Evolving Virtual Creatures Revisited

Peter Krčah
Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25
118 00 Prague 1, Czech Republic
peter.krcah@matfyz.cz

ABSTRACT

Thirteen years have passed since Karl Sims published his work on evolving virtual creatures. Since then, several novel approaches to neural network evolution and genetic algorithms have been proposed. The aim of our work is to apply recent results in these areas to the virtual creatures proposed by Karl Sims, leading to creatures capable of solving more complex tasks. This paper presents our success in reaching the first milestone - a new and complete implementation of the original virtual creatures. All morphological and control properties of the original creatures were implemented. Laws of physics are simulated using ODE library. Distributed computation is used for CPU-intensive tasks, such as fitness evaluation. Experiments have shown that our system is capable of evolving both morphology and control of the creatures resulting in a variety of non-trivial swimming and walking strategies.

Categories and Subject Descriptors

I.2.9 [**ARTIFICIAL INTELLIGENCE**]: Robotics; I.2.8 [**ARTIFICIAL INTELLIGENCE**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation

Keywords

Artificial life, Computer aided/automated design, Evolutionary robotics, Evolving virtual creatures

1. INTRODUCTION

Many problems seem to require human creativity to be successfully solved. Finding solutions often involves searching large multidimensional spaces of all possible solutions, which, luckily, humans can often do intuitively. However, the process of invention can be challenging, time-consuming and tiring. Therefore, several methods for automatization of such processes have been proposed in the last few decades. One such approach is inspired by Darwinian theory of evolution. Several effective evolutionary search methods have been proposed (survey of methods can be found in [1, 4]) and have even led to automatic invention of several human-competitive results recently [5].

Artificial evolutionary search methods are based on a genetic algorithm, which in many ways imitates the process of evolution in the nature. Possible solutions of the problem are compactly coded into a *genotype*, analogous to genetic description of animals. Genotypes are then translated to *phenotypes* (analogous to animals' physical bodies), which compete for survival in virtual arenas. Fitness function is defined to evaluate each phenotype according to its success in solving the problem. The fittest organisms are than allowed to reproduce, with number of offspring proportional to their fitness value. Random mutations are introduced in newly created offspring to maintain genetic variability.

In our opinion, one of the most interesting applications of genetic algorithms is the evolution of two- or three-dimensional self-controlling virtual creatures with both morphology and control system discovered by the evolution. Research in this area was pioneered by Sims [10]. His three-dimensional creatures successfully evolved various locomotion, swimming, jumping and following strategies, often resembling behaviors found in the nature. Sims also simulated a competition of the creatures [9], successfully producing the "Red Queen effect".

Sims' work has inspired several other works on three-dimensional virtual creatures. Shim and Kim have evolved various flying creatures with different types of wings [8]. Ray has created a system for interactive evolution of a colorful three-dimensional creatures in physical environment [7]. Komosinski and Ulatowski have created the Framstick project for evolving virtual creatures composed of sticks [3]. Pollack and Lipson have evolved three-dimensional walking robots and also introduced a method of their automatic fabrication [6]. Pollack and Hornby have used L-systems for generative encoding of both morphology and control system of their creatures [2]. Ventrella has evolved two-dimensional Swimbots [12] to show how can sexual selection inhibit optimization of locomotion.

In summary, several works besides Sims' have successfully evolved interesting virtual creatures. However, the virtual creatures evolved by Sims still impress and inspire us the most. We believe, that many researchers would like to reproduce Karl Sims' experiments or even enhance the original

virtual creatures and continue the research in this particular area. However, original Sims' implementation is not available and no other suitable implementation had existed. Therefore, we decided to reimplement the creatures "from scratch" and provide our implementation freely to everyone.

This work introduces ERO (Evolution of Robotic Organisms) – a framework for arbitrary evolutionary experiments, with an implementation of Karl Sims' evolving virtual creatures. It also presents our approach to several problems encountered during ERO development and creature design. ERO can be used to evolve any kind of organism, while reusing all organism-independent components. We aimed to make the framework user-friendly and evolution transparent, so we included many tools, which allow better insight into inner workings of evolution (advanced fitness graph, tracing of organism "family tree", etc.) and organism design (an editor of organism genotype and phenotype graphs, an interactive simulation and visualization of creature's neural network, etc.). The computation of a genetic algorithm is speeded up by a built-in engine for distributed computation of CPU-intensive tasks (such as fitness evaluation). All parts of ERO framework are cross-platform (and successfully tested on both Linux and Windows operating systems). ERO framework has been successfully used to evolve several interesting (and often amusing) types of creature behavior for both swimming and walking.

The first part of this paper (section 2) describes our implementation of virtual creatures. The second part (sections 3, 4, 5 and 6) summarizes features of ERO, which are not related to any specific type of organism (such as distributed computation system and various tools for evolution debugging). Section 7 describes our experiments with virtual creatures and shows some examples of evolved behavior.

2. CREATURES

Creature design is inspired by the work of Karl Sims [10, 9]. Much effort was dedicated to making creature implementation general and extensible. Most organism components were designed to be fully replaceable. Examples of these "black box" components include the shape of a morphological node, a node controller, a creature brain, sensors and effectors. This design decision opens the possibility for adding, for example, entirely different types of node controller, or adding a new type of node shape (sphere, cylinder) easily.

