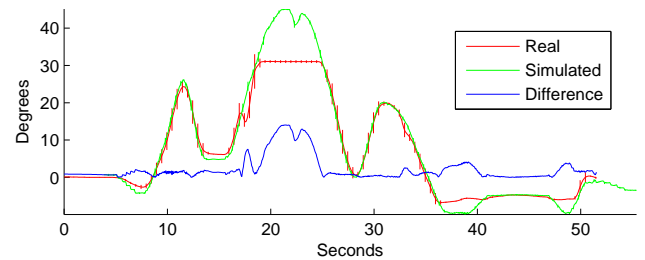


Figure 8: Joint angles and standard deviation of the RAnkleRoll joint while executing the Tai Chi Chuan dance. Results were averaged over ten runs. The red line shows the angles trajectory of the real Nao, while the green line shows the same for the simulated Nao. The blue line shows the difference between the two angles trajectories.

Fig. 8 shows the average joint angles for the RAnkleRoll joint. This joint is interesting because it shows a difference in the angles trajectories of the real and simulated Nao.

The command angle around 22 seconds is about 45 degrees. The real Nao joint is unable to follow the command angles. Most likely this is caused by the movement of other joints, resulting in a force being put on the parts around the joint. When sufficient force is put on the joint it will be unable to maintain the correct position (due to the motor not putting enough force in maintaining that position). In the case of the simulated Nao RAnkleRoll joint there is either not enough force pushing on the joint or the force of the joint used to maintain the position is too high.

4.2.2 Walking

Realistic walking comparable to the walking behavior of the real Nao is crucial in a robot simulation. During a RoboCup match a robot will have to walk a large part of the time.

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.*[6] and is solved using Quadratic programming[10]. When walking at full speed it can reach a velocity of 9.52cm/s and 42deg/s when turning.

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.

Fig. 9 shows the average joint angles of the LKneePitch joint (i.e. the left knee) with standard deviation over ten recordings of the real and simulated Nao. In contrast to section 4.2.1 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.

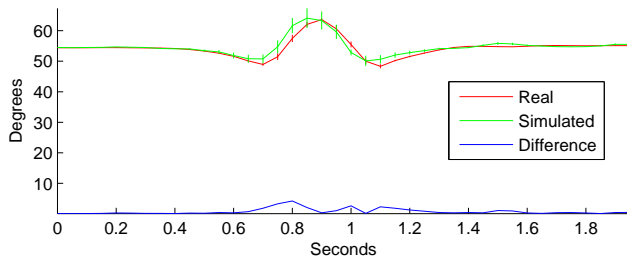


Figure 9: Average joint angles with standard deviation of the LKneePitch joint while executing a single step. Joint angles were averaged over ten runs for the real (red) and simulated (green) Nao. The blue line shows the difference between the joint angles trajectories.

For both the real as simulated Nao the forward walking was recorded ten times. The real Naos all walked around the expected distance (0.48 meter), while the simulated Naos only reached about 0.37 meter. This result for the simulated Naos could be tweaked (for instance by enlarging the motor force), but this makes the robot less stable.

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 again 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 Naos. Fig. 11 and 10 shows the path trajectories of three different real Naos and a 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.

Most of the real Naos successfully walked a circle like shaped path when replaying the commands, although there

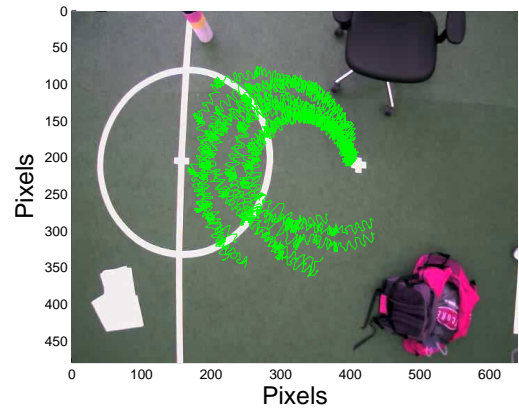


Figure 10: Trajectory of the simulated Nao walking in a circle with the same diameter as the white circle, repeated ten times for each Nao.

is a lot of variation in the paths.

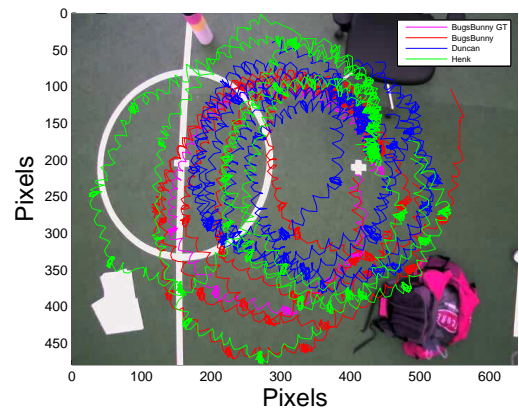


Figure 11: Trajectory of three different Naos walking in a circle, repeated five times for each Nao (recording using camera ground-truth).

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

5. FULL APPLICATION EXPERIMENT

To test how well the performance is for real applications, the source code of the Dutch Nao Team[9] has been tested with USARSim.

