Users' Guide

Răzvan V. Florian
Center for Cognitive and Neural Studies (Coneural)
Str. Saturn 24, 400504 Cluj-Napoca, Romania
www.coneural.org/florian
florian@coneural.org

Version 1.1
September 1, 2005

www.thyrix.com

Contents

1 Introduction

Thyrix is a very fast agent / environment simulator. It was developed to be used for embodied artificial intelligence or artificial life research, especially for evolutionary experiments. The simulator can also be used as a simple physics engine for games.

1.1 Features

The simulator has the following features:

- C++ code, optimized for speed;
- Object primitives: circles, capped rectangles, borders (semi-planes);
- Composite objects, articulated agents;
- Contact detection and resolution;
- 2-dimensional space;
- Quasi-static dynamics;
- Visual sensors;
- Tactile sensors integrated with contact detection;
- User interaction by dragging objects with the mouse;
- Cross-platform portable graphical user interface;

1.2 License

The simulator is freely available under the GNU General Public License (GPL)[1]. Special licenses for commercial usage are available upon request.

1.3 How to install and use Thyrix

The simulator is written in C++. The simulation engine is stand-alone and does not require any non-standard external libraries.

Thyrix includes code that helps building graphical interfaces for the simulator, and this code uses the wxWidgets framework. WxWidgets is a open source library that allows cross-platform compilation. Thus, graphical interfaces for the simulator based on this framework can be easily ported to any major operating system (including Windows, Linux, and MacOS). For installing and compiling the

[1] http://www.thyrix.com/lite/license.php

wxWidgets framework, please refer to the wxWidgets website [2]. Usage of wxWidgets is not mandatory, and users that prefer other APIs for the graphical interface can freely use them.

We provide files that define projects for compilation under Microsoft Visual C++ 6.0. The compilation of the source code was tested with Microsoft Visual C++ 6.0 and MinGW compilers, on Windows 98 and XP. The provided projects that use the wxWidgets framework need the system variable *WXWIN* to be set to the path to the wxWidgets libraries.

2 Purpose of the simulator

2.1 Embodied artificial intelligence

There is an increasing awareness among the scientific community that genuine intelligence (adaptable, flexible, robust) can emerge only in a system that is embodied (i.e., has a body through which can interact with the external environment, using sensors and effectors), and is situated in an environment it can interact with (e.g., Steels and Brooks, 1995; Pfeifer and Scheier, 1999; Brooks, 1990, 1991; Bickhard, 1993; Varela et al., 1992; Chiel and Beer, 1997; Ziemke, 2001; Florian, 2003). The essential implication of embodiment is the bidirectional, circular interaction between the body of the cognitive agent and the environment: some of the agent's actions change the state of the environment, thus changing also the influence of the environment on the agent (partly perceived through the sensors). This coupling permits the exploration by the agent of the structure of the environment and the discovery of structural invariants, through a process which depends on the sensorimotor capabilities of the agent and its goal. The agent can thus develop its own conceptualization of the environment, through self-organization and learning. The grounding of concepts on the sensorimotor interaction with the environment eliminates the problems of classical artificial intelligence (lack of robustness; the lack of access to the semantic content of designer-provided symbols or categories; the confusion between the agent's perspective and the observer's perspective).

2.2 Simulated embodied agents

While embodiment generally implies a real physical body, like those of animals and robots, several studies (Quick et al., 1999; Oka et al., 2001; Riegler, 2002) have argued that the importance of embodiment is not necessarily given by materiality, but by its special dynamic relation with the environment. This relation can also emerge in environments other than the material world, such as computational ones. The environment can be a simulated physical environment, or a genuinely computational one, such as the internet or an operating system. Simulated physical environments may be connected to the sensors and effectors of real physical agents,

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

as in virtual reality, or may also simulate the body of the agent. Thyrix simulates both the agents and the environments with which they interact.

2.3 Requirements for a simulator suitable for embodied artificial intelligence research

Many experiments in embodied artificial intelligence research use navigation or object manipulation tasks. There are studies that argue that object manipulation capabilities ground higher cognitive abilities. Thus, a simulator suitable for this research area should minimally implement agents that are able to move within an environment and interact with objects within it.

Tactile perception is important in object manipulation. Simulating tactile perception implies detecting the contact between the tactile sensor and external objects. Since contact detection has high computational burden, simulation of tactile perception should be integrated with general contact detection procedures that avoid the interpenetration of simulated objects.

Many studies involve artificial evolution of controllers for embodied intelligent agents. This method implies repeated evaluations of the performances of the controllers, by letting the agents perform tasks in the environment. The complexity of these studies is limited by the long computing time needed for finding interesting solutions. Thus, these studies need fast simulators, in order to reduce the computational burden and the duration of the experiments.