In the following section, we describe creature morphology and the procedure of creating creature phenotypes from their genetic template. Section 2.2 presents a distributed control system of the creatures. Section 2.3 shortly describes mutation operator and mating of creatures. Section 2.4 summarizes fitness evaluation. In section 2.5, we introduce testing mechanism to remove faulty creatures quickly and in section 2.6 we summarize various tools for debugging creatures in ERO.

2.1 Creature morphology

Creature phenotype is represented by a hierarchy of morphological nodes. Each node corresponds to a single body part (i.e. a box in current implementation) and each connection between two nodes corresponds to a joint between two body parts.

Creature phenotype is created from a corresponding genotype template. Creature genotype is a directed graph with

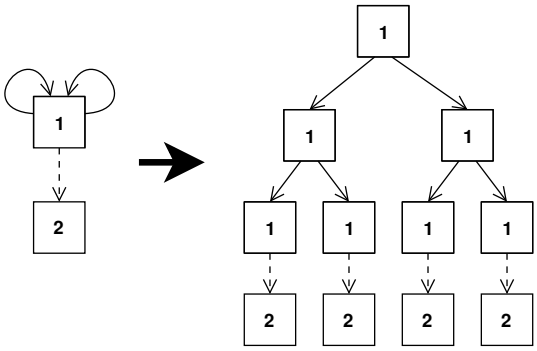


Figure 1: Creature genotype and phenotype (dashed-line represents terminal connection).

a single special node called a *root node*. Phenotypes are created from genotypes by first copying the genotype *root node* and then recursively traversing connections and adding encountered nodes and connections to the phenotype graph. To prevent infinite cycles, each genotype node has a *recursive limit*, which limits the number of passes through the given genotype node. Each genotype connection also has a *terminal flag*, which, if set, forces a connection to activate only if there are no other copies of the current genotype node in the phenotype subtree starting in the current phenotype node. The terminal flag can be used to create hand-like structures, various tail endings, etc. Transition from a genotype to its phenotype is illustrated in Figure 1.

Both genotype node and genotype connection have various other properties used for building their phenotype counterparts. A genotype node contains information about the *shape* of resulting morphological node (in the case of a box, its dimensions are specified) and a *joint-type*. A joint-type defines the physical properties of the joint connecting current node to its parent in the phenotype graph. Currently implemented joint types are: *fixed*, *hinge* and *universal*.

Genotype connection contains information about the position of a child node relative to its parent node. The position is represented by child and parent *anchor points*, relative *rotation* and a *scaling factor*. Each anchor point must be on the surface of the corresponding node. When creating a phenotype, child and parent nodes are first aligned, so that child and parent anchor points become identical and surface normals at anchor points become opposite. Afterwards, the child node is scaled by the scaling factor and rotated by the *rotation* parameter.

2.2 Control system

Creature's control system is distributed over its entire body. Each morphological node has its local sensors, effectors and a neural controller. Besides local controllers, a single global controller (i.e. the "brain") is also present to allow global coordination among organism parts. Global sensors and effectors are also possible, although not currently implemented. Local controllers in a child node can also contain connections to its parent node controller (both incoming and outgoing connections are allowed). This way, a neural signal can spread through the organism body in a similar fashion of that in real organisms. The example of a creature local control system is shown in Figure 2.

Controllers are designed to be easily replaceable by an

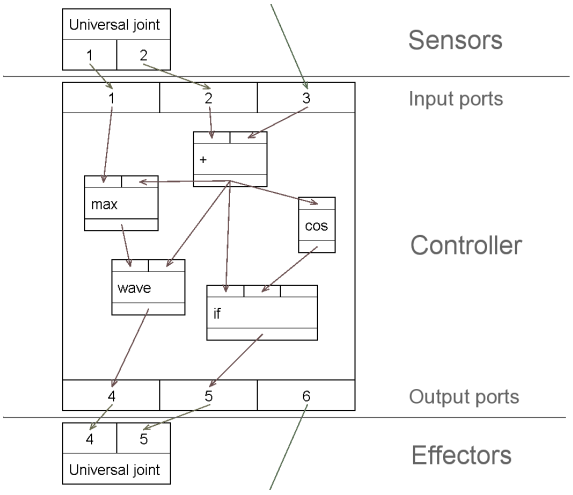


Figure 2: Example creature local control system.

entirely different type of controller. A controller can be any component having *input ports* (which can receive signals from local sensors, output ports of parent/child controllers or output ports of the brain) and *output ports* (which can send signals to local effectors, input ports of parent/child controllers and input ports of the brain).

Currently implemented controller type consists of neurons connected by directed weighted connections (as proposed by Karl Sims in [10]). There are no constraints put on the topology of the neural graph. Connections can, for example, form feedback loops, or divide network into separate subnetworks.

2.2.1 Sensors

Each sensor is contained in a specific morphological node. Sensors allow a creature to respond to changes in its environment. The value of each sensor is measured during each simulation step and propagated to the input port of the corresponding controller. Currently implemented sensor types are the following:

1. The *joint angle sensor* measures a single value for each degree of freedom of a joint. Each creature node contains a single joint sensor attached to a joint between a node and its parent node.

2. The *gyroscopic sensor* measures relative orientation of a morphologic node. Orientation is provided in the form of a vector pointing upwards, transformed to the reference frame of a body part containing the sensor.

3. The *touch sensor* is located on each face of each box and returns 1 if a contact occurred and -1 if there was no contact.

