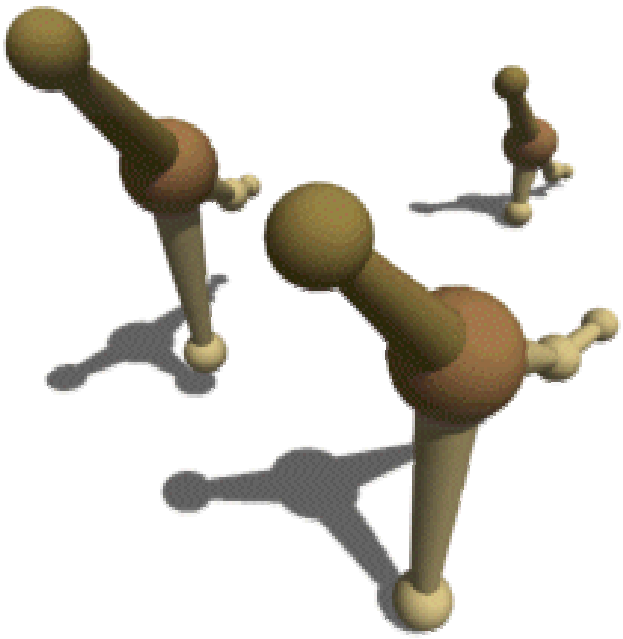
Autonomous Dynamically Simulated Creatures for Virtual Environments

Paul Urban (urban@rucus.ru.ac.za)

Computer Science Honours 2001



Submitted in partial fulfilment of the requirements for the degree of

Bachelor of Science (Honours) of Rhodes University.

Abstract

In this thesis we explore a solution to the problem of creating virtual creatures with autonomous behaviour and physically realistic motion. This solution is to construct artificial animals (animats) with simulated physical bodies, sensors, actuators and a suitable control mechanism.

We designed and implemented a modular, extensible object-oriented dynamics engine for the *greatdane* VR system. We used this engine to build and simulate two animats using dynamics primitives such as spherical masses, springs and springy angular joints. The two animats are a “roobot” – a monopod that hops on the ground and a jellyfish that swims in a simulated underwater environment. Pure finite state machines control both animats. Notably, we describe an experiment in which we used a genetic algorithm to evolve stable and fast hopping motion for the roobot.

Our models produce physically realistic motion and are sufficiently simple to be simulated and rendered in real- to near real-time on a dual Intel Pentium III 800 MHz PC, making them suitable for use in interactive virtual environments.

From experience of designing and implementing animats using this approach, we conclude that an engineering-style approach is necessary, in that strong dynamics knowledge is critical and care must be taken to create models that are simple enough to control and simulate in real-time.

Contents

1 Introduction

Virtual reality researchers are seeking to make their worlds more engaging and realistic. One developing area of study is the incorporation of animals into virtual environments.

Arguably the biggest hurdle to overcome in the design of virtual animals is that their behaviour and motion cannot be completely scripted. In traditional computer animation the animator specifies the state of the system at key instants (so-called keyframing) and the computer interpolates between them. This technique works well for short animation sequences in which the animal behaviour is finely controlled and directed, but is not useful in fully interactive virtual environments. Animals in VR should react to unpredictable events that occur around them. What's more, they should interact with their environment, and do so realistically, if there is any hope to make the user fully immersed. The motion of the creature should obviously occur in real-time because it exists in a virtual environment that is potentially populated by human spectators.

To create suitable animals for virtual environments, it is necessary to take a radically different approach to that of laborious keyframing animation. This approach is to create complete artificial animals (animats). The animats are built from the "physics-up" by first defining a simulated physical body, then adding simulated sensors and actuators and finally adding a control mechanism to act as a simulated brain, controlling the body. This project focuses on creating creatures using this method, with emphasis on producing physically realistic motion. Different approaches to producing motion exist, but only a few are suitable for producing real-time interactive motion. Other researchers have demonstrated that when using suitable dynamics models, this approach to creature creation can indeed yield motion and behaviour of unprecedented complexity and realism.

We now outline the basic architecture we have chosen to use for our animats, and then briefly introduce the fields of physically based modelling, artificial life and autonomous mobile robots, from which we draw background theory.

1.1 Anatomy of an Animat

The animat architecture we have chosen is a simplification of that used by Tu [Tu, 1996] for the creation of virtual fish, and views the essential components of an animat (for the purposes of animation) as three interacting layers (Figure 1).

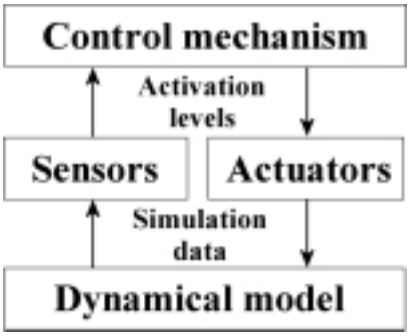


Figure 1. Functional layer diagram of an animat.

Construction of an animat starts at the lowest layer and proceeds upwards. First, in the "dynamical model" layer, a dynamical system representing the creature's body is defined. When the animat is placed in a virtual environment, a dynamics engine takes care of animating the model through time according to physical laws. Next, a sensor-actuator layer is added above this. Sensors allow the creature to obtain information about itself and its environment, and actuators allow it to control its body. Finally a control layer, the creature "brain", is added at the top. This topmost layer takes sensory data as input and responds by producing actuator commands as output. In practice this control layer is sub-divided when creating higher-order creatures. Each section in this layer may manage a function such as intention generation, attention control or task-level planning.

1.2 Background Theory

This research draws on theory and results from the three fields of physically based modelling, artificial life and autonomous mobile robots. We now briefly introduce these fields and indicate their relevance to this project.

1.2.1 Physically Based Modelling

The study of the motion of physical systems in response to applied forces is termed "dynamics". Although physics encompasses a vast amount of theory, only a small subset of dynamics has actually been applied to simulation in computer graphics. In

particular, it is used to automatically animate dynamical systems in strict accordance with a set of physical laws (Newton's laws).

In this project, physically based modelling theory is used to implement a dynamics engine that animates models of creature bodies (and other systems) according to the laws of dynamics.

1.2.2 Artificial Life

Artificial Life (ALife) is concerned with synthesizing artificial systems that resemble biological ones. ALife researchers construct models of living and imaginary biological systems at various levels of abstraction, from individual cells to entire populations of organisms.

The ALife approach is invaluable to biologists because simulated systems are more easily manipulated and studied than real ones, allowing thorough testing of biological theories.

Our work on creating artificial creatures for virtual environments falls into this broad field.

1.2.3 Autonomous Mobile Robots

Mobile robotics is a field that is beginning to attract a lot of attention. Autonomous mobile robots are robots that can navigate through their environment with little to no human intervention.

A lot of work is being done into designing mechanisms that control a robot's actuators in response to its sensors. In creating our creatures we face similar challenges to those creating autonomous mobile robots. Specifically, our creatures possess simulated physical bodies and have simulated sensors and actuators, so we use similar control mechanisms.

Interestingly, many mobile robots developed in recent years were in fact simulated on computers prior to being built, in order to test and refine their control mechanisms.

1.3 Overview of this Thesis

Chapter 1 introduces the problem of creating suitably autonomous and realistic animals for virtual environments, and outlines the approach investigated. Chapter 2 describes and evaluates related work in the fields of artificial life and physical modelling. Chapter 3 describes the design of the dynamics engine that was added to greatdane; the three animats created and the experiments conducted with them. Chapter 4 discusses the implementation of the dynamics engine and the three animats. Chapter 5 presents the results of the experiments and some observations made. Chapter 6 summarises the project and lists the conclusions reached.

2 Related Work

This chapter starts with a brief description of the *greatdane* VR system on which this work was implemented as well as some previous work on physics modelling that was performed with it.

The bulk of the chapter is devoted to a more thorough introduction to the fields of artificial life and physical modelling. For each field we start by describing its basic philosophy and theory and then proceed to outline and analyse previous research and its applicability to creating animats for real-time interactive virtual environments.

2.1 Greatdane

VR research conducted in the Computer Science department at Rhodes University is supported by the local Virtual Reality Special Interest Group (VRSIG). VRSIG members work on a shared VR software system developed at Rhodes University, called *RhoVeR* [Bangay et al, 1996]. *RhoVeR* was created to serve as a platform for developing virtual environments and testing critical aspects of VR.

The *RhoVeR* code has gone through many revisions since it was first released in 1996. This project was done during the course of 2001, using the *greatdane* release. The *greatdane* release is written primarily in Java, but certain critical functions such as device interfacing are written in C and C++. *Greatdane* development is done with Linux. The strong component architecture of *greatdane* makes it particularly extensible. In addition, the basic functionality is encapsulated in a number of shared libraries (e.g. basis, vr, network and audio), which are simply linked into VR applications.

2.1.1 Physically Based Modelling

Previous VRSIG members have explored Physically Based Modelling with *RhoVeR*. Dembovsky [Dembovsky, 1996] implemented a system for collision detection and simple physical modelling in *RhoVeR* and used it to create a three-dimensional table-tennis simulation. In Dembovsky's system all objects involved in the physics simulation are derived from a class known as *VRPhysicsEntity*, which in its definition refers to specific forces such as gravity and supports interactions between only a

limited number of object types. The concepts of mass and force are not completely formalised and separated in the object model. In addition, every object and its behaviour are implemented in a fairly ad-hoc manner. Finally, only first-order integration is supported by the *VRPhysicsEnvironment*, and no provision is made for trial integrations used by higher-order integrators.

For creating virtual creatures with simulated bodies a system that can represent and simulate fairly complex mechanisms in arbitrary configurations is needed. In this project we design and implement a completely separate dynamics engine in the component-based spirit of *RhoVeR* and use it to build and simulate our creatures.

2.2 Artificial Life

Our effort towards creating virtual creatures falls into the young field known as Artificial Life. We now attempt to define the Alife movement and then introduce some of the major branches of ALife research. Along the way we describe and evaluate previous work.

2.2.1 ALife Defined

The Artificial Life (ALife) movement has been defined by Langton [Langton et al, 1992] as:

"…a field of study devoted to understanding life by attempting to abstract the fundamental dynamical principles underlying biological phenomena, and recreating these dynamics in other physical media -- such as computers -- making them accessible to new kinds of experimental manipulation and testing.
...
In addition to providing new ways to study the biological phenomena associated with life here on Earth, *life-as-we-know-it,* Artificial Life allows us to extend our studies to the larger domain of "bio-logic" of possible life, *life-as-it-could-be*..."

The assumption implicit in Langton's definition, often referred to as "the strong ALife principle", is that one can separate the logical form of an organism from its material base, and that its "aliveness", its capacity to live and reproduce, is a property of the form, not the matter [Emmeche, 1992]. The question of whether the systems created

by ALife researchers can really be considered “alive” is certainly a contentious one. The “weak ALife principle” by contrast merely asserts that it is possible to create systems that resemble life in certain respects. The work done in this project relies on the weak principle alone.

Other researchers have noted that ALife it is a truly multi-disciplinary endeavour [Ronald et al, 1999]:

“Some researchers aim at evolving patterns in a computer; some seek to elicit social behaviors in real-world robots; others wish to study life-related phenomena in a more controllable setting, while still others are interested in the synthesis of novel lifelike systems in chemical, electronic, mechanical, and other artificial media.”

Some early work in ALife was performed by a mathematician, John Conway, who invented the Game of Life [Internet: GOL]. The Game of Life is a 2-dimensional grid of cells, each of which can be in one of two states, alive or dead. On each iteration all dead cells with exactly three live neighbours become live and all live cells with less than 2 or more than three live neighbours become dead. Through the repeated application of these simple rules alone incredibly complex patterns can emerge. Conway’s Game of Life is one example of a class of systems called Cellular Automata.

Although low-level models such as cellular automata aren’t suitable for building complete creatures, the work and philosophy of early pioneers such as Langton and Conway are the inspiration for present ALife work.

2.2.2 Animats

One major branch of ALife concerns the construction of artificial organisms (animats). Much in-depth research into their design and implementation has been done.

Karl Sims [Sims, 1994] explored an interesting way to create animats, which is to run genetic algorithms for generating novel animat forms and behaviours simultaneously

on massive supercomputers (a Connection Machine was used). In his system, each creature has a genotype (in the form of a directed graph), one for its body and another for its brain. To create a new individual, the genotype is decoded into a specific phenotype. The body genotype encodes an articulated system of blocks connected by joints. The brain genotype encodes a generalised neural network as well as sensors and effectors (actuators). The individual is simulated in a physical world (land and water worlds were used) and its performance is evaluated according to some criterion.

Sims evolved a menagerie of weird and wonderful creatures that can walk, jump, swim and compete for control of a cube[1].

Completely automatic methods of creating creatures are attractive in that little designer intervention is necessary beyond specifying the criterion on which to evaluate the trial creatures. As far as virtual environments go though, the majority of the time the creator will want to populate them with animats that actually resemble well-known living creatures. Nevertheless, genetic algorithms are a powerful tool for automating certain phases of creature creation.

A lot of work has gone into creating realistic virtual humans. This area is led by The Centre for Human Modelling and Simulation at the University of Pennsylvania [Badler et al, 1995]. They have created a digital human actor, *Jack*, which moves its jointed body in real-time and with realistic degrees of freedom[2].

Jack supports forward and inverse kinematics, real-time balancing, collision avoidance and multiple goal-directed behaviour control algorithms (among other things). It has been used recently in interactive multi-user environments and has proved invaluable in ergonomics analysis and human factors engineering.

The *Jack* system has been used by a number of other researchers. Hodgins and her colleagues [Hodgins et al, 1995] have developed control algorithms for performing a

[1] Images available at: http://biota.org/ksims/blockies/

[2] Images available at: http://www.cis.upenn.edu/~hms/

range of athletic motions with significant dynamics, including running, bicycling and vaulting[3].

A number of advanced techniques for modelling the outer appearance of real creatures have been developed. Wilhelms et al have, for instance, developed a system for realistic anatomical modelling using simulated muscles, tissues and skin [Wilhelms et al, 1997]. Their system animates these components in response to motion of the animal's skeleton. They start by attaching muscles and masses of generalised tissue to the skeleton, and finish by covering this with a layer of skin[4]. Their technique faithfully re-creates the stretching and bulging behaviour of real skin and has been used in a large number of recent films.

Such realistic models are unfortunately far too complex to be used for real-time virtual environments, at least on standard desktop PCs. The models we can use are very primitive, but as technology advances, adding spectacular visual effects to our animats will become feasible.

Tu created a virtual marine world populated with various types of autonomous fish with realistic appearance, movement and behaviour [Tu, 1996]. The fish were modelled as actuated spring-mass-damper networks to provide flexibility. The fish swam through a simulated water medium via simulated hydrodynamic forces. The model was computationally efficient enough to run ten artificial fishes, fifteen food particles and four static obstacles at about 4 frames/s using a Silicon Graphics R4400 Indigo Extreme workstation[5]. Tu gave her fish a set of simple behaviours and an attention and action selection mechanism to switch between them. She created the behaviours by hand. Each behaviour called a number of parameterised motor controllers that she optimised.

A very positive result in Tu's work is that the fish were completely autonomous and required zero animator intervention during the simulation. One could, if desired,

---

[3] Images available at: http://www.cc.gatech.edu/gvu/animation/

[4] Images available at: http://www.cse.ucsc.edu/~wilhelms/fauna/Monkeys/index.html

[5] Images available at: http://www.dgp.utoronto.ca/people/tu/images.html

direct a fish at a very high level by changing some of its internal parameters such as its mood and desires.

Another result that Tu demonstrated is that a damped spring (a viscoelastic unit) is a fair approximation to a real muscle, and a versatile element for building all sorts of physical models. By varying the relaxation length of the spring, tension will be generated that will tend to lengthen or shorten it. We have made heavy use of this idea in the construction of our creatures.

2.2.3 Autonomous Mobile Robots

Autonomous mobile robot designers are faced with similar challenges to animat designers and hence we can borrow ideas that they have already applied successfully.

With regards the creation of autonomous agents, Pattie Maes observes [Internet: AL Entertain]:

“The Artificial Life community has initiated a radically different approach towards this goal which focuses on fast, reactive behavior, rather than knowledge and reasoning, as well as adaptation and learning.”

In the early years of robotics and artificial intelligence (pre-1985), a strict planning approach was taken to achieving goals. After observing the outside world through sensors and building an internal representation of it, a course of action that would achieve the desired goal would be methodically planned and executed. The new approach is often termed “bottom-up” or “behaviour-based” AI as opposed to “top-down” or “knowledge-based” AI. In the field of mobile robotics, the main proponent of this approach has been Brooks [Brooks, 1985], who asserts that we should “use the world as its own best model”. He asserts that building fast, simple, reactive mechanisms that can be connected and layered is the correct approach to take when designing autonomous agents. This approach is largely biologically inspired. Brooks introduced a new distributed control mechanism that he calls the subsumption architecture, and used it to construct a number of autonomous lab robots[6].

[6] Images of some of the robots at: http://www.ai.mit.edu/projects/mobile-robots/robots.html

Essentially, a number of augmented finite state machines perform simple behaviours, but they have connections between them to inhibit or suppress each other. Each finite state machine typically performs a simple “reflex”, such as swinging a leg to attempt to maintain balance. New levels of behaviour can be added as higher-layer networks that selectively subsume some of the lower layer’s functions.

Very little work on implementing the subsumption architecture on animats appears to have been done, presumably because higher-level capabilities (such as complex task-selection) are difficult to implement directly. The “bottom-up” approach to robot design and control is still relevant to animat creation.

A more traditional engineering-style approach to creating control algorithms has been developed by Raibert and Hodgins [Raibert et al, 1991]. They have successfully implemented control algorithms for stable legged locomotion of simulated and real autonomous mobile robots at the MIT leg lab.

Their approach was to separate the robot gaits into distinct phases and use a finite state machine to switch to the appropriate control algorithm for each phase. They exploited symmetry to extend the single-leg control algorithm to creatures with more legs, with only minor changes. We use this idea to control our own simulated hopping monopod.