3 Design and implementation of the simulator

The software was designed to be a simple simulator (or even the simplest, if needed) in which agents are capable of navigating in an environment populated with solid objects, and of interacting with these objects using articulated arms. The agents can grasp some of the objects with their arms and move them relative to their body, carry them from place to place in the environment, explore them haptically and visually. The agents have sensors for vision, proprioception, and also tactile sensors distributed on the surface of their body. As effectors, agents can control the angles of their joints. Agents may also have "rockets" that allow them to move in the 2D space.

Simplicity was desired because it implies computational efficiency (rapidity in simulations) and stability in operation. A simple simulator can also be extended, as needed. Computational efficiency is very important if the simulator is used to artificially evolve controllers, when many generations of agents have to be tested repeatedly in the environment.

3.1 Dimensionality of the space

The simplest physical environment that fits our purposes has a two-dimensional (2D) space. A 2D simulation is computationally much simpler / faster than a 3D

one, but retains the needed features of the environment (spatial relationships, the discreteness of the objects). The simulator was implemented in 2D, but can be extended to 3D in the future.

3.2 Objects

The simulated environment may contain solid articulated agents and other solid objects. We used as solid primitives circles and capped rectangles (rectangles having two opposite sides capped with semicircles). These primitives were chosen for the computational simplicity of detecting the contact between them.

The objects in the environment or the parts of the articulated agents may be composed of a single solid primitive, or of several primitives bound together to form a composed object.

3.3 Dynamics

For computational efficiency, we have not used an environment implementing the Newtonian dynamics of the real world ($\mathbf{F} = m\,\mathbf{a}$), but an environment with a simplified, Aristotelian (quasistatic) dynamics: the velocity $\mathbf{v}$ of an object of mass m directly depends on the force $\mathbf{F}$ applied to the object, according to $\mathbf{F} = m\,\mathbf{v}$. There is thus no inertial movement: a body moves as long a force is applied to it, but it immediately stops if no force moves it. Objects cannot be thus thrown in the environment simulated by Thyrix; they stop after the contact ceases, as in the real word they would stop if confronted with a large friction force. This is a radical change from the laws of real physics. However, this difference is not essential from a cognitive point of view. Actually, many noneducated people believe that the real world obeys Aristotelian laws: in elementary physics courses, children must be unteached the Aristotelian principles, sometimes with great pain (diSessa, 1982). Besides balls or other small objects thrown or kicked in the air or water, or on the surface of ice or other smooth surfaces, most other objects around us stop after one ceases pushing or pulling them, because of friction.

There are several computational advantages of using Aristotelian dynamics versus classical dynamics. The computation and integration of dynamical quantities are slightly simpler. We have no collisions to deal with, but only contacts. The treatment of friction is much simpler: dynamic friction in Aristotelian physics is equivalent with static-only friction in real (Newtonian) physics, and thus much easier to simulate computationally. Dynamic friction in real physics can yield configurations that are inconsistent or intractable computationally, and may require not only contact forces, but also contact impulses (besides collision impulses) (Baraff, 1991). The movement of objects in the environment is mainly initiated by the agents, so we can optimize the computations by not updating on each cycle objects that are out of the reach of the agents.

3.4 Contacts

The simulator detects and resolves the contact between the objects. The contact detection is integrated with the updating of the haptic sensors of the agents. For contact resolution, we have implemented a fast algorithm devised by Baraff (1994) and also an iterative algorithm inspired by the Gauss-Seidel method (Preda and Florian, 2005).

3.5 Articulations

The articulated agents have a body and one or more tree-shaped articulated limbs connected to the body. Each link of the articulated limb can rotate relative to the joint; the rotation angle can be limited to a predefined range.

The simulator does not allow loops in the structure of the articulated agent, because of the algorithm used. We have implemented a modified Featherstone-type algorithm (Featherstone, 1983, 1987), which allows the fast simulation of the dynamics of a chain of N articulated links in a computational time linearly proportional with N. Our implementation was based on the implementations described by McMillan (McMillan, 1994; McMillan et al., 1995*b*,*a*, 1996). We have changed the algorithm to comply with the characteristics of our environment (2D, rather than 3D; and Aristotelian dynamics, rather than Newtonian). The algorithm uses the so-called "spatial notation", where the corresponding angular and linear components of velocity, acceleration and force are combined in a single vector. In 3D, these vectors are 6-dimensional, with 3 angular components and 3 linear components. In 2D, these vectors are 3-dimensional, with one angular component and 2 linear components. We have also changed the algorithm to allow contact resolution for the parts of the articulated body. An algorithm similar to the algorithm implemented in Thyrix, but slightly slower, was subsequently published by (Kokkevis, 2004).

