VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Adaptive Behaviour in Physically-Based Simulation

Timothy Field

Supervisor: Marcus Frean

5 November 2001

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Two of the broad aims of Artificial Intelligence are to understand the nature of cognition and to engineer rational agents which adapt to their environment. Evidence suggests that these goals are inseparable: engineering agents which exhibit adaptive behaviour may be the best way to understand cognition.

This report describes the building of a framework for research into adaptive behaviour using physical simulation of evolving agents which learn through reinforcement.

Contents

Chapter 1

Motivation

1.1 Adaptive Behaviour

An agent in the Artificial Intelligence sense is anything which perceives its environment through sensors and acts upon its environment through actuators [RN95]. An *adapting* agent maintains some coherent structure under varying environmental conditions [Pfe98]. Adaptive behaviour is exhibited by adaptive agents which optimize their behaviour to suit this changing environment bounded by their physical and computational resources.

Engineering adaptive agents has long been a goal of Artificial Intelligence. This project investigates the task of engineering such adaptive agents in a physically simulated environment.

1.2 Embodiment and Situatedness

> "An ant, viewed as a behaving system, is quite simple. The apparent complexity of its behaviour over time is largely a reflection of the complexity of the environment in which it finds itself."
>
> *Herbert A. Simon (1969)*

The influence of the environment and the body in adaptive behaviour has traditionally been ignored in Artificial Intelligence. Early efforts to understand intelligence in a computational paradigm sought enlightenment through the study of algorithms. Applying the familiar tools of abstraction, top-down analysis and divide-and-conquer resulted in the subdivision of the intelligence problem into the separate subproblems of perception, planning, and action. These subproblems were further specialised into the techniques such as natural language processing, vision and knowledge representation.

Unfortunately, as researchers began to build robots, the realisation slowly dawned that the hardest issues in the study of intelligence do not involve playing chess, but rather perception and action in the real world. Their failure in engineering robots to perform even simple tasks lead to a widespread rejection of so-called Good Old-Fashioned AI. This heralded a new era of connectionism and bottom-up robotics which recognised the complexity and nonlinearity in the real-world. The fundamental tenets of this new approach stress the importance of embodiment and situatedness in the real-world. These principles emphasize the role of an agent's own body and its immediate physical environment in its cognition [Bee01]. In particular, the grounding afforded by embodiment dramatically simplifies problems that seem intractable at the purely computational level.

An example of a method which embraces these principles is Brooks' behaviour-based approach [Bro91]. Instead of decomposing intelligence along the problem dimension, the overall behaviour of an agent is decomposed into a network of behaviours which communicate via excitation and inhibition. While this approach produced relatively impressive results, its dependence on designing behaviours ignores the fundamental insight of embodiment that behaviour is a result of the dynamical interaction between the agent and the environment.

This insight suggests that intelligence is an emergent property of complex dynamical interactions between the coupled systems of control, body, and environment. The complexity of an agent's behaviour is not necessarily due to a complex behaviour control mechanism but may simply arise from the complexity of the environment. While the behaviour-based approach couples the agent with the real-world, the problem still remains that an explicit design of behaviours must be made. The difficulty of engineering these lies in the fact that, in general, there does not exist a one-to-one mapping between subcomponents of an agent's control system and the subcomponents of the agent's observed behaviour [NF00]. Thus, engineering these subsystems based on an intuitive understanding of the desired distal behaviour cannot succeed in general.

1.3 "Simulated" Embodiment?

Brooks made great efforts to avoid simulating robots and stressed the importance of physical embodiment. The shunning of simulation was defended by two arguments [Bro92]. Firstly, he argues that, in simulating robots, effort will inevitably go into solving problems that do not come up in the real world. Secondly, he argues that a faithful simulation of the dynamics of the real world is difficult and costly, whereas in physical embodiment, the real world provides agents with a consistent model "for free".

However, it is debatable whether the resulting abandonment of simulation is desirable or even necessary. The crucial aspect of embodiment and situatedness is that it forces the agent into sensing and acting in a complex dynamical system. If we are investigating the emergence of adaptive behaviour then the first argument is not applicable. In fact, simulations have the desirable property of providing the ability to more easily investigate which aspects of the environment are necessary for agents to be able to adapt their behaviour [Jak98]. Secondly, due to the progress of computational resources and research, the quality of physical simulation has surpassed that of ten years ago when Brooks' concerns were voiced.

More important than these concerns, however, is the fact that the interdependence resulting from the coupling of the body and controller means that, given an inability to decompose and analyse the mechanisms of behaviour, designing agents from the top-down which perform sensory-motor coordination tasks is infeasible. Moreover, the contrary approach of incrementally improving designs is also infeasible for physical robots. Brooks argued that such modification may be achieved through progressively enabling sensors and actuators [Bro92]. However, this ignores the immense engineering difficulties involved in incrementally modifying the shape of real-world robots and their sensor and effector placements [MC95].

While the computational resources to simulate an accurate, consistent and stable physical model are not insignificant, there are several other advantages to using simulations over physical robots. Firstly, we avoid the difficulty, cost and time involved in building physical robots. Secondly, the ability to reproduce results is improved.

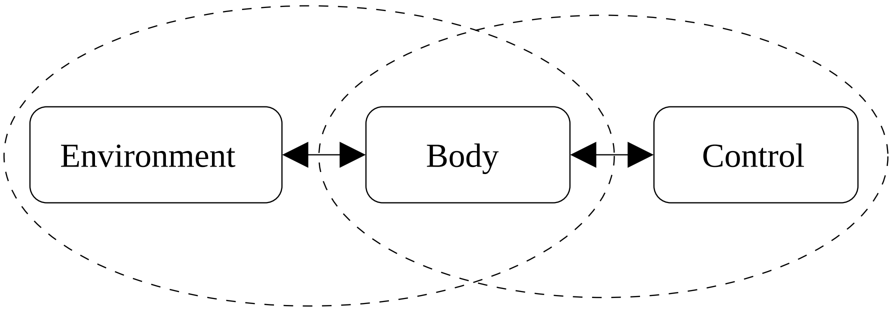

Figure 1.1: **Interdependence of the elements of an adaptive system.**

1.4 Ecological Balance

The close interrelationship of the body, mind and environment in adaptive behaviour has been termed "ecological balance" by Pfeifer [Pfe00]. He states that "given a particular task environment, there is a harmonious relationship between morphology, materials and control." His viewpoint refines the position of embodiment and situation, by implying that there exists a complexity tradeoff between these interdependent aspects. An example of this tradeoff is the fixing of one aspect while adapting the others. For example, in [LE99] the sensor distribution of a mobile robot was evolved, while the controller remain fixed. In contrast, having the right morphology and the right materials may significantly simplify control. An example of this tradeoff between morphology and materials for computation is the passive dynamic walker introduced by McGeer [McG90]. This robot is capable of walking down an incline without control (i.e. with no actuation) by taking advantage of the physical characteristics of itself and the environment.

These examples demonstrate the interrelationship between all aspects of an agent. However, in [BP01] a stronger argument is made that, in order to produce truly complex behaviour, simultaneous adaptation of these aspects is required. The argument is based on the concept of an extra-dimensional bypass as introduced by Conrad in [Con90]. Given a fitness landscape with multiple adaptive peaks in the topology of the search space, it is possible that the introduction of an extra-dimension in the search space allows a localised search to move between the otherwise unconnected peaks via "bridges" in the higher-dimensional search space. These connections are called extra-dimensional bypasses. The hypothesis argues that restricting adaptation to one aspect of the system, such as the controller, makes the optimization problem harder as the extra-dimensional bypasses cannot be utilised. It is not argued that the addition of the extra morphological dimensions to the problem gives better behaviour because evolution discovers objectively better morphologies. Rather, the addition of morphological parameters transforms the topology of the search space through which the evolving population moves, creating connections in the higher dimensional space between the separated adaptive peaks in the lower dimensional space.

1.5 Previous Work

Adaptive behaviour in physically-based simulated environments is a young field. The seminal work in [Sim94b, Sim94a] used simultaneous evolution of morphology and control structures of agents in physically-based simulations. Sims highlighted the importance of simultaneous adaptation of morphology and controller by evolving virtual creatures which demonstrated life-like behaviour across several locomotion tasks. The morphologies were specified using a recursive, graph-based representation. The controllers consisted of networks of functional nodes computing outputs such as sinusoidal and sawtooth waves, relational and arithmetic operators, thresholding and rate of change functions. The sensory input consisted of proprioceptive, contact and photo-sensors.

Ventrella [Ven96] evolved similar creatures, although his controllers were restricted to an oscillating function parametrised by the joint position. Framsticks [KU99] again evolves similar creatures with a tree-like morphology simulated by finite-element methods. They evolve the weights and activation functions of neural network controllers. In [BP01, PB01] locomotion of a biped robot was evolved under fixed morphological structure, but varying mass distribution. In [BP00] the correlation between morphological symmetry and locomotive efficiency is explored in evolved agents. Finally, in [TRG96], a learning algorithm is used to generate behaviours for fish with three-dimensional body plans, which can deform and locomote within a simulated physics-based environment.

Having argued that the simulation of the real-world as a complex dynamical system is a viable and desirable approach to the engineering of adaptative agents, we now turn to details of simulating such a system.

Chapter 2

Simulating a Physical Environment

The physical world can be regarded as a continuous dynamical system. A dynamical system is a mathematical object which unambiguously describes how its state evolves over time. More formally, it can be regarded as a tuple $\langle T, S, \phi_t \rangle$ consisting of an ordered time set T, a state space S, and an evolution operator $\phi_t : S \to S$ that transforms an initial state $x_0 \in S$ at time $t_0 \in T$ to another state $x_t \in S$ at time $t \in T$. The important distinction to note is that the transition from one state to the next is dependent only on the state of the system and not on the history of states.

Simulating the physical world involves finding representations of the state of the world, and choosing an evolution operator to describe state transitions. The exact nature of the state space and the evolution operator depend upon which physics we choose to simulate. Because only a subset of physical behaviour is interesting or applicable to an adaptive agent, assumptions can be made about the types of physics to model. Classical physics can arguably[1] account for the real-world interaction experienced by adaptive organisms.

2.1 Assumptions

Classical physics describes the evolution of the physical world by a set of differential equations. The primary equation is Newton's second law of motion, $\ddot{x} = F/m$, which relates the evolution of the position x of a body to the total force being applied to it, F, and its mass m. The forces being applied to bodies can come from a variety of sources, including force laws (such as gravity, drag, springs, attraction and repulsive forces), contact forces and constraint forces. This second-order equation is converted to a first-order one by introducing another variable to create a pair of coupled first-order ordinary differential equations, $\dot{v} = F/m$ and $\dot{x} = v$.

Inevitably, some simplifying assumptions must be made in order to yield a tractable system. The first simplifying assumption that is typically made in physical simulation is one of perfect rigidity. That is, the objects being modeled have static shape and are non-deformable. Other simplifying assumptions include limiting the geometry of the objects to convex polyhedra with constant density. This simplifies collision detection and volume calculation.

Thus, given these assumptions, an object's state at any point in time can be uniquely determined by its position, orientation, and linear and angular momentum[2]. The physical state of the entire system is simply the combination of the state of all of the bodies of the system.

[1] However, see [Pen94] for an argument that interactions at a quantum scale in the brain are essential for cognition.

[2] Substituting velocities instead of momentum into the state is possible, however it complicates the derivations.

2.1.1 Solving the Differential Equations

While an analytical solution of the equations of motion is possible in special cases (such as pendulums or single-body orbits), in general it cannot provide the extensibility needed to simulate the interaction of general geometries with arbitrary constraints in a wide range of configurations. Thus, because the differential equations specify continuous motion and we are limited to simulating discrete time steps on a computer, the simulation must use numerical methods to approximate the evolution of the system.

Numerical integration calculates an approximate change in state over a small time interval and then adds this change to the existing state to evolve the system forward in time. Techniques for numerical integration are based on approximating the Taylor series expansion of the function, which expresses the value at the end of the step as an infinite sum involving the value and derivatives at the beginning:

$$x(t+h) = x_t + h\dot{x}_t + \frac{h^2}{2!}\ddot{x}_t + \frac{h^3}{3!}\frac{\partial^3 x}{\partial t^3} + \ldots + \frac{h^n}{n!}\frac{\partial^n x}{\partial t^n} + \ldots$$

The accuracy of the integration is determined by the number of higher-order derivatives calculated and the step-size, h. Inaccuracy in integration can cause problems as shown in Figure 2.1 or, in the worst case, can lead to instability and cause the system to diverge.

While a more accurate solution allows for a larger step-size, evaluating these higher-order derivatives increases the computational cost per step. Thus, a trade-off exists between the greater step-size afforded by more sophisticated integration methods and the computational cost of evaluating it. Typically, a good trade-off is achieved by the fourth-order Runge-Kutta algorithm which gives an accuracy of $O(h^4)$.