2.2.2 Neurons

Each neuron has a transfer function, which is one of *min, max, sum, sum-and-threshold, product, sign-of, greater-than, abs, exp, sin, cos, atan, if-then-else, divide, oscillate-wave, oscillate-saw, log*. The neuron receives input values either from a controller input port (which can be connected to a sensor, a brain or another controller), another neuron's output or it simply receives a constant value. Neuron's output

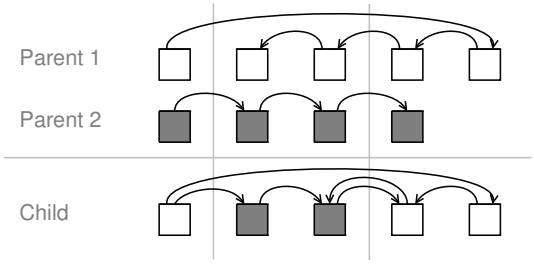


Figure 3: Crossover.

is connected to a controller's output port (which, again, can be connected to an effector, a brain or another controller) or to another neuron's input.

2.2.3 Effectors

Effectors allow the creature to move in the physical world by applying torques to the physical joints of the creature body parts. Each degree of freedom of a joint is controlled separately, by a single controller output.

The straightforward application of the controller output values to physical joints has shown to be inconvenient, because unconstrained output values often cause undesirable effects (twitching and jittering) and instability in the simulation. Therefore, several transformations are applied to effector values before their application.

Effector values are first clipped to range $[-1, 1]$ and then scaled by a factor proportional to the mass of smaller of two connected body parts. This transformation limits the maximum size of a force to some reasonable value and consequently improves simulation stability.

The value is then smoothed by a process of averaging a previous fixed number of samples (ten samples were used in this work) and only this average is used as the effector output. This modification eliminates the previously mentioned jittering and twitching of the creatures.

2.3 Genetic operators

Creature mating assures, that advantageous genetic information discovered by different individuals can be combined in their offspring. Same methods as those proposed by Karl Sims, i.e. grafting and crossover, were used for creature mating.

Both mating methods work by combining morphology graphs of the parents. During mating, the control system within each body part is copied along with that body part. Local control systems thus change only during mutation. The global controller (i.e. the brain) is simply copied from the first parent (the brain could also be combined from parent genotypes using one of the methods described below, but we chose the simpler solution in this work). Neural connections between parent and child node controllers, whose source or target node have been changed during reproduction, are pointing to the same relative locations or are randomly reassigned if needed (i.e. if a corresponding input or output port is not present or is already used by another neural connection).

2.3.1 Crossover

In this method, nodes of each of two parent genotypes are first aligned in a row. Then one or more crossover points

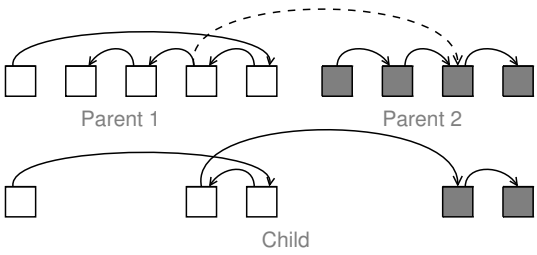


Figure 4: Grafting.

are chosen. An offspring is created by copying nodes from the first parent until the initial crossover point occurs. The copying then continues from the second parent and the copying source switches at each following crossover point. Example of a crossover operation is shown in Figure 3.

2.3.2 Grafting

Grafting works by first copying nodes from both parents to a resulting offspring, setting the root-node of the offspring to be the root-node of the first parent. Then, a single connection from the first parent is chosen and redirected to point at a random node in the second parent. Finally, all nodes, which are unreachable from the root-node are deleted (both from the first and second parent). Example of a grafting operation is shown in Figure 4.

2.4 Fitness evaluation

Each creature genotype is evaluated by first preforming a validity test as described in section 2.5. If the genotype passes the testing procedure, the creature phenotype is generated and placed in a virtual 3D world. The physical engine is used to simulate rigid body dynamics in a virtual world. Simulation proceeds in discrete simulation steps at a rate of 30 steps per second. During each time step, following phases occur:

1. Sensor values are filled according to current creature environment.

2. Neural signal is propagated throughout a distributed control system.

3. Torque is applied to each joint between body parts, according to the current value of the corresponding effector.

4. Simulation step is taken and the positions of all objects in the world are updated.

Simulation runs for a specified amount of time (60 seconds), during which the creature is evaluated. Walking, swimming and jumping fitness is defined as proposed by Sims in [10]. We also, like Sims, experienced problems with a "falling tower" strategy taking over during evolution. Special initial phase of simulation, as proposed by Sims, was added to prevent this.

2.4.1 Swimming

The water environment is simulated by turning off gravity and adding a viscosity effect. An organism fitness is determined as the distance traveled by an organism during the simulation. Forward motion is favored over a circular motion by computing fitness as a distance between starting and ending position of the creature.

2.4.2 Walking

Simulation is started with gravity turned on. A single infinite plane representing the ground is added to the scene. The creature is then placed on top of the plane. A special initial phase of simulation is needed to prevent a "falling tower" strategy to take over in the population. Therefore, organisms are first simulated for a specified amount of time without a fitness measurement. After the initial phase, the creature's current position is marked as a starting position and the simulation continues. The creature's fitness value is the distance between its starting and final position.

2.4.3 Jumping

Simulation is set up in the same way as in walking fitness. The resulting fitness is computed as the largest height reached by an organism (an organism reaches some height if his lowest point is at that height).