4 Packages

4.1 Package list

Thyrix includes the following packages:

- *Tools*: some classes that define basic tools;
- *ThyrixLite*: the basic version of the Thyrix simulator;
- *ThyrixPro*: additional classes of the Thyrix simulator, for simulating articulated agents;
- *Controller*: classes that define a basic agent controller that generates random action;

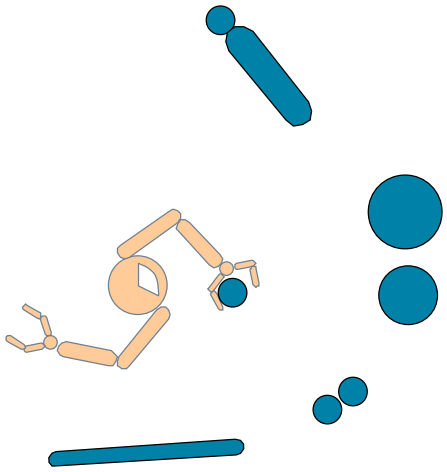

Figure 1: A screenshot of a Thyrix simulation. An articulated agent (Iunctus) catches a circular object with its limb.

- *GUIWx*: some classes that allow building a simple graphic interface for the simulator, using the wxWidgets framework;
- *Spherus*: an example of an agent developed using Thyrix Lite; demonstrated by the following applications:
  - *SpherusText*, an example console application that runs the Spherus agent;
  - *SpherusWx*, a wxWidgets graphical interface that runs the Spherus agent;
- *Pac*: another example of an agent developed using Thyrix Lite; demonstrated by:
  - *PacWx*, a wxWidgets graphical interface that runs the Pac agent;
- *Iunctus*: an example of an articulated agent developed using Thyrix Pro; demonstrated by:
  - *IunctusWx*, a wxWidgets graphical interface that runs the Iunctus agent;

4.2 Tools

This package includes the following classes or modules:

- *purgeContainer*: Purges a container (e.g. $std :: vector$) that contains pointers to objects, by deleting these objects, and emptying the container;

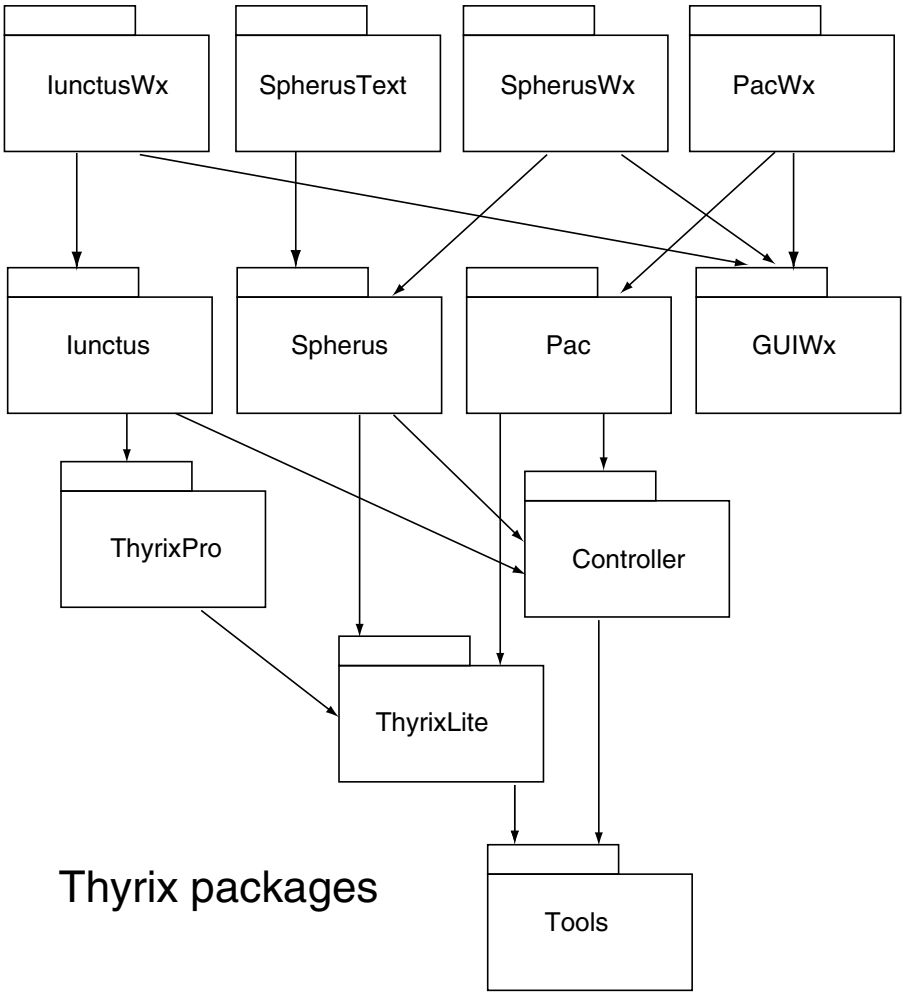

Figure 2: The Thyrix packages and their dependencies.