A promising future control mechanism is a pure neural network. Hugo de Garis [de Garis, 1990] used a genetic algorithm to evolve time-dependent neural networks that were used to control a pair of stick legs. He managed to teach the neural network how to get the stick legs to walk. One disadvantage of using neural nets is that they are notoriously difficult to analyse and debug. At present, more traditional designed control algorithms are preferred.

2.2.4 Autonomous Agents

Pattie Maes describes adaptive autonomous agents as follows [Maes, 1994]:

"An agent is a system that tries to fulfil a set of goals in a complex, dynamic environment. An agent is situated in the environment: it can sense the environment through its sensors and act upon the environment through its actuators.
…An agent is called autonomous if… it decides itself how to relate its sensor data to motor commands in such a way that its goals are attended to successfully. An agent is said to be adaptive if it is able to improve over time."

She identifies two related subproblems that must be solved in order to create effective agents, namely *action selection* and *learning from experience*. Action selection may be described as deciding what to do next so as to progress towards reaching a set of goals. Learning from experience is the process of improving future performance based on past experience. Successful animats in virtual environments should necessarily solve both of these problems.

Learning from experience implies that the animat should change its internal state in response to the apparent success of its past interactions with its environment. For instance, if a course of action was chosen that was followed by an undesirable effect, that course of action should be avoided in future. Neural Networks are ideally suited to this because methods for implementing on-line learning (as the simulation progresses) exist.

The creators of the commercial game, Creatures, have implemented an advanced neural network brain for their Norns, with regions specialising in separate functions [Grand et al, 1996]. Their Norns learn on-line by interacting with their environment (the user can train the creatures by administering reward and punishment). They have gone so far as to incorporate a fairly complex biochemical simulation, complete with organs that produce and consume chemicals selectively, chemical reactions and interactions, glands and receptors. These techniques are well beyond the scope of this project, but these promising results indicate that creating highly realistic animats for virtual environments is almost certainly achievable, just not with current hardware.

2.2.5 Genetic Algorithms

Genetic algorithms (GAs) arise from the idea of applying the biological principle of evolution to artificial systems [Internet: GA]. GAs have been successfully applied to numerous problems from different domains, including optimisation, automatic programming, machine learning, economics, operations research, ecology, population genetics, studies of evolution and learning, and social systems.

A genetic algorithm is an iterative procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols, known as the genome, encoding a possible solution in a given problem space. This space, referred to as the search space, comprises all possible solutions to the problem at hand. Generally speaking, the genetic algorithm is applied to spaces that are too large to be exhaustively searched.

The Darwinian model of evolution consists of three phases, selection, crossover and mutation. Genetic algorithms usually have corresponding phases.

An initial population of individuals is generated at random or heuristically. At every evolutionary step, known as a generation, the individuals in the current population are decoded (from genotype to phenotype) and evaluated according to some predefined quality criterion, referred to as the fitness function. To form a new population (the next generation), individuals are selected according to their fitness. After selection, additional operations such as crossover and mutation are performed. Crossover exchanges portions of two selected individuals' genomes. Mutation randomly alters portions of the genome.

In this project we have used a genetic algorithm to perform parameter optimisation that would have been extremely time-consuming to achieve manually.

2.3 Physical Modelling

Our animats are constructed from the "physics-up", meaning that we start by defining a physical model of the creature's body. We describe some of the theory from physically based modelling that we have used in this project as well as the issues of

importance when constructing the various layers of animats with simulated physical bodies.

2.3.1 Physically Based Modelling

Dynamics is the study of force and motion. Physically Based Modelling (PBM) is an important new approach in computer graphics, concerned with methods of simulating dynamical structures for the purposes of animation [Baraff et al, 1999]. Systems are animated by calculating all the forces present and solving the equations of motion.

Dynamical systems can be modelled in many different ways, with varying results in terms of realism and computational complexity.

Some of the most important topics (in approximate order of difficulty) are:

- Numerical integration: how to integrate the equations of motion to yield animation.
- Particle systems: how to represent and animate systems of simple movable particles with mass.
- Rigid bodies: how to represent non-deformable, rotatable objects with shape and volume.
- Collision: how to detect and resolve collisions.
- Contact: how to model resting contact and friction between objects.
- Constraints: how to constrain the system's behaviour (e.g. joints).

When constructing a dynamical simulation these issues must be kept in mind. In the following three sections we describe some of the theory of relevance to the dynamics engine we created and used to build our animats.

2.3.1.1 Numerical integration

Numerical integration is a powerful mathematical technique that can be applied to all sorts of problems, including solving the equations of motion of dynamical systems [Baraff et al, 1999]. In this section we introduce numerical integration in the context of the equations of motion of idealised point masses. It must be stressed that we are solving the equations of motion of a single non-rotational point mass particle.

In Newtonian mechanics each point mass has at least these five values (Table 1):

Symbol	Description
m	Mass
x	Position
v	Velocity
a	Acceleration
f	Force

Table 1. Basic attributes of point masses.

The attribute symbols above are in bold to indicate that they are vectors in general. The mass is generally a constant. The position, velocity and acceleration are related as follows:

$$\dot{\mathbf{x}} = \mathbf{v} \qquad \text{Eq. 1}$$

$$\ddot{\mathbf{x}} = \dot{\mathbf{v}} = \mathbf{a} \qquad \text{Eq. 2}$$

The notation above means that velocity is the first derivative of position, and acceleration is the second derivative of position. A derivative in this case is an instantaneous rate of change with respect to time. Newton's third law states that acceleration and applied force are related as follows:

$$\mathbf{f} = m\mathbf{a} \qquad \text{Eq. 3}$$

Re-arranging equation 3 and substituting equation 2 yields:

$$\ddot{\mathbf{x}} = \mathbf{f} / m \qquad \text{Eq. 4}$$

A differential equation is an equation that describes the relation between an unknown function and its derivatives. A solution is a function that satisfies the equation. In our case the unknown function that we are looking for is the position $\mathbf{x}(t)$, which varies with time, and the known derivatives are the velocity and acceleration:

$$\dot{\mathbf{x}} = f(\mathbf{x}, t) \qquad \text{Eq. 5}$$

$$\ddot{\mathbf{x}} = f(\mathbf{x}, \dot{\mathbf{x}}, t) \qquad \text{Eq. 6}$$

Equation 6 involves a second time derivative, making it a second order ordinary differential equation (ODE). For integration purposes it is useful to introduce an extra variable, **v**, and split this equation into two coupled first order ODEs (after substitution equation 4):

$$\dot{\mathbf{v}} = \mathbf{f} / m \qquad \text{Eq. 7}$$

$$\dot{\mathbf{x}} = \mathbf{v} \qquad \text{Eq. 8}$$

The variables **x** and **v** in equations 7 and 8 are the state of the point mass. For anything but trivial dynamical systems, analytical solution methods are impractical, and so numerical methods are used instead. Standard dynamical simulations fall into a class known as initial value problems. For each point mass we have an initial state (**x**, **v**) at some starting time, t_0 and wish to find the state at some later time, $t = t_0 + \Delta t$. We do this by simultaneously integrating the two state variable ODEs, equations 7 and 8.

The simplest numerical integration method is Euler's method. For each of the two differential equations we are solving (one for **x**, one for **v**), we calculate the new state variable **x**(t) after some time interval Δt using the equation:

$$\mathbf{x}(t_0 + \Delta t) = \mathbf{x}(t_0) + \Delta t \dot{\mathbf{x}}(t_0) \qquad \text{Eq. 9}$$

Euler's method is termed first order because it only uses derivatives up to first order. Higher-order methods such as the second order Midpoint method and the fourth order Runge-Kutta method also exist. These methods use equations similar to equation 9, but have additional terms tacked on to the end. These terms originate from the Taylor series, which is used to approximate any continuous function to arbitrary accuracy by simply using more terms from the series. A first order method stops after only two terms and is thus not a very faithful approximation [Baraff et al, 1999].

All numerical methods are merely approximations – the error in our integration methods is their order plus 1. Under some circumstances the approximation may be so bad that the system becomes unstable and explodes. Decreasing the time interval will improve the result, as will increasing the order. It is generally accepted that physical systems should be simulated using at least a fourth order method, such as the Runge-Kutta method. Even so, the maximum timestep that can safely be used before introducing severe numerical instability (in my experience it is **always** severe!) is finite and varies with the system. For virtual environments our integrator should preferably operate in or near real-time, so we are limited to choosing systems whose maximum safe timestep is larger than the inverse of the frame rate.

For each timestep a simple integrator typically:

- somehow calculates the nett force acting on every point mass;
- divides the nett force on each point mass by its mass to find its acceleration;
- integrates the velocity and position ODEs of each point mass over one timestep to yield its new state.

2.3.1.2 Particle systems

Particle systems are a useful way of model for a variety of systems. They are typically used to model objects with ill-defined edges, such as clouds, or any object that is made up of a number of particles that interact in complex ways [Reeves, 1983]. Particles may enter and leave the system at any time, and this process is usually controlled stochastically. Particle systems have been used to model fire, smoke, rain, snow and sand, amongst other things.

We have chosen to view the point masses in our dynamical models as the particles in a particle system, in which the interactions between the particles arise due to other entities, such as springs, that connect them. This model is the essence of simplicity and means that we can build and simulate virtually any "particle system"-like model.

2.3.1.3 An Object-Oriented Decomposition

It is possible to efficiently decompose classic dynamical systems into an object-oriented (OO) system for simulation purposes. The decomposition hinted at by Andrew Witkin [Baraff et al, 1999] has two main components:

- Particles (as described in 2.3.1.1) – objects with mass, dynamical attributes (position, velocity etc.) and any number of additional attributes.
- Force objects – objects with apply a force to a list of particles.

In addition there should be a system integrator that instructs the force objects to apply their forces to the particles they influence and then looks at the nett force on each particle and calculate its new dynamical state variables after a certain timestep. In addition, the system integrator should detect and handle events such as collisions.

Such OO decompositions are attractive because they are highly intuitive. We have extended Witkin's decomposition somewhat by formalising the roles of the numerical

integrator and event handler. Our system allows the user to declare any number of integrators and associate them with any number of point mass particles. In addition, new events such as collisions can be added without modifying the integrator, and new forces can be added at any time.

2.3.2 Creature Modelling

As we have shown in our earlier analysis of Artificial Life work, an animat is composed of a number of distinct layers, from physical model to control mechanism. A huge variety of mechanisms and algorithms have been used in each of these layers, with varying results. We will now discuss computationally simple models that have been used in real-time systems.

2.3.2.1 Finite Element Dynamics Models

Simulation on computer implies the use of numerical methods. There are at least two popular dynamics models, both of which are composed of a finite set of elements.

For models of creatures with skeletons, the elements are the bones that are modelled as rigid bodies, and these are connected with joints of varying degrees of freedom. For an in-depth discussion of rigid body and constrained dynamics, see [Baraff et al, 1999].

For deformable creatures or very simple structures, the elements are point masses, and these are connected with springs. This model has been put to good use for simulating a variety of objects including leaves, cloth and chains as well as creatures such as fish [Tu, 1996]. We make extensive use of this model, because of its computational simplicity and versatility.

When designing a finite element model of a real animal, care must naturally be taken to ensure that the model is a fair approximation to the original.

2.3.2.2 Sensors

Sensors should produce specific information about the creature’s body and/or surrounding environment. Sensor range and accuracy should be limited appropriately.

For the purposes of real-time simulation, sensors should perform as little unnecessary computation as possible. Following Tu's example, we make our sensory input highly abstract / high level. From a creature designer's point of view, the actual processing details should be ignored when designing the other layers of the creature.

We find it useful to formalise the sensors, so that they act as consistent interfaces to information from the dynamical simulation.

For compatibility with artificial neural networks, the sensory input should be one or more distinct real values within a strict range (e.g. [0..1]).

2.3.2.3 Actuators

Similar to sensors, actuators should have a certain strict operating range. Typically, each actuator takes an activation level within a certain range and adjusts some property in the dynamical simulation accordingly.

Tu has shown that the very springs used to define the creature body can also function as actuators that approximate muscle-pairs [Tu, 1996]. An actuator that controls a spring as if it were a muscle might take an activation level and adjust the relaxation length of the spring.

We extend the idea of springs with a relaxation length and stiffness constant to angular joints with a relaxation angle and an angular stiffness constant. Our angular joint model approximates an actuated joint. We also introduce the idea of a torque joint that applies a torque to one end-mass, about another mass and a torque axis. Our torque joint model approximates a motor. These objects can be used to build dynamical body models with very few components.

Once again the activation level provided to the actuator should fall into a strict range (e.g. [0..1]) for compatibility with neural networks.

2.3.2.4 Action Selection Mechanisms

Arguably the trickiest layer is the control layer, for this is where everything from reflexes to habits to reasoning and planning may occur.

As discussed earlier, the control layer may be anything from a pure network to a subsumption network. These two architectures are not optimal from a manual design and real-time simulation point of view.

Using Tu's fish model with a set of standard behaviours, the problem reduces to selecting which behaviour is appropriate at each instance and executing it. This is known as the "action selection problem". Tu used a decision tree to arbitrate between competing behaviours.

An even simpler architecture is to use a finite state machine to switch between behaviours based on simple ad-hoc rules. This approach is used by Raibert and Hodgins [Raibert et al, 1991], and is the one we adopt.

3 Design

In this chapter we describe the design of the dynamics engine and three animats built with it. We indicate the major design considerations and motivate the choices made. For the dynamics engine design section we describe the four central abstract interfaces and all the classes that implement each one. Later, in the animat design section we cover each of the animats in turn, by describing each of the layers in the animat model as well as any special issues concerning each animat.

3.1 Dynamics Engine

After an evaluation of Dembovsky's previous work on implementing a physics engine in *RhoVeR* (2.1.1) we set out to design and implement a separate new dynamics engine. The design criteria are that the engine be:

- Conceptually simple;
- Applicable to as many systems as possible;
- Computationally inexpensive.
- Extensible;

To meet the first three criteria the engine is designed to handle only particle systems. Each particle in the dynamical system is a point mass with a number of additional properties (volume, bounciness etc.). To meet the final criterion, abstract interfaces are used wherever possible, so that additional components can be added at a later stage. To thoroughly meet the first and last criteria, a fully object-oriented system design is followed. With the fourth criteria in mind it is decided to make the establishment of inter-object associations explicit, so that the user has full control of which forces apply to which objects, thereby minimising unnecessary computation. Starting from Witkin's description of the functioning of a particle system simulator [Baraff et al, 1999], we choose to define the following core objects in the engine:

- *PointMass* – a basic mass particle;
- *Force* – an object that calculates and applies a force to a number of PointMass objects;

- *DiscreteEvent* – an object that reports whether some discrete event (a very short-lived event, such as a collision) has occurred or is occurring;
- *Integrator* – an object that operates on a number of *PointMass*, *Force* and *DiscreteEvent* objects that comprise the system, in order to advance it through some time interval.

The exact way in which the associations between the objects are created and maintained is not specified – this is an implementation detail. A number of useful primitives for building creature bodies exist in the engine. Each primitive implements one or more of these abstract interfaces. The core interfaces and the primitives implemented from them are now described in more detail. Additional application-specific objects are defined by following a similar process, but these objects are not described.

3.1.1 The *PointMass* Interface

The *PointMass* interface is designed so that any classes that implement it should contain the following variables, with their associated dynamics meanings (Table 2):

Attribute	Description
position	Position of the centre of mass
velocity	Velocity of the centre of mass
force	Accumulated force on the centre of mass
trial	Current integration trial number
mass	Mass
volume	Volume
radius	Radius
mu_s	Coefficient of static friction
mu_k	Coefficient of kinetic friction
r	Coefficient of restitution

Table 2. Atributes implied by the *PointMass* interface.

The **position**, **velocity** and **force** variables are actually lists of vectors, because the integrator may perform a number of trial integrations for each timestep, and the results of each trial must be recorded. The **trial** variable indicates the current trial

number. The trial integrations are used by the higher-order integrators, such as the popular 4th Order Runge-Kutta integrator.

The **force** variable is actually a force accumulator. It is cleared at the start of every simulation timestep, and each force object merely adds its force to each particle's accumulator. After every force object has done that, the force accumulator vector represents the nett force on the point mass.

In an early design the properties of the point masses were separated into a number of interfaces, such as *Buoyant* for properties relevant to buoyancy in a fluid, *Contact* for properties relevant to surface collision and contact and so on. Although it was a good abstract design, it resulted in a rather less elegant implementation, primarily because of Java's single inheritance model. In the final model all these properties are included in our single *PointMass*.

3.1.1.1 SpherePointMass

The *SpherePointMass* is intended to be a strict implementation of the *PointMass* interface. It calculates quantities such as cross-sectional area under the assumption that it is a perfect sphere, and should draw itself as such.

3.1.2 The *Force* Interface

The *Force* interface is simple. It consists of a single method definition:

```
applyForce(t, timestep);
```

All objects that implement this interface should calculate and apply a force to all the point masses they are associated with. Applying a force to a point mass is achieved by adding the desired force to the point mass' force accumulator. The parameter **t** indicates the current simulation time (according to the integrator's clock). By using this variable in their calculations, forces take on a time-varying property. The **timestep** parameter is only used by some unusual forces that run the risk of generating fictional oscillations in the system. An example of such a force is sliding friction.

We now describe the actual force objects. All force equations are taken from a standard physics text [Halliday et al, 1996].

3.1.2.1 Gravity

The *Gravity* object applies a force:

$$\mathbf{F} = m\mathbf{g} \qquad \text{Eq. 10}$$

to every particle it influences, with mass *m*. The vector **g** is the gravitational acceleration vector, and is by default a vector of length 9.8, pointing along the negative y-axis (down). The value of **g** can be changed by the user.

3.1.2.2 Spring

The *Spring* object is the standard idealised spring of physics theory fame. Each spring has a characteristic relaxation length, *r*, and if stretched or squashed it will apply a force to try to keep itself at the relaxation length. The rate at which the stretching force increases with extension, **k**, is a characteristic of the spring, as is the damping force constant, $\mathbf{k_d}$, that dissipates the kinetic energy of the spring. The *Spring* object applies an equal but opposite force to each of its two end masses, a and b (Figure 2).