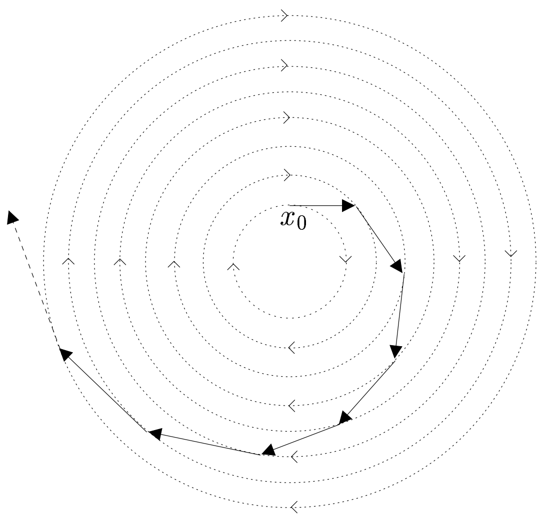

Figure 2.1: **An example of inaccuracy in physical simulation.** Consider a particle lying on the given circular vector field which specifies its velocity, $\dot{x}_t$. The correct trajectory given the initial point x_0 is for the particle to maintain a stable orbit around the origin. If, however, the particle's position is evolved using only the first-order derivative (i.e. its velocity) as in Euler's method for numerical integration, then the particle is doomed to spiral away from the origin.

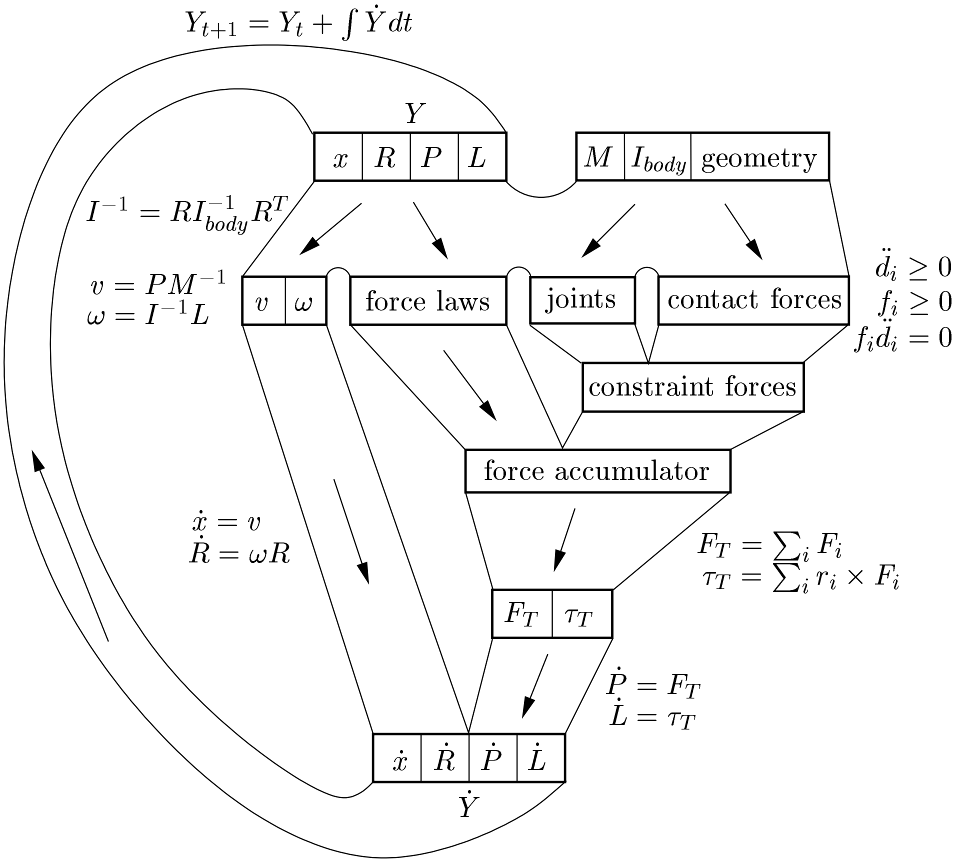

Figure 2.2: **Flow of information in a rigid-body dynamics simulation.** State vectors, Y, describing the position, orientation, linear and angular momentum of bodies are combined with masses (M, I_{body}) and geometries to compute the linear and angular velocity, v and ω. Force laws describing global and n-body forces such as gravity, springs and drag, are accumulated with forces computed to maintain non-interpenetration and joint constraints resulting in the total force and torque acting on bodies. These auxiliary quantities determine the state derivative, $\dot{Y}$, which is integrated to produce the new state, Y_{t+1}.

2.2 Simulation Complexities

Simulating the motion of objects as they float through space following parabolic trajectories is relatively straightforward. The complexities of rigid-body dynamics stem from two aspects of the simulation: *contacts* and *constraints*.

2.2.1 Contacts

Contact exists between two bodies when they touch each other. There are three particular difficulties in handling contacts:

- **Interpenetration detection.** Efficient detection of interpenetrating bodies is a difficult problem in general. Approaches to the problem typically use a hierarchy of detection methods. Initial pruning of possible colliding pairs using coarse bounding volumes (such as boxes or spheres) is applied before more expensive methods such as closest-feature tracking and, if necessary, per-polygon intersection testing. However, restricting bodies to regular geometries (such as boxes or spheres) as opposed to arbitrary polyhedra simplifies the detection problem immensely.

- **Collision resolution.** If two bodies are contacting and they have a velocity towards each other then a collision occurs. Poisson's empirical law of collisions relates the velocities of the bodies before and after the collision. The rigid-body assumption means that this change must occur over infinitesimal time in the form of an impulse. The result is a discontinuous change in the velocity of the colliding objects. While the computation of these impulses is straightforward, difficulties arise from this discontinuity. The numerical integrator assumes that the system is continuous so any discontinuity means that the integrator must be stopped and restarted. Algorithms for managing this interaction between collision response and integration include: *conservative advancement* (where objects are evolved forward based on conservative estimates of interpenetration time), *retroactive detection* (where detection of interpenetration forces the system to roll-back to a pre-interpenetration state), and more sophisticated techniques such as the *Timewarp* algorithm [Mir00] (which partitions the simulation into contact groups and allows parallel evolution of bodies at different rates).

- **Resting contact force computation.** If two bodies are touching and a collision doesn't occur, then the result is resting contact. Approaches to simulating resting contact include modelling it as a series of micro-impulses [Mir98b] or exactly computing contact forces via analytical solution [Bar94]. The analytical solution involves solving a linear complementary problem which contains the conditions of non-interpenetration, and conservative and repulsive contact forces. While solving this LCP is difficult in general, the algorithm can take advantage of structure of inequalities to improve performance. Under worst case conditions, the solution takes exponential time, but in practice these degenerate cases never occur and the algorithm performs in $O(n)$.

2.2.2 Constraints

Constraints are algebraic equations of state which are required to be satisfied at each time-step. They are primarily used to simulate joints, such as fixed joints, hinges, and ball-socket joints. The computation of forces which maintain constraints is, in general, a difficult problem. Instability and stiff systems can arise in non-analytic methods such as the penalty method (which doesn't enforce the constraints totally but pulls bodies towards satisfaction of the constraint over time). Luckily, many useful holonomic constraints (i.e. those constraints that are strictly geometric) can be enforced using the contact force algorithm by specifying an additional non-separation condition.

Other alternatives to explicitly enforcing constraints include finite element methods and reduced-coordinated formalisms (such as the $O(n)$ articulated-body dynamics algorithm [Fea84]). The latter is particularly relevant for simulating robot morphologies as it depends on the connectivity of the objects being trees.

2.3 The DROLE Simulation Engine

In order to support research for the project, I developed a group of applications to simulate adaptive agents. The DROLE simulation engine is one application which simulates three-dimensional physically-based environments. The relevant features of this application are:

- **Extensible object-oriented simulation.** Every physical body in the environment is an object whose properties can be accessed and modified by the user at run-time. Objects can be added and removed from the environment at run-time. Objects form hierarchies based on joint connectivity which can be loaded at run-time from XML description files[3]. The system is built using the model-view design pattern which ensures high modularity and low coupling between the various subsystems. Properties of the subsystems can be accessed and modified at run-time.

- **Rigid body dynamics simulation.** The motion of objects in the system is governed by the laws of rigid-body physics as described in this chapter. Various physical subsystems were built which rely on existing libraries such as the ODE engine[4] for contact and constraint force computation and the MultiBody dynamics library [Mir98a] for articulated body simulation. Collision detection is implemented using the SWIFT++ library [EL01]. The main simulation loop of DROLE uses retroactive detection (with binary search to discover interpenetration times) to resolve collisions.

- **3D visualization.** The user interacts with the environment via a fully 3D interface. The user is free to move and orientate the viewpoint anywhere in the environment. Objects can be focused upon by selecting with the mouse or keyboard and their properties are overlaid on the 3D view of the environment. Objects can be manipulated by dragging the mouse which imparts forces on the objects. The interface utilises textures, lighting, reflections and shadows to provide orientation and distance cues to the user. Figure 2.3 shows some examples of this visualization.

Given this physically simulated environment, we now turn to the problem of representing and situating agents in this environment.

[3]See Appendix B for details.

[4]See *http://www.q12.org/ode/ode.html*

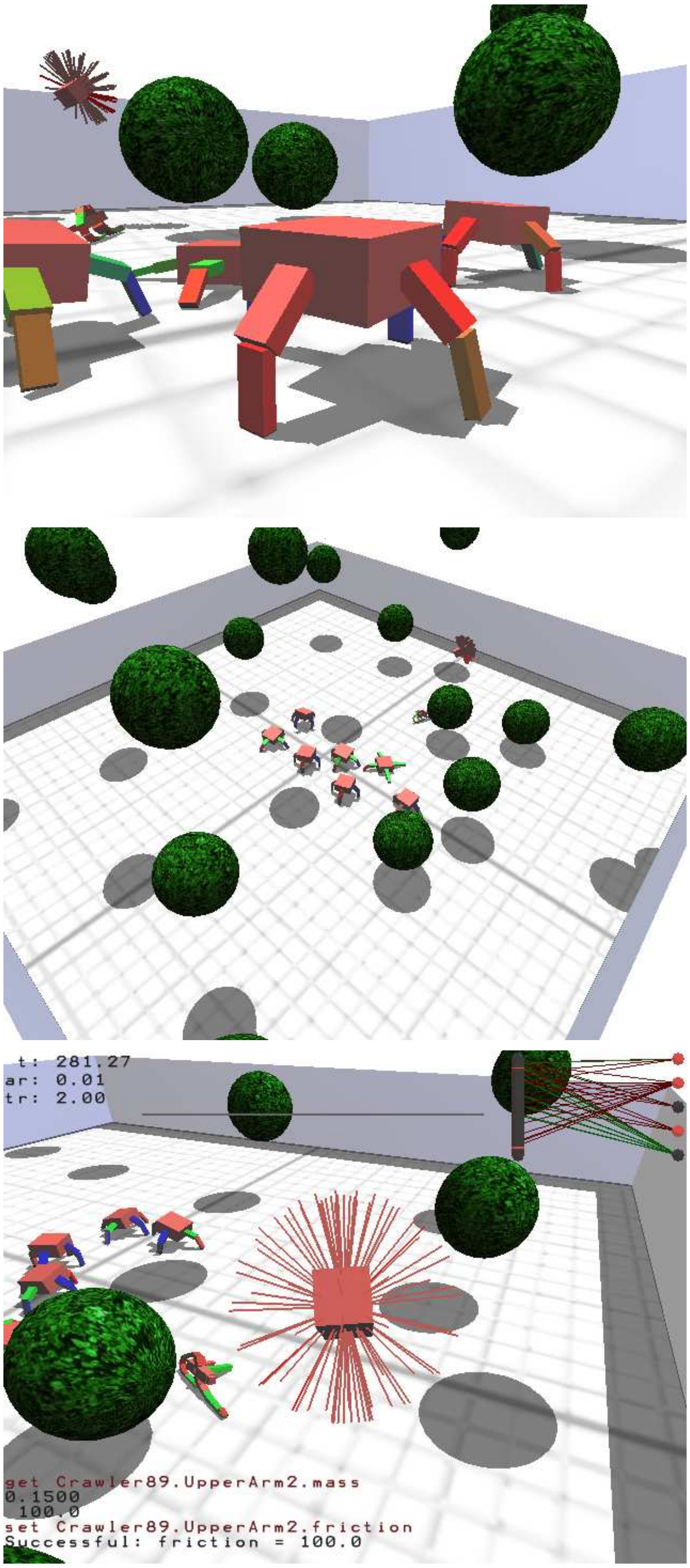

Figure 2.3: **The DROLE simulation engine.**

Chapter 3

Situating Agents

A situated agent can be partitioned into three separate (but interdependent) aspects: its *morphology*, *materials* and *control*. This chapter looks at representing the first two aspects of agents in a physical environment.

3.1 Morphology

The morphology of an agent consists of the shape and connectivity of its parts and the placement of its sensors and actuators. It includes the various geometries and sizes of each of the parts, as well as how each part is connected together. While in biology morphology has always been a central research topic, the influence of morphology on adaptive behaviour has largely been unexplored.

Given that we are simulating agents as rigid bodies connected by joints, the structure of the agent may be represented as a tree. Each vertex of the tree corresponds to a segment of the agent's body, and the edges connecting vertices correspond to joints. In DROLE the morphology is simplified by assuming that these segments are boxes and are only connected via two types of joints: the hinge joint (with 1 degree of freedom) and the fixed joint (with no degrees of freedom). Also, joints are restricted to locations on 18 sites on the segment: the 6 faces and 12 edges of the box.

The hinge joints connecting the segments have two parameters which influence its behaviour:

- **Range.** The range of a hinge determines which angles the hinge can move through. Constraint forces are applied to restrict motion to within this range.