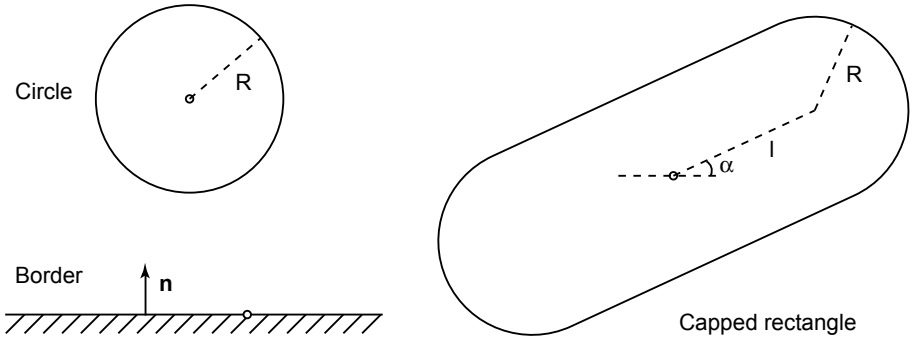

Figure 3: Geometrical primitives used by the Thyrix simulator.

- *Random*: provides easy generation of random numbers.
- *MTRandom*: an implementation of the Mersenne Twister random number generator.
- *MathTools*: provides several mathematical functions.

4.3 ThyrixLite

This package containes the core of the Thyrix simulator.

4.3.1 Basic data structures and modules

ThyrixParameters contains basic definitions. It defines the type *real*, which is the basic floating point data type used in the physics computations, and may be defined either as *float* or *double*.

Vector2 and *Vector3* define 2D and respectively 3D vectors. *Matrix3* defines a 3x3 matrix, while *SymmetricMatrix3* is a specialization of *Matrix3* for symmetric matrices.

4.3.2 Geometrical primitives

GraphicalObject is a base class for all objects that can be drawn (including physical objects, but eventually also elements like markers, contacts, forces, etc.).

PhysicalObject is a base class for all simulated physical objects.

The class *Border* defines a line that delimitates a semiplane containing the simulation area. The line has infinite length; it is defined by one point belonging to the line and the normal to this line. The location of the point on the line does not matter.

The class *Circle* defines a circle, characterized by its center and a radius.

The class *CappedRectangle* defines an elongated rounded object, a rectangle having the thin ends capped with half circles. The base position (relative to which

the rotation angle is measured) is along the x axis. l is the half length of the rectangle, R its half width. The end half circles have a radius R.

The class *ComposedPhysicalObject* defines a complex rigid object composed of several different graphic primitives (circles or capped rectangles).

The class *VisualSensor* defines a simple visual sensor, composed of unit sensors (pixels) arranged on a circular segment. The sensor is rigidly attached to an arbitrary object (e.g., the body of an agent). It currently has infinite range. It considers that circles and capped rectangles emit light. The activation of a unit sensor is proportional with the value of its view angle covered by an object.

4.3.3 Contact-related classes

ContactInfo is a class that stores, at the level of the object, the information about a contact on an object. It is characterized by the position $\mathbf{p}$ of the contact on the surface of the object, relative to the object's point of reference; by the direction $\mathbf{n}$ of the contact force (normal to the surface of the object) and by the sign σ of the direction of the contact force. The contact can be of type force (defines a contact force that acts on the object) or of type torque (a contact that appears on joints where the rotation angle is limited).

GlobalContactInfo is a class that stores globally information about contacts.

ContactSolver contains the algorithm for contact resolution.

4.3.4 The simulator

World is a base class defining a process that is updated with discrete time steps. It can be used with *GUIWx* to graphically monitor generic simulators.

Simulator is a class that governs the simulation, dealing with the interaction of objects and their evolution in time.

Integrator provides first-order integration for the movement of the objects.

4.3.5 Graphical user interface

GUI is an interface providing functionality for graphical illustration of the simulated agents and environment. *Color* and *ColorDefinitions* provide a cross-platform interface for using colors in the graphical user interface.

4.3.6 Elastoid

Elastoid is a base class for building agents composed of objects linked by elastic links. *ElasticLink* defines links used for this purpose.

4.4 ThyrixPro

This package contains advanced capabilities of the Thyrix simulator related to the fast simulation of articulated objects.

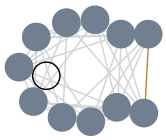

Figure 4: The Pac example agent. It is composed by several circles (filled in the picture) linked by elastic links. The open circle is an external object that was eaten by the agent. The orange link is the controllable link that allows the agent to open its mouth.

ArticulatedComponent is a component for articulated agents: it may represent the body of the agent, or (extended) an articulated link.

ArticulatedLink is a link that composes an articulated agent. A link has an articulation on which the agent can produce a torque.

ArticulatedLimb is a limb formed by a tree-like structure of articulated links.

ArticulatedAgentBase is a base class for articulated agents. Should be specialized for either classical mechanics or quasistatic mechanics.