2.4.4 Physical simulation

ODE physics library was used to simulate rigid body dynamics. ODE offers a complete and reasonably robust solution for physical simulations and has proved to be sufficient for purposes of simulation of evolving creatures. We, however, encountered similar problems, as those encountered by Karl Sims, when we observed our evolving creatures exploiting errors and instabilities in the physics engine to their advantage.

There are three possible solutions of this problem: correcting the physical simulation so that it is more stable, creating creatures which do not cause instability, or simply changing fitness function so that instable creatures are handicapped and they extinct quickly. In this work, we used the combination of all three methods.

Following mechanisms are used to ensure that organisms do not abuse the physical engine:

1. *Simulation watchdog.* During the fitness evaluation, we continuously watch relative displacement of each two connected parts of each creature and when the maximum displacement rises above a specified threshold (which typically means that the creature starts to behave unrealistically), the organism is immediately assigned zero fitness and its simulation is stopped.

2. The *organism testing* has been introduced to quickly remove organisms, which are likely to abuse physics engine. The testing procedure is described in detail in the following section.

3. Physical forces used by creatures to move, are carefully controlled so that creatures cannot apply forces, which would result in unrealistic simulation. The details of this mechanism are described in section 2.2.3.

4. The *physical engine configuration.* ODE was configured to be as robust as possible to all kinds of creature behavior, while retaining a realistic simulation. In ODE physical engine, this was accomplished by the proper setting of CFM and ERP parameters (see [11] for details).

ODE engine does not automatically include the simulation of a water environment. The water environment is simulated by adding viscous forces and turning off gravity. The viscous force is added for each face of box in a direction opposing surface normal, proportional to the surface area and velocity of that face.

2.5 Testing

Testing mechanism was introduced to speed up the evolution process. As mentioned in the previous section, many organisms tend to exploit properties of a physical simulation to their advantage. However, several pre-simulation tests can be made to early discover abusive creatures. The following creature properties are currently tested before the simulation starts:

1. The count of nodes in a phenotype graph must be in a specified range (organisms with a large number of nodes tend to be highly unstable, and organisms with a small number of nodes are uninteresting).

2. Organisms must not interpenetrate themselves in their initial position.

3. The volume of each body part must be larger than the specified threshold (extremely small body parts cause instability in the physical engine).

Simple organisms tests are much faster to compute than an entire physical computation and moreover, it is convenient to exclude unsuitable creatures as early as possible, so that they do not consume computer resources. Therefore, every newly introduced genotype (by the creature generation, mutation or crossing) is being immediately tested and if the test fails, the genotype is discarded and a new genotype is introduced using the same method. This operation continues until a genotype passing the test is introduced or a specified number of unsatisfactory genotypes are created, in which case, the last created genotype is used.

The process of testing slows down the operations of crossing, mutation and creature generation, but significantly increases the overall computation speed, because genotypes do not have to be evaluated by the computationally-expensive fitness function to be discarded.

Another approach would be to completely avoid generating faulty creatures. This approach results in a more effective and faster computation and we use it wherever it is possible (e.g. the scaling of effector forces). There are few cases, however, when avoiding faulty creatures would inadequately complicate the operations of generation, mutation and crossing, making them less flexible (e.g. avoiding self-penetrating creatures). In these cases, we choose the simpler approach and repeat the process until a satisfactory creature is produced.

2.6 Debugging creatures

ERO introduces several tools allowing better understanding of a creature evolution and early detection of problems in creature design.

The most useful tool is the creature viewer and editor. Users can create new creature genotypes or view and edit existing genotypes (even during the process of evolution). Genotypes can also be saved for later use or loaded from a previously saved genotype. Each property of the genotype

Figure 5: Creature genotype editor.

can be changed from user's interface, although some parameters might be difficult to edit, because their genetic code was designed to aid the evolution process, not for design by hand (e.g. anchor point position). Both creature morphology and control system can be edited. The genotype editor is shown in Figure 5.

Users can also view the phenotype graph corresponding to the selected genotype. The phenotype can be interactively simulated in an environment corresponding to the chosen fitness function. Similarly, multiple creatures can be selected for simulation in a single 3D world. Users can use this feature to compare creatures or to study a specific evolutionary innovation.

The simulation can run in the anaglyph mode, when the image for the left eye is painted with cyan color and the image for the right eye with red color. Viewing this image with anaglyph glasses results in a convincing illusion of the 3D effect, which is useful for visualizing complex creatures.

Phenotype editor can also be automatically refreshed during an interactive creature simulation. Current values for each neuron, sensor and effector are shown. This is a valuable information while detecting flaws or improving the performance of the creature control system. A screenshot of the phenotype editor during the creature simulation is shown in Figure 6.

3. OVERVIEW OF ERO COMPONENTS

ERO was designed to be useful not just for evolutionary computations, but for other computations as well. Each computation is split into subtasks running in parallel on several computers. A single computation is called *a project*. For each project, ERO provides an interactive project configuration, a parallel computation on different computers and a viewing of the project results. Currently implemented projects include a testing project which computes an approximation of π and two evolutionary projects: binary organisms and virtual creatures.