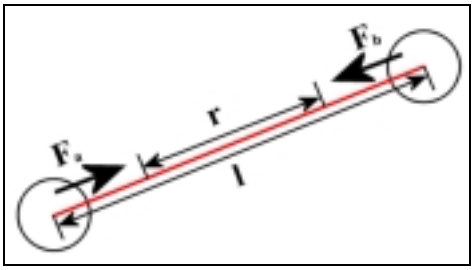


Figure 2. Idealised spring model.

The spring vector, **s**, is a vector pointing from b to a, with length *l* ($\mathbf{p}_x$ below denotes the position of point mass x):

$$\mathbf{s} = \mathbf{p}_a - \mathbf{p}_b$$
$$l = |\mathbf{s}|$$

The magnitude of the force is given by Hooke's law (equation 11):

$$F = k(l - r) - k_d dl \qquad \text{Eq. 11}$$

where *k* is the stiffness constant, *l* is the length, *r* is the relaxation length, k_d is the damping constant and *dl* is the rate of extension (change of length). The actual force vectors applied to the end masses are then:

$$\mathbf{F}_b = F\hat{\mathbf{s}}$$
$$\mathbf{F}_a = -F\hat{\mathbf{s}}$$

3.1.2.3 AngleJoint

The *AngleJoint* is basically a springy component that tries to maintain an angular separation between three point masses.

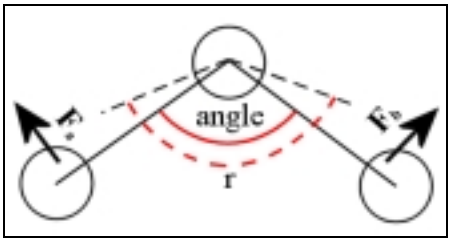


Figure 3. angular joint model.

The *AngleJoint* is very similar to a *Spring*, except that it has three point masses, two end masses and a third joint mass, and works on the angle of separation between its two end masses, whereas the *Spring* simply works on the distance of separation between its two end masses.

Two vectors, **a** and **b**, each pointing from the joint mass j to the end masses a and b are found as follows:

$$\mathbf{a} = \mathbf{p}_a - \mathbf{p}_j$$
$$\mathbf{b} = \mathbf{p}_b - \mathbf{p}_j$$

The angle between **a** and **b** is found from their dot product:

$$\theta = \cos^{-1}\left(\frac{\mathbf{a} \bullet \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right)$$

This difference between this angle and the relaxation angle, *r* is used to calculate a torque magnitude, *T* in a similar fashion to that used for the *Spring*. The joint vector, **j** is calculated from the cross product of **a** and **b**:

$$\mathbf{j} = |\mathbf{a} \times \mathbf{b}|$$

The actual forces applied to each end mass have magnitude:

$$F_a = \frac{T}{|\mathbf{a}|}$$

$$F_b = \frac{T}{|\mathbf{b}|}$$

and are given by:

$$\mathbf{F}_a = F_a |\mathbf{j} \times \mathbf{a}|$$

$$\mathbf{F}_b = -F_b |\mathbf{j} \times \mathbf{b}|$$

3.1.2.4 TorqueJoint

The *TorqueJoint* applies a torque to one end mass about a second joint mass. The torque axis vector is specified by the user, as is the torque magnitude. The actual torque to force translation is done in a similar fashion to that done by the *AngleJoint* as described above.

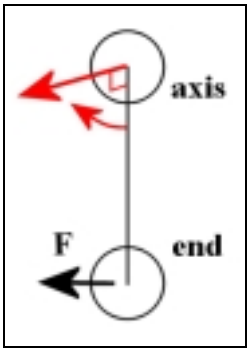


Figure 4. Torque joint model.

3.1.2.5 ConstantForce

The *ConstantForce* simply applies a constant force vector, **f**, to all the point masses it affects.

3.1.2.6 WorldNail

The *WorldNail* attempts to keep all the points it affects stationary in the world by applying a force equal but opposite to the accumulated nett force on each point mass. In practice this object may simply clear the force accumulator on each point mass.

3.1.2.7 Floor

The *Floor* influences point masses in two ways, friction and collision. In this section we will describe how friction is modelled.

Friction acts on all the point masses that are in contact with the world plane (y = 0). A point mass is said to be in contact with a plane if three conditions are met:

- It is touching the plane;
- It is not moving towards or away from the plane;
- It is being forced towards the plane.

Under these three conditions friction is applied to the point mass. First, the normal force is saved ($\mathbf{N} = -\mathbf{F}_y$ for the world plane, y = 0). An equal but opposite force (termed the "normal reaction force" is then applied. The friction model we use is divided into two parts, static friction and kinetic friction.

If the point is not moving tangentially (or it is moving slower than a critical threshold speed), static friction is applied. The static friction force $\mathbf{F}_s$ opposes any force that tends to move a point mass tangentially to a plane, but has a limit:

$$\mathbf{F}_s \leq \mu_s |\mathbf{N}| \qquad \text{Eq. 12}$$

where μ_s is the point-mass-to-plane coefficient of static friction.

If the point mass is moving tangentially or the static friction force limit is exceeded, kinetic friction is applied. The kinetic friction force $\mathbf{F}_k$ opposes any movement tangential to the plane, and is given by the equation:

$$\mathbf{F}_k = \mu_k |\mathbf{N}| \hat{v}_r \qquad \text{Eq. 13}$$

where μ_k is the point-mass-to-plane coefficient of kinetic friction and v_r is the tangential velocity of the point mass relative to the plane.

One caveat is that the kinetic friction force is capped (truncated) if it is predicted to cause the point mass to reverse direction after integration. Without this measure, point masses sliding across a surface would start to oscillate when they were nearly stationary.

3.1.2.8 FluidMedium

The *FluidMedium* applies two distinct forces, turbulent fluid drag and buoyancy, to every point mass it influences (Figure 5).

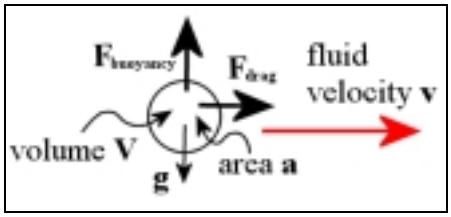


Figure 5. Fluid medium model.

Turbulent fluid drag is calculated with the formula:

$$\mathbf{F} = -\tfrac{1}{2} C\rho A(\mathbf{v}_r)\mathbf{v}_r^{\,2} \qquad \text{Eq. 14}$$

where C is an "experimentally-determined" drag coefficient, ρ is the fluid density, $A(\mathbf{x})$ is the cross-sectional area of the point mass with respect to the normal vector $\mathbf{x}$, $\mathbf{v}_r$ is the velocity of the point mass relative to the fluid. It is worth emphasising that the fluid need not be static – fluid flow can be represented by providing some function that returns a flow vector (a velocity) at each point in space. This is done by the *ReefWater* object, which defines some interesting swaying currents that are similar to those found on a coral reef some distance below the waves.

Buoyancy is calculated from Archimedes's principle, which basically says that a buoyant force equal to the weight of the displaced fluid acts on the submerged object. This translates into the formula:

$$\mathbf{F} = -V\rho\mathbf{g} \qquad \text{Eq. 15}$$

where V is the total volume of the point mass, ρ is the fluid density again and $\mathbf{g}$ is the gravitational acceleration vector.

For these equations to work, each point mass should clearly have an associated volume and cross-sectional area (i.e. shape). That is why these attributes are included in the basic point mass class.

3.1.3 The *DiscreteEvent* Interface

The *DiscreteEvent* interface is implemented by all objects that detect and handle very short-lived events. An example of such a short-lived event is a collision between a point mass and the world plane, during which the point mass' velocity changes almost instantaneously by an impulse force that is applied by the world plane. These large, but short-lived forces cannot be handled using normal integration and a large timestep, and so have to be detected and handled at the moment when they occur. Two methods are defined in the interface:

```
getEventState();
handleEvent();
```

After a trial integration, `getEventState()` should indicate which of three states applies to the event in question:

Event State	Description
None	No event detected during the previous timestep
Occurring	An event is occurring now and can be handled
Missed	An event occurred during the timestep

Only if the event is occurring it should be handled when `handleEvent()` is called. If the event has been missed, the integrator should backtrack.

3.1.3.1 Floor

Floor is the only object in the base engine that implements *DiscreteEvent*. The discrete event it detects is a collision between a point mass in its list with the world plane (y = 0).

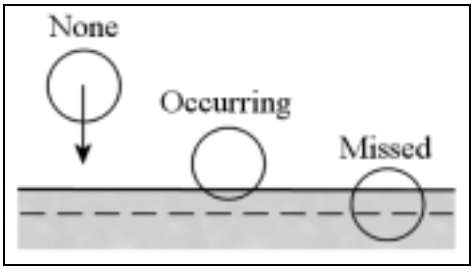


Figure 6. Point-to-world plane collision detection.

Interpenetration occurs when the lowest point of the point mass (which is actually a sphere) is more than the collision envelope below the world plane. In this case the collision event state **Missed** is returned.

Collision occurs when the lowest point of the point mass is below the world plane. In this case the collision event state **Occurring** is returned.

If neither of these conditions is met the collision event state **None** is returned.

The floor also handles collisions. The collision response is to move the point mass so it just touches the plane and then assign it a new velocity after the collision, from the formula:

$$v_y = -rv_y \qquad \text{Eq. 16}$$

where *r* is the coefficient of restitution (bounciness) of the point-mass-plane collision.

3.1.4 The *Integrator* Interface

All integrators should implement the *Integrator* interface, implying that they should contain at least the following variables:

Attribute	Description
t	Time (simulation clock)
timestep	Integration timestep

Table 3. Attributes implied by the *Integrator* interface.

In addition they should implement this method:

```
advance();
```

When `advance()` is called, the system should be advanced through one timestep to time **t**+**timestep** (by numerical integration), and all discrete events occurring during this time period should be detected and handled.

3.1.4.1 EulerIntegrator

The *EulerIntegrator* class implements the classic first order integrator. With an eye to deriving higher-order integrators from *EulerIntegrator*, its operation is broken into the following functions, which should exist in all integrators:

- applyForces – instruct all Force objects to apply themselves
- setTrial – set the current trial number of all point masses
- copyTrial – copy all the state variables from one trial to another
- linearDerive – derive the state variables of the current trial linearly from those of the first trial

The most work is done by a function, *integrate* which advances the system from the starting time to some later time, but does not take collisions into account. This function can simply be overridden by other derived classes to implement higher-order integrators. The *EulerIntegrator* also detects and handles collisions.

3.1.4.2 MidpointIntegrator

The *MidPointIntegrator* is a second order integrator that simply overrides the *integrate* function of *EulerIntegrator*.

3.1.4.3 RungeKuttaIntegrator

Similary, the *RungeKuttaIntegrator* is the popular fourth order integrator that simply overrides the *integrate* function of the *EulerIntegrator*. This integrator is a good compromise between stability and computational complexity. The stability offered by a high order integrator is particularly important when using oscillating components such as masses on springs, because low-order integrators tend to perform too gross an approximation and cause the springs to blow up.

3.2 Animats

In this section we describe the design of three animats. The animats are the *roobot*, a hopping monopod, the *jellyfish* and the *quadruped*.

After briefly describing each animat we describe each of its layers and finish by discussing special issues relevant to each animat. For the *roobot*, we describe an

experiment in which we used a genetic algorithm to evolve a stable and fast hopping gait. For the *jellyfish*, we describe the underwater environment that they live in.

3.2.1 The Roobot – A Hopping Monopod

The roobot is an imaginary creature with a single telescoping leg, rotating hip and ankle joints, a head and a springy two-segment tail (Figure 7). The intention behind the design of the roobot is to create a simple dynamically unstable hopping creature. It resembles a triple-amputee kangaroo.

Figure 7. Three hopping roobots.

3.2.1.1 Dynamical Model

For simplicity, this model is planar, that is, it is confined to a 2-dimensional plane. An additional variable representing the roobot's angle of rotation in the world allows it to move in 3-d space (Figure 8).

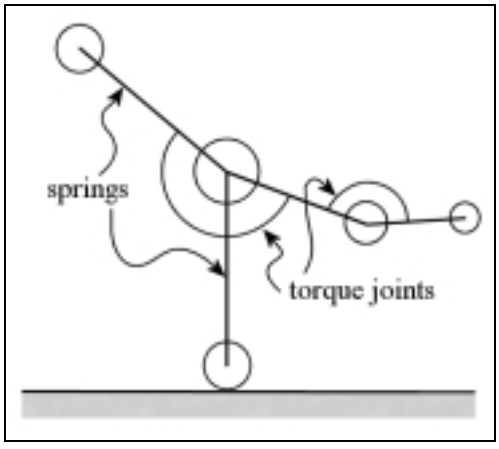


Figure 8. Roobot dynamical model.

The main elements are the hip and foot *PointMass*es and the *Spring* that connects them. Two *TorqueJoint*s, one placed at the hip and the other at the foot, are intended to generate rotations about each, similarly to how hip and ankle joints do in legged animals. The model also includes a few passive elements. One *PointMass* represents a head, and a *Spring* connects this to the hip. Another two *PointMass*es make up the tail, and these are also connected to each other and the hip with *Spring*s. Springy *AngleJoint*s link the neck and tail to the leg to stop them from drooping, while retaining the model's elasticity. All of the point masses are subject to the force of gravity and contact with the world plane. The model is dynamically unstable – it will fall over if left alone, either forwards or backwards.

3.2.1.2 Actuators

In order to keep the control problem as simple as possible, just three components are chosen to function as actuators:

- The relaxation length of the leg spring can be changed;
- A torque about the hip can be applied to the foot;
- A torque about the foot can be applied to the hip.

It was found that it was necessary to distinguish between the flight (foot not touching the ground) and stance (foot touching the ground) phases to improve physical realism. When in flight the ankle *TorqueJoint* must apply zero torque to the hip, and when in stance, the hip *TorqueJoint* must apply zero torque to the foot.

3.2.1.3 Sensors

Once again, to keep the animat as simple as possible only four sensors are defined:

- The height of the foot off the ground;
- The angle of the leg, relative to the vertical;
- The hip's forward speed;
- The estimated time till foot touchdown.

The first three sensors may be implemented as trivial manipulations of quantities taken directly from the dynamics simulation. The fourth involves slightly more

calculation. In a sense this implementation is “cheating”, because a real creature and a robot would not have access to the universe’s “simulation data”. In reality all of these “senses” would be obtained through complex processing of more basic sensory input, such as the sight of the ground approaching, the feeling of wind rushing and so on. To follow our approach strictly we should generate more basic, realistic sensory data and leave the creature to interpret it. This may be necessary if one is performing research in which a simple “black-box” abstraction of the sensor is insufficient (for example simulating a vision-based robot). In this instance though, adding these intermediate steps would introduce unnecessary complexity. We should imagine that the creature has some innate ability to guess how long it will be till its foot touches the ground and ignore the implementation details.

3.2.1.4 The Hopping Gait FSM

Taking a lead from other research on legged robots the hopping gait control mechanism is designed as a finite state machine. Each state of the machine corresponds to a distinct phase in the hopping motion (Figure 9). State transitions are triggered by sensor levels passing critical values. Each state is characterised by a unique sensor-actuator feedback mechanism. For example while in the **flight** state a desired leg angle (sensor input) is calculated, based on forward speed. The torque applied by the hip actuator is regulated to try and get the leg angle sensor input as close to this desired value as possible.

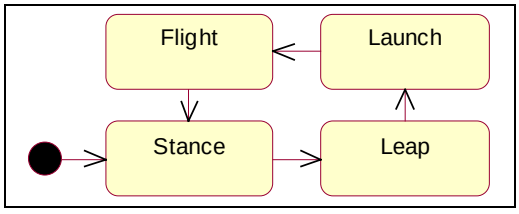


Figure 9. States of the hopping control FSM.

The control algorithms and state transitions for the four states are now described:

In the **stance** state, the leg actuator level is reduced at a rate called the “leg retraction rate”, which is parameter 3 times the difference between the leg actuator level and the target leg actuator level in stance (parameter 1), divided by the difference between the target leg angle sensor level in stance and the current leg angle sensor level. At the

same time, the ankle actuator level is set to the sum of parmameter 4 times the difference between the target leg angle sensor level in stance (parameter 1) and the actual leg angle sensor level and parameter 5 times the difference between the hip forward speed sensor level and the target hip forward speed sensor level (parameter 2).

So, briefly, while in the **stance** state, the leg is compressed to a desired length and at some rate, which is a function of a few parameters and the current sensor readings, and at the same time the ankle rotates the hip forwards towards some desired angle and speed via a similar calculation.

The transition from the **stance** state to the **leap** state is triggered by the leg angle sensor level passing the desired leg angle sensor level in stance (parameter 1).

In the **leap** state the leg actuator is controlled so as to explosively extend the leg, in a similar fashion to that used in the stance phase (source code for the finite state machine and the feedback algorithms appears available in Appendix B).

The transition from the **leap** state to the **launch** state is triggered by the leg actuator level passing the target leg actuator level in leap (parameter 6).

In the **launch** state the leg actuator is controlled so as to retract the leg quickly.

The transition from the **launch** state to the **flight** state is triggered by the foot height sensor level passing the minimum launch height (parameter 9).

In the **flight** state, the leg actuator is controlled to retract the leg at a desired rate until a desired length. At the same time, the hip actuator is controlled to swing the foot forwards at a desired rate and to a desired angle. The actual rate depends on the predicted time till foot touchdown sensor level, and the target leg angle depends on the hip forward speed sensor level.

The transition from the **flight** state back to the **stance** state is triggered by the foot height sensor level reaching zero.