ArticulatedAgentQuasistatic defines an articulated agent characterized by quasistatic (Aristotelian) physics. This class contains the Featherstone-type algoritm for solving the constraints imposed by the articulations.

LinkContactInfo is a data structure used in relation with the computing of the effects of contacts on articulated links.

4.5 GUIWx

This package provides an implementation of the graphical user interface for the wxWidgets multi-platform framework. It also provides a thread implementation, video buffering and basic interactivity with the user.

4.6 Controller

This package provides a controller that can move randomly the agents. In common usage of Thyrix, this controller can be overridden by a neural network or other intelligent controller.

4.7 Spherus

Spherus provides an example of an agent developed using ThyrixLite (see Fig. 6). Section 8 discusses in more detail the characteristics of this agent. Spherus is demonstrated by two programs. *SpherusText* is a console application that simulates the evolution of Spherus in an environment populated by circles for a finite number of timesteps and prints at each timestep the position of the center of the agent. *SpherusWx* illustrates graphically the simulation using the wxWidgets framework.

4.8 Pac

Pac (named after Pac-Man, the game character) is another example of an agent developed using ThyrixLite. It is composed of several circles linked with elastic links, in a circular structure. One link can be controlled by the agent, and thus it functions as a mouth, through which the agent can eat objects from the environment and then carry them inside its body (see Fig. 4). *PacWx* illustrates graphically a simulation of two PacAgents.

4.9 Iunctus

Iunctus is an example of articulated agent that can be developed by ThyrixPro (Iunctus means "articulated" in Latin; see Fig. 1). *IunctusWx* illustrates graphically the simulation using the wxWidgets framework.

5 Important data structures

Each object in the simulation should inherit from *PhysicalObject*. A physical object has the following properties: position $\mathbf{r}$, velocity $\mathbf{v}$, mass m, angle α, angular velocity ω. The linear properties (r, v) are the properties of the center of mass, relative to the laboratory reference system. The rotational (α, ω) properties refer to the rotation around the axis perpendicular to the simulation plane that passes through the center of mass of the object, relative to the laboratory reference system. The object also has a list of contacts, representing eventual contacts with other objects, and may have an array of tactile sensors on its surface, activated by the contact forces.

Each instance of the simulator uses an instance of the class *Simulator*. The objects inside the simulator (agents, as well as passive objects, like obstacles or landmarks) are contained in *Simulator::objects*. After creation, each object used in the simulation must be registered with the simulation using *Simulator::registerObject*; the method includes the object in *Simulator::objects*.

Simulator::contacts contains information about all contacts between the simulated objects, and is used for contact force computation.

6 How the simulator works

The simulator updates the simulated world in discrete timesteps. The method *Simulator::advanceTime* performs a complete update of the simulated world, corresponding to one timestep. The simulation unfolds by repeated calls to *Simulator::advanceTime*. During each timestep, the following actions are performed.

First, the old information regarding contacts is erased. This phase is performed here, and not at the end of the cycle, in order to keep in memory the information regarding contacts between cycles, and have it available for updating the graphical interface (for displaying contact forces and activations of contact or visual sensors).

The simulator then searches for contacts(*Simulator::detectContacts()*). If there are contacts, they are indexed (a number is given to each contact; *Simulator::indexContacts()*). If the user interacts with the simulation by dragging an object with the mouse, the corresponding force is applied to the dragged object (*Simulator::applyMouseForce()*).

Then, the simulator computes the velocities of the objects, ignoring the eventual contact forces, by calling the *computeDerivativesWithoutContacts* method of each object registered with the simulator (we use the word "derivatives" because in the case of quasistatic dynamics, currently implemented by the simulator, the method computes velocities, while in the case of real physics the method should compute accelerations). In the case that there are no contacts, these velocities are the final velocities of the objects. If there are contacts, these velocities will be used during contact force computation.

Contact forces are then computed. Using these forces, the simulator then computes the final velocities of the objects.

The simulator then calls the *controll* methods of the registered objects. For passive objects, this method will not do anything, but for agents or other objects that have a self-generated movement, the method will communicate to their controller the perceptual information, and will set eventual motor forces or torques according to the command of the controller (these forces will be applied in the next timestep). The *controll* method is called at this point during the timestep cycle because it may need information about contact forces (if the objects have tactile sensors) or proprioceptive information about the object velocity. Visual or proximity sensors were also updated during the contact detection phase.

Then, the simulator computes the new positions of the objects, given the previously calculated velocities, by calling the *integrate* method of the registered objects.

6.1 Treatment of penetrations

Currently, if penetrations between rigid objects is detected, the simulator computes contact forces that restrain the objects to interpenetrate further, and adds a small elastic penalty force, proportional to the penetration surface. Thus, penetrations are not treated systematically, e.g. by rolling back the time to the moment of contact. To avoid penetrations, the parameters of the simulations (the timestep, maximum forces, object masses) should be set such as the maximum velocity possible times the timestep is less than the tolerance for interpenetration.