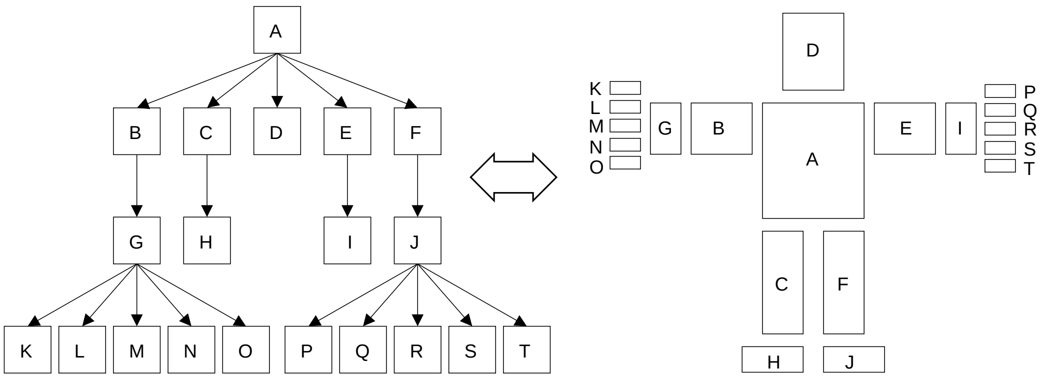

Figure 3.1: **An example of representing complex morphologies as a tree.**

- **Bounciness.** The bounciness of a hinge determines how elastic the joint constraint is.

The details of the representation of the morphology can be found in Appendix B.

3.2 Materials

Each segment in the morphology has several scalar parameters corresponding to its material properties:

- **Density.** The density of a segment is used to compute its mass and inertia tensor via integrating over the volume.
- **Friction.** The friction property is a measure of the static and dynamic friction of the segment. It can range from 0 (no friction, the segment slides around) to infinity (total friction, no sliding).
- **Elasticity.** The elasticity property is a measure of the coefficient of restitution of the segment. This property is used to determines how much energy is lost in collisions. A value of 1 corresponds to a perfectly elastic collision which conserves all the energy of the bodies.
- **Damping.** The damping property is used as an approximation to fluid resistance. A segment's linear and angular momentum is multiplied by its damping parameter to apply a force resisting its movement.

3.3 Sensors

An agent's sensors observe the state of the environment. Perfect sensors which correctly observe the entire state of the world are unrealistic. For a situated agent, the only observables available to it are the relative positions and orientations (and their respective rate of changes) of the objects in the environment and the forces being applied to them. This respects the requirement of embodiment that only grounded representations are used [Har90], and no abstract information is available to the agent.

Sensors are situated on segments and hence all observations are relative to the agent. The sensors defined in DROLE are:

- **Proprioceptive.** Proprioceptive sensors detect the motion or position of segments of the agent's own body. For hinge joints, two types of proprioceptive sensors are possible: one measuring the angle of a hinge joint connecting two segments while the other measures the rate of change of the angle.
- **Haptic.** Haptic sensors detect contact between an agent and any other body in the environment.
- **Vision.** A vision sensor emits a signal when an object falls within its viewing cone as defined by its field of view and range parameters.

Other possible sensor types which weren't modeled include: *tactile* sensors for detecting surface properties of objects and pressure change during contact, *audio* sensors for sound detection and location and *olfactory* sensors.

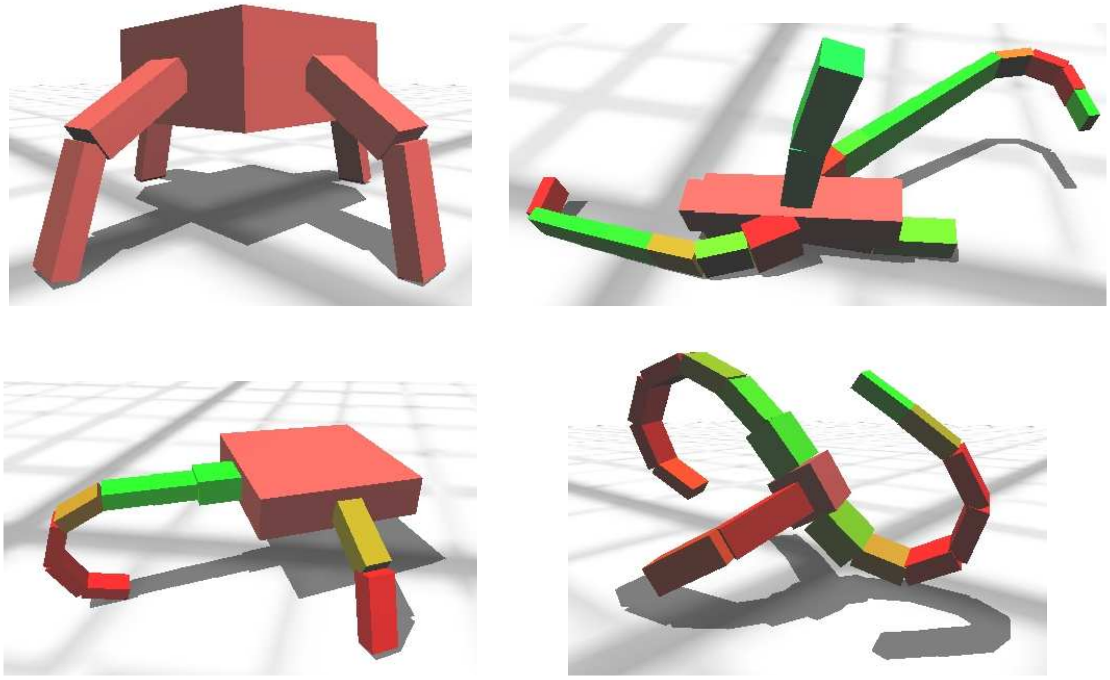

Figure 3.2: **Examples of situated agents in DROLE.**

3.4 Actuators

An agent interacts with its environment via actuators. Actuators translate signals received from the agents control system into forces or torques to provide *locomotion* (moving the agent through the environment), *manipulation* (moving other objects) and *modification* (changing physical aspects of other objects) functions. In DROLE the actuators apply forces to the segments they are placed on or, alternatively, apply torque to hinge joints.

Given we have situated agents in a physically-based environment, we are in a position to discuss the behaviour of agents and the adaptation and optimization of this behaviour.

Chapter 4

Adapting Agents

This chapter describes how agents interact with their environment, and discusses the methods used to adapt an agent's behaviour to its environment.

4.1 Interacting with the Environment

We have seen how an agent is connected to its environment via perception and action with its sensors and actuators. So, how does the interaction with the environment proceed?

At time t, an agent receives a percept from the environment, y_t, which is its observation of the state x_t, of the environment. It then chooses an action, a_t, which changes the environment. In discrete simulations, this interaction is usually iterated once for every simulation step. However, in physical simulations, the time steps of the simulation are so small that the state evolves very slowly. Thus, the period between interaction is usually a multiple of the simulation step size.

This sequence of states of the environment and the actions that the agent performs in them at each interaction step defines the behaviour of the agent. More formally, the behaviour is defined as the agent's trajectory through state-action space.

4.2 Optimizing Behaviour

Adaptation is the problem of optimizing this behaviour. In order to discuss the optimality of a behaviour, we must specify an objective performance measure. This performance measure is a total order on the set of trajectories through state-action space. This corresponds to the

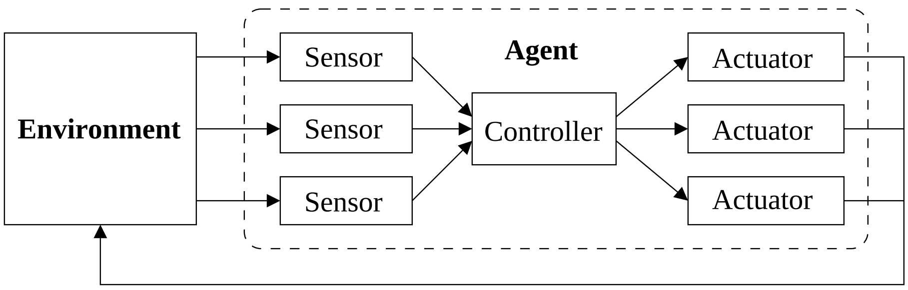

Figure 4.1: **The interaction between sensors, actuators, the controller and the environment.**

notion of the 'fitness' of an agent. The performance measure (or 'fitness function') depends on the specified task environment.

An agent's behaviour is the result of interaction between the agent and its environment. This interaction depends on all the parameters specifying the agent, including the morphology and control, as well as the dynamics of the environment. Thus, there is a mapping from these parameters to behaviour via the dynamics. Composing this mapping from parameters to behaviour with our mapping from behaviour to fitness, gives the intuitive notion of a 'fitness landscape'. The problem of generating adaptive behaviour is akin to searching for the peaks of this fitness landscape. As we have seen, an agent situated in a physical environment consists of a bewildering range of physical parameters without even considering its controller. We need some way of searching in this high-dimensional space.

4.3 Evolutionary Algorithms

One technique for optimizing these parameters is by using *evolutionary algorithms*. Evolutionary algorithms are meta-heuristic optimization methods inspired by Darwin's theory of natural selection.

The algorithm initially generates a population of possible solutions (individuals). From these, two or more of the best individuals are selected as parents. A pool of offspring for the next generation is created through recombination of the parents' parameters (genomes) and some of these offspring are perturbed (mutated). The best offspring are selected to form the next generation of solutions. The method iterates until a stopping criteria is met. As in Darwinian evolution, any changes to the parameters made throughout its lifetime are not incorporated back into the genome, but die with the individual.

4.3.1 Evolution in DROLE

The DROLE evolution engine (Figure 4.2) is a tool which interacts with the simulation engine to implement an evolutionary strategy. It maintains multiple populations of individuals at a persistent central repository, which the simulation engine queries to receive simulations to run. The simulation engine then uploads results to the web site. The evolutionary algorithm is implemented as an application which also interacts with this repository.

For population maintenance, a *steady-state* genetic algorithm is used to continuously add and remove individuals from the population. The steady-state approach lends itself to the distributed nature of the tool. The tool monitors the state of the population, and once all active individuals in the population have completed a minimum number of simulations, then a new offspring is added to the population and an individual is made inactive. A number of reproduction selection methods are implemented, including roulette-wheel selection (where parents are chosen proportional to their fitness), elitist selection (where the best individuals are chosen), and tournament-based selection (where a group of random individuals are chosen, and the highest-ranking individual is chosen with some probability).

The numerical parameters of an agent are converted to a vector by flattening its structure via depth-first traversal. Mutation is performed by perturbing this vector by Gaussian noise. The validity of solutions is maintained by repairing the parameters after mutation. Any information lost through this flattening process (e.g. the morphological structural and symbolic parameters) are reconstructed using its parents as templates. This simplifies the algorithm by avoiding the need to encode and evolve the morphological structure of the agent. No cross-over is performed in DROLE. Parent's genomes are simply recombined by performing an averaging before mutation is applied.

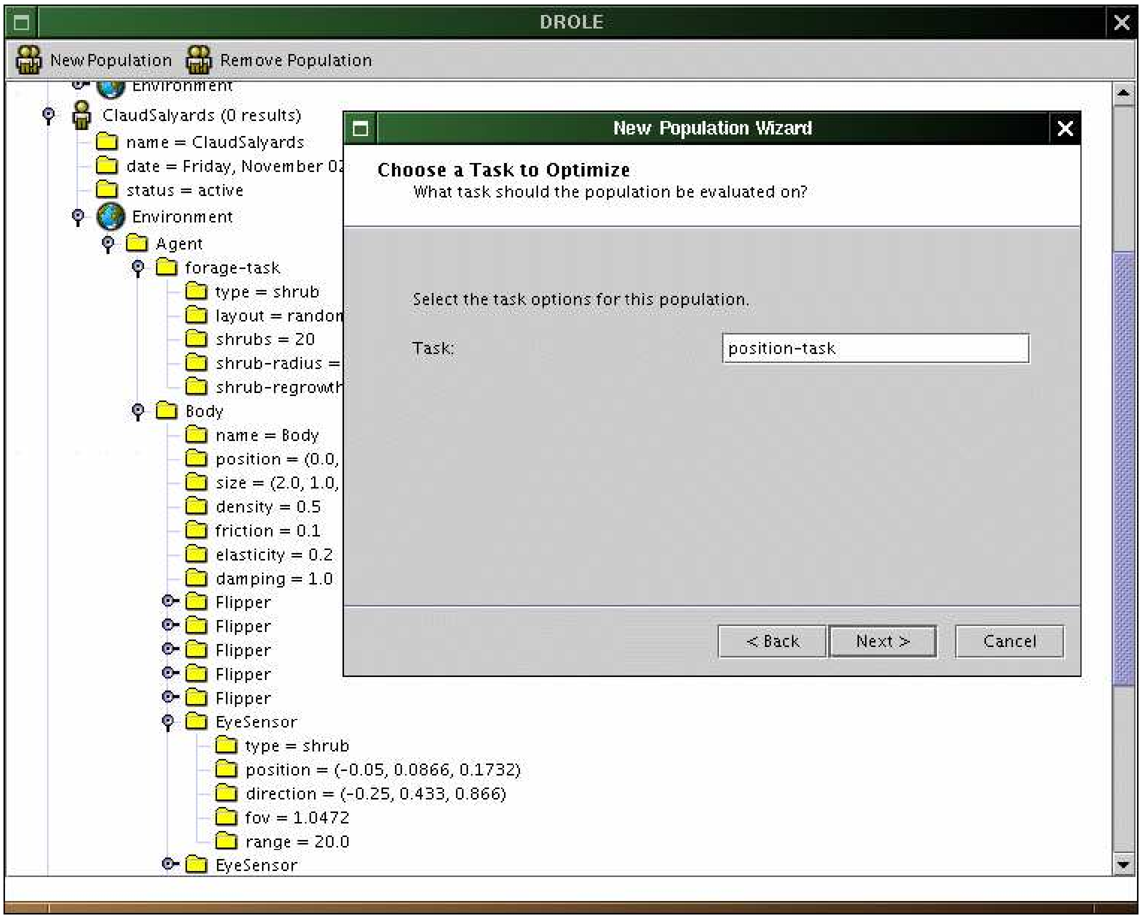

Figure 4.2: **The DROLE evolution engine.** This tool evolves populations of individuals given a task environment and initial population characteristics. It allows a user to browse an individual's parameters as well as the results from all of its runs.