ERO is composed of three different applications: *a user client*, *a server* and a computational client (or *worker*). The relations among these components are illustrated in Figure 7. The server and workers together form a computational layer. The purpose of the computational layer is to

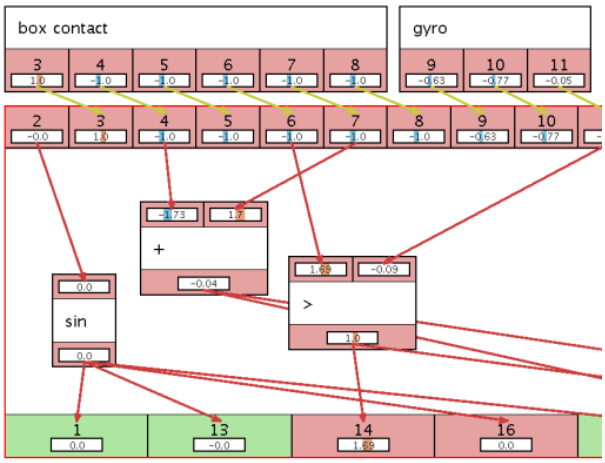


Figure 6: Detail of phenotype editor during organism simulation.

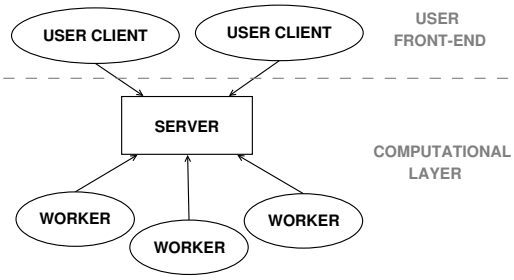


Figure 7: Overview of main project components.

receive a project from the user client, then run it and return computation results to all user clients currently watching the project. We describe the computational layer in detail in the following section.

The user client, on the other hand, is a user-friendly interface to ERO. It is the only interactive application among the three. Users can configure new projects, send configured projects to a server, view project results, pause/unpause projects or remove existing projects from the server. User clients can work with any project on any server.

4. PARALLEL COMPUTATION

The computational layer consists of the server and workers. The server runs projects and coordinates user clients and workers. Several projects may run on a server simultaneously, although they all share the same computational resources.

Workers, on the other hand, are simple computational clients. They receive a small part of computation from the server, compute it, and return the result to the server (server then hands it back to the project). The number of workers connected to the server is not limited.

Typically, a user client sends a project to the server and the server immediately runs it. A project may, while running, generate *tasks*, which are atomic computations (preferably small in size). Tasks are independent of each other, so they can run on different computers. Tasks produced by a project are distributed by a server among computational

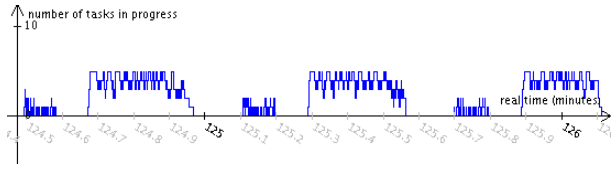


Figure 8: Computational efficiency (5 workers).

clients for computation. After a task is computed, its result is given back to the project (and the project, may, in turn generate new tasks for computation).

Various information about parallel computation is shown by the user client. Besides basic statistics (number of generated/finished tasks, running time, etc.) a graph showing the number of tasks concurrently computed by workers over time is shown. The graph has proved to be useful as a tool for detecting problems with a parallel computation and for improving computation efficiency. Parts of the project, which cannot run in parallel (such as organism selection or fitness transformation), are computed directly by the server. These computations cause gaps in the computational graph, as can be seen in Figure 8.

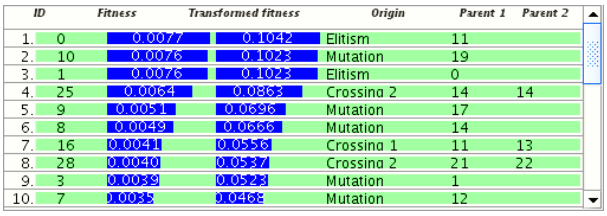
5. EVOLUTION

This chapter describes evolutionary projects (EP). EPs differ by organisms they work with. Each EP has to provide the following: organism genotype and phenotype representation, routines for generating, mutating and mating these genotypes and the fitness function. Additionally, EPs can optionally provide an organism viewer and editor and visualization of the fitness computation. All other properties (e.g. genetic algorithms or selection types) and user client features (project configuration, viewing the results of evolution) are preprogrammed and automatically available for each new EP. This also means, that when a new type of organism selection is introduced, all EPs can be immediately interactively configured to use it, with no need of any additional programming.

Currently implemented project properties common to all evolutionary projects are organism selection, fitness transformation and genetic algorithm itself. These properties are configured interactively by the user during a project configuration in user client application. The project, therefore, does not choose which genetic algorithm to use and the choice (and configuration of genetic algorithm) is left solely to the user.

An organism selection in ERO is a method of choosing k organisms from the pool of n organisms by their fitness (each organism can be selected multiple times, so k might be larger than n). Currently implemented methods of organism selection are roulette wheel selection, truncation, stochastic universal sampling and tournament selection (described in detail in [1, 4]).