6.2 Integration

The simulator currently uses simple Euler integration of the positions or angles of the objects. More complex integration can be used by modifying the integrator used by the simulator (*Simulator::integrator*).

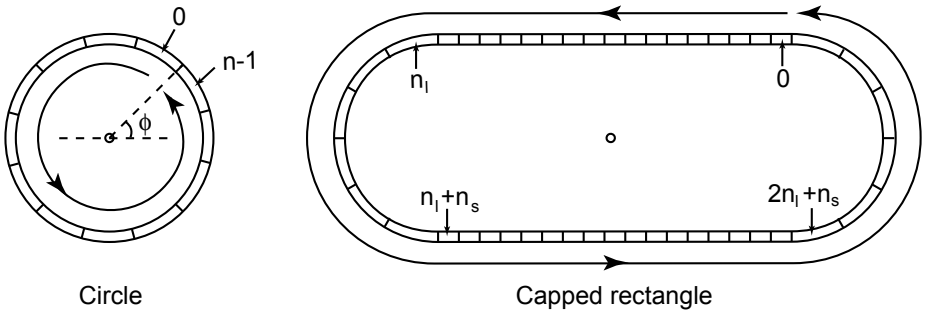

Figure 5: The position and numbering of tactile sensors. Circle: n is the number of sensors, ϕ is the sensors start angle. Capped rectangle: n_l is the number of lateral sensors (per side), n_s is the number of circular sensors (per semicircle). The figure illustrates the objects without rotation ($\alpha = 0$).

6.3 External forces

In this context, external forces/torques are forces/torques applied to the simulated objects, other than the contact forces/torques. An example of external force is the eventual force generated by the user by dragging an object with the mouse in the graphical interface. External forces/torques applied to objects should be accumulated in *PhysicalObject::externalForce*. They are taken into account by the simulator in the *PhysicalObject::computeDerivativesWithoutContacts* method, and are immediately reset to zero. Thus, external interactions should be updated at each time step, in the current setup (this can be changed quite simply).

6.4 Tactile sensors

Each geometrical primitive can have tactile sensors on its surface, that are activated when there are contacts with another objects (See fig. 5 for more information about the position of the sensors). The updating of the tactile sensors at each timestep is performed if the number of sensors *nSensors* is set to a positive value.

6.5 Bounding boxes

Objects have associated axis-aligned bounding boxes (with the lower-left and upper-right opposite corners given by the *boxMin* and *boxMax* vectors). These boxes are used to determine quickly whether two objects may be in contact or not, during broad-phase contact detection (culling). If the bounding boxes intersect, a more thorough investigation of the contact is performed, which takes into consideration the exact shape of the objects (see *Simulator::detectContacts*).

6.6 Further information

You may find more information about the simulator in the Reference Manual, generated from comments in the source code.

7 Usage guide

A user would typically perform the following steps when running an experiment based on Thyrix:

- Compile the Thyrix packages (if they are used for the first time);
- Create classes defining the morphology of the agents involved in the experiment;
- Create controllers for the agents;
- Create a simulator class that would define the experiment;
- Verify the behavior of the created agents in the simulated environment through a graphical interface;

The user may also run simulation sessions in batch mode to artificially evolve the agents, or analyze the behavior of agents using advanced tools or graphical user interfaces.

7.1 Creating the agents

The packages Spherus and Iunctus illustrate the creation of agents.

With ThyrixLite, one can build agents composed of a single solid object (a circle, a capped rectangle, or a rigid object composed of several circles and/or capped rectangles), or agents composed of several solid objects connected through elastic links (for example). Thus, an agent will typically inherit from *Circle*, *CappedRectangle*, *ComposedPhysicalObject*, or *PhysicalObject*. The objects composing the agents can be created to have tactile sensors, and proximity (vision) sensors can be also attached to the agent's body. The sensors are read by the controller in the *controll* method. The action of the forces corresponding to the motor command of the controller can be computed in the *computeDerivativesWithoutContacts* method, where the resulting velocities are applied to the components of the agent.

With ThyrixPro, one can build articulated agents, that will typically inherit from *ArticulatedAgentQuasistatic*. The agents need a body (which is a simple or composed object), to which are attached several limbs. Each limb can be composed of multiple articulated links and can have a tree-like structure.

7.2 Creating the simulators

The classes *SpherusSimulator* and *IunctusSimulator* illustrate the creation of simulators.

One would typically create, in the constructor of the simulators, the simulated objects, and register them with the simulator.

7.3 Creating graphical interfaces

The creation of graphical interfaces with the wxWidgets framework is illustrated by *GUIWx*, *SpherusWx* and *IunctusWx*.

8 The Spherus example agent

Spherus is an example of an agent that can be simulated with ThyrixLite.

8.1 The agent's morphology

Spherus's morphology was chosen as the simplest one which would allow the agent to push the circular objects in its environment without the slipping of the objects on the surface of the agent. This slipping may appear, for example, if a circle pushes another circle, and the pushing force is not exactly oriented on the line connecting the centers of the two circles.