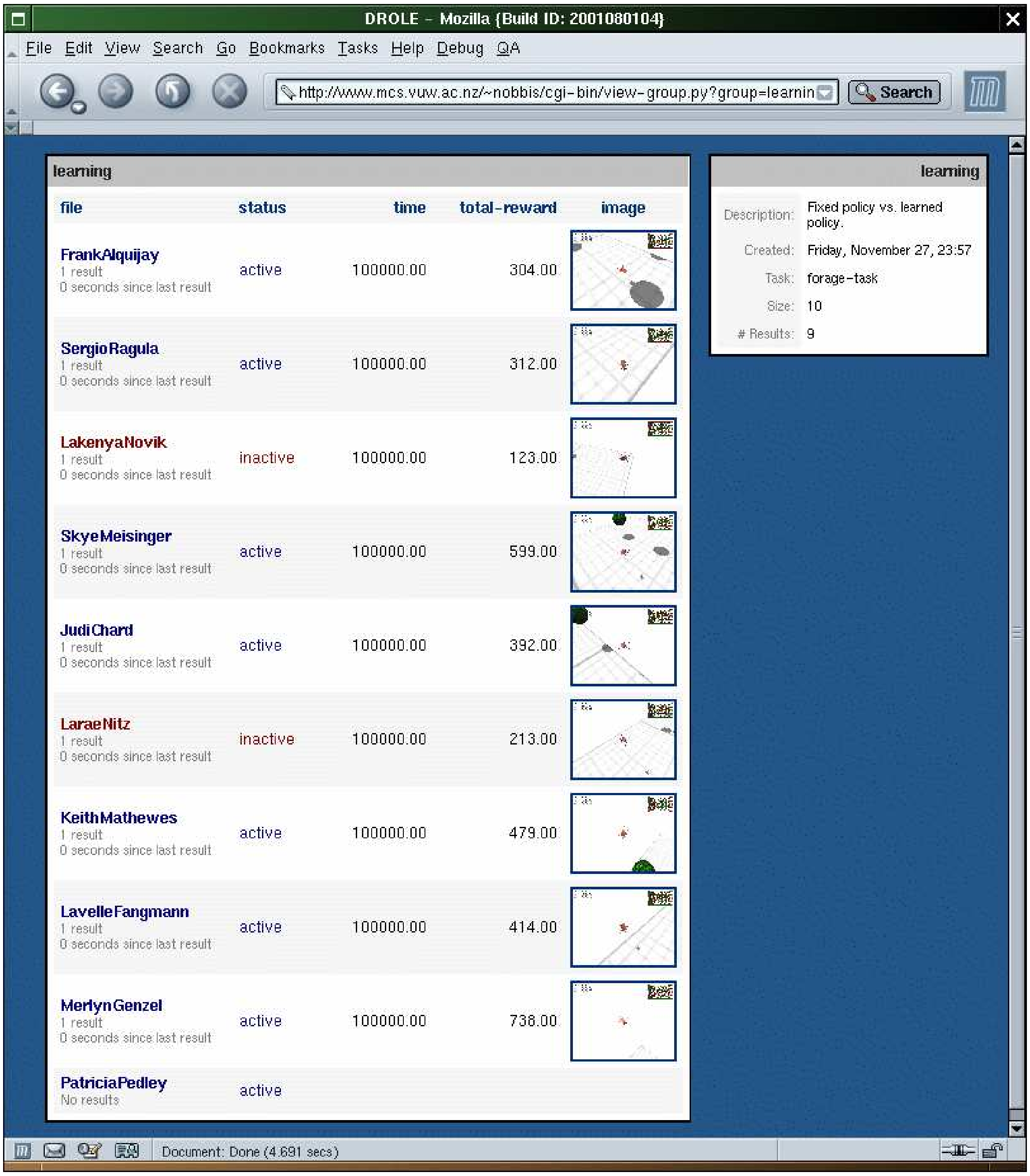

Figure 4.3: **Viewing a population on the DROLE web site.** This display allows the user to browse the individuals in a population. Each row contains an individual solution with the number of results that have been collected for it via simulation, the (simulation) time that the run lasted for, and the total reward for the experiment. A thumbnail screenshot of the experiment is given. Experiments can be disabled/enabled by clicking on the deactivate/activate hyperlink.

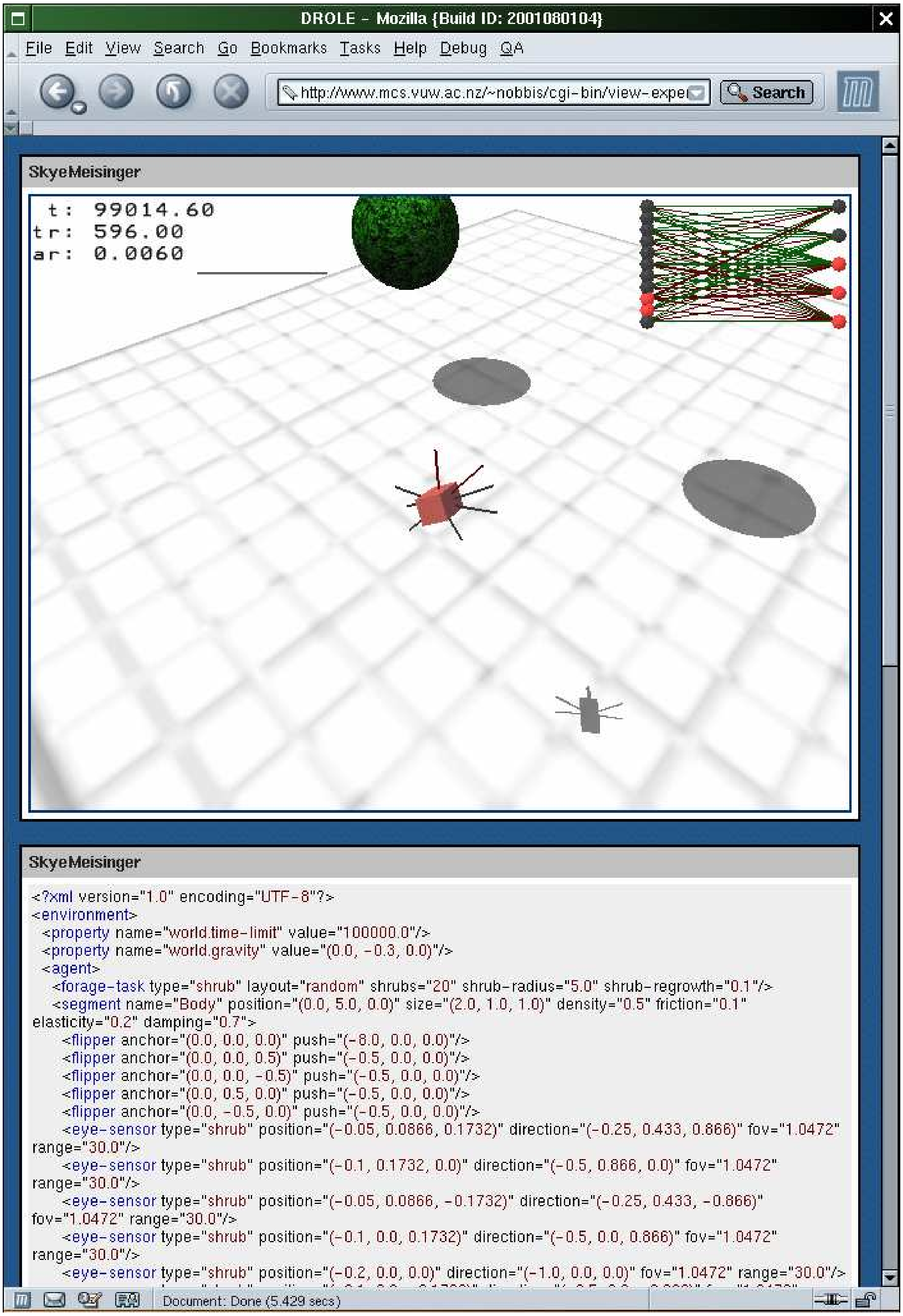

Figure 4.4: **Viewing an individual on the DROLE web site.** Selecting a particular individual allows details of it to be presented. An enlarged screenshot is given, as well as the XML file which describes all of the parameters of the experiment.

4.4 Policies

Up until now, the details of the control system of an agent have been ignored. An agent's control system can be thought of as defining a *policy*. A policy is a rule which specifies which action to perform at each time step as a function of the entire history, i.e. the complete sequence of observation-action pairs since time 0. The simplest possible policy is a *reactive policy*. In reactive policies, the choice of the next action is only dependent on the previous observation. Even with such restricted policies, reactive agents have been shown to demonstrate seemingly complex behaviour [Bra84].

Policies may be deterministic or stochastic. Deterministic policies map each observation to an action, while stochastic policies map each observation to a probability distribution over all possible actions.

4.5 Learning

The policy of the controller may be fixed over the lifetime of the agent, or it may adapt to the environment. It is important to note that, as far as the policy is concerned, the physical characteristics of its own body constitute part of the environment. The adaptation of a policy to the environment is analogous to the lifetime learning that exists in organisms.

Lifetime learning shares many similarities to evolution. They are both forms of biological adaptation. They are both concerned with optimization with respect to an evaluation function. In principle, every behaviour that can be generated through evolution and lifetime learning can be generated through evolution alone. Nolfi et al [NF00] introduce the concepts of "plastic general" and "full general" behaviour to describe these processes. A full general solution is a strategy which is suited to different environments, while a plastic general solution detects which it is in environment and modifies its strategy accordingly.

4.5.1 Costs and Benefits of Learning

A full general strategy is more effective as there is no adaptation process. An agent using a plastic general strategy must undergo a learning phase where its behaviour is necessarily suboptimal compared with a genetically specified behaviour. Moreover, not only is a learning agent delayed in its ability to acquire potential fitness, but its suboptimal behaviour might irrevocably damage the agent. Also, learning is inherently unreliable as it depends on the interaction that the agent has with the environment which is dependent on initial conditions.

However, a full general strategy is necessarily more complex than a plastic general strategy, and experimental results suggest that, even in simply environments, a full general strategy does not exist [NF00].

Plastic general strategies in the form of lifetime learning allows individuals to adapt to changes that are unpredictable at a generational level (i.e. changes that occur in the lifespan of an individual or across a few generations). For example, they allow individuals to adapt to peculiarities of their own body which are unpredictable at a generational level, or which are too complex to develop a full general solution for. From a biological perspective, examples of mechanisms which produce these unpredictable changes include:

- **Gestation.** Factors during gestation (such as parents' diet and disease) can affect the initial physical and neural capabilities of an organism.
- **Maturation.** The capabilities of an organism are not fixed over its lifetime. Their development is influenced by factors such as muscle use and diet.

- **Injury.** Organisms exist in dangerous environments and are prone to injury which alters their physical capabilities.

While biological evolution could, in theory, develop a full general solution to these changes, the immense range of possible changes precludes this possibility in reality. For example, consider the number of ways in which an organism can be injured. The complexity of a solution which compensated for each and every possibility would be immense. On the other hand, a plastic general solution would be expected to evolve as an adaptation because any small improvement in an organism's ability to compensate for injury would be naturally selected for.

Another advantage of lifetime learning is that it helps to guide the evolutionary process by using information extracted from the environment [Tur96]. This is known as the Baldwin Effect.

4.5.2 Learning Methods

Given that learning is an integral part of an adaptive agent operating in a changing environment, how can we go about implementing a learning behaviour? What mechanism can be used to adapt the policy based on the environment?

The traditional approach to behaviour would be to generate plans of future actions. However, such an approach requires the agent to build a general and accurate model of the world, which is intractable outside of toy domains. Another possible approach is to use supervised learning, where agents are told the optimal action that it could have taken at each interaction step. However, supervised learning is insufficient for autonomous adaptive agents which must be able to learn from their own experiences.

A good approach for when we don't know have a model of the world is *reinforcement learning*. In reinforcement learning problems, the value of each state transition is communicated to the agent through a scalar reinforcement signal, $r_t = R(s_t, a_t)$. The goal of the agent is to choose actions which tend to increase the long-run sum of values of the reinforcement signal. It learns to do this over time by systematic trial and error, guided by a learning algorithm. The goal of reinforcement learning is to obtain an optimal policy when the transition and reward functions are not known (see Figure 4.5).

The fundamental difficulty in reinforcement learning is the credit assignment problem. This problem consists of two aspects. Firstly, the reward signal may be delayed so an action may not affect not only the immediate reward, but also the next state, and thus all subsequent rewards. Thus, a reinforcement learning agent must be able to deduce which actions in the past contributed to the reward (the *temporal assignment* problem). The other aspect is deducing which parts of the actions contributed to the reward (the *structural assignment* problem).