Fitness transformation can be used by a genetic algorithm to transform fitness values of an entire population. Currently implemented fitness transformations are fitness normalization (divides each value by the sum of all fitness values) and linear ranking parametrized by selection pressure (fitness values are transformed to be equally distributed in range $[1 - s, 1 + s]$, where $s \in [0, 1]$ is the selection pressure and the order of organisms sorted by their fitness values is



	ID	Fitness	Transformed fitness	Origin	Parent 1	Parent 2
1.	0	0.0077	0.1042	Elitism	11	
2.	10	0.0076	0.1023	Mutation	19	
3.	1	0.0076	0.1023	Elitism	0	
4.	25	0.0064	0.0863	Crossing 2	14	14
5.	9	0.0051	0.0696	Mutation	17	
6.	8	0.0049	0.0666	Mutation	14	
7.	16	0.0041	0.0556	Crossing 1	11	13
8.	28	0.0040	0.0537	Crossing 2	21	22
9.	3	0.0039	0.0523	Mutation	1	
10.	7	0.0035	0.0468	Mutation	12	

Figure 9: Population window.

maintained after transformation).

A genetic algorithm in ERO can be any algorithm working with organisms ranging from simple genetic algorithm to hunter/prey model or even island genetic algorithm. The currently implemented algorithm, however, is a simple genetic algorithm, similar to that used by Karl Sims. First population is prepared by the user (either created by hand, or assembled from the results of previous evolutions), however, a user can leave some (or all) slots in the population empty. Empty slots are automatically filled with randomly generated organisms when the project starts. Organisms in current population are then evaluated by the fitness function. Resulting fitness values are transformed by chosen fitness transformation. A new population is constructed using the following sequence of steps: first, elite 20% of the previous population is copied to the new population, then 40% of remaining free slots is filled with mutated variants of organisms selected from previous population using the chosen selection method. Finally, remaining free slots are filled with organisms created by sexual reproduction. Parents are, again, chosen from previous population using the chosen selection method. Both grafting and crossover are used for mating (50% of descendants are created by grafting and 50% by crossover). All parameters of genetic algorithm (such as percentage of mutated organisms) can be changed by the user during the project configuration.

5.1 Debugging evolution

This chapter describes various tools which allow better understanding of what is actually happening "under the covers" of genetic algorithms. All of the following features are available when a user client connects to a running evolutionary project.

After a successful connection to a project, a screen displaying the current state and history of the genetic algorithm is shown. A part of the screen is occupied by a window showing a list of all generations with basic statistics (average and maximum fitness) for each generation. The user can select a specific generation and view its details in the population window. The population window contains a list of all organisms along with details about each organism. Details include the organism raw fitness value, transformed fitness value, organism's origin (whether it was created by elitism, mutation, crossing or random generation) and organism parents. The screenshot of the population window is shown in Figure 9. Users can also view genotype and phenotype of the organism or its parents or watch their fitness evaluation in a real-time 3D simulation.

The entire evolution history is stored on the server, so users can trace an organism genotype back to its oldest ancestors. Each organism can also be saved to disk for later use.

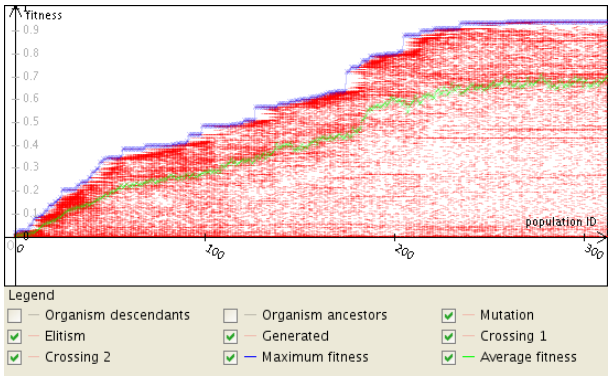


Figure 10: Fitness graph.

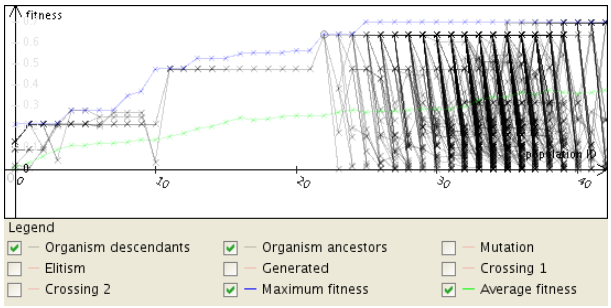


Figure 11: The "family tree" of an organism.

The process of evolution is best summarized by a graph of fitness values of all organisms yet created. The horizontal axis contains a generation number and the vertical axis contains a fitness value. Each organism is marked on the graph. This graph is useful for watching long-term trends or formation of subpopulations within the population. Average and maximum fitness values are also shown in the graph. Graphs are shown for both raw fitness (Figure 10) and transformed fitness. The marks of the organisms are partly transparent, so when multiple organisms organisms have the same fitness value, the mark is darker. This allows user to see the fine structure in the fitness graph, where there would be a solid area otherwise.

Each graph also contains options for showing/hiding organisms, which were created using a specified method. A user can, for example, filter organisms to see only those created by mutation. This is a particularly useful feature when searching for the source of faulty organisms.

Users can also view the entire "family tree" of an organism in the fitness graph. The family tree algorithm starts by marking current organism and continues by connecting the organism with its parents and its children. The procedure then recursively continues connecting parents to grandparents and children to grandchildren, etc. "Family tree" can be used for watching the migration of organisms in the fitness graph. The example of a fitness graph with a "family tree" is shown in Figure 11. Single best organism in generation 22 was chosen as a starting point. Graph shows, besides other things, that this organism was not created from the best organisms of the previous generation and his descendants did not absorb the next innovation (which occurred in generation 25) until the generation 35.