This application not only involves walking around, but also perception and dedicated behaviors like kicks and standing up.

To test real applications an intermediate program has been created, UsarNaoQi, which works as a proxy server, converting NaoQi messages in USARSim messages and vice versa. NaoQi is the framework provided by Aldebaran and allows the user to control the Nao in various programming

languages (C++, Python, C# or Urbi).

The source code of the Dutch Nao Team is written in Python, and could be directly applied. The code was fully functional, the robots could standup, position themselves on the field, locate the ball and kick the ball. The only observed difference is in the approach of the ball; the Dutch Nao Team code makes a number of small steps to get in a good position behind the ball. In simulation those steps are too small; the Nao needs too much time to position itself.

The experiment was performed by putting a number of Nao robots in the simulated RoboCup environment. The average frames per second (FPS) was recorded for two different scenarios. In the first scenario the Nao is simply standing and doing nothing. In the second scenario we executed the Nao with robot controller from the Dutch Nao Team. The controller was set in *play* mode. In this mode the Naos will walk around scanning for the ball.

The experiment was performed on a computer with an Intel iCore 7 920 processor and an AMD Radeon HD 6850 graphics card. USARSim was used in combination with the UDK December build 2011. UsarNaoQi was set to use a time step of 10ms; the Naos in USARSim sent status updates at a rate of 100 times per seconds (joint angle updates).

Table 1 shows the frame rate of the simulation with different numbers of Naos. The base FPS shows the frame rate when the Naos are standing on the ground doing nothing, while FPS DNT shows the Naos in the *play* state of the game.

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

Table 1: Frame rate results with UsarNaoQi time step of 10ms

Without any Naos the scene is rendered at a FPS of 320. With one and two Naos the FPS drops to around 110 and 55 respectively, which is enough for running a decent simulation. With three Naos the FPS drops to 30, which is still acceptable (see section 4.1.3). With four Naos the simulation frame rate drops to 10 FPS, resulting in incorrect movements.

To find the performance bottlenecks in the simulation various profiler tools provided by UDK are used (PhysX statistics and UnrealScript code profiler). Using these tools reveals that when simulating four Naos half of the frame time is spent in the physics. The remaining part of the time goes to the sonar sensor (tracing), 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.

6. CONCLUSION

In this paper we demonstrated that the simulation of the Nao in USARSim resembles reality quite closely. Our current model is usable in practice on the condition that one keeps in mind the limits of the method; like the walking behavior and the scaling issues with the number of Naos. The combination of Unreal/USARSim provides several ad-

vantages over other robot simulators. The simulation is at such a level that transparent migration of code between real robots and their simulated counterparts is possible. In this paper this is demonstrated with an intermediate program, UsarNaoQi, which enables access to the simulated robot with its native interface. Using this interface several experiments have been performed with both the real and simulated robot. The experiments consisted of movements where most of the 21 DOF were needed to maintain balance, which allowed us to monitor unexpected correlation between joints. The model developed for this humanoid robot demonstrates that robots with complex dynamics could be realistically modeled inside USARSim, which could be the basis of the introduction of other models of complex robots into USARSim like two-arm manipulators and/or service robots.

7. ACKNOWLEDGMENTS

The authors like to thank Hayley Hung for proofreading the manuscript. Part of the research is funded by the Dutch IIP Cooperation Challenge 'Sensor Intelligence for Mobility Systems'.

8. REFERENCES

- [1] D. Baraff. An introduction to physically based modeling: rigid body simulation I - unconstrained rigid body dynamics. SIGGRAPH Course Notes, 1997.
- [2] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. Usarsim: a robot simulator for research and education. In *Proceedings of the International Conference on Robotics and Automation (ICRA'07)*, pages 1400–1405, 2007.
- [3] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the Symposium on Interactive 3D graphics*, pages 189–196. ACM, 1995.
- [4] K. Fu, R. Gonzalez, and C. Lee. *Robotics: control, sensing, vision, and intelligence*. McGraw-Hill, 1987.
- [5] A. Held. Creating an abstract physics layer for simspark. Studienarbeit im Studiengang Informatik, Universität Koblenz-Landau, November 2010.
- [6] S. Kajita and K. Tani. Experimental study of biped dynamic walking. *Control Systems Magazine, IEEE*, 16(1):13–19, 1996.
- [7] O. Michel. Cyberbotis ltd - webotsTM: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, March 2004.
- [8] T. Röfer et al. B-human team report and code release 2011. Published online, November 2011.
- [9] C. Verschoor et al. Dutch nao team - code release 2011 and technical report 2011. Published online, Universiteit van Amsterdam, October 2011.
- [10] P. Wieber. Trajectory free linear model predictive control for stable walking in the presence of strong perturbations. In *Proceedings of the International Conference on Humanoid Robots*, pages 137–142, 2006.
- [11] M. Zaratti, M. Fratarcangeli, and L. Iocchi. A 3D simulator of multiple legged robots based on USARSim. In *Robocup 2006: Robot Soccer World Cup X*, volume 4434 of *Lecture Notes in Artificial Intelligence*, pages 13–24. Springer, 2007.