4.5.3 Value Function Approaches

The traditional approach to solving reinforcement learning problems is to attempt to learn a value function. The value function makes estimates which predict the expected reward given the current state description and the current decision policy. This approach relies on dynamic programming techniques. The algorithms such as temporal difference learning and Q-learning [KLM96] incrementally improve estimates of the value of a state or of actions in states (so-called Q values). These dynamic programming approaches typically use a sample backup rule of the following form which uses observations of prediction differences from consecutive states to measure the prediction error and update the value function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

If these updates are performed using table-lookup to represent the value function exactly then there is provable asymptotic converge of the Q values to optimality and thus globally optimal behaviour [Tsi94]. When the Q values are nearly converged to their optimal values, it is appropriate for the agent to act greedily, taking, in each situation, the action with the highest Q value. During learning, however, there is a difficult exploitation versus exploration trade-off to be made. There are no good, formally justified approaches to this problem in the general case [KLM96].

4.5.4 Problems with Value Function Approaches

While table-lookup is feasible for small domains, applying it to large discrete or continuous state-action spaces (such as those found in physical simulation) is intractable. Some form of generalisation over these spaces is required. Retaining the dynamic programming approach requires one of two compromises: either variables must be discretized, or the value function must be approximated.

- **Discretization.** Discretizing variables has two problems related to the size of the discretization. If the discretization is too coarse then hidden state is introduced which may make it impossible to find the optimal behaviour. Conversely, if the discretization is too fine then we lose the ability to generalise over the state-action space. Also, the amount of training data need is increased. In practice, discretization is intractable due to the curse of dimensionality (the number of discrete states is exponential in the number of dimensions).

- **Function Approximation.** The other option is to replace the lookup table with a parameterized system to approximate the value function. An immediate difficulty is that it becomes hard to compute $max_a Q(s,a)$ for continuous actions. However a more catastrophic problem is that this method foils any guarantee of convergence to a globally optimal policy and, even worse, has been shown to diverge from a stable policy even on simple MDPs [BM95]. This divergence is due to the fact that any supervised learning algorithm will suffer from prediction error when trained on real data set. Using an approximator for value function results in the small prediction errors quickly accumulating and rendering the approximation useless. From the sample backup equation, we see that the new value for $Q(s,a)$ is based on the current approximate value for $Q(s,a)$ and values for actions from state s'. Any error in these predictions will be incorporated into the update for the new value. Likewise, the new value for $Q(s,a)$ is then used to update the approximation for other state-action pairs. Thus, the error propagates across the values, snowballing until it dominates the approximation.

4.6 Policy Search

An alternative approach to computing the value function is to explicitly represent the agent's policy and then search over the space of policies. An intuitive justification for this approach is hypothesized in [MSG99]. The argument is made that the problem that value function methods seem to be trying to solve is a more difficult problem than the one posed by reinforcement learning. The goal of reinforcement learners is to choose the best action at each step. However, value function approaches are attempting to predict future payoffs. In a sense, they not only tell you *what* to do, but *why*.

We can roughly compare the size of the hypothesis search spaces to demonstrate this difference. If we use direct policy representation then the size of the search space is $|S \rightarrow A| = |S \times A| = |S|^{|A|}$ while the size of the hypothesis space for a value function representation is $|(S \rightarrow A) \rightarrow V| = |V|^{|S||A|}$. If $|V|$ is unbounded then this hypothesis space is infinite and, even if it is bounded, then the size of the value function search space is at least several orders of magnitude larger than direct policy space.

4.6.1 Policy Gradient Search

The primary technique for searching over policy space is gradient-based search. In policy gradient search algorithms, the agent's policy is given by a function approximator defined by a set of modifiable parameters. The algorithms use estimates of the gradient of the reward function with respect to the parameterization of the agent's policy to incrementally improve the policy. The agent's policy is represented as a stochastic matrix, $\pi(s, a; \theta)$, which gives the probability that the agent chooses action a in state x given the parameters of the function approximator, θ. A reward function $\rho(\theta)$ is defined and learning is done by estimating the performance gradient $\partial\rho/\partial\theta$, which is used to modify the policy in a direction that increases the reward. Thus, the agent learns by estimating this gradient $\widehat{\partial\rho}/\partial\theta$ for the current policy π, and then updating the policy parameters θ as follows:

$$\theta_{t+1} = \theta_t + \alpha \frac{\widehat{\partial\rho}}{\partial\theta}$$

where α is a small positive step size. In practice, $\widehat{\partial\rho}/\partial\theta$ is calculated from estimates of the state action value function $Q^\pi(s, a)$, which is the value of executing action a in state s, under the current policy π.

A stochastic policy is used because, in general, greedy policies are non-differentiable [BB99] and stochastic policies perform better than deterministic ones in POMDPs due to the problem of perceptual aliasing.

Advantages of Policy Gradient Search

Three advantages of the policy gradient search were proposed in [GU00]:

- **Generalization.** Reinforcement learning problems with large continuous state spaces require some form of generalization. Policy gradient search uses a function approximation representation of the agent's policy which directly addresses this need.

- **Cost.** Traditional reinforcement learning algorithms have exponential growth in computational cost [KLM96]. However, computing an estimate of the policy gradient is linear in the number of parameters describing the policy. This makes it attractive for high-dimensional problems.

- **Convergence.** Finally, policy gradient algorithms are provably convergence to locally optimal policies.

These aspects of policy gradient search make it suitable for use in physically based environments with continuous state-action spaces.

Policy Gradient Search Algorithms

The first use of policy gradient search for reinforcement learning was the REINFORCE algorithm [Wil87, Wil92]. William's algorithm relied on the task having an identified recurrent state and the algorithm returned a gradient estimate each time this recurrent state was entered into. This algorithm has since been generalized to the infinte-horizon setting and to more general reward structures.

Policy-gradient algorithms for which convergence results have been proven all rely on the existence of recurrent states [BB00]. The main problem with policy gradient search is that it is prone to high variance in its estimates of the gradient reward. This variance is related to the time between visits to the recurrent state. The "VAPS" algorithm described in [Bai98] is one attempt to combine the advantages of the value-function and policy-gradient approaches in order to reduce this variance. These "Actor-Critic" approaches [KT00] attempt to learn the value function and policy simultaneously.

Other examples of policy gradient search include [PKMK00, MKKC99].

4.7 Direct Reinforcement Learning

The Direct Reinforcement Learning algorithm (DRL) [BB99, BWB99] is an example of a policy search algorithm for POMDPs. DRL guarantees convergence of policies to a local optimum in POMDPs under the assumptions of stationary distribution of the state-transition function, and differentiability of the function approximator.

The DRL algorithm records an *eligibility* matrix which maintains a decaying trace of the gradient of the log probabilities of past actions. The eligibility of an action given an input represents how likely it was that that action was taken given the input. The eligibilities decay away at each time step. The parameters of the approximator are updated whenever reinforcement is received in direct proportion to the eligibility of the parameter and the learning rate.

The primary advantage of DRL over other forms of policy gradient search is that its running time is not bounded by the requirement to visit recurrent states. Instead, it is bounded by the "mixing" time of the POMDP which is loosely defined as the time needed to approach stationarity. This mixing time is always shorter than recurrence time and often substantially so [BB00].

Policy gradient algorithms can learn in an offline or online fashion depending on when the policy is updated based on the estimated gradient. In offline learning, the agent makes successive gradient estimates under a fixed policy. In online learning, the agent updates its policy while making the gradient estimate. Thus, online learning is more applicable to provide lifetime learning for autonomous agents. In DROLE, the online version of DRL is used (see Figure 4.6) to enable agents to modify the weights in their policies which are represented as feedforward neural networks (see Figure 4.7).

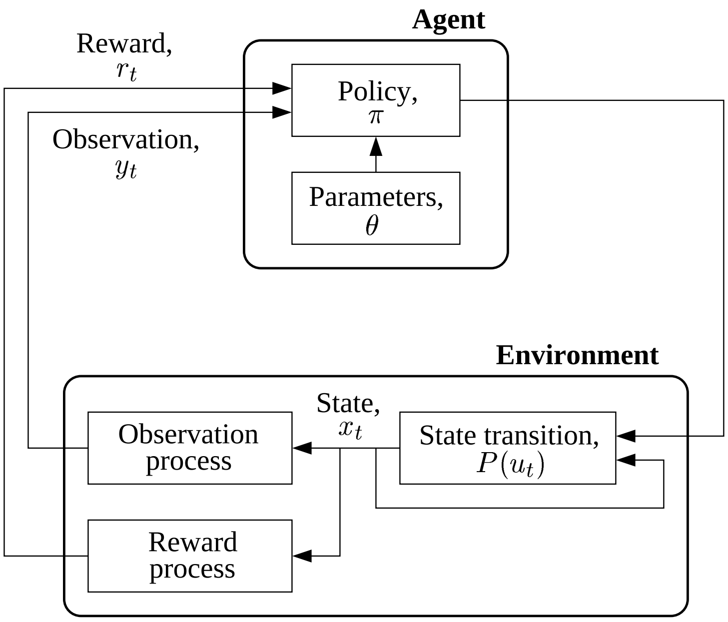

Figure 4.5: **A Partially Observable Markov Decision Process.** The POMDP model provides a formal definition of the problem faced by agents that learn by reinforcement in partially observable environments.

Given:
- $\beta \in [0, 1)$.
- $T > 0$.
- Initial parameter values $\theta_0 \in \mathbb{R}^K$.
- Randomized parameterized policies $\{\mu(\theta, \cdot) : \theta \in \mathbb{R}^K\}$ satisfying Assumptions 1 and 2.
- POMDP with rewards satisfying Assumption 3, and which when controlled by $\mu(\theta, \cdot)$ generates stochastic matrices $P(\theta)$ satisfying Assumption 4.
- Step sizes $\gamma_t, t = 0, 1, \ldots$ satisfying $\sum \gamma_t = \infty$ and $\sum \gamma_t^2 < \infty$.
- Arbitrary (unknown) starting state i_0.

Set $z_0 = 0$ ($z_0 \in \mathbb{R}^K$).
for $t = 0$ to $T - 1$ **do**
  Observe y_t (generated according to $v(i_t)$).
  Generate control u_t according to $\mu(\theta, y_t)$
  Observe $r(i_{t+1})$ (where the next state i_{t+1} is generated according to $p_{i_t i_{t+1}}(u_t)$.
  Set $z_{t+1} = \beta z_t + \frac{\nabla \mu_{u_t}(\theta, y_t)}{\mu_{u_t}(\theta, y_t)}$
  Set $\theta_{t+1} = \theta_t + \gamma_t r(i_{t+1}) z_{t+1}$
end for
return θ_T

Figure 4.6: OLPOMDP **Algorithm** [BWB99] The on-line version of the DRL algorithm specifies a function which, given a learning rate, a number of time steps, and an initial parameter vector returns a parameter vector which maximises the expected reward.

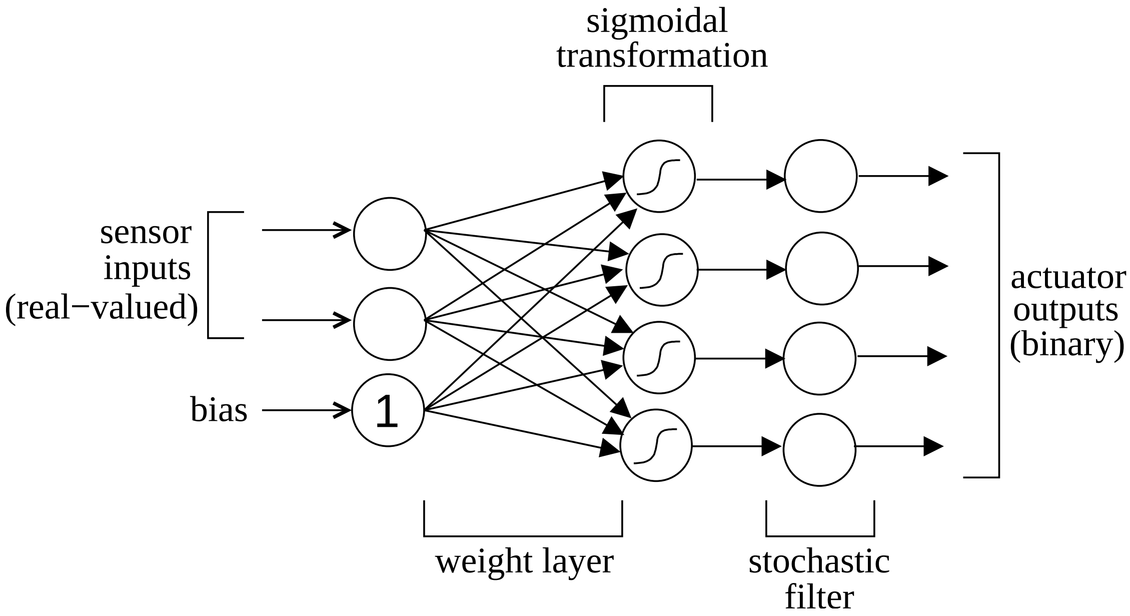

Figure 4.7: **The DRL agent's stochastic neural network controller.** Sensor inputs (normalised to lie in the range $[-1, 1]$) are fed through a neural network whose internal neurons sum the signals from their input neurons and then emit a signal based on a sigmoidal transformation function ($\frac{1}{1+e^{-\phi}}$). This signal is the probability of firing the corresponding actuator.

Chapter 5

Examples of Adapting Agents

Several task environments were created to demonstrate the application of reinforcement learning in DROLE. Further ongoing experiments can be viewed at the DROLE web site[1].