We wanted maximum simplicity both for economy (to allow evolution and development in less computing time) and for having few degrees of freedom, which may allow dynamical analysis and simpler statistical analysis of the behavior of the agent. However, we have tried to respect the principle of ecological balance (Pfeifer and Scheier, 1999, pp. 455–463) in the design of the agent's morphology and sensorimotor capabilities.

Thus, the agent is composed of two circles, connected by a variable length link. The link is "virtual", in the sense that it provides a force that keeps the two circles together, but it does not interact with other objects in the environment, i.e. external objects can pass through it without contact. With this morphology, the agent can easily push other circles in its environments, by keeping them between its two body circles, without the need of balancing them to prevent slipping.

8.2 The agent's effectors and sensors

The agent can apply forces to each of its two body circles. The forces originate from the center of the circles and are perpendicular to the link connecting them. They may be considered to originate from some virtual "rockets". Two effectors correspond to each of the two body circles, one commanding a forward-pushing force, and one commanding a backward-pushing force. There are thus four "rocket" effectors. The net motor force acting on one body circle is the sum

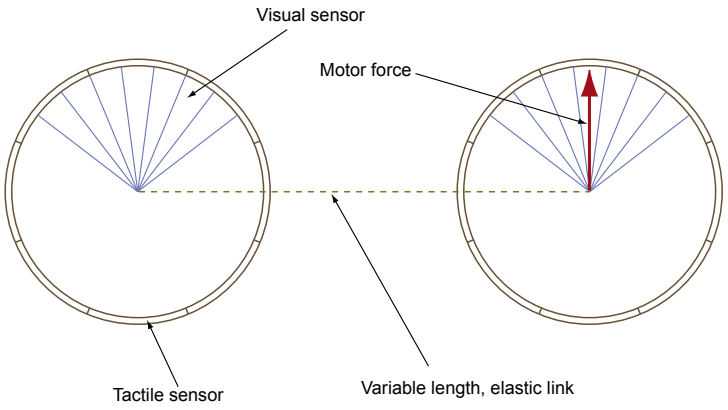

Figure 6: The agent's morphology.

of the forward and backward forces. These effectors allow thus the agent to move backward or forward, to rotate in place, and, in general, to move within its environment.

A fifth effector commands the length of the virtual link connecting the two body circles, between zero and a maximum length. If the actual length of the link is different from the commanded length, an elastic force (proportional with the difference between the desired and actual length) acts on the link, driving it to the desired length.

The agent has contact sensors equally distributed on the surface of its two body circles (8 contact sensors per circle, spanning a 45° angle each). The activation of the sensors is proportional to the sum of the magnitudes of the contact forces acting on the corresponding surface segment, up to a saturation value.

Each circle also has 7 visual sensors. Each sensor has a 15° view angle, originating from the center of the circle. Thus, each circle has a 105° viewing angle, centered around the "forward" direction. The activation of the sensors is proportional to the fraction of the viewing angle covered by external objects. The range of the sensors is infinite.

The agent also has proprioceptive sensors corresponding to the effectors. Each body circle has two velocity sensors, measuring the velocity in the forward and backward directions, respectively. The sensors saturate at a value corresponding to the effect of the maximum motor force that can be commanded by the effectors. The agent also has a proprioceptive sensor that measures the actual length of the link connecting the two body circles, that saturates at the maximum length that can be commanded by the link effector.

Thus, the agent has a total of 5 effectors and 35 sensors (16 contact sensors, 14 visual sensors, and 5 proprioceptive ones). Each sensor or effector can have an activation between 0 and 1.

8.3 Demonstrations

Spherus is demonstrated by two programs. *SpherusText* is a console application that simulates the evolution of Spherus in an environment populated by circles for a finite number of timesteps and prints at each timestep the position of the center of the agent. *SpherusWx* illustrates graphically the simulation using the wxWidgets framework.

9 Acknowledgements

The development of the simulator was supported by Arxia (http://www.arxia.com/). The simulator was developed by Razvan Florian, with contributions from Mihai Preda and Sorin Stan.