The 14 parameters in the complete FSM are:

#	Description
0	target leg extension in stance
1	target leg angle in stance
2	target hip forward speed in stance
3	leg retraction rate f((leg extension - target leg extension) / (target leg angle - leg angle)) in stance
4	ankle torque f(leg angle - target leg angle) in stance
5	ankle torque f(hip forward speed - target hip forward speed) in stance
6	target leg extension in leap
7	leg extension rate in leap
8	leg retraction rate in launch
9	minimum launch height
10	target leg extension in flight
11	leg retraction rate in flight
12	target leg angle in touchdown f(hip forward speed)
13	hip torque f((target leg angle - leg angle) / (timetilltouchdown))

Table 4. The 14 parameters in the hopping control finite state machine.

3.2.1.5 Evolving A Stable Hopping Gait

A problem is that the optimal values of all the parameters that define the state transitions and feedback mechanisms in the finite state machine are unknown. In fact, this parameter optimisation problem is a serious one that faces all who take this approach to animat creation.

Fourteen parameters control the roobot's hopping gait. Changing any single parameter alters the roobot's resulting motion, often in an unpredictable way. There is no single optimal value for each parameter – changing the value of one affects the others in a complicated way. The aim of the experiment is to generate a fast and stable hopping gait. In order to do this, a set of optimal parameters must be found. This is a non-trivial task, because the quality of any set of parameters is not easily guessed, it can

only be found by performing a full simulation of a roobot using those parameters. For evolutionary purposes the “fitness” of a roobot is calculated using the equation below:

$$fitness = timeup + 50 \times dist \qquad \textbf{Eq. 17}$$

Each evaluation runs until the earliest of the hip touching the floor or the timeout occurring. *Timeup* is the duration of the evaluation and *dist* is the distance travelled from the start.

During an initial experiment the fitness value was simply calculated as the time until the hip touched the ground, but this often yielded roobots that stayed motionless at the starting point! The final fitness formula is chosen to favour fast as well as stable roobots.

3.2.1.6 Evolutionary Model

As mentioned, a genetic algorithm is used to find progressively better sets of parameters through trial and error. In nature, evolution operates according to the Darwinian “selection”, “crossover” and “mutation” model. We choose to use a simpler “selection” and “mutation” model. Specifically, we always retain the best parameter set found so far. For each new generation we generate a new set by slight random mutation of the best set. If this new set is better than the current best, it replaces it otherwise it is discarded.

3.2.2 The Jellyfish

The intention in the design of the jellyfish animat is to create a creature that resembles real jellyfish in both appearance and motion (Figure 10). Real jellyfish have a rounded dome that deforms as they swim. This is simply modelled graphically as a curved parametric surface that changes shape smoothly during the pulsing motion.

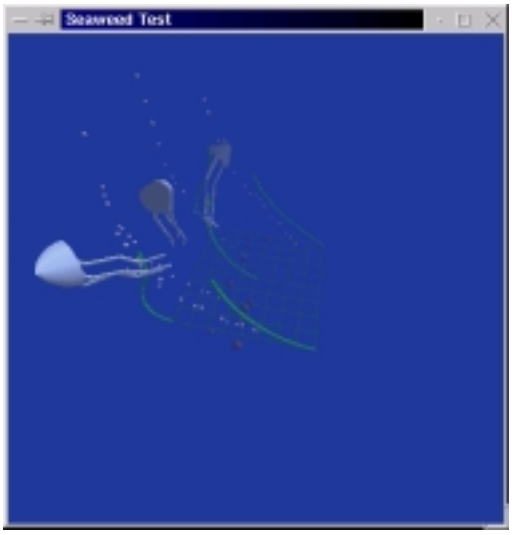


Figure 10. Three swimming jellyfish.

3.2.2.1 Dynamical Model

The jellyfish model is fully 3-dimensional and exists in a fluid medium, such as water (Figure 11).

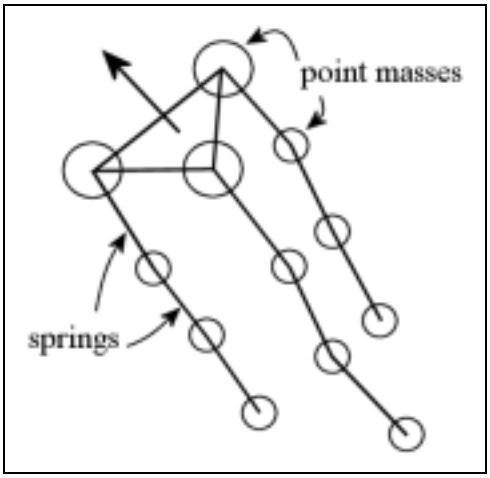


Figure 11. Jellyfish dynamical model.

The dome is represented as a ring of three *PointMass*es connected by *Spring*s. A long trailing tentacle comprised of *PointMass*es and *Spring*s is attached to each of the *PointMass*es in the dome. All the point masses are subject to gravity as well as the buoyancy and drag forces exerted by the water medium. To approximate the nett hydrodynamic forces generated by the dome during the pulsing motion, a shared *ConstantForce* object applies a force vector to each of the point masses in the dome. To facilitate steering of the main body, each of the point masses has an associated

ConstantForce object. Steering is achieved by applying small different forces to each point mass in the dome during the pulsing motion.

3.2.2.2 Actuators

Although real jellyfish swim by changing the shape of their flexible dome we choose to approximate the results of this formidable computational problem with a single force vector. The differential steering forces are represented similarly, yielding four actuators:

- The magnitude of the pulse force can be varied;
- The magnitude of the steering force on each of the three point masses in the dome can be varied.

The direction of all of these forces is set equal to the surface normal of the triangle defined by the three point masses that represent the dome.

3.2.2.3 Sensors

To allow the jellyfish to steer towards food, a food sensor is located at each of the point masses in the dome. It may be imagined that the food sensors detect the concentration of a substance in the water. In our simplified model the food sensors track a single *PointMass* that represents the “food” and report a sensor level that falls off with distance from the food.

3.2.2.4 Locomotion

The pulsing action of the main body is what powers the jellyfish’s swimming motion. Each pulse is viewed as an uninterruptible process. The current point in the pulse is recorded as a phase variable that is ramped up from 0 to 2π. The actuators are controlled at the same time to generate synchronised forces. The same phase variable controls the shape of the graphical surface representing the main dome.

At the behavioural level, the jellyfish has a repertoire of three simple behaviours. The action selection problem is solved with a finite state machine (Figure 12). Each behaviour has a characteristic energy usage rate and which changes a simulated internal energy level.

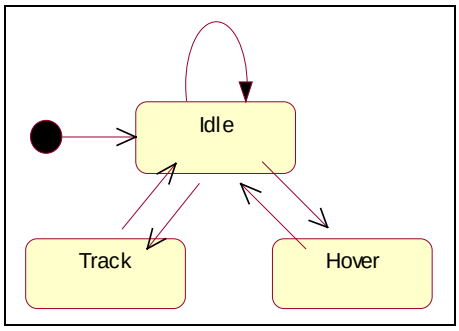


Figure 12. States in the jellyfish control FSM.

In the **track** state, the jellyfish pulses once and steers uses its three food sensors to steer towards the source of the food. After completing one pulse a transition back to the **idle** state occurs. While in the **track** state, the energy level falls at a rate of 0.1 units/s.

In the **hover** state, the jellyfish saves its current depth and pulses just often enough to maintain this depth. A transition back to the **idle** state occurs after each pulse or if no pulse is necessary because it is above the starting depth. While in the **hover** state and pulsing, the energy level remains constant.

In the **idle** state, the jellyfish decides which of the three states is most appropriate and causes a transition to that state. The decision is based on a simulated internal "energy" level that changes appropriately in each of the states. The energy level ranges, and the corresponding transitions appear below:

Energy	Transition
[0..0.2)	Go to **idle**.
[0.2..0.5]	Go to **hover**.
[0.5..0.7]	Return to **track,** otherwise go to **hover**.
(0.7..1.0]	Go to **hover**.

Table 5. Energy level ranges and FSM transitions.

While in the **idle** state (which is visited periodically when hovering) the energy level increases at a rate of 0.4 units/s.

In summary, the jellyfish switches smoothly between behaviours based on its hidden internal state.

3.2.2.5 The Underwater Environment

To explore the idea of creating complete virtual environments for animats, an underwater environment is created for the jellyfish to swim in (Figure 13).

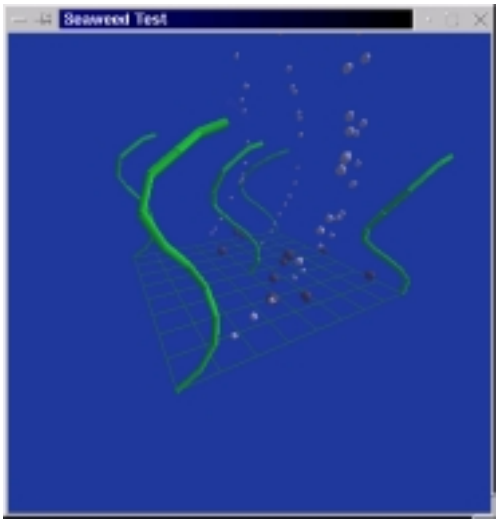


Figure 13. The underwater environment.

A new fluid medium, *ReefWater*, is defined. *ReefWater* approximates water with currents by generating velocity vectors that vary in time and space. As implemented, the currents resemble those found near a reef with waves rolling by overhead. To add to the visual impact of the environment, swaying fronds of deformable seaweed, sinking stones and rising air bubbles are all incorporated. It should be emphasised that the *ReefWater* affects all the objects in the environment, including the jellyfish.

The seaweed is modelled as point mass and spring chains. Each point mass has a density slightly lower than the surrounding water and thus experiences a slight nett upward force because the buoyancy force is stronger than the gravitational force. The base of each seaweed frond is anchored with a *WorldNail* object.

The bubbles are encapsulated in a *BubbleCollumn* object. Each column is a true particle system – bubbles are spawned, move through the world and die (the topmost bubble is actually recycled whenever a new bubble is generated). Each bubble is

spawned stochastically and has a random volume within some range. The bubble density is much lower than the surrounding water, so the nett force is upwards.

The rocks are individual point masses, each with a random volume within some range, but all with a density higher than the water, so they sink.

In principle we could change any parameter such as the gravitational acceleration or the density of the water and the simulation would behave realistically.

3.2.3 The (Attempted) Quadruped

The idea behind the quadruped is to extend the work on the hopping roobot to creatures with more legs. A number of problems with constructing a stable dynamics model are encountered during the implementation phase. Three different models are designed and implemented, and are described below.

3.2.3.1 Dynamical Models

All three quadruped dynamics models have a flat, horizontal, rectangular torso defined by four *PointMass*es completely connected by 6 *Spring*s. All three models have four legs attached, but the design of the legs varies.

In the first model (Figure 14) the legs are comprised of two segments, the first segment points up and away from the torso; the second points down and away from the torso, and touches the ground. An *AngleJoint* is added to maintain the angular separation of the two leg segments. A new 2-DOF joint, at the point where the upper leg joins the torso controls the uppermost leg segment – to maintain the angle between the upper segment and the torso normal as well as the angle of the upper segment around the torso normal.

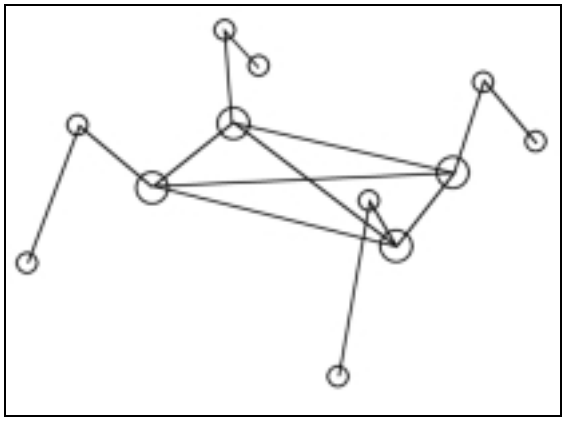


Figure 14. Quadruped model 1.

In the second model (Figure 15) the legs are simplified to a single segment. Two *AngleJoint*s are added to support the leg, the first to the torso spring running from front to back, and the second to the torso spring running from left to right.

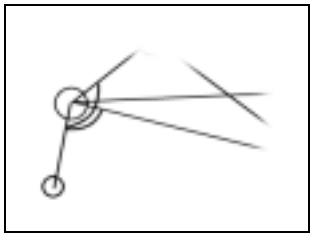


Figure 15. Quadruped leg model 2.

In the third model the *AngleJoint* pairs are replaced with pairs of *Spring*s (Figure 16).

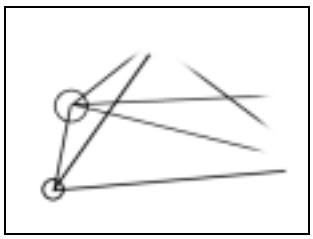


Figure 16. Quadruped leg model 3.

The reasons why these models are unstable will be discussed later, in the implementation section.

3.2.3.2 Actuators

The second and third models are stable in the standing position, and so components are chosen to function as actuators, to attempt to obtain basic stable locomotion.

For the second model:

- The eight *TorqueJoint*s supporting the legs can swing the feet forwards and backwards as well as left and right;
- The relaxation length of the four leg springs can be changed.

For the third model:

- The eight springs supporting the legs can push the feet forward and backwards as well as left and right;
- The relaxation length of the four leg springs can be changed.

4 Implementation

In this chapter we discuss the implementation of the dynamics engine and the animats, and cover the major issues that arise. For the dynamics engine we also work through an example program that illustrates how it is typically used.

4.1 Dynamics Engine

The dynamics engine is implemented entirely in Java, the primary language of *greatdane*. When compiled, the classes are combined into a single shared library that is linked into *greatdane* applications, in the same way that the pre-existing basis, audio, video and vr libraries are.

New dynamics classes can easily be derived from the existing ones, thereby building upon the existing functionality. This is thanks to the careful use of interfaces. Once written, these new classes can either be compiled and linked in an application's directory, or added to the dynamics engine source directory and become part of the shared library.

4.1.1 Class Hierarchy

The class diagram for the complete dynamics engine appears in Appendix A.

4.1.2 An Example Program – The N-body Gravitational Simulator

To demonstrate the process of using the dynamics engine, we describe the creation of my old favourite, the n-body gravitational simulator. What this program does is show how a number of point masses (n bodies) move under the influence of mutual gravitational attraction.

First of all, a new force has to be defined – one that will calculate and apply the mutual gravitational attraction between a number of point masses. A new Java source file is created, and the class defined:

```
class NBodyGravity implements Force {
  /// gravitational constant.
  protected double G;
  /// list of point masses.
```

```
  protected Vector pointmasses;
```

Accessors for the variable **G** and the list **pointmasses** are added. Finally, the `applyForce()` method is implemented.

```
  /// apply the gravitational force.
  public void applyForce(double t, double timestep) {
    PointMass thispointmass, otherpointmass;
    for(int i = 0; i < pointmasses.size(); i++) {
      thispointmass = (PointMass)pointmasses.elementAt(i);
      for(int j = i+1; j < pointmasses.size(); j++) {
        otherpointmass = (PointMass)pointmasses.elementAt(j);
…
```

The gravitational force **F** is calculated using Newton's law of universal gravitation.

```
…
        thispointmass.addForce(Vector3D.Multiply(F, direction));
        otherpointmass.addForce(Vector3D.Multiply(-F, direction));
      }
    }
  }
```

Then the application is created. After standard *greatdane* preliminaries, the objects in the dynamics simulation are instantiated and linked. The objects have to be explicitly linked in this model so that the engine only performs as much work as is absolutely necessary. Each *ForceObject* has its own lists of the point masses is influences, and these have to be constructed explicitly. Likewise, it is possible to declare multiple integrators. The *PointMass*, *Force* and *DiscreteEvent* objects have to be explicitly registered with an integrator or they will not be simulated.

```
EulerIntegrator integrator = new RungeKuttaIntegrator(timestep);
NBodyGravity gravity = new NBodyGravity();
integrator.addForce(gravity);
for(int i = 0; i < numpoints; i++) {
  PointMass pointmass = new PointMassWithTrail(mass,...);
  pointmass.setPosition(...);
  pointmass.setVelocity(...);
…
```

Specify a nice “sphere with trail” representation for the body.

```
…
  gravity.addPointMass(pointmass);
  integrator.addPointMass(pointmass);
}
```

When the call to `Component.RunComponents()` is made in the application loop, the integrator advances the system through one timestep and greatdane renders the point masses in their new positions (Figure 17).

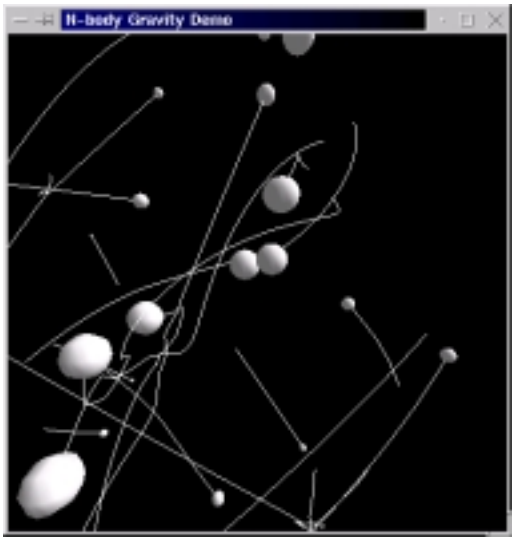


Figure 17. The complete n-body gravitational simulator.

4.2 Animats

The animats are implemented entirely in Java. The dynamics library is linked into the test applications. Some new dynamics classes are written, but not included in the dynamics library.

To ensure that our formal approach to animat design is followed, the actuators and sensors in the animat models are implemented as actual classes.

With a view to possible future work into using artificial neural networks in the control layer, the sensors and actuators are designed to be compatible. The activation levels are confined to the range [0..1]. In addition, classes that simulate a neural network that operates in the same range are implemented.

4.2.1 The Actuator Class

The actuators are all derived from a single *Actuator* class. The idea is to present a uniform interface for the control of any underlying physical component, and to ensure that a strict “operating range” is enforced.