5.1 Task 1: Foraging.

In the foraging task, the environment is populated with a number of spherical food sources of a specified radius. Agents eat the food by making contact with them which causes the food to be transported to a new location in the environment. The radius of the food is then reduced to zero, and it gradually regrows over time. Agents are given a reward of 1 for eating a food source.

For this task, the agent's morphology simply consisted of a single block with no joints. The agent floats through the environment, pushed by 5 actuators which apply a thrusting force as well as applying forces off-centre to generate pitch and yaw turning behaviour.

The agent has 9 vision sensors with range 30 which look forward from the front of the agent. The segment contains a contact sensor to determine when it has touch a food source. The controller weights are initialised to random values sampled from a Gaussian distribution with a standard deviation of 0.8 and mean 0.

Twenty food sources with a radius of 5 and a regrowth rate of 1% per simulation step (and an upper limit of providing 1 reward per contact) are randomly distributed throughout the environment. In order to avoid agents being stuck in the extremes of the environment, the agent is wrapped around upon contact with the boundary.

The simulation was run for 100,000 seconds of simulation time with a step size 0.2. Figure 5.1 shows the results of using DRL with a range of learning rates of to control the agent with no evolution of parameters. For each learning rate 10 different agents were created and simulated for 3 runs. Thus, 105 million steps ($= 100,000/0.2 \times 7 \times 10 \times 3$) were simulated. The average reward of the agents was recorded every 100 simulation seconds.

As expected, the agents with no learning showed static average reward over their lifetime. These agents moved unguided throughout the environment, randomly contacting the food. These random static policies had the worst average reward by the end of the simulation. However, it is interesting to note these random policies initially perform better than learning agents with too high a learning rate. This is due to these agents, in a sense, "overlearning" reinforced actions. In contrast, agents with $\beta \leq 0.5$ performed well essentially right from the start. This dependence of the performance of DRL on the learning rate is a known problem [BB00].

[1] http://www.mcs.vuw.ac.nz/ nobbis/drole-index

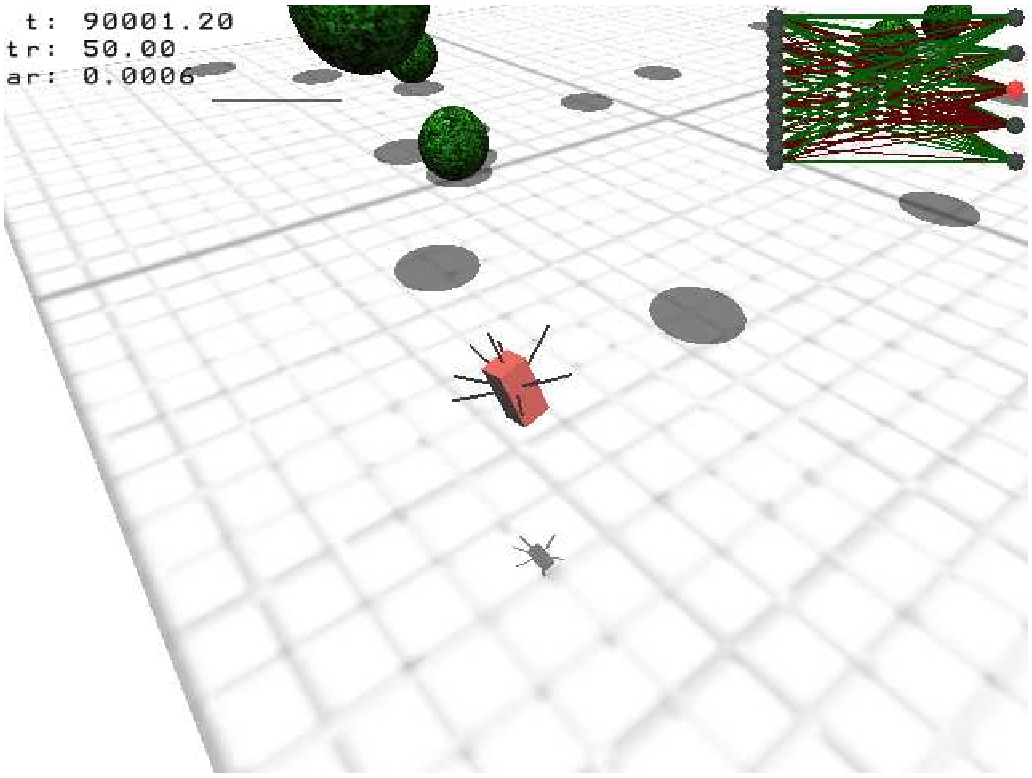

Figure 5.1: **Learning to forage.** The lines from the body represent the direction of the agent's vision sensors.

The DRL agents showed monotonic increase in average reward over the course of the simulation, thus displaying converging toward a local optimum.

5.2 Task 2: Crawling.

In this task, the goal for the agent is to move in a certain direction. The agent is rewarded for moving in that direction, and penalized for moving in the opposite direction. The agent's morphology consisted of a main body with a single arm connected to the top of it. Absolute and rate of change angle sensors are situated on each joint, and actuators apply torques to both joints.

A crawling behaviour was easily and quickly learnt given appropriate physical parameters. Figure 5.2 shows screenshots of a movie[2] showing the crawling behaviour. While this behaviour was learnt given the right parameters, the physical parameters affected the agent's behaviour significantly:

- **Friction.** If the friction of the body is too large, then the arm is unable to generate enough force to move the body. However, if the friction is too small, then the temporal credit assignment problem is prevalent as the agent is being rewarded for actions which did not contribute to the movement.
- **Mass.** If the mass of the body is too large, then the arm will be unable to generate enough force to move the body. However, if the body is too light, then the arm would generate too much force and end up toppling the body over.

[2] *http://www.mcs.vuw.ac.nz/ nobbis/media/crawl1.mpeg*

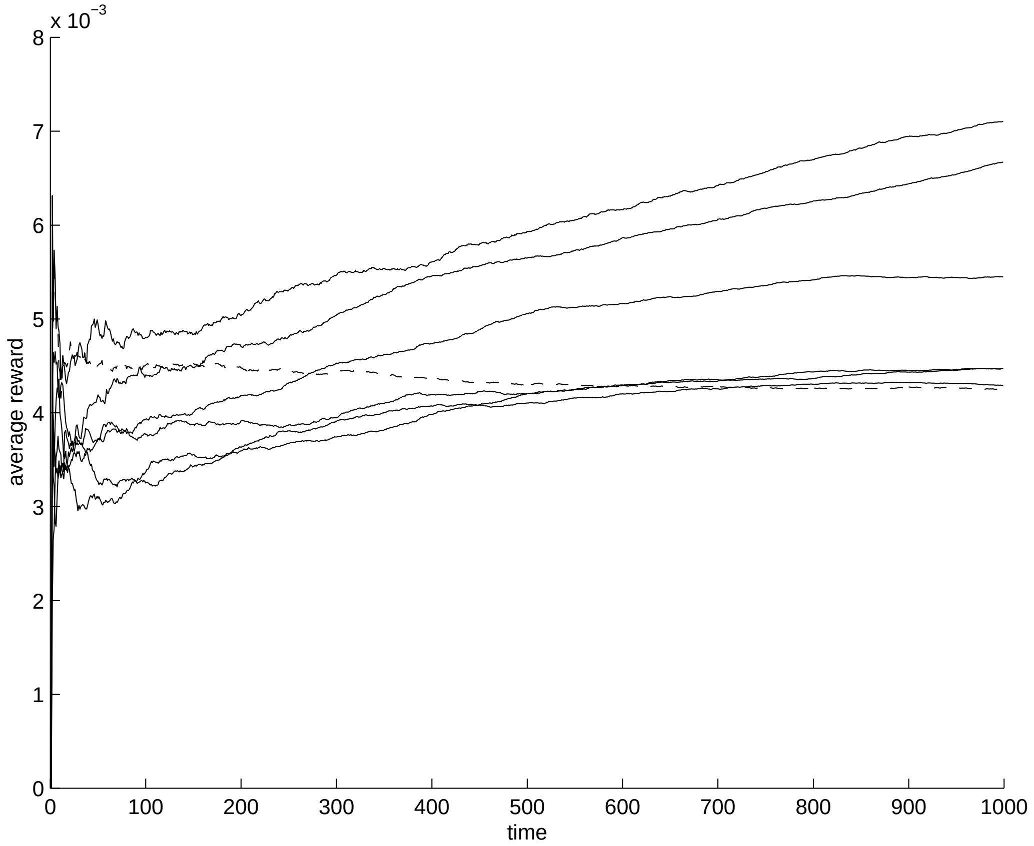

Figure 5.2: **Results for the foraging task.** The solid lines from top to bottom (at step 1000) correspond to agents with learning rates of 0.01, 0.02, 0.05, 0.1, 0.2, 0.5 respectively. The dotted line shows the results for the unlearned static policies. Each line represents results averaged over 30 runs.

- **Lower arm length.** A longer lower arm increases the path length of the arc needed to move the body, resulting in more work needing to be done to drag the arm back. However, a shorter arm produces a less efficient dragging movement and more of a shuffling action.

- **Lower arm mass.** A heavier lower arm means that it is harder to lift and drag back, while reducing its mass means that the force needed to drag the body is so great that, if there's no impedance to its movement, then it can flip the agent over.

- **Upper arm mass.** The lower arm has to work against the weight of the upper arm when dragging back. A large mass in the upper arm makes it harder to push up.

Theoretical guarantees of convergence for policy gradient algorithms do not exist for problems in which the reward structure changes. However, I experimented with altering the reward structure of this task over time by inverting the rewards once the agent reached a goal-line meaning that the agent is instead rewarded for moving backwards and penalized for moving forwards. An agent (with a learning rate of 0.2) was able to detect this change in the environment quickly. The behaviour of the agent changed abruptly from a circular movement, to a more random-looking action as the weights of its controller began to adjust. After a while, the agent learnt to move backwards. Interestingly, the agent was unable to relearn the forwards motion after inverting the reward direction once more.

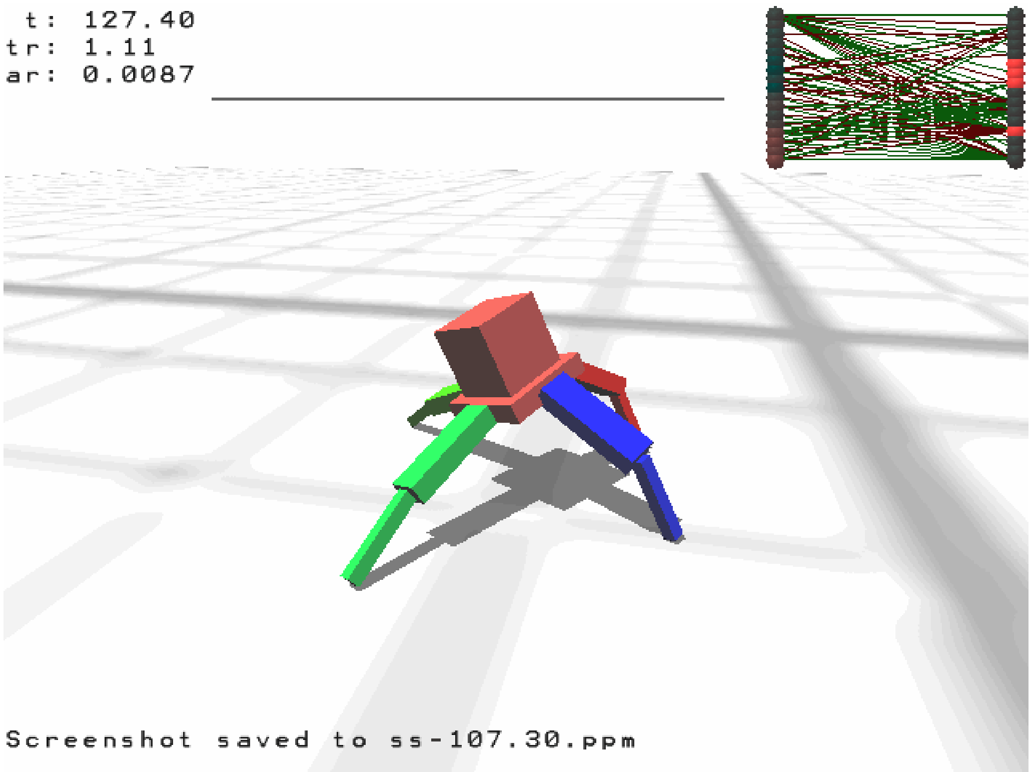

Figure 5.3: **Learning to stand.**

5.3 Task 3: Standing.

Finally, an interesting direction would be to get a multi-legged agent to crawl. As a first stage, the agent would need to be able to hold its body above ground. While, multi-legged walking wasn't attempted, an agent was easily able to learn to hold its body (with an additional weight on top) above the ground (given positive reinforcement for doing so).

These examples demonstrate the flexibility of the DROLE system and the ease with which simulated agents can be engineered to learn simple tasks given an appropriate morphology. Obviously, there are many more possible directions for adapting behaviour with agents in this simulated environment.