References

Baraff, D. (1991), 'Coping with friction for non-penetrating rigid body simulation', *Proceedings of SIGGRAPH 1991, Computer Graphics* **25**, 31–40.
http://www-2.cs.cmu.edu/~baraff/papers/sig91.pdf

Baraff, D. (1994), Fast contact force computation for nonpenetrating rigid bodies, *in* A. Glassner, ed., 'Proceedings of SIGGRAPH 1994, Computer Graphics Proceedings, Annual Conference Series', ACM Press, pp. 23–34.
http://www-2.cs.cmu.edu/~baraff/papers/sig94.pdf

Bickhard, M. H. (1993), 'Representational content in humans and machines', *Journal of Experimental and Theoretical Artificial Intelligence* **5**, 285–333.
http://www.lehigh.edu/~mhb0/repconpage.html

Brooks, R. A. (1990), 'Elephants don't play chess', *Robotics and Autonomous Systems* **6**, 3–15.
http://www.ai.mit.edu/people/brooks/papers/elephants.pdf

Brooks, R. A. (1991), 'Intelligence without representation', *Artificial Intelligence Journal* **47**, 139–159.
http://www.ai.mit.edu/people/brooks/papers/representation.pdf

Chiel, H. J. and Beer, R. D. (1997), 'The brain has a body: Adaptive behavior emerges from interactions of nervous system, body and environment', *Trends in Neurosciences* **20**, 553–557.
http://vorlon.ces.cwru.edu/~beer/Papers/TINS.pdf

diSessa, A. (1982), 'Unlearning aristotelian physics: A study of knowledge based learning', *Cognitive Science* **6 (2)**, 37–75.

Featherstone, R. (1983), 'The calculation of robot dynamics using articulated-body inertias', *The International Journal of Robotics Research* **2**, 13–30.

Featherstone, R. (1987), *Robot dynamics algorithms*, Kluwer Academic Publishers, Boston, MA.

Florian, R. V. (2003), Autonomous artificial intelligent agents, Technical Report Coneural-03-01, Center for Cognitive and Neural Studies, Cluj, Romania.
http://www.coneural.org/reports/Coneural-03-01.pdf

Kokkevis, E. (2004), 'Practical physics for articulated characters', Game Developers Conference 2004.
http://www.research.scea.com/research/pdfs/VangelisK_GDC2004.pdf

McMillan, S. (1994), *Computational dynamics for robotic systems on land and under water*, Ph.D. Thesis, The Ohio State University, Columbus, OH.
http://dynamechs.sourceforge.net/Papers/dissertation.tar.gz

McMillan, S., Orin, D. E. and McGhee, R. B. (1995*a*), Dynamechs: An object oriented software package for efficient dynamic simulation of underwater robotic vehicles, *in* J. Yuh, ed., 'Underwater Robotic Vehicles: Design and Control', TSI Press, pp. 73–98.
http://dynamechs.sourceforge.net/Papers/yuh_chapter.ps.gz

McMillan, S., Orin, D. E. and McGhee, R. B. (1995*b*), 'Efficient dynamic simulation of an underwater vehicle with a robotic manipulator', *IEEE Transactions on Robotics and Automation* pp. 606–611.
http://dynamechs.sourceforge.net/Papers/ab_inertia.ps.gz

McMillan, S., Orin, D. E. and McGhee, R. B. (1996), 'A computational framework for simulation of underwater robotic vehicle systems', *Journal of Autonomous Robots* **3**, 253–268.
http://dynamechs.sourceforge.net/Papers/jar95.ps.gz

Oka, N., Morikawa, K., Komatsu, T., Suzuki, K., Hiraki, K., Ueda, K. and Omori, T. (2001), Embodiment without a physical body, *in* R. Pfeifer, G. Westermann, C. Breazeal, Y. Demiris, M. Lungarella, R. Nunez and L. Smith, eds, 'Proceedings of the Workshop on Developmental Embodied Cognition, Edinburgh', Edinburgh, Scotland.
http://www.cogsci.ed.ac.uk/~deco/posters/oka.pdf

Pfeifer, R. and Scheier, C. (1999), *Understanding intelligence*, MIT Press, Cambridge, MA.

Preda, M. and Florian, R. V. (2005), Simple iterative algorithm for contact force computation, Technical Report Coneural-05-01, Center for Cognitive and Neural Studies, Cluj, Romania.
http://coneural.org/reports/Coneural-05-01.php

Quick, T., Dautenhahn, K., Nehaniv, C. and Roberts, G. (1999), 'The essence of embodiment: A framework for understanding and exploiting structural coupling between system and environment', *Proceedings of the Third International Conference on Computing Anticipatory Systems, Liege, Belgium. August 9-14, 1999 (CASYS'99)* .
http://homepages.feis.herts.ac.uk/~comqkd/quick_casys99.ps

Riegler, A. (2002), 'When is a system embodied?', *Cognitive Systems Research* **3**, 339–348.
http://pespmc1.vub.ac.be/riegler/papers/riegler02embodiment.pdf

Steels, L. and Brooks, R., eds (1995), *The artificial life route to artificial intelligence: Building embodied, situated agents*, Lawrence Erlbaum Associates, Hillsdale, NJ.

Varela, F. J., Thompson, E. and Rosch, E. (1992), *The embodied mind: Cognitive science and human experience*, MIT Press, Cambridge, MA.

Ziemke, T. (2001), 'The construction of 'reality' in the robot: Constructivist perspectives on situated artificial intelligence and adaptive robotics', *Foundations of Science* **6**, 163–233.
http://researchindex.com/ziemke00construction.html