All actuators that extend the actuator class map an activation level in the range [0..1] on to a component value in the range [min..max], and vice versa. Each new actuator overrides the following two methods:

```
protected double getValue()
protected void setValue(double value)
```

The methods above get and set the actual physical parameter in the dynamical model. A new actuator is created by specifying the object it controls and the bounds of the operating range, **min** and **max**. The constructor should save the actuation level corresponding to the object’s initial state, and revert to that when its `reset()` method is invoked. In normal use, the animat will control the actuator by calling its `setLevel(double level)` method.

4.2.2 The Sensor Interface

The Sensors all implement the *Sensor* interface, consisting of a single method:

```
public double getLevel()
```

All sensors should by convention return a level in the range [0..1].

4.2.3 Behaviour Routines

All the creatures contain a method called `BehaviourRoutine()`, which executes one step in the behaviour routine algorithm each time it is called. In the case of the roobot and the jellyfish, this is an action selection finite state machine that in turn calls a function to execute the current behaviour.

4.2.4 Upper Bounds on Timestep

Virtual reality applications should run in real-time. If a virtual environment contains a dynamics simulation, it is usually desirable that the simulation clock runs at the same speed as the real world, thus the simulation timestep should be adjusted accordingly, and this can easily be done automatically as the application runs. It was found that even when using a 4th order integrator it is necessary to place an upper bound on the timestep, because each system has a value at which it becomes numerically unstable, and this should not be exceeded.

4.2.5 Quadriplegic Quadrupeds

As mention in section 3.2.3, numerous attempts to implement a stable standing and walking quadruped were met with disaster.

The first model, with two-segment legs, fails to remain standing. The reason for this is that no object exerts a force to prevent the "knees" from twisting. As a result, the torso begins to rotate slowly; the legs gradually twist and lean further and further to one side until the torso reaches the ground. The problem of defining all the additional forces required to prevent the knees from twisting is a formidable one given the particle system nature of the dynamics engine. A more advanced engine with rigid bodies and proper joint hierarchies would presumably solve this problem, at the cost of complicating the model.

The second model, with single-segment legs supported by *AngleJoint*s, stands stably, but lifts itself right up and over onto its back as soon as a pair of feet are raised from the floor. This problem is also traced to the simplistic particle system nature of the dynamics engine. A more advanced engine typically solves the differential equations defining the entire system as one, whereas our system merely integrates the equations for each point mass independently.

The third model, with single-segment legs supported by *Spring*s, stands stably, but is virtually impossible to control to the degree that walking can be achieved. The reason for this is that three spring actuators affect each foot. When the relaxation length of any one of them is adjusted, this generates tension in the other two. These springs in turn push or pull the foot in a different direction, so that the foot follows a very

different path to that intended. To move a foot properly, all three springs must be actuated simultaneously while taking their effect on each other into account. This artificial problem is again a side effect of our relatively primitive joint model.

5 Results and Observations

In this chapter we describe the major results obtained in this project. We start by evaluating the dynamics engine implementation on three criteria: firstly its applicability in simulating dynamic creature bodies, secondly its ability to produce physically realistic motion and thirdly its ability to operate in real-time. Next, we describe the results of the experiment in which stable and fast roobot hopping motion was successfully evolved with a genetic algorithm. Finally, we discuss our observation that creating animats from the “physics-up”, as we have done, is an “engineering problem” in that the creator needs a thorough understanding of dynamics and the models have to be carefully designed to be simple enough to control and to simulate in real-time.

5.1 Dynamics Engine Evaluation

The object-oriented decomposition of the dynamics engine is validated, because it has been used to define and realistically simulate a variety of particle systems, as intended. Experience shows that the class framework allows easy implementation of additional objects, such as forces.

The important attribute of particle systems is that the motion of individual point masses can be calculated with little consideration to the exact nature of the links to the other point masses. The difficulties encountered when attempting to construct a useful quadruped model indicate that the engine is however not suitable for systems dominated by joints, multiple points of contact, or where the rotation of any of the parts is significant enough to warrant simulating. It would be worth experimenting with a more advanced dynamics engine that can simulate rigid bodies and joints. At least one such promising open source package exists, *DYNAmechs*. This is expected to complicate the process of defining simple physical models like the ones used in this project, because additional properties will have to be specified for every single component, e.g. shape, rotational inertia, orientation, the location of joints and joint ranges.

One area of the engine which could be developed further is collision detection and resolution. Currently only simple point to world plane collisions are detected and

handled, but in principle additional objects which implement *Force* and *DiscreteEvent* could be added which would facilitate other collisions, such as those between pairs of point masses or between point masses and springs.

5.2 Complex Physically Realistic Motion Achieved

Interesting, physically realistic motion is generated automatically using the dynamics engine. All animats created according to our approach necessarily exhibit motion in accordance with the laws of dynamics and the forces at play in the system. Therefore, our approach to generating creatures that exhibit physically realistic motion is validated.

Naturally, if the animat creator chooses to define forces that are not faithful to those that exist in the real world, or models a real animal poorly, the resulting motion of the animat may not resemble that of the real animal, but it will still be consistent with the laws of dynamics.

5.3 Real-time Operation

Although no thorough benchmarks of the creature models were performed, during the course of the work most models (using typical physical parameters such as masses and spring constants) with less than a hundred components (masses, forces) could be run in real-time.

Benchmarking is complicated by the fact that much of the simulation work is actually distributed between all the *Force* and *DiscreteEvent* objects that are registered with the *Integrator*, and each of these can refer to and influence any number of point masses. Indeed, the scaling (Order) of the dynamics engine is not fixed, it varies with the nature and configuration of the system being simulated.

Nevertheless, we have some experience simulating a variety of systems that may be considered “typical” as far as models of simple animats goes. In some test applications the timestep was adjusted online to keep the simulation running at the correct rate. When run on a dual Intel Pentium III 800MHz PC with Red Hat Linux 7.0 and an Nvidia GeForce graphics card, five roobot models could easily be simulated and rendered in real-time at a ridiculous resolution of several hundred

thousand pixels. In practice, during the evolution experiment the timestep is fixed so that the effects of a changing timestep do not affect the process.

The *SeaWorld* test application for the jellyfish, however, had to be modified to limit the timestep used, because the seaweed fronds become unstable when the timestep exceedes a certain threshold, which varies with the strength of the water current. This is due to the characteristic oscillation frequency of the point masses in the chains. If the simulation timestep is too large relative to the oscillation frequency, they blow up. The only remedy is to either change the actual model parameters (such as masses and spring constants) or to use a faster machine.

5.4 Evolved A Stable Hopping Gait

Starting from an initial set of guessed parameters that defined a roobot that fell over after landing for the second time, we successfully evolved a set of parameters that yielded a stable hopping motion after less than a hundred generations. After several hundred generations, the motion is still stable and significantly faster. Thus, the parameter optimisation approach to developing motor controllers is validated. A graph of best fitness value versus generation for 6 separate runs appears below (Figure 18). The graphs are dominated by a logarithmic tapering off of the fitness curves indicating that the returns from continued optimisation diminish over time. Interestingly, this pattern does not always hold. The graph of best fitness over several thousand generations, and the actual parameter data from the star achiever appear in Appendix C. This roobot (run 6) continued to improve its fitness long after the other runs had tapered off. This suggests that the evolutionary model can be improved upon. In particular, the effect of a crossover phase may accelerate the evolutionary process by causing widely separated but successful parameter sets to be combined and the result tested. Animations of selected generations during this roobot's evolution are available on the project website. A copy of the full website, including poster, presentation, write-up, source code, animations and screenshots is on the CD included with the hardcopy thesis.

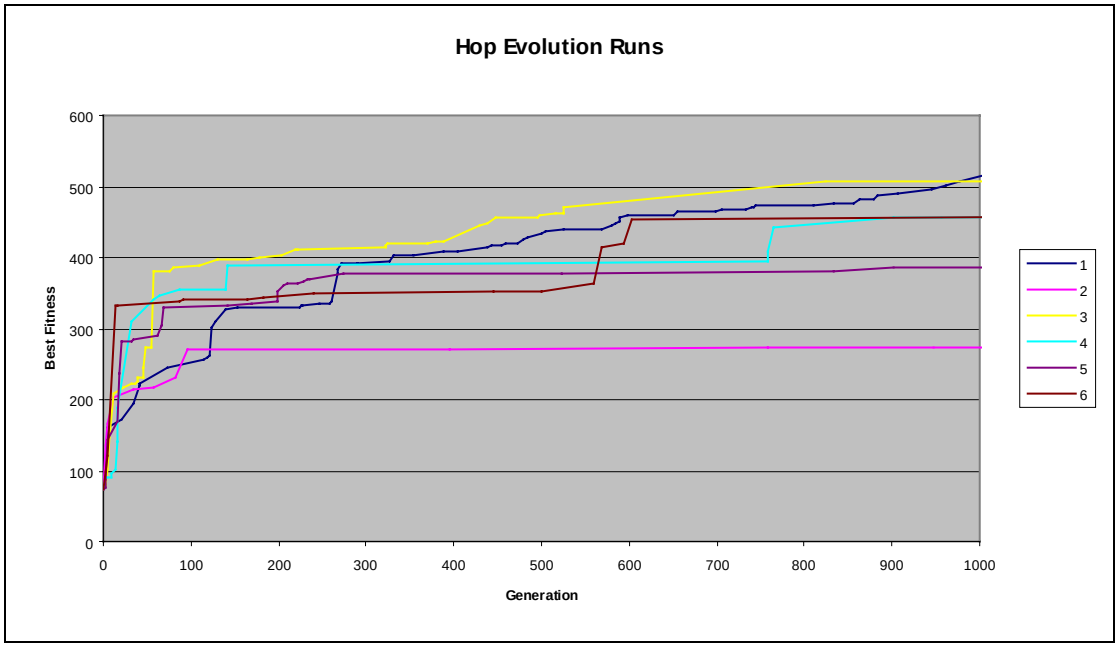


Figure 18. Best fitness as a function of generation for 6 different runs, over 1000 generations.

An interesting observation is that several parallel evolutions were actually run, all starting from the same initial parameter set, but each generated startlingly different motions. There are indeed many ways to achieve a given goal. This suggests that the fitness landscape for the roobot model is highly complex, which lends supports to the strategy of searching for sufficiently optimal sets as opposed to a single “best” set.

5.5 Animat Design: An Engineering Problem

The animat design process is complicated because the added physical realism of a full dynamical model introduces new challenges.

When constructing a dynamical model, far more than simply the appearance of the model has to be borne in mind. Factors such as the distribution of mass, elasticity and density must be considered, because they affect the motion of the model. The relative importance of all the forces at work in a real animal must also be evaluated and a decision made as to which are significant enough to be modelled.

5.5.1 Dynamics Knowledge Necessary

Experience of designing and implementing animats has shown that the creature designer must have a firm grasp of dynamics concepts and dynamical simulation, for two reasons.

Firstly, one cannot build a model that behaves in a certain way unless one understands how all the components work and interact. One must understand the significance of all the parameters involved in order to tweak them to fine-tune the model. An intuitive graphical interface for defining and experimenting with dynamical models would go a long way towards relaxing this requirement.

Secondly, when constructing minimal dynamical models it is frequently necessary to define new objects such as forces (in a manner similar to that done in the n-body gravitational simulator demo above). The object-oriented nature of the engine supports this process, but it cannot be done without a thorough understanding of the entity (e.g. force) being modelled (from a physics point of view), the current structure and operation of the dynamics engine, Java and 3-d vector maths.

5.5.2 Lazy Modelling Is Optimal

Following a principle reminiscent of Occam's Razor may be useful:

"the simplest possible model that performs as desired is probably the best".

By making a model overly complicated one tends to introduce many "degrees of freedom" into it. This means that the number of different configurations that the model can assume is increased. Although we want to build "physically realistic" models that are capable of occasionally behaving in surprising but still realistic ways, as animat designers we are also trying to create animats with bodies that usually behave in a number of standard ways. The more degrees of freedom in the physical model layer, the more complex the other layers (e.g. sensors, actuators and control mechanism) must be to overcome them.

When designing the roobot, the animat design and implementation is simplified substantially by restricting the dynamical model to a 2-dimensional plane, while still allowing it to execute the desired hopping motion.

Another significant motivation for using simpler models is that they are less computationally expensive. Each force that is added to the system means more work for the dynamics engine to do for each timestep. As mentioned previously, dynamical simulations in virtual environments should ideally run in or near real-time, so computationally expensive models should be avoided. Specifically, to operate in real-time, the maximum stable timestep for the model must be larger than 1/*framerate*.

6 Conclusions

In this project we have explored a solution to the problem of creating autonomous creatures with physically realistic motion that inhabit virtual environments. This solution is to build creatures from the “bottom-up”, in three distinct functional layers: the physically simulated body, the sensors and actuators and the brain.

We have implemented a simple, modular, extensible, object-oriented dynamics engine in the *greatdane* VR system. At the heart of the engine are three numerical integrators of different orders that solve the equations of motion of the system in order to animate it. The engine supports systems comprised of point masses, generalised forces, collision detectors and resolvers, and includes a number of primitives, including spherical masses, gravity, springs, angular joints, water, air and a world plane.

We have used our dynamics engine to construct two creatures with distinctly different bodies that exhibit interesting, physically realistic motion. The first creature was a “roobot”, a hopping monopod, and the second was a jellyfish. For the roobot, we created a finite state machine to control its hopping motion and used a genetic algorithm to evolve stable and fast hopping. For the jellyfish, we implemented an underwater environment complete with water currents, swaying seaweed, rising bubbles and sinking rocks, and used a finite state machine to switch between different swimming behaviours as appropriate.

Our creature models were sufficiently simple to simulate in real- or near real-time on a dual Intel Pentium III 800MHz PC with an Nvidia GeForce graphics card, making them suitable for interactive virtual environments.

After experience of designing and implementing animats using the “bottom-up” approach we conclude that an engineering-style approach to the process is necessary, for three reasons. Firstly, an understanding of the significance of each physical property is necessary when designing and tweaking dynamics models. Secondly, new entities such as forces will frequently have to be implemented for each new creature. Finally, care must be taken to keep the models as simple as possible, to simplify the

control problem and to ensure that the model can be simulated in real-time, which is a strict requirement for virtual environments.

We believe that if further work on creating dynamically simulated creatures were carried out with *greatdane*, it would be worthwhile using an advanced dynamics simulator such as *DYNAmechs*, which can accurately simulate rigid bodies and articulated joints, and would allow the creation of creatures with proper skeletal structure.

7 References

Internet: GA Intro

Sipper, M., *A Brief Introduction To Genetic Algorithms*, available via the WWW: http://lslwww.epfl.ch/~moshes/ga.html

Internet: GOL

The Game of Life, available via the WWW: http://www.brunel.ac.uk/depts/AI/alife/al-gamel.htm

Internet: NR

Numerical Recipes Home Page, available via the WWW: http://www.nr.com/

Badler et al, 1995

Badler, N.I., Metaxas, D., Webber, B., Steedman, M., *The Centre for Human Modelling and Simulation*, Presence 4 (1), pp. 81-96, 1995.

Bangay et al, 1996

Bangay, S., Gain, J., Watkins, G., Watkins, K., *RhoVeR: Building the Second Generation of Parallel/Distributed Virtual Reality Systems*, First Eurographics Workshop on Parallel Graphics & Visualisation, Bristol (UK), 26-27 September 1996.

Baraff et al, 1999

Baraff, D., Witkin, A., Kass, M., *Physically Based Modelling*, SIGGRAPH '99 Course Notes, 1999.

Brogan et al, 1998

Brogan, D.C., Metoyer, R.A., Hodgins, J.K., *Dynamically Simulated Characters in Virtual Environments*, IEEE Computer Graphics and Applications, 1998.

Brooks, 1985

Brooks, R. A., *A Robust Layered Control System for a Mobile Robot*, MIT AI Lab Memo 864, September 1985.

Dembovsky, 1999

Dembovsky, C., *VRPhysicsEnvironment - A Framework For Collision Detection and Physical Modelling in a Virtual Environment*, Honours Thesis, Computer Science Department, Rhodes University, November 1999.

Emmeche, 1992

Emmeche, C., *Life as an abstract phenomenon: is Artificial Life possible?*, In Francisco J. Varela and Paul Bourgine (eds.): Toward a Practice of Autonomous Systems. Proceedings of the First European Conference on Artificial Life, The MIT Press, Cambridge, Mass., 1992.

de Garis, 1990

de Garis, H., *Genetic programming: Evolution of a time dependent neural network module which teaches a pair of stick legs to walk*, In Luigia Carlucci Aiello (editor), ECAI 90 9th European Conference on Artificial Intelligence, pp. 204-206, Pitman Publishing, London, 1990.

Grand et al, 1996

Grand, S., Cliff, D., Malhotra, A., *Creatures: Artificial Life Autonomous Software Agents for Home Entertainment*, University of Sussex Technical Report CSRP434, 1996.

Halliday et al, 1996

Halliday, D. Resnick, R., Walker, J., *Fundamentals of Physics*, Extended, 5th Edition, Wiley, 1996.

Hodgins et al, 1995

Hodgins, J., Wooten, W., Brogan, D., and O'Brien, J.1995. Animating human athletics, In Computer Graphics (SIGGRAPH ’95 Proceedings), pp. 71-78, 1995.

Langton et al, 1992

Langton, C. G., *Preface*, In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen (eds.), Artificial Life II, Volume X of SFI Studies in the Sciences of Complexity, pages xiii-xviii, Addison-Wesley, Redwood City, CA, 1992.

Maes, 1994

Maes, P., *Modeling Adaptive Autonomous Agents*, Artificial Life Journal 1 (1 & 2), pp. 135-162, MIT Press, 1994.

Maes, 1995

Maes, P., *Artificial Life Meets Entertainment: Lifelike Autonomous Agents*, Communications of the ACM Special Issue on Novel Applications of AI, 38 (11), pp. 108-114, 1995.

Mataric, 1997

Mataric, M.J., *Behaviour-Based Control: Examples from Navigation, Learning and Group Behaviour*, Journal of Experimental & Theoretical Artificial Intelligence 9 (2/3), pp. 323-336, 1997.

Raibert et al, 1991

Raibert, M.H., Hodgins, J.K., *Animation of Dynamic Legged Locomotion*, Computer Graphics (SIGGRAPH '91 Proceedings), Volume 25, pp. 349-358, July 1991.