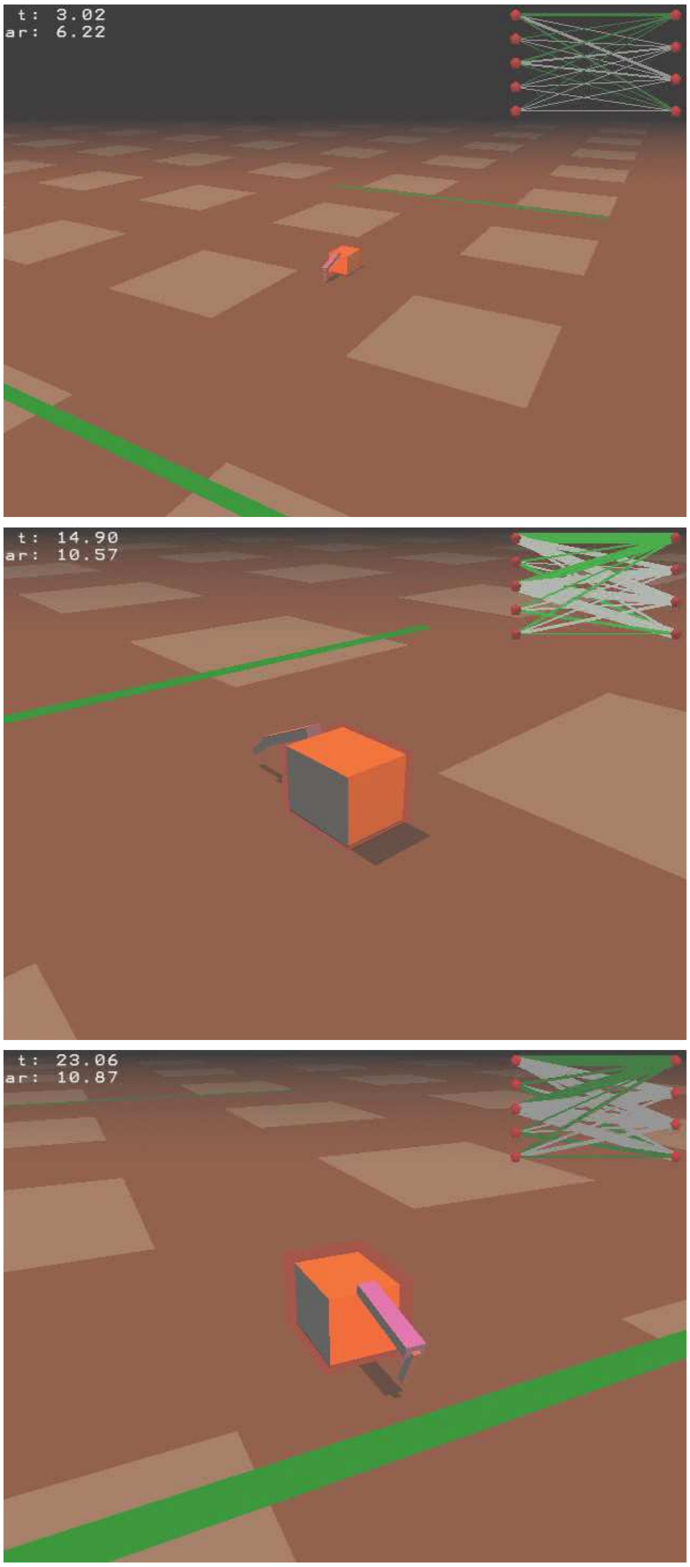

Figure 5.4: **Learning to crawl.**

Chapter 6

Summary

The work described in this report has evolved significantly from its initial proposal, which was to compare and contrast two approaches to reinforcement learning. As a result of background reading, I became interested in applying reinforcement learning to autonomous agents inhabiting a simulated environment that incorporates some realistic physics. As described in chapters 2 and 3, simulating agents in such a world is non-trivial. The early experiments in this domain made it clear that reinforcement learning could be successful only if other characteristics of the agent were suitable, and this led to an interest in the interplay between learning and evolution.

Accordingly, an extended system was developed, as described in chapter 4. DROLE has been constructed to allow the exploration of both learning and evolution of simulated creatures. It consists of over 20,000 lines of code, and features:

1. physically realistic dynamics
2. distributed evolution
3. an XML agent specification language

Chapter 5 described examples of simple behaviours generated by this system. This study has shown that, in concert with an evolutionary "outer loop", policy-gradient search in the form of direct reinforcement learning can demonstrate adaptive behaviour in physically realistic environments.

The above results are promising but tentative - they are perhaps best viewed as "proof of concept" demonstrations. They show that the approach is flexible and usable, and that it fosters the development of sensible behaviours by embodied autonomous agents without substantial "hard-wiring" or other human intervention. This work suggests many avenues for future research in adaptive behaviour. Some of the more interesting directions include:

- **More complex task environments.** Adaptation in more complex task environments could be investigated. Examples include environments with barriers or sloping and uneven terrain.

- **Varying environments.** Further research into environments in which the reward structure changes over the course of the task could be investigated to compare and contrast the performance of reinforcement learning and evolution in such environments.

- **Varying agents.** The application of learning to an agent's changing capabilities over its lifetime (as discussed in Chapter 4) could be investigated. Examples include continuously or discretely changing the morphology of an agent over its lifetime, or randomly removing or perturbing weights in its controller.

- **Imperfect sensors and actuators.** One consequence of physical embodiment is that sensory information and actuation is not perfect. Sensors in the real world have limited accuracy and are error-prone and motors can slip, muscles can cramp and joints can break. Modelling these effects could be used to investigate the usefulness of learning in the face of degrading sensory information and actuation.
- **More complex controllers.** Neural networks with more complicated architectures (such as hidden layers or recurrent connections) could be investigated as well as the use of other types of controller (such as function generators as used in [Sim94b]).
- **Biologically plausible actuators.** Modelling the actions of muscles in animals which provide continuous increase in the application of force based on effort required could reduce the 'brittleness' of agents' actions.
- **Evolving morphological structure.** Previous work has evolved the physical structure of agents [Sim94b, Sim94a, Ven96, KU99]. It would be interesting to see what effect lifetime learning had on evolution of this structure, e.g. whether learning helps guide evolution and whether evolution provides learning with the initial knowledge it needs.
- **Reinforcement learning of morphology and materials.** Another direction would be to apply online reinforcement learning to the physical parameters. While such a direction is less biologically plausible than learning the control, it may bring insight into how damaged robots could repair themselves. For example, if a bipedal robot damages one of its legs and is less able to do work with it, then the robot could adapt its gait by reducing the leg's hinge ranges in its knee and hip to offload more work to the other leg.
- **Extending the simulation.** Particle-spring systems could be used to simulate deformable morphologies. Other forms of locomotion could be investigated such as flying and swimming (by adding fluid dynamics) and wheeled agents (by adding wheel-plane constraints).
- **Making the transition to the real world.** Transferring evolved simulated robots to the real world is a promising direction in engineering autonomous robots [NF00]. Robots could take advantage of lifetime learning to cope with simulation inaccuracies.
- **More complex morphologies.** A greater variety of geometries, joint types (e.g. ball-socket, sliding) and sites for connecting segments could be provided.
- **Improving simulation accuracy and efficiency.** The simulation engine could be improved in several ways: adaptive step-sizes based on the simulation error rate could be used to improve both the efficiency and accuracy of the simulation and further efficiency gains could be had by implementing a more sophisticated integration loop algorithm.
- **Multi-agent learning.** Task environments involving multiple agents with co-operative (as in [PKMK00]) or competitive (as in [Sim94a]) goals could be investigated.

Acknowledgements

I wish to thank Dr. Marcus Frean for his continual support and inspiration throughout the project. May he never become a dentist.

Appendix A

Software systems

The goal of the development of DROLE was to produce an extensible framework for the analysis of evolving learning agent performance in physically-based environments. One of the goals of the framework was to allow the simulation to be distributed over many machines. To this end, the implementation is divided into three applications:

- The **Evolution Engine** generates, selects, reproduces, recombines and mutates the populations of agents.
- The **Simulation Engine** implements the test phase of the genetic algorithm.
- The **Population Manager** persistently stores the parameters of populations of agents and the statistics generated by their simulation, as well as allowing the accessing and modification of populations and individuals to provides the link between the above two application.

A.1 Evolution Engine

The `Evolver` object uses subclasses of `Selector` to choose individuals to add and remove from the population. A `Genome` object extracts `Gene`'s from `Entity`'s (which represents any part of the agent).

The Evolution Engine is implemented in Java (over 5,000 lines of code) and communicates with the population manager via HTTP/CGI.

A.2 Simulation Engine

The Simulation Engine is an interactive application which allows the user to move through a virtual environment while observing and manipulating multiple objects. These objects are subjected to the laws of physics.

The Engine is implemented in C++ (over 15,000 lines of code) and communicates with the Population Manager with Python scripting (200 lines) via HTTP/CGI. A Perl script is used to batch process the generation of MPEG movies.

A.2.1 Environment

The Simulator consists of an instance of `Environment` which is responsible for managing the physical objects which can be added, removed and modified at run-time to the system. The `Environment` maintains a number of `ObjectListener` subsytems which are notified whenever an `Object` is created, modified (via dynamical simulation or user interaction), or deleted.

The `ObjectListener`'s implemented include `CollisionDetector` (which is responsible for computing object intersection, contacts, and distances), `Physics` (which is responsible for integrating the physical state of the objects forward by one timestep while obeying constraints), and `UI` (which is responsible for displaying the environment to the user and processing any interaction).

`Experiment` subclasses `Environment` to provide utility functions useful for performing experiments (such as time limits and saving statistics to file).

A.2.2 Object

The `Object` class represents a physical object which is influenced by the laws of physics. `Object`'s are connected to each other via `Joint`'s, forming a tree hierarchy. The physical state of an `Object` includes its position, orientation, velocity and angular velocity. Its geometry is stored in a `Polyhedron` object which stores vertices and faces which connect them as well as face normals, edge neighbour information, and bounding sphere for use in contact calculation. A few geometries are inbuilt: `Box`, `Octahedron`, and `Sphere` (constructed by recursive subdivision of the `Octahedron`) while others can be imported from a number of file formats (DXF, TRI, POLY, VRT).

Other physical properties include the `Object`'s density (assumed to be uniform), mass, inertia tensor, elasticity (i.e. its coefficient of restitution), and a simple friction parameter.

The appearance of the `Object` is influenced by its `Material` which includes its ambient, diffuse, specular and emissive colors as well as its level of shininess. Several predefined `Material`'s are included.

A.2.3 Physics

The physics subsystem is responsible for computing the object trajectories subject to the constraints in the system. All of the objects are assumed to be rigid bodies and thus respect hard contact conditions. A number of physics libraries were used to offload the physical simulation `DynamoPhysics`, `MultiBodyPhysics` and `ODEPhysics`. The `VolumeIntegrator` class computes an object's density and geometry via integration over its volume.

A.2.4 CollisionDetector

The `CollisionDetector` subsystem is responsible for computing contact between multiple objects in the environment. This information is fed to the main simulation loop in the form of `Contact` objects in order to resolve collisions, compute contact forces (provided by the `ContactForceCalculator`) and to control the forward integration of state. The subsystem integrates with the SWIFT++ collision detection library to perform actual detection.

A.2.5 UI

Multiple user interface subsystems are supported. Currently, only 3 are implemented:

- `GLUI` Uses the low-level OpenGL/GLUT 3D graphics libraries.
- `MaverikUI` Uses the high-level Maverik3D VR microkernel developed at Manchester University.
- `TextUI` A simple textual non-interactive interface.

The `Material` and `Texture` classes the appearance of an object when a `Light` hits it. `Texture` loads imports BMP or TGA images from disk. The `Font` class is used to overlay text on the display. Screenshots can be taken at any time in the simulation and saved as GIF files, and MPEG2 movies can be recorded (movies are generated by saving successive image frames to disk in the PPM format and feeding them into the `mpeg2encode` utility).

A.2.6 Agent

An `Agent` is a non-deterministic `Object`: it has the ability to influence its environment as well as reacting to it. It achieves this through its `Controller`, which receives input from a number of `Sensors` and computes a response to send to its `Actuators` which influence the environment through the application of forces and torques. A number of actuators are defined: `Flipper`, `Flinch`, `Mouth`, `Muscle`. A number of sensors are defined: `AngleSensor`, `AngleChangeSensor`, `ConstantSensor`, `EyeSensor`, `TouchSensor`. The only controller implemented was the `DRLController`. The `DRLController` requires a `FunctionApproximator` to represent its current policy. A feedforward `NeuralNet` consisting of multiple `NeuralNetLayers` is defined.

An `Agent` is evaluated on its ability to perform a `Task` which computes a reinforcement value based on any state of the environment, and decides when a goal is reached. Tasks defined included `PositionTask` (which rewards an agent based on its distance from a goal) and `ForageTask` (which rewards agents for moving to target areas).

A.2.7 Utility

A number of utility classes are implemented to support the framework. These include a `Range` class, general-purpose multi-dimensional `Vector` and `Matrix` classes, as well as full vector mathematics library including 3-dimensional points and vectors (`Point3`, `Vector3`), and a 3x3 matrix and 4-tuple axis-angles and quaternion classes (`Matrix3`, `AxisAngle4` and `Quat4`) to represent orientations. Other geometric primitives include `Edge`, `Plane`, and `Polygon`.

A.3 Population Manager

The population manager provides persistent storage and allows for communication between the simulation and the evolution engines. Multiple populations of agents are stored as XML documents on a web server. The population manager has the following functionality:

- add/remove experiment groups
- add/remove experiments
- submit experiment results
- get a particular experiment/experiment group or all experiment groups

The population manager is written in Python and is around 1,500 lines of code.