6. IMPLEMENTATION NOTES

ERO is implemented in the Java programming language. ODE (see [11]) is used for a rigid body dynamics simulation using modified OdeJava for Java bindings. OpenGL is used for real-time 3D rendering using JOGL for Java bindings. RMI is used for TCP/IP communication. XML file format is used for saving organisms and a project configuration.

Choosing Java as a primary programming language brings the advantage of the cross platform software. The usage of ERO is, therefore, only limited to platforms supported by ODE/JavaODE and OpenGL/JOGL, which are native C/C++ libraries. ERO was tested (and fully works) on recent Linux and Windows operating systems on 32-bit architectures.

Java's ability to dynamically load classes is used to automatically download code for projects from the user client to the server and from the server to the workers. New versions of projects can be thus tested without the need to restart either the server or the workers.

7. EXPERIMENTS AND RESULTS

Experiments were conducted using the setup described in section 5, i.e. 20% of new population was formed by survivors copied from the previous generation and the rest of the population was created by mutation, grafting and crossover operations in the ratio of 4:3:3. We used population size of 300 individuals. Evolution was stopped typically after 50-200 generations. Stochastic universal sampling was used as a selection method and normalization for a fitness transformation. These settings are also the default settings of an evolutionary project in ERO. Using 5 workers (all of them Pentium IV 2.0GHz, 512MB RAM), project with 100 generations and swimming fitness was finished in about 1.5 hours, achieving speed of slightly more than one generation per minute. Creatures were evolved for swimming, walking and jumping.

Walking fitness produced large number of simple small creatures which moved by series of small jumps. These creatures differed in the way they were balanced. Some evolved wide "heads", which prevented them from falling on the side (Creatures 2 and 6 in Figure 12). Others added fixed boxes to their body. Boxes had no apparent meaning, but when manually removed, creature flipped on its back and failed to resume forward movement. One of these organisms continued to extend its small jumps to longer and higher jumps until it became even better jumper than jumpers evolved by the jumping fitness (Creature 4, balanced by a small box attached to its central box). Some organisms moved by simply wagging one of their boxes back and forth in the air (Creature 3). The most creative solution to the balancing problem was, however, discovered by a creature, which added a copy of itself to its body. This double-creature then evolved into a "crab-like" organism with large arms and a perfect balance (Creature 1). This was also the fastest creature evolved for walking during our experiments.

Swimming fitness produced even larger diversity of interesting behaviors. A frequent strategy of evolved creatures was to move forwards by wagging their tail in a sinusoidal pattern (e.g. Creatures 9 and 10 in Figure 13). Besides moving forwards, these creatures often rotate around the direction of their movement. This already successful strategy was even more enhanced by a creature, which gradually invented a complex fin-like appendage at the end of its tail (Creature 8). Another creature used one appendage to pull itself forwards and another one as a fixed paddle to compensate the rotating effect and keep the forward direction (Creature 11). Simple and effective crocodile-like motion was invented by a creature, which moved forward by swinging its long and thin head side-to-side (Creature 12).

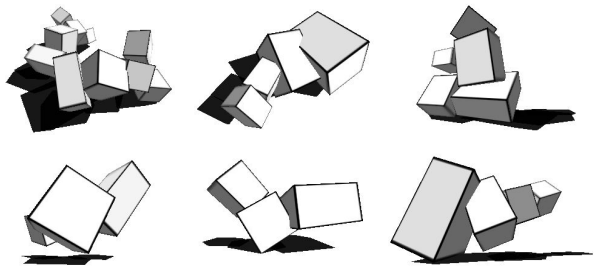


Figure 12: Creatures 1-6, evolved for walking.

The fastest and the most surprising motion was, however, developed by several creatures, all of which have independently invented a propeller device by tweaking the parameters of physical joints in ERO. This improvement was unexpected, since we had believed that physical joints did not allow development of such propeller-like structures. The intended use of a hinge joint is shown in Figure 15, Case A. Smaller cube can rotate around the hinge joint (illustrated by a thick line at the base of the cube). In order to restrict the smaller cube from rotating freely "through" the large cube, the collision detection is performed between the top half of the smaller cube (colored in gray) and the large cube (this is a regular mechanism of collision detection, used for all physical joints in ERO). Case B shows the misuse of a hinge joint in order to create a propeller. Hinge joint is shifted close to the edge of the larger cube and the smaller cube is rotated by 90° (using the relative *rotation* parameter, as described in section 2.1). The larger cube now does not restrict the smaller cube from rotating freely, thus the smaller cube can rotate continuously, acting as a propeller.

Many variations of this basic principle were discovered. Creatures differed in the number and structure of paddles attached to the main propeller (Creatures 7 and 13). The most successful propelling creature (and also the overall fastest swimming creature) was composed of two main body parts, connected by an evolved propeller-joint, with each body part having two symmetrical paddles (Creature 16 in Figure 14).

Jumping fitness has not produced such satisfactory results. Creatures mostly evolved to a chaotic group of boxes moving in random directions. Jumping behavior during the creature simulation seemed to occur only by accident.

In summary, several interesting locomotion strategies have evolved, many of them resembling behaviors found in the nature. Largest diversity of evolved creatures was produced in the water environment. This is probably due to the fact, that the water environment immediately reflects even the smallest mutations, making the selection more efficient.