Reeves, 1983

Reeves, W.T., *Particle Systems – A Technique for Modeling a Class of Fuzzy Objects*, ACM Transactions on Graphics, Vol. 2, No. 2, pp. 91-108, April 1983.

Reynolds, 1987

Reynolds, C. W., *Flocks, Herds, and Schools: A Distributed Behavioral Model*, Computer Graphics, Volume 21, pp. 25-34, 1987.

Ronald et al, 1999

Ronald, E. M. A., Sipper, M., Capcarrère, M.S., *Design, Observation, Surprise! A Test of Emergence*, Artificial Life Journal, Volume 5, Number 3, pp. 225-239, The MIT Press, Cambridge, MA, 1999.

Sims, 1994

Sims, K., *Evolving Virtual Creatures*, Computer Graphics (SIGGRAPH '94 Proceedings), pp. 15-22, July 1994.

Thalman

Thalman, D., Dynamic Simulation as a Tool for Three-Dimensional Animation

Tu, 1996

Tu, X., *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception, and Behavior* (Ph.D Dissertation, University of Toronto, 1996) ACM Distinguished Ph.D Dissertation Series, Springer-Verlag, 1999.

Wilhelms et al, 1997

Wilhelms, J., Van Gelder, A., *Anatomically Based Modeling*, Computer Graphics (SIGGRAPH '97 Proceedings), pp. 173-180, August 1997.

8 Appendices

A. Dynamics Engine Class Hierarchy

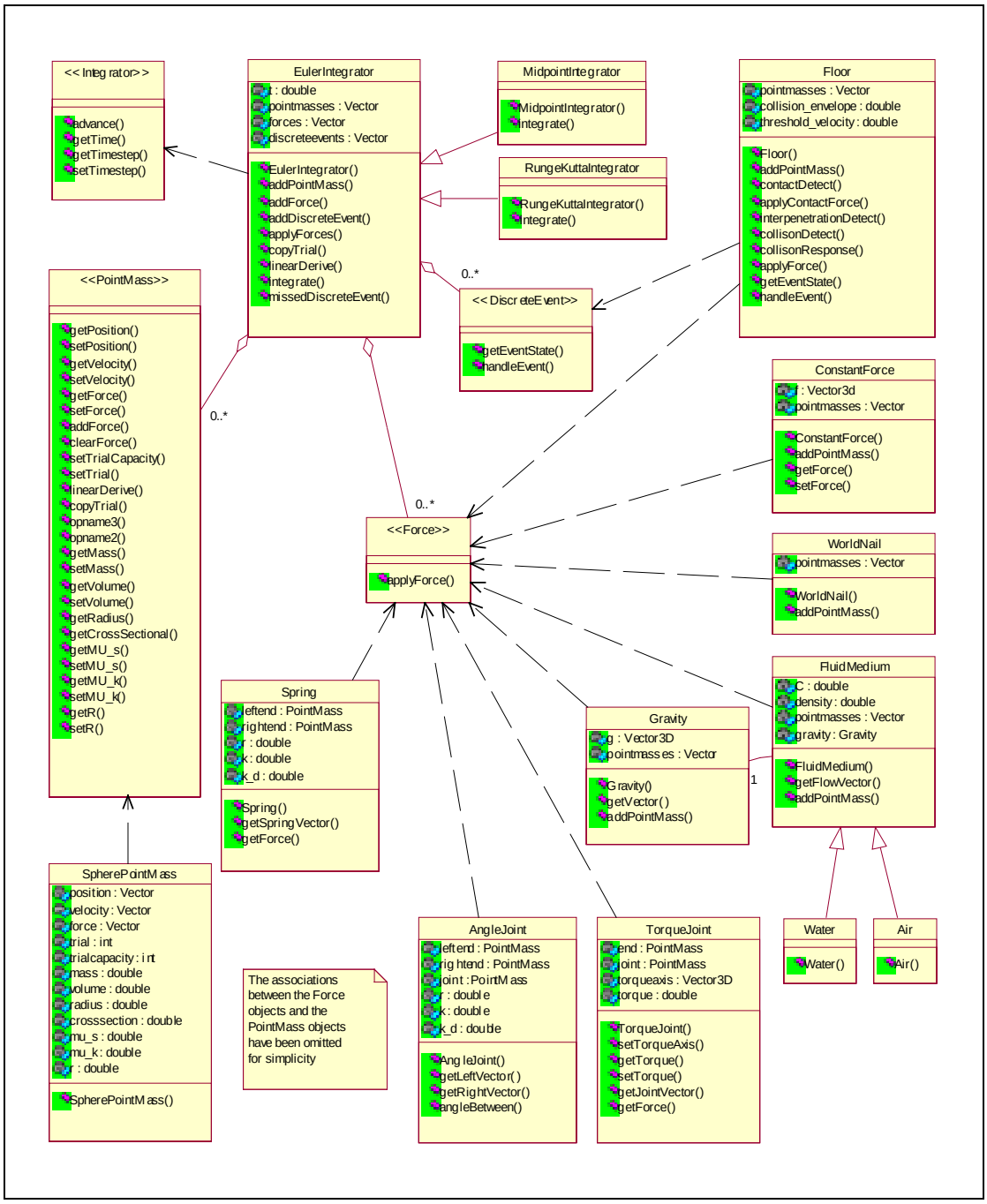


Figure 19. Dynamics engine class hierarchy.

B. Selected Source Code

FSMRoobot.java

```
/** class FSMRoobot

   a hopping 'roobot', controlled with a FSM, second generation.
*/

class HeightSensor implements Sensor {
  protected PointMass foot;

  public HeightSensor(PointMass foot) {
    this.foot = foot;
  }

  public double getLevel() {
    double height = foot.getPosition().y-foot.getRadius();
    return Math.max(0.0, Math.min(1.0, height));
  }
}

class LegAngleSensor implements Sensor {
  protected PointMass hip, foot;

  public LegAngleSensor(PointMass hip, PointMass foot) {
    this.hip = hip;
    this.foot = foot;
  }

  public double getLevel() {
    Vector3D legvector = Vector3D.Subtract(hip.getPosition(), foot.getPosition());
    double legangle = Math.atan(legvector.x/legvector.y);
    double leglevel = 0.5 + (legangle/2.0*Math.PI);
    //System.out.println("leg angle sensor: "+leglevel+" "+Math.max(0.0, Math.min(1.0,
leglevel))+" ("+legangle+")");
    return Math.max(0.0, Math.min(1.0, leglevel));
  }
}

class TimeTillTouchdownSensor implements Sensor {
  protected PointMass foot;
  protected Sensor heightsensor;
  protected static double g = 9.8;

  public TimeTillTouchdownSensor(PointMass foot, Sensor heightsensor) {
    this.foot = foot;
    this.heightsensor = heightsensor;
  }

  public double getLevel() {
    double timetilltouchdown = (foot.getVelocity().y + Math.sqrt(foot.getVelocity().y
* foot.getVelocity().y + 2.0 * g * heightsensor.getLevel())) / g;
    return Math.max(0.0, Math.min(1.0, timetilltouchdown));
  }
}

class SpeedSensor implements Sensor {
  protected PointMass hip;

  public SpeedSensor(PointMass hip) {
    this.hip = hip;
  }

  public double getLevel() {
    double speed = hip.getVelocity().x;
    return Math.max(0.0, Math.min(1.0, speed));
  }
}

class FSMRoobot extends Roobot {
  /// the array of hopping parameters.
```

```
  private double[] parameters;
  /// foot height sensor.
  protected Sensor heightsensor;
  /// time till foot touchdown pseudo-sensor.
  protected Sensor timetilltouchdownsensor;
  /// leg tilt sensor.
  protected Sensor leganglesensor;
  /// hip forward speed sensor.
  protected Sensor speedsensor;
  protected static final int IDLE_BEHAVIOUR = 0;
  protected static final int HOP_BEHAVIOUR = 1;
  /// the behaviour FSM state.
  protected int behaviour = IDLE_BEHAVIOUR;

  protected static final int HOP_STANCE = 0;
  protected static final int HOP_LEAP   = 1;
  protected static final int HOP_LAUNCH = 2;
  protected static final int HOP_FLIGHT = 3;
  /// the hopping FSM state.
  protected int hopstate = HOP_STANCE;

  /* Parameters are as follows:
   0   target leg extension in stance
   1   target leg angle in stance
   2   target hip forward speed in stance
   3   leg retraction rate f((leg extension - target leg extension) / (target leg
angle - leg angle)) in stance
   4   ankle torque f(leg angle - target leg angle) in stance
   5   ankle torque f(hip forward speed - target hip forward speed) in stance
   6   target leg extension in leap
   7   leg extension rate in leap
   8   leg retraction rate in launch
   9   minimum launch height
   10  target leg extension in flight
   11  leg retraction rate in flight
   12  target leg angle in touchdown f(hip forward speed)
   13  hip torque f((target leg angle - leg angle) / (timetilltouchdown))
  */

  public FSMRoobot(double hipmass, double footmass, double[] parameters,
EulerIntegrator integrator, Gravity gravity, FluidMedium air, Floor floor) {
    super(hipmass, footmass, integrator, gravity, air, floor);
    heightsensor = new HeightSensor(foot);
    speedsensor = new SpeedSensor(hip);
    timetilltouchdownsensor = new TimeTillTouchdownSensor(foot, heightsensor);
    leganglesensor = new LegAngleSensor(hip, foot);
    loadParameters(parameters);
    Reset();
  }

  public void Reset() {
    super.Reset();

    legactuator.reset();
    hipactuator.reset();
    ankleactuator.reset();
    hopstate = HOP_STANCE;
  }

  /// copy a parameter array.
  public void loadParameters(double[] parameters) {
    if (parameters.length != 14) System.exit(1);
    this.parameters = new double[14];
    for(int i = 0; i < 14; i++) {
      this.parameters[i] = parameters[i];
    }
  }

  /// the idle behaviour.
  public void IdleBehaviour () {
    behaviour = HOP_BEHAVIOUR; // always hop.
  }

  /// the hopping behaviour.
  public void HopBehaviour () {
    switch (hopstate) {
```

```
      case HOP_STANCE: { // Stance phase - compress leg and tilt over to the desired
angle, moving at the desired velocity
                if (leganglesensor.getLevel() < parameters[1]) {
                  double legretractionrate = parameters[3] * (legactuator.getLevel() -
parameters[0]) / (parameters[1] - leganglesensor.getLevel());
                  double leglevel = Math.max(parameters[0], legactuator.getLevel() -
legretractionrate * integrator.getTimestep());

                  legactuator.setLevel(Math.max(0.0, Math.min(1.0, leglevel)));
                  hipactuator.reset();
                  double anklelevel = 0.5 + parameters[4] * (parameters[1] -
leganglesensor.getLevel()) + parameters[5] * (parameters[2] - speedsensor.getLevel());
                  ankleactuator.setLevel(Math.max(0.0, Math.min(1.0, anklelevel)));
                } else {
                  hipactuator.reset();
                  ankleactuator.reset();
                  hopstate = HOP_LEAP;
                  //System.err.println("Stance -> Leap");
                }
              } return;
      case HOP_LEAP: { // Leap phase - extend leg to the desired length
                if (legactuator.getLevel() < parameters[6]) {
                  double leglevel = Math.min(parameters[6], legactuator.getLevel() +
parameters[7] * integrator.getTimestep());
                  legactuator.setLevel(Math.max(0.0, Math.min(1.0, leglevel)));
                } else {
                  hopstate = HOP_LAUNCH;
                  //System.err.println("Leap -> Launch");
                }
              } return;
      case HOP_LAUNCH: { // Launch phase - retract leg till foot at the minimum height
                if (heightsensor.getLevel() < parameters[9]) {
                  double leglevel = Math.max(parameters[10], legactuator.getLevel() -
parameters[8] * integrator.getTimestep());
                  legactuator.setLevel(Math.max(0.0, Math.min(1.0, leglevel)));
                } else {
                  hopstate = HOP_FLIGHT;
                  //System.err.println("Launch -> Flight");
                }
              } return;
      case HOP_FLIGHT: { // Flight phase - retract leg and bring foot forwards to
desired angle
                if (heightsensor.getLevel() >  0.0) {
                  double leglevel = Math.max(parameters[10], legactuator.getLevel() -
parameters[11] * integrator.getTimestep());
                  legactuator.setLevel(Math.max(0.0, Math.min(1.0, leglevel)));

                  ankleactuator.reset();

                  double targetlegangle = 0.5 + Math.max(0.0, Math.min(1.0, -
parameters[12] * (hip.getVelocity().x)));
                  double hiplevel = 0.5 + parameters[13] * (targetlegangle -
leganglesensor.getLevel()) / (timetilltouchdownsensor.getLevel() + 0.05);
                  hipactuator.setLevel(Math.max(0.0, Math.min(1.0, hiplevel)));
                } else {
                  hopstate = HOP_STANCE;
                  behaviour = IDLE_BEHAVIOUR;
                  //System.err.println("Flight -> Stance");
                }
              } return;
    }
  }

  /// the behaviour routine.
  public void BehaviourRoutine() {
    if (behaviour == IDLE_BEHAVIOUR) IdleBehaviour();

    switch (behaviour) {
      case IDLE_BEHAVIOUR : break;
      case HOP_BEHAVIOUR : { HopBehaviour(); break; }
    }
  }
}
```