Appendix B

DROLE XML

B.1 Example Environment

The following is an example of an XML document used to specify an environment for the DROLE engine to simulate. It contains a single agent with four legs connected to its torso by hinges which have actuators and proprioceptive sensors. The torso contains a single contact sensor and four vision sensors pointing in each compass direction. The task of this agent is to find randomly placed shrubs. Its controller uses a feed-forward neural network which learns via the OLPOMDP algorithm. Several of the network's initial weights are specified.

```
<?xml version="1.0" standalone="yes"?>
<environment>
  <property name="world.gravity" value="(0, -9.81, 0)"/>

  <agent>
    <forage-task type="shrub" layout="random" shrubs="20" shrub-radius="3.0"
                 shrub-regrowth="0.02"/>

    &lt;segment name="Body" position="(0, 2, 0)" size="(2, 1, 2)" density="1"
             friction="10" elasticity="0">
      &lt;segment name="FrontLeg" size="(1, 0.3, 0.5)" density="1" friction="1"
               elasticity="0.2">
        <hinge-joint name="FrontHip" anchor-base="front" range="[0, 45]"
                     bounciness="0.1"/>
        <muscle name="Flex"   strength=" 10"/>
        <muscle name="Extend" strength="-10"/>
        <angle-sensor name="HipAngle"/>
        <angle-change-sensor name="HipAngleChange"/>
      &lt;/segment&gt;

      &lt;segment name="LeftLeg" size="(0.5, 0.3, 1)" density="1" friction="1"
              elasticity="0.2">
        <hinge-joint name="LeftHip" anchor-base="left" range="[0, 45]"
                     bounciness="0.1"/>
        <muscle name="Flex"   strength=" 10"/>
        <muscle name="Extend" strength="-10"/>
        <angle-sensor name="HipAngle"/>
        <angle-change-sensor name="HipAngleChange"/>
```

```
      &lt;/segment&gt;

      &lt;segment name="RightLeg" size="(0.5, 0.3, 1)" density="1" friction="1"
               elasticity="0.2"&gt;
        <hinge-joint name="RightHip" anchor-base="right" range="[-45, 0]"
                     bounciness="0.1"/>
        <muscle name="Flex"   strength=" 10"/>
        <muscle name="Extend" strength="-10"/>
        <angle-sensor name="HipAngle"/>
        <angle-change-sensor name="HipAngleChange"/>
      &lt;/segment&gt;

      &lt;segment name="BackLeg" size="(1, 0.3, 0.5)" density="1" friction="1"
               elasticity="0.2"&gt;
        <hinge-joint name="BackHip" anchor-base="back" range="[-45, 0]"
                     bounciness="0.1"/>
        <muscle name="Flex"   strength=" 10"/>
        <muscle name="Extend" strength="-10"/>
        <angle-sensor name="HipAngle"/>
        <angle-change-sensor name="HipAngleChange"/>
      &lt;/segment&gt;

      <touch-sensor type="shrub"/>
      <eye-sensor type="shrub" position="( 0.5, 0,    0)"
                  direction="(1, 0, 0)" fov="45" range="20.0"/>
      <eye-sensor type="shrub" position="(-0.5, 0,    0)"
                  direction="(-1,  0,  0)" fov="45" range="20.0"/>
      <eye-sensor type="shrub" position="( 0.5, 0,  0.5)"
                  direction="( 0,  0,  1)" fov="45" range="20.0"/>
      <eye-sensor type="shrub" position="( 0.5, 0, -0.5)"
                  direction="( 0,  0, -1)" fov="45" range="20.0"/>

      <drl-controller learning-rate="0.2" eligibility-decay="0.8"
                      clock="1"/>
        <link weight="0.05"/>
        <link weight="-0.05"/>
        <link weight="0.1"/>
        <link weight="0.2"/>
        <link weight="-0.1"/>
      </drl-controller>
    &lt;/segment&gt;
  </agent>
</environment>
```

B.2 DTD

The DTD for these XML documents is:

```
<?xml version="1.0"?>

<!ELEMENT environment (property*, agent*)>

<!ELEMENT property (EMPTY)>
<!ATTLIST property name  CDATA #REQUIRED>
<!ATTLIST property value CDATA #REQUIRED>

<!ELEMENT agent (segment+, position-task?. forage-task?)>

<!ELEMENT segment (segment*, fixed-joint?, hinge-joint?, muscle*,
                  flipper*, angle-sensor*, angle-change-sensor*,
                  eye-sensor*, drl-controller*)>
<!ATTLIST segment name       CDATA #IMPLIED>
<!ATTLIST segment position   CDATA #IMPLIED>
<!ATTLIST segment size       CDATA #REQUIRED>
<!ATTLIST segment density    CDATA #REQUIRED>
<!ATTLIST segment friction   CDATA #REQUIRED>
<!ATTLIST segment elasticity CDATA #REQUIRED>
<!ATTLIST segment damping    CDATA #REQUIRED>

<!ELEMENT position-task (EMPTY)>
<!ATTLIST segment goal  CDATA #IMPLIED>

<!ELEMENT forage-task            (EMPTY)>
<!ATTLIST segment type           CDATA #IMPLIED>
<!ATTLIST segment layout         CDATA #IMPLIED>
<!ATTLIST segment shrub-radius   CDATA #REQUIRED>
<!ATTLIST segment shrub-regrowth CDATA #REQUIRED>

<!ELEMENT fixed-joint (EMPTY)>
<!ATTLIST fixed-joint name        CDATA #IMPLIED>
<!ATTLIST fixed-joint anchor-base CDATA #REQUIRED>

<!ELEMENT hinge-joint (EMPTY)>
<!ATTLIST hinge-joint name        CDATA #IMPLIED>
<!ATTLIST hinge-joint anchor-base CDATA #REQUIRED>
<!ATTLIST hinge-joint range       CDATA #IMPLIED>
<!ATTLIST hinge-joint bounciness  CDATA #IMPLIED>

<!ELEMENT muscle (EMPTY)>
<!ATTLIST muscle strength CDATA #REQUIRED>

<!ELEMENT flipper (EMPTY)>
<!ATTLIST flipper anchor CDATA #REQUIRED>
<!ATTLIST flipper push   CDATA #REQUIRED>
```

```
<!ELEMENT angle-sensor (EMPTY)>

<!ELEMENT angle-change-sensor (EMPTY)>

<!ELEMENT eye-sensor (EMPTY)>
<!ATTLIST eye-sensor type      CDATA #IMPLIED>
<!ATTLIST eye-sensor position  CDATA #REQUIRED>
<!ATTLIST eye-sensor direction CDATA #REQUIRED>
<!ATTLIST eye-sensor fov       CDATA #REQUIRED>
<!ATTLIST eye-sensor range     CDATA #REQUIRED>

<!ELEMENT touch-sensor (EMPTY)>
<!ATTLIST touch-sensor type CDATE #REQUIRED>

<!ELEMENT drl-controller (link*)>
<!ATTLIST drl-controller learning-rate     CDATA #REQUIRED>
<!ATTLIST drl-controller eligibility-decay CDATA #REQUIRED>
<!ATTLIST drl-controller clock             CDATA #REQUIRED>

<!ELEMENT link (EMPTY)>
<!ATTLIST link weight CDATA #REQUIRED>
```

Bibliography

[Bai98] L. C. Baird. Gradient descent for general reinforcement learning. In *Neural Information Processing Systems, 11*, 1998.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics*, 28(Annual Conference Series):23–34, 1994.

[BB99] J. Baxter and P. L. Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. Technical report, Research School of Information Sciences and Engineering, Australian National University, Jul 1999.

[BB00] Jonathan Baxter and Peter L. Bartlett. Reinforcement learning in POMDP's via direct gradient ascent. In *Proc. 17th International Conf. on Machine Learning*, pages 41–48. Morgan Kaufmann, San Francisco, CA, 2000.

[Bee01] Randall D. Beer. The dynamics of active categorical perception in an evolved model agent. Technical report, Dept. of Electrical Engineering and Computer Science, Dept. of Biology, Case Western Reserve University, OH, 2001.

[BM95] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.

[BP00] Josh C. Bongard and Chandana Paul. Investigating morphological symmetry and locomotive efficiency using virtual embodied evolution. In *The Sixth International Conference on the Simulation of Adaptive Behaviour*, pages 420–429, Paris, France, Aug 2000.

[BP01] J. Bongard and C. Paul. Making evolution an offer it can't refuse: Morphology and the extradimensional bypass. In *Proceedings of the Sixth European Conference on Artificial Life (ECAL 2001)*, Prague, CZ., 2001.

[Bra84] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, Cambridge, Massachusetts, 1984.

[Bro91] Rodney A. Brooks. Intelligence without reason. In Ray Myopoulos, John; Reiter, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann.

[Bro92] R. Brooks. Artificial life and real robots. In *European Conference on Artificial Life*, pages 3–10, 1992.

[BWB99] J. Baxter, L. Weaver, and P. L. Bartlett. Direct gradient-based reinforcement learning: Ii. gradient ascent algorithms and experiments. Technical report, Research School of Information Sciences and Engineering, Australian National University, Jul 1999.

[Con90] M. Conrad. The geometry of evolution. In *Biosystems 24*, pages 61–81, 1990.

[EL01] Stephen A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using surface decomposition. In *Computer Graphics Forum (Proceedings of Eurographics)*, 2001.

[Fea84] Roy Featherstone. *Robot dynamics algorithms.* PhD thesis, University Of Edinburgh, 1984.

[GU00] Gregory Z. Grudic and Lyle H. Ungar. Localizing policy gradient estimates to action transitions. In *Proc. 17th International Conf. on Machine Learning*, pages 343–350. Morgan Kaufmann, San Francisco, CA, 2000.

[Har90] S. Harnad. The symbol grounding problem. In *Physica D, 42*, pages 335–346, 1990.

[Jak98] N. Jakobi. The minimal simulation approach to evolutionary robotics. In T. Gomi, editor, *Evolutionary Robotics - From Intelligent Robots to Artificial Life (ER'98)*, Ontario, Canada, 1998. AAI Books.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[KT00] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms, 2000.

[KU99] Maciej Komosinski and Szymon Ulatowski. Framsticks: Towards a simulation of a nature-like world, creatures and evolution. In *European Conference on Artificial Life*, pages 261–265, 1999.

[LE99] L. Lichtensteiger and P. Eggenberger. Evolving the morphology of a compound eye on a robot. In *Eurobot'99.*, 1999.

[MC95] Maja Mataric and Dave Cliff. Challenges in evolving controllers for physical robots. In *Robotics and Autonomous Systems, 1995. Special issue on Evolutional Robotics*, 1995.

[McG90] T. McGeer. Passive dynamic walking. *The International Journal of Robotics Research, 9(2)*, pages 62–82, Apr 1990.

[Mir98a] B. Mirtich. Multibody dynamic package, 1998.

[Mir98b] B. Mirtich. Rigid body contact: Collision detection to force computation. Technical Report TR-98-01, Mitsubishi Electrical Research Laboratory, 1998.

[Mir00] Brian Mirtich. Timewarp rigid body simulation. Technical Report TR-2000-17, Mitsubishi Electrical Research Laboratory, Apr 2000.

[MKKC99] Nicolas Meuleau, Kee-Eung Kim, Leslie Pack Kaelbling, and Anthony R. Cassandra. Solving POMDPs by searching the space of finite policies. In *UAI99*, 1999.

[MSG99] D. Moriarty, A. Schultz, and J. Grefenstette. Reinforcement learning through evolutionary computation. In *Journal on Artificial Intelligence Research (JAIR)*, 1999.

[NF00] S. Nolfi and D. Floreano. *Evolutionary Robotics: The Biology, Intelligence and Technology of Self-Organizing Machines.* MIT Press, 2000.

[PB01] Chandana Paul and Josh C. Bongard. The road less travelled: Morphology in the optimization of biped robot locomotion. In *IEEE International Conference on Intelligent Robots and Systems (IROS 2001)*, 2001.

[Pen94] Roger Penrose. *Shadows of the Mind.* Oxford University Press, New York, 1994.

[Pfe98] Rolf Pfeifer. Embodied system life. In *Proceedings of the 1998 International Symposium on System Life*, 1998.

[Pfe00] Rolf Pfeifer. On the role of morphology and materials in adaptive behaviour. In *SAB2000*, 2000.

[PKMK00] L. Peshkin, K. Kim, N. Meuleau, and L. Kaelbling. Learning to cooperate via policy search. In *Sixteenth International Conference on Uncertainty in Artificial Intelligence*, 2000.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[Sim94a] Karl Sims. Evolving 3d morphology and behavior by competition. *Artificial Life*, 1:353–372, 1994.

[Sim94b] Karl Sims. Evolving virtual creatures. *Computer Graphics*, 28(Annual Conference Series):15–22, 1994.

[TRG96] Demetri Terzopoulos, Tamer Rabie, and Radek Grzeszczuk. Perception and learning in artificial animals. In *Artificial Life V: Proc. Fifth International Conference on the Synthesis and Simulation of Living Systems*, pages 313–320, Nara, Japan, May 1996.

[Tsi94] J. N. Tsitsiklis. Asynchronous stochastic approximation and q-learning. In *Machine Learning*, volume 16, pages 185–202, 1994.

[Tur96] P. Turney. Myths and legends of the baldwin effect. In T. Fogarty and G. Venturini, editors, *Proceedings of the ICML-96 (13th International Conference on Machine Learning)*, 1996.

[Ven96] J. Ventrella. Explorations of morphology and locomotion behaviour in animated characters. In *Artificial Life IV*, pages 436–441. MIT Press, Cambridge, MA, 1996.

[Wil87] R. J. Williams. A class of gradient-estimating algorithms for reinforcement learning in neural networks. In M. Caudill and C. Butler, editors, *IEEE International Conference on Neural Networks*, volume 2, pages 601–608, San Diego, 1987. IEEE, New York.

[Wil92] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.