8. FUTURE WORK

One direction of the future work lies in *completing* the creatures to include all features of Karl Sims' creatures. This

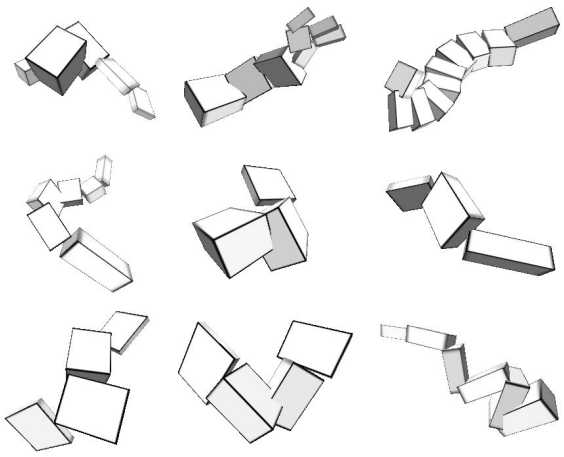


Figure 13: Creatures 7-15, evolved for swimming.

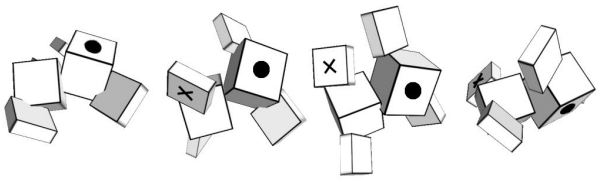


Figure 14: Creature 16, the fastest swimming creature.

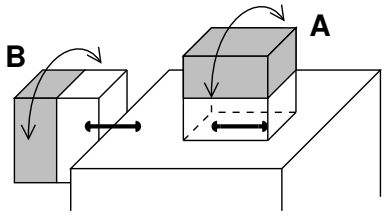


Figure 15: Typical use of a hinge joint (A) and hinge joint tweaked to form a propeller (B).

is, however, not a difficult task, since only a few features have not been implemented yet: a reflection component of parent-child relative position (as described in section 2.1) and several types of physical joints between organism parts (three joint types are currently available in ERO, as opposed to seven joint types implemented by Karl Sims).

Creatures could also be *extended* in various ways. Body parts could have different shapes (e.g. sphere, cylinder or even general convex triangular mesh). Current control system of the creatures can be replaced by a completely different distributed control system (such as multi-layer perceptron network, recurrent neural networks, etc.). Our creature design even allows these different systems to coexist within a single creature. This would open an interesting possibility of letting organisms pick the best control system during their evolution.

Another work lies in fine-tuning genetic operators to increase organism diversity and speed up the adaptation process. ERO is also ready for experiments with other types of genetic algorithms, such as the island genetic algorithm, hunter/pray model or competitions of individuals (as described in [9]).

Computational layer could also be enhanced in several ways. For example, projects could be assigned priorities, so that the project with a higher priority would have an access to more workers than a project with a smaller priority. This would increase a user's comfort when working with several projects simultaneously. Computational layer could also be more workers are available than necessary.

9. CONCLUSION

We have introduced a framework for evolutionary experiments and used it to reproduce some of the experiments of Karl Sims. A built-in engine for parallel computation of CPU-intensive tasks significantly speeds up the evolution. Our implementation of creatures is extensible and ready for new experiments and enhancements.

Behaviors of evolved creatures often resemble those of the real animals. Perhaps evolved creatures could help us understand evolution of locomotion strategies in the nature (e.g. the invention of a propeller-like structure from non-propeller components might resemble the invention of a similar locomotion system found in the nature – a propelling bacterial flagellum).

10. ACKNOWLEDGMENTS

The framework could not be finished without the help of Miroslav Blaško, Alexander Kutka, Rudolf Vlk and Daniel Toropila, my colleagues at the Faculty of Mathematics and Physics, Charles University in Prague. The author would like to thank to Russell Smith and ODE community for ODE physics engine.

The work on this paper was supported by the Grant Agency of Charles University in Prague under Grant-No.358/2006/A-INF/MFF.

11. REFERENCES

[1] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, January 1989.

[2] G. S. Hornby and J. B. Pollack. Body-brain co-evolution using L-systems as a generative encoding.

In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 868–875, San Francisco, California, USA, 7-11 2001. Morgan Kaufmann.

[3] M. Komosinski and S. Ulatowski. Framsticks: Towards a simulation of a nature-like world, creatures and evolution. In *ECAL '99: Proceedings of the 5th European Conference on Advances in Artificial Life*, pages 261–265, London, UK, 1999. Springer-Verlag.

[4] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[5] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[6] H. Lipson and J. Pollack. Evolving physical creatures. In *Artificial Life VII*. MIT Press, 2000.

[7] T. S. Ray. Aesthetically evolved virtual pets. In *Artificial Life 7 Workshop Procs*, pages 158–161, 2000.

[8] Y.-S. Shim and C.-H. Kim. Generating flying creatures using body-brain co-evolution. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 276–285, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[9] K. Sims. Evolving 3d morphology and behavior by competition. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 1994. ACM Press.

[10] K. Sims. Evolving virtual creatures. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 1994. ACM Press.

[11] R. Smith. ODE manual. Available at http://www.ode.org.

[12] J. Ventrella. Attractiveness vs. efficiency (how mate preference affects location in the evolution of artificial swimming organisms). In *ALIFE: Proceedings of the sixth international conference on Artificial life*, pages 178–186, Cambridge, MA, USA, 1998. MIT Press.