Jellyfish.java

```
/** class Jellyfish

   a floating jellyfish with active pulsing and passive flexible tentacles.
*/

import java.util.Vector;
class FoodSensor implements Sensor {
  protected PointMass sensor, food;

  public FoodSensor(PointMass sensor, PointMass food) {
    this.sensor = sensor;
    this.food = food;
  }

  public double getLevel() {
    Vector3D foodvector = Vector3D.Subtract(food.getPosition(), sensor.getPosition());
    foodvector.y = 0.0;
    double foodlevel = 0.1/foodvector.length();
    return Math.max(0.0, Math.min(1.0, foodlevel));
  }
}

class HeightSensor implements Sensor {
  protected PointMass pointmass;

  public HeightSensor(PointMass pointmass) {
    this.pointmass = pointmass;
  }

  public double getLevel() {
    double height = pointmass.getPosition().y;
    return Math.max(0.0, Math.min(1.0, height));
  }
}

class Jellyfish extends VRComponent implements VisualRepresentation {
  protected objectID me;

  protected PointMass[] bodymass, tentaclemass;
  protected Spring[] bodyspring, tentaclespring;
  protected ConstantForce pulseforce;
  protected ConstantForce[] steerforce;
  protected ConstantForceActuator pulseactuator;
  protected ConstantForceActuator[] steeractuator;
  protected Sensor[] foodsensor;
  protected Sensor heightsensor;

  protected EulerIntegrator integrator;

  protected double bodyforcemag;
  protected Vector3D bodydir;

  protected double bodydensity = 1100.0, tentacledensity = 1200.0, bodyradius = 0.03;
  protected double bodyspring_k = 50.0, bodyspring_mu_k = 5.0;
  protected double tentaclespring_k = 20.0, tentaclespring_mu_k = 1.0;
  protected double tentaclespringlength = 0.1;
  /// current body position
  protected Vector3D pos;
  protected int numtentacles = 3, numtentaclesegs = 4;
  protected double phase = 0.0, pulseperiod = 1.0;

  protected static final int IDLE_BEHAVIOUR = 0;
  protected static final int TRACK_BEHAVIOUR = 1;
  protected static final int HOVER_BEHAVIOUR = 2;
  protected int behaviour, prev_behaviour;
  protected double hoverheight = -1.0;

  protected double energy = 1.0;

  public Jellyfish(double bodymass_mass, double tentaclemass_mass, Vector3D pos,
EulerIntegrator integrator, Gravity gravity, FluidMedium water, PointMass food, Floor
floor) {
```

```
    this.pos = pos;
    bodydir = new Vector3D(0.0, 1.0, 0.0);
    pulseforce = new ConstantForce(new Vector3D(0.0, 0.0, 0.0));
    integrator.addForce(pulseforce);
    steerforce = new ConstantForce[numtentacles];
    pulseactuator = new ConstantForceActuator(pulseforce, -5.0, 5.0);
    steeractuator = new ConstantForceActuator[numtentacles];
    foodsensor = new FoodSensor[numtentacles];
    this.integrator = integrator;
    behaviour = prev_behaviour = IDLE_BEHAVIOUR;

    double factor = 2.0*Math.PI/(double)numtentacles;
    bodymass = new SpherePointMass[numtentacles];
    for (int i = 0; i < numtentacles; i++) {
      bodymass[i] = new SpherePointMass(bodymass_mass, bodymass_mass/bodydensity, 0.6,
0.5, 0.6);
      bodymass[i].setPosition(Vector3D.Add(pos, new
Vector3D(bodyradius*Math.cos(factor*(double)i), 0.0,
bodyradius*Math.sin(factor*(double)i))));
      gravity.addPointMass(bodymass[i]);
      water.addPointMass(bodymass[i]);
      floor.addPointMass(bodymass[i]);
      integrator.addPointMass(bodymass[i]);
      steerforce[i] = new ConstantForce(new Vector3D(0.0, 0.0, 0.0));
      steerforce[i].addPointMass(bodymass[i]);
      pulseforce.addPointMass(bodymass[i]);
      integrator.addForce(steerforce[i]);
      steeractuator[i] = new ConstantForceActuator(steerforce[i], -0.5, 0.5);
      foodsensor[i] = new FoodSensor(bodymass[i], food);
    }
    heightsensor = new HeightSensor(bodymass[0]);

    bodyspring = new Spring[numtentacles];
    for (int i = 0; i < numtentacles; i++) {
      bodyspring[i] = new Spring(bodymass[i], bodymass[(i+1)%numtentacles],
bodyspring_k, bodyspring_mu_k);
      integrator.addForce(bodyspring[i]);
    }

    tentaclemass = new SpherePointMass[numtentacles*numtentaclesegs];
    tentaclespring = new Spring[numtentacles*numtentaclesegs];
    int tm = 0;
    for (int i = 0; i < numtentacles; i++)
    for (int j = 0; j < numtentaclesegs; j++) {
      tentaclemass[tm] = new SpherePointMass(tentaclemass_mass,
tentaclemass_mass/tentacledensity, 0.6, 0.5, 0.6);
      tentaclemass[tm].setPosition(new Vector3D(bodymass[i].getPosition().x,
bodymass[i].getPosition().y - (j+1)*tentaclespringlength,
bodymass[i].getPosition().z));
      gravity.addPointMass(tentaclemass[tm]);
      water.addPointMass(tentaclemass[tm]);
      floor.addPointMass(tentaclemass[tm]);
      integrator.addPointMass(tentaclemass[tm]);

      if (j == 0) {
        tentaclespring[tm] = new Spring(tentaclemass[tm], bodymass[i],
tentaclespring_k, tentaclespring_mu_k);
      } else {
        tentaclespring[tm] = new Spring(tentaclemass[tm], tentaclemass[tm-1],
tentaclespring_k, tentaclespring_mu_k);
      }

      integrator.addForce(tentaclespring[tm]);
      tm++;
    }

    registerWithVREnvironment();
  }

  public void deregisterFromVREnvironment() {
    VREnvironment.destroyThing(me);
  }

  public void registerWithVREnvironment() {
    me = VREnvironment.createThing ();
    VREnvironment.setPhysicalRepresentation (this, me);
  }
```

```
  /// get the current body position.
  public Vector3D getPosition() {
    pos = new Vector3D(0.0, 0.0, 0.0);
    for (int i = 0; i < bodymass.length; i++) {
      pos.Add(bodymass[i].getPosition());
    }
    pos = Vector3D.Multiply(1.0/(double)bodymass.length, pos);
    return pos;
  }

  /// render a spring as a cyliner.
  public void RenderSpringAsCylinder(Spring spring, int segments, float upperradius,
float lowerradius) {
    GL.PushMatrix();

    Vector3D springvector = spring.getSpringVector();
    Vector3D springdir = springvector.Normalize();
    float xangle =
180f*(float)Math.atan2(Math.sqrt(springdir.z*springdir.z+springdir.x*springdir.x),
springdir.y)/(float)Math.PI;
    float yangle = 180f*(float)Math.atan2(springdir.x, springdir.z)/(float)Math.PI;
    Vector3D basepos = spring.getRightPointMass().getPosition();
    if (xangle < 0.0) xangle = -xangle;

    GL.Translate((float)basepos.x, (float)basepos.y, (float)basepos.z);
    GL.Rotate(yangle, 0.0f, 1.0f, 0.0f);
    GL.Rotate(xangle, 1.0f, 0.0f, 0.0f);
    Cylinder(segments, upperradius, lowerradius, (float)springvector.length(), true);

    GL.PopMatrix();
  }

  /// render a cylinder, optionally capped.
  public void Cylinder(int segments, float upperradius, float lowerradius, float
height, boolean cap) {
    double foo = 2.0*Math.PI/(double)segments;
    GL.Begin(GL.GL_QUAD_STRIP);
      for(int i = 0; i <= segments; i++) {
        double angle = i*foo;
        GL.Normal3f((float)Math.cos(angle), 0.0f, (float)Math.sin(angle));
        GL.Vertex3f(upperradius*(float)Math.cos(angle), height,
upperradius*(float)Math.sin(angle));
        GL.Vertex3f(lowerradius*(float)Math.cos(angle), 0.0f,
lowerradius*(float)Math.sin(angle));
      }
    GL.End();

    if (cap) {
      GL.Begin(GL.GL_POLYGON);
        for(int i = 0; i < segments; i++) {
          double angle = i*foo;
          GL.Normal3f(0.0f, -1.0f, 0.0f);
          GL.Vertex3f(lowerradius*(float)Math.cos(angle), 0.0f,
lowerradius*(float)Math.sin(angle));
        }
      GL.End();

      GL.Begin(GL.GL_POLYGON);
        for(int i = 0; i < segments; i++) {
          double angle = i*foo;
          GL.Normal3f(0.0f, 1.0f, 0.0f);
          GL.Vertex3f(upperradius*(float)Math.cos(angle), height,
upperradius*(float)Math.sin(angle));
        }
      GL.End();
    }
  }

  /// render the dome of the body. phase specifies the pulse phase.
  public void RenderDome (double height, double maxradius, int slices, int segments,
double phase) {
    GL.PushMatrix();

    float xangle =
180f*(float)Math.atan2(Math.sqrt(bodydir.z*bodydir.z+bodydir.x*bodydir.x),
bodydir.y)/(float)Math.PI;
```

```
    float yangle = 180f*(float)Math.atan2(bodydir.x, bodydir.z)/(float)Math.PI;
    Vector3D basepos = Vector3D.Add(getPosition(), Vector3D.Multiply(-0.5*height,
bodydir));
    if (xangle < 0.0) xangle = -xangle;

    GL.Translate((float)basepos.x, (float)basepos.y, (float)basepos.z);
    GL.Rotate(yangle, 0.0f, 1.0f, 0.0f);
    GL.Rotate(xangle, 1.0f, 0.0f, 0.0f);

    double prevradius = 0.0;
    for (int slice = 1; slice < slices; slice++) {
      double currradius = maxradius*Math.sqrt((double)slice/(double)slices);
      currradius *= 0.3*Math.sin(0.1 /*+ Math.PI*/ -phase +
Math.PI*(double)slice/(double)slices) + 1.4;
      GL.PushMatrix();
        GL.Translate(0.0f, (float)height*(1.0f-(float)slice/(float)slices), 0.0f);
        Cylinder(segments, (float)prevradius, (float)currradius,
(float)height/(float)slices, false);
      GL.PopMatrix();
      prevradius = currradius;
    }

    GL.PopMatrix();
  }

  /// render the jellyfish.
  public void Render (VRSinkMonitor monitor, Vector3D viewvector) {
    GL.Color3f(0.7f, 0.8f, 1.0f);
    for (int i = 0; i < tentaclespring.length; i++) {
      RenderSpringAsCylinder(tentaclespring[i], 3, 0.008f, 0.008f);
    }
    RenderDome(0.1, 0.05, 7, 10, phase);
  }

  public void RenderBoundary (Vector3D viewvector) {
  }

  /// idle, decide to do something.
  public void IdleBehaviour() {
    /* This creature's behaviour is governed by it's internal "energy level" and it's
height
       energy [0..0.2) or height > 0.8 -> IDLE_BEHAVIOUR
       energy [0.2..0.5] -> HOVER_BEHAVIOUR
       energy [0.5..0.7] -> continued TRACK_BEHAVIOUR, otherwise HOVER_BEHAVIOUR
       energy (0.7..1.0] -> HOVER_BEHAVIOUR
    */

    energy = Math.max(0.0, Math.min(1.0, energy + 0.4*integrator.getTimestep()));

    pulseactuator.reset();
    for(int i = 0; i < steerforce.length; i++) {
      steeractuator[i].reset();
    }

    if (heightsensor.getLevel() >= 1.0 || energy < 0.2) return; // too high or too
tired, carry on idling.

    if (energy > 0.9 || (energy > 0.5 && prev_behaviour == TRACK_BEHAVIOUR)) { // got
energy, track food.
      prev_behaviour = TRACK_BEHAVIOUR;
      behaviour = TRACK_BEHAVIOUR;
      hoverheight = -1.0;
      return;
    } else { // just hover.
      if (hoverheight == -1.0) {
        hoverheight = heightsensor.getLevel();
      }
      prev_behaviour = HOVER_BEHAVIOUR;
      behaviour = HOVER_BEHAVIOUR;
      return;
    }
  }

  /// track the food.
  public void TrackBehaviour() {
    energy -= 0.1*integrator.getTimestep();
```

```
    phase += 2.0*Math.PI*integrator.getTimestep()/pulseperiod;
    if (phase >= 2.0*Math.PI) {
      phase = 0.0;
      behaviour = IDLE_BEHAVIOUR;
      return;
    }

    double pulselevel = 0.5*Math.sin(phase);
    if (pulselevel < 0.0) pulselevel *= 0.1;
    pulseactuator.setLevel(0.5 + pulselevel);

    int leastfood = 0; // find which food sensor is least active.
    double leastfoodsense = foodsensor[0].getLevel();
    for (int i = 1; i < foodsensor.length; i++) {
      double thisfoodsense = foodsensor[i].getLevel();
      if (thisfoodsense < leastfoodsense) {
        leastfood = i;
        leastfoodsense = thisfoodsense;
      }
    }

    for (int i = 1; i < steeractuator.length; i++) {
      if (i == leastfood) {
        steeractuator[i].setLevel(1.0*Math.abs(pulseactuator.getLevel()));
      } else {
        double randomsteerlevel =
0.5*Math.sin((double)i*5.0*integrator.getTime()+heightsensor.getLevel());
        steeractuator[i].setLevel(0.5 +
randomsteerlevel*Math.abs(pulseactuator.getLevel()));
      }
    }
  }

  /// hover at a certain height.
  public void HoverBehaviour() {
    phase += 2.0*Math.PI*integrator.getTimestep()/pulseperiod;
    if (phase >= 2.0*Math.PI || heightsensor.getLevel() >= hoverheight) {
      phase = 0.0;
      behaviour = IDLE_BEHAVIOUR;
      return;
    }

    double pulselevel = 0.5*Math.sin(phase);
    if (pulselevel < 0.0) pulselevel *= 0.1;
    pulseactuator.setLevel(0.5 + pulselevel);
  }

  /// the behaviour routine.
  public void BehaviourRoutine() {
    bodydir = Vector3D.CrossProd(bodyspring[0].getSpringVector(),
bodyspring[2].getSpringVector()).Normalize();
    pulseactuator.setDirection(bodydir);
    for(int i = 0; i < numtentacles; i++) {
      steeractuator[i].setDirection(bodydir);
    }

    switch (behaviour) {
      case IDLE_BEHAVIOUR  : { IdleBehaviour(); break; }
      case TRACK_BEHAVIOUR : { TrackBehaviour(); break; }
      case HOVER_BEHAVIOUR : { HoverBehaviour(); break; }
    }
  }

  public void ThreadRoutine () {
    BehaviourRoutine();
  }
}
```

HopEvolver.java

```
/** class HopEvolver.

    an application to evolve roobot hopping gaits using a genetic algorithm.
    we find optimum combinations of the 14 hopping gait parameters through "selection"
and "mutation".
*/

import java.io.*;

class HopEvolver
  {
    /// simulation timestep
    private static double timestep = 0.008;

    public static void main (String args [])
      {
        VREnvironment vrenv = new VREnvironment ();
        VREnvironment.setCurrentEnvironment (vrenv);

        VRSink testsink  = new VRSink("Roobot Hop Evolver", 400, 300, null);

        objectID viewer = VREnvironment.createThing ();
        Point3D viewerpos = new Point3D (0.0, 0.0, 1.0);
        Quaternion viewerori = new Quaternion ();
        VREnvironment.setAbsolutePosition (viewer, viewerpos);
        VREnvironment.setAbsoluteOrientation (viewer, viewerori);
        testsink.setViewPoint (viewer);

        //VRUserActor vruser = new VRUserActor (viewer);
        //Connection userinput = new Connection (vrsource, vruser);

  /* Parameters are as follows:
   0    target leg extension in stance
   1    target leg angle in stance
   2    target hip forward speed in stance
   3    leg retraction rate f((leg extension - target leg extension) / (target leg
angle - leg angle)) in stance
   4    ankle torque f(leg angle - target leg angle) in stance
   5    ankle torque f(hip forward speed - target hip forward speed) in stance
   6    target leg extension in leap
   7    leg extension rate in leap
   8    leg retraction rate in launch
   9    minimum launch height
   10   target leg extension in flight
   11   leg retraction rate in flight
   12   target leg angle in touchdown f(hip forward speed)
   13   hip torque f((target leg angle - leg angle) / (timetilltouchdown))
  */
        int numparams = 14;
        double[] parameters = new double[numparams];
        parameters[0] = 0.2;
        parameters[1] = 0.55;
        parameters[2] = 0.01;
        parameters[3] = 0.5;
        parameters[4] = 5.0;
        parameters[5] = 5.0;
        parameters[6] = 1.0;
        parameters[7] = 3.0;
        parameters[8] = 2.0;
        parameters[9] = 0.02;
        parameters[10] = 0.3;
        parameters[11] = 0.5;
        parameters[12] = 0.01;
        parameters[13] = 4.0;

        /*parameters[0] = -0.1;
        parameters[1] = 0.02; //0.1
        parameters[2] = 0.01;
        parameters[3] = 0.2;
        parameters[4] = 50.0;
        parameters[5] = 10.0;
        parameters[6] = 0.2;
```

```
        parameters[7] = 2.5; //1.0
        parameters[8] = 1.0; // 0.5
        parameters[9] = 0.02;
        parameters[10] = -0.05;
        parameters[11] = 1.3;
        parameters[12] = 0.005; // 0.01
        parameters[13] = 1.0; // 2.0*/

        /*parameters[0] = -0.07958562184471549;
        parameters[1] = 0.02;
        parameters[2] = 0.006276146797725384;
        parameters[3] = 0.15547574418862486;
        parameters[4] = 20.68949246461333;
        parameters[5] = 4.151543233328644;
        parameters[6] = 0.09486924350049752;
        parameters[7] = 3.6366549461105575;
        parameters[8] = 1.799154303263847;
        parameters[9] = 0.017674946156207107;
        parameters[10] = -0.052735512474537984;
        parameters[11] = 1.0829032317477343;
        parameters[12] = 2.76363332020045E-4;
        parameters[13] = 0.33842574813409493;*/

        int numbots = 3; // number of robot lanes

        double[] bestscores = new double[numbots];              // array of best
scores per lane
        double[][] bestparams = new double[numbots][numparams]; // array of best
parameters per lane
        double[][] botparams = new double[numbots][numparams];  // array of current
parameters per lane
        double[] timeup = new double[numbots];                  // array of current
time off the ground per lane
        double[] botscores = new double[numbots];               // array of scores per
lan
        int[] generations = new int[numbots];                   // array of generation
counters

        for(int r = 0; r < numbots; r++) {
          for(int i = 0; i < numparams; i++) {
            bestparams[r][i] = parameters[i];
            botparams[r][i] = parameters[i];
          }
        }

        WorldPlane worldplane = new WorldPlane(10.0f, 1.0f, 5.0f, 0.0f, 40, 4);

        EulerIntegrator integrator = new RungeKuttaIntegrator(timestep);

        Gravity gravity = new Gravity();
        integrator.addForce(gravity);
        Air air = new Air(gravity);
        Floor floor = new Floor(0.01, 0.2);

        FSMRoobot[] roobot = new FSMRoobot[numbots];
        for (int r = 0; r < numbots; r++) {
          // create a roobot
          roobot[r] = new FSMRoobot(0.4, 0.1, bestparams[r], integrator, gravity, air,
floor);
          // position it
          roobot[r].setFootPosition(new Vector3D(0.0, 0.0, -
0.5+((double)r/((double)numbots-1.0))));
          // reset some values...
          timeup[r] = 0.0;
          botscores[r] = 0.0;
          bestscores[r] = 0.0;
          generations[r] = 1;
        }

        integrator.addForce(air);
        integrator.addForce(floor);
        integrator.addDiscreteEvent(floor);

        int currentbest = 0;

        Vector3D viewpos = new Vector3D(0.0,0.0,0.0);
```

```
        double t = 0.0;         // test timestep
        boolean running = true; // "simulation running" flag
        while (running) {

          for (int r = 0; r < numbots; r++) {
            if ((roobot[r].getPosition()).y > 0.0 && timeup[r] < 20.0 &&
roobot[r].getLength() < 1.0) {
              timeup[r] += timestep; // save time spent up
              botscores[r] = timeup[r] + (50.0 * roobot[r].getPosition().x); //
calculate score
              if (botscores[r] > botscores[currentbest]) currentbest = r; // test if
this score is better than the current best running bot
            } else {
              if (botscores[r] > bestscores[r]) { // test if this score is a new best
for this lane ("selection")
                for(int i = 0; i < numparams; i++) {
                  bestparams[r][i] = botparams[r][i]; // if so, save these parameters
                }
                bestscores[r] = botscores[r];

                System.out.print(r+" "+generations[r]+" "+botscores[r]+" "); // print
the lane, generation and score
                for(int i = 0; i < numparams; i++) {
                  System.out.print(botparams[r][i]+" "); // print the parameters
                }
                System.out.println();

              } else {
                for(int i = 0; i < numparams; i++) {
                  botparams[r][i] = bestparams[r][i]; // copy the parameters from the
best parameters for this lane
                }
              }

              // no "crossover" phase - asexual reproduction model

              int g = 1 + (int)Math.round(3.0*Math.random());
              for (int gc = 0; gc < g; gc++) { // change a few parameters a little
("mutation" phase)
                int pn = (int)(((double)(numparams))*Math.random());
                if (pn >= numparams) { System.err.println("Warning: pn capped from
"+pn+" to "+(numparams-1)); pn = numparams-1; }
                botparams[r][pn] += botparams[r][pn]*0.5*(r+1)*(-0.5 + Math.random());
              }

              roobot[r].Reset(); // reset the location of the various parts
              roobot[r].setFootPosition(new Vector3D(0.0, 0.0, -
0.5+((double)r/((double)numbots-1.0)))); // re-position
              // reset some variables
              timeup[r] = 0.0;
              botscores[r] = 0.0;
              roobot[r].loadParameters(botparams[r]);
              generations[r]++;
            }
          }

          t += timestep; // increment simulation clock

          Vector3D pos = roobot[currentbest].getPosition(); // track current best
running roobot

          viewpos =
Vector3D.Add(Vector3D.Multiply(0.990,viewpos),Vector3D.Multiply(0.01,pos));
          viewpos.z = 0.0;
          viewerpos = new Point3D(viewpos.x - 1.3 *Math.sin(t), viewpos.y+0.4,
viewpos.z + 1.3*Math.cos(t)); // orbit around target point
          VREnvironment.setAbsolutePosition (viewer, viewerpos);
          viewerori = new Quaternion(t, new Vector3D(0.0, -1.0, 0.0));
          viewerori = Quaternion.Multiply(viewerori, new Quaternion(-0.4, new
Vector3D(1.0, 0.0, 0.0)));
          VREnvironment.setAbsoluteOrientation (viewer, viewerori);

          Component.RunComponents ();
        }
    }
}
```

SeaWorld.java

```
/** class SeaWorld

   test application for swirling seaweed, rising bubbles, falling rocks and swimming
jellyfish.
*/

import java.io.*;

class SeaWorld
  {
    private static double timestep = 0.01, maxtimestep = 0.011, adjustmentinterval =
1.0;

    /// convert a double to a string and truncate to "length".
    static public String doubleToString(double number, int length) {
      String foo = (new Double(number)).toString();
      return foo.substring(0, Math.min(length, foo.length()));
    }

    public static void main (String args [])
      {
        VREnvironment vrenv = new VREnvironment ();
        VREnvironment.setCurrentEnvironment (vrenv);

        VROrbitInputDevice orbitinput = new VROrbitInputDevice ();
        DeviceSource vrsource = new DeviceSource (orbitinput);

        VRSink testsink  = new VRSink("Seaweed Test", 400, 400, null);

        objectID viewer = VREnvironment.createThing ();
        Point3D viewerpos = new Point3D (0.0, 0.0, 1.0);
        Quaternion viewerori = new Quaternion(0.0, new Vector3D(0.0, -1.0, 0.0));
        VREnvironment.setAbsolutePosition (viewer, viewerpos);
        VREnvironment.setAbsoluteOrientation (viewer, viewerori);
        testsink.setViewPoint (viewer);

        VRUserActor vruser = new VRUserActor (viewer);
        Connection userinput = new Connection (vrsource, vruser);

        WorldPlane worldplane = new WorldPlane(1.0f, 1.0f, 0.0f, 0.0f, 8, 8);

        EulerIntegrator integrator = new RungeKuttaIntegrator(timestep);

        Gravity gravity = new Gravity();
        integrator.addForce(gravity);
        ReefWater water = new ReefWater(gravity);
        integrator.addForce(water);

        Floor floor = new Floor(0.01, 0.0001);

        double minrockmass = 0.05;
        double maxrockmass = 0.3;
        for(int i = 0; i < 10; i++) {
          double rockmass = minrockmass+(maxrockmass-minrockmass)*Math.random();
          PointMass rock = new SpherePointMass(rockmass, rockmass/2000.0, 0.9, 0.7,
0.4);
          rock.setPosition(new Vector3D(Math.random()-0.5, Math.random()+0.2,
Math.random()-0.5));
          gravity.addPointMass(rock);
          water.addPointMass(rock);
          integrator.addPointMass(rock);
          floor.addPointMass(rock);
          PointMassRepresentation rockrep = new
PointMassRepresentation((SpherePointMass)rock, 0.4f, 0.3f, 0.3f);
        }

        integrator.addForce(floor);
        integrator.addDiscreteEvent(floor);

        Seaweed seaweed = new Seaweed(12, new Vector3D(0.0, 0.0, 0.0), integrator,
gravity, water);
```

```
        Seaweed seaweed2 = new Seaweed(12, new Vector3D(0.5, 0.0, 0.5), integrator,
gravity, water);
        Seaweed seaweed3 = new Seaweed(12, new Vector3D(-0.5, 0.0, 0.5), integrator,
gravity, water);
        Seaweed seaweed4 = new Seaweed(12, new Vector3D(0.5, 0.0, -0.5), integrator,
gravity, water);
        Seaweed seaweed5 = new Seaweed(12, new Vector3D(-0.5, 0.0, -0.5), integrator,
gravity, water);

        BubbleCollumn bubbles = new BubbleCollumn(15, 0.1, 0.4, new Vector3D(0.3, 0.0,
-0.4), integrator, gravity, water);
        BubbleCollumn bubbles2 = new BubbleCollumn(15, 0.1, 0.4, new Vector3D(-0.2,
0.0, 0.25), integrator, gravity, water);
        BubbleCollumn bubbles3 = new BubbleCollumn(15, 0.1, 0.4, new Vector3D(0.1,
0.0, 0.45), integrator, gravity, water);
        BubbleCollumn bubbles4 = new BubbleCollumn(15, 0.1, 0.4, new Vector3D(-0.1,
0.0, -0.2), integrator, gravity, water);

        PointMass target = new SpherePointMass(1.0, 1.0/1000.0, 0.7, 0.6, 0.5);
        target.setPosition(new Vector3D(0.0, 1.0, 0.0));
        Jellyfish jellyfish = new Jellyfish(0.2, 0.02, new Vector3D(-0.3, 0.6, 0.4),
integrator, gravity, water, target, floor);
        Jellyfish jellyfish2 = new Jellyfish(0.3, 0.03, new Vector3D(0.4, 0.8, -0.2),
integrator, gravity, water, target, floor);
        Jellyfish jellyfish3 = new Jellyfish(0.15, 0.011, new Vector3D(-0.1, 0.5,
0.5), integrator, gravity, water, target, floor);

        Vector3D viewpos = new Vector3D(0.0,0.4,0.0);
        Vector3D pos = new Vector3D(0.0,0.4,0.0);

        float fogcolour [] = {0.1f, 0.2f, 0.6f, 1.0f};
        GL.ClearColor (0.1f, 0.2f, 0.6f, 0.0f);
        GL.Fogfv (GL.GL_FOG_COLOR, fogcolour);
        GL.Fogi (GL.GL_FOG_MODE, GL.GL_LINEAR);
        GL.Fogf (GL.GL_FOG_DENSITY, 0.5f);
        GL.Fogf (GL.GL_FOG_START, 0.0f);
        GL.Fogf (GL.GL_FOG_END, 3.0f);
        GL.Enable (GL.GL_FOG);

        //double angle = 0.0;
        boolean running = true;
        long oldtime = System.currentTimeMillis();
        int frames = 0;
        while (running) {
          //angle += 0.005;

          //viewpos =
Vector3D.Add(Vector3D.Multiply(0.995,viewpos),Vector3D.Multiply(0.005,pos));
          /*viewpos = jellyfish.getPosition();
          viewerpos = new Point3D(viewpos.x - 0.3 *Math.sin(angle), viewpos.y+0.4,
viewpos.z + 0.3*Math.cos(angle)); // orbit the origin
          VREnvironment.setAbsolutePosition (viewer, viewerpos);
          viewerori = new Quaternion(angle, new Vector3D(0.0, -1.0, 0.0));
          viewerori = Quaternion.Multiply(viewerori, new Quaternion(-0.4, new
Vector3D(1.0, 0.0, 0.0)));
          VREnvironment.setAbsoluteOrientation (viewer, viewerori);*/

          Component.RunComponents ();

          frames++;
          long time = System.currentTimeMillis();
          double elapsedtime = (double)(time - oldtime)/1000.0;
          if (elapsedtime >= adjustmentinterval) { // time to adjust the timestep
            System.out.println();
            System.out.print(doubleToString((double)frames/adjustmentinterval,5) + "
frames/s > simulated-real time ratio: " +
doubleToString((double)frames*integrator.getTimestep()/elapsedtime, 4));
            double newtimestep = Math.min(maxtimestep, (3.0* integrator.getTimestep()
+ adjustmentinterval/(double)frames) / 4.0);
            System.out.println(", adjusting timestep to " +
doubleToString(newtimestep, 5));
            integrator.setTimestep(newtimestep);
            oldtime = time;
            frames = 0;
          }
        }
    }
```

```
}
```

C. Sample Hopping Gait Evolution Data

Table 6. Hop evolution data from run 6.

Generation	Fitness	param[0]	param[1]	param[2]	param[3]	param[4]	param[5]	param[6]	param[7]	param[8]	param[9]	param[10]	param[11]	param[12]	param[13]
1	72.83171	0.2	0.55	0.01	0.5	5	5	1	3	2	0.02	0.3	0.5	0.01	4
2	98.82743	0.2	0.55	0.01	0.651943	5	5	0.779134	3	2.354429	0.02	0.3	0.5	0.01	4
4	120.4535	0.2	0.55	0.00286	0.651943	5	5	0.779134	3	2.354429	0.023823	0.3	0.5	0.01	4
13	332.256	0.2	0.815559	0.00286	0.651943	5	5	0.779134	3.510822	2.354429	0.032443	0.3	0.5	0.01	4
15	332.541	0.2	0.815559	0.00286	0.651943	5	5	0.779134	3.510822	2.354429	0.032443	0.3	0.483968	0.01	2.516526
87	337.7446	0.2	0.815559	0.004657	0.993326	5	5	0.779134	3.510822	2.079132	0.032443	0.3	0.483968	0.01	2.516526
92	341.008	0.2	0.815559	0.00736	0.993326	5	5	0.779134	3.510822	2.079132	0.035186	0.3	0.483968	0.01	2.516526
164	341.8737	0.2	0.815559	0.00736	0.993326	4.96562	5	0.779134	3.510822	2.079132	0.025689	0.3	0.483968	0.01	2.516526
183	344.4472	0.2	0.815559	0.00736	0.993326	4.96562	4.735305	0.779134	3.510822	1.821418	0.025689	0.3	0.483968	0.01	2.516526
239	349.7336	0.2	0.815559	0.00736	0.993326	4.96562	4.735305	0.779134	3.510822	1.058726	0.025689	0.3	0.483968	0.01	2.516526
446	350.8979	0.2	0.815559	0.00736	0.993326	4.96562	4.735305	0.779134	3.510822	1.058726	0.025689	0.3	0.483968	0.014602	2.516526
460	352.6674	0.2	0.815559	0.00736	0.993326	4.96562	4.735305	0.779134	3.510822	1.058726	0.023601	0.3	0.483968	0.008691	2.516526
474	352.7621	0.2	0.815559	0.00736	0.993326	4.96562	4.735305	0.779134	3.510822	1.058726	0.023601	0.3	0.483968	0.008691	2.21518
501	352.7622	0.2	0.815559	0.00736	0.993326	4.96562	4.735305	0.779134	3.510822	1.058726	0.023601	0.3	0.483968	0.013814	2.21518
559	362.6665	0.2	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	3.510822	1.058726	0.023601	0.3	0.483968	0.013814	3.392832
568	414.8785	0.2	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	3.510822	1.583167	0.023601	0.3	0.483968	0.013814	3.392832
593	420.0207	0.202996	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	3.510822	1.583167	0.03039	0.3	0.483968	0.013814	3.392832
602	454.6375	0.202996	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	3.510822	2.387092	0.03039	0.3	0.483968	0.013814	1.405062
1332	459.1914	0.202996	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	3.510822	2.387092	0.03039	0.3	0.483968	0.010281	1.405062
1725	470.7535	0.202996	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	3.510822	2.387092	0.03039	0.3	0.483968	0.005484	1.405062
3621	540.1346	0.202996	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	5.408322	3.961812	0.03039	0.254057	0.483968	0.005484	1.405062
3657	542.7413	0.202996	0.815559	0.00736	0.993326	8.505862	4.735305	0.779134	7.277455	3.961812	0.03039	0.254057	0.483968	0.005484	1.405062
3666	552.0757	0.202996	0.815559	0.002323	1.393056	8.505862	4.735305	0.779134	7.277455	3.961812	0.03039	0.254057	0.483968	0.005484	1.405062
3672	558.6879	0.202996	0.815559	0.002323	1.393056	8.505862	4.735305	0.779134	7.277455	3.961812	0.033132	0.254057	0.49136	0.005484	1.405062
3680	599.9052	0.202996	0.815559	0.001758	1.393056	8.505862	4.735305	0.779134	7.277455	3.961812	0.049836	0.254057	0.49136	0.005484	1.153858
3801	601.232	0.202996	0.815559	0.001758	1.393056	8.505862	4.735305	0.779134	7.277455	3.961812	0.049836	0.254057	0.392194	0.005484	1.153858
4029	603.6041	0.202996	0.815559	0.001758	1.393056	8.505862	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.392194	0.005484	1.153858
4054	663.1124	0.202996	0.815559	0.001758	1.393056	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.392194	0.005484	1.153858
4206	669.2489	0.202996	0.815559	7.11E-04	1.393056	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.392194	0.009531	1.153858
4564	671.0384	0.202996	0.815559	7.11E-04	2.263017	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.392194	0.009531	1.153858
4690	687.8718	0.202996	0.815559	7.11E-04	2.263017	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.392194	0.013685	1.153858
4903	688.3639	0.202996	0.815559	7.11E-04	1.113043	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.619986	0.011176	1.153858
5104	699.1869	0.202996	0.815559	6.56E-04	1.113043	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.619986	0.011176	1.153858
5168	705.3337	0.202996	0.815559	8.91E-04	1.113043	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.619986	0.011176	1.153858
5312	714.6791	0.202996	0.815559	8.91E-04	1.113043	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.827603	0.019272	1.153858
5340	734.3799	0.202996	0.815559	8.91E-04	0.841864	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	0.827603	0.024613	1.153858
5579	763.1431	0.202996	0.815559	8.91E-04	0.858001	9.368097	4.735305	0.779134	7.277455	2.4011	0.049836	0.254057	1.44706	0.024613	1.153858
5924	764.1444	0.202996	0.815559	8.91E-04	0.858001	9.368097	4.735305	0.779134	7.277455	1.290349	0.031487	0.2491	1.44706	0.024613	1.153858
5937	776.2422	0.202996	0.815559	0.001426	0.858001	9.368097	4.735305	0.779134	7.277455	1.290349	0.031487	0.2491	1.44706	0.041202	1.153858
5944	780.2547	0.202996	0.815559	0.001426	0.858001	9.368097	4.735305	0.779134	7.277455	1.290349	0.031487	0.2491	1.44706	0.041202	1.13311
6076	793.8003	0.202996	0.815559	0.001426	1.462128	9.368097	4.735305	0.779134	7.087769	1.290349	0.040517	0.2491	1.44706	0.051436	1.13311
6090	827.2725	0.202996	0.815559	7.21E-04	1.462128	9.368097	4.735305	0.803132	7.087769	1.290349	0.040517	0.2491	1.44706	0.051436	1.13311
6299	830.5289	0.202996	0.815559	7.21E-04	1.462128	9.368097	4.735305	0.803132	9.522653	1.972362	0.040517	0.2491	1.44706	0.051436	1.548576
6373	844.2516	0.202996	0.815559	9.73E-04	1.462128	9.368097	4.735305	0.803132	9.522653	1.972362	0.040517	0.2491	1.44706	0.051436	1.548576
6483	846.5078	0.202996	0.815559	9.73E-04	1.462128	9.368097	4.735305	0.803132	9.522653	1.972362	0.051647	0.2491	1.44706	0.051436	1.9154
6740	862.2766	0.202996	0.815559	9.73E-04	1.462128	9.368097	4.735305	0.803132	10.03219	1.972362	0.051647	0.2491	1.530421	0.036266	1.9154
7038	878.8873	0.202996	0.815559	9.73E-04	1.462128	9.368097	4.735305	0.803132	10.03219	1.972362	0.051647	0.2491	1.530421	0.036266	3.157892
7902	881.4445	0.202996	0.815559	9.73E-04	1.462128	9.368097	4.735305	0.803132	10.03219	1.972362	0.051647	0.2491	1.530421	0.022208	3.157892
8158	883.721	0.202996	0.815559	9.73E-04	1.462128	9.368097	4.735305	0.803132	10.03219	1.972362	0.051647	0.2491	1.530421	0.009517	3.157892
8927	887.432	0.202996	0.815559	9.73E-04	1.462128	9.368097	4.735305	0.803132	10.03219	1.972362	0.051647	0.2491	1.530421	0.008652	3.157892

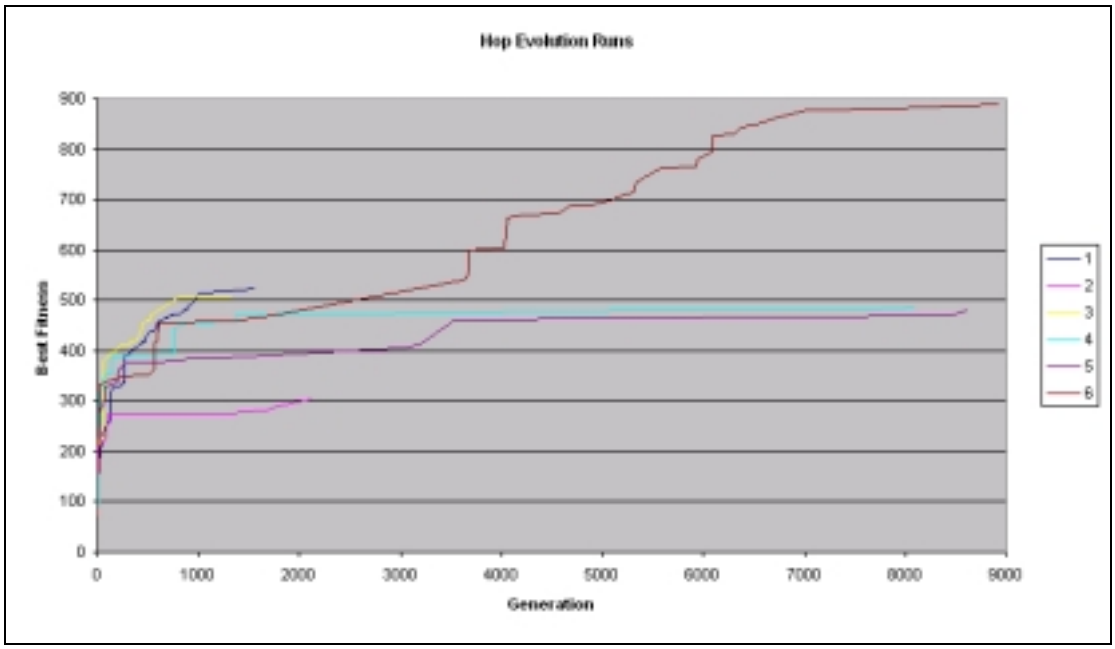


Figure 20. Best fitness as a function of generation.