Danmarks Tekniske Universitet

Walking with Dinosaurs

The project that hasn't much to do with dinosaurs

Bachelor Project

Written by:

s103476 - Rasmussen, Kasper

s103454 - Lund, Gustav

DTU Informatik
Institut for Informatik og Matematisk Modellering

Bachelor Project

Supervisor: Joseph Kiniry
Delivery date: 28/06-13
Contains 51 pages including the front page

Abstract

This paper describes the design and implementation of an ALife system, inspired by Karl Sims work (Evolving Virtual Creatures). The objective of the project is to create, test and evolve 3D creatures in a virtual physics environment. The creatures with the best test scores are chosen for further evolution, following a Darwinian survival of the fittest approach. Evolution is done by crossbreeding and mutating a DNA representation of the creatures. We present a controlled way to crossbreed a DNA string that has greater chances of successful cross-breeding, and a mutating method that changes mutate frequency based on the actual population of creatures. We furthermore present an approach to controlling the virtual evolution, by the use of different test methods. The different tests are chained together, or used combined to yield better results.

Distribution of work

This project has been a joint effort, and thus most of the work has been distributed evenly in the group. A few subjects has however been distributed for specific members of the group to handle. Thus Gustav is the one responsible for the rendering solutions in the project, and Kasper is responsible for the interprocess communication between server and client parts, of the project.

We would like to thank Joseph Kinery for his help in guiding us through this project as our advisor, and providing helpful support.
We would also like to thank Maja Lund for helping, by sending us several interesting creatures evolved with our program.

Table of Contents

Figures

Tables

Introduction 1

1.1 Project Description

Design an `ALife` system, inspired by Karl Sims work (Evolving Virtual Creatures), where 3D creatures are evolved and fitness tested in a 3D environment, using a physics engine. The creatures have neural nets that combined with sensors and effectors, enable them to move. The creatures with low test scores are terminated and replaced, which should mimic the survival of the fittest scheme. The creatures should resemble dinosaurs.

Primary Goals

- The creatures will be defined and simulated in a 3D environment.
- The creatures should act accordingly to physics.
- The creatures should optimize themselves through evolution. The evolution should be guided by survival of the fittest, which is evaluated by fitness tests.
- The fitness test should evaluate how well the creatures move (distance).
- We will be using a learning system approach with neural networks, to make creatures act.
- The creatures should be based on a fixed dino-like starting point.
- The program will be parallelized on the CPU.
- The simulation can be visualized from a saved file.

Secondary Goals

- Several other fitness test will be implemented (walk, jump, react to outside stimuli).
- A GUI of the simulator will be implemented, enabling the user to set certain simulation parameters, and view the progress.
- The creatures will be evaluated based on several different fitness tests, instead of just one.
- We will speed up the simulation, by using the GPU for further parallelization.

Tertiary Goals

- The creatures should be skinned.
- Several more outlandish fitness tests will be devised (reaching for objects high up without jumping, battle, etc.).
- We will use decision trees for the creatures.
- Varied landscapes will be used for the simulations.
- We should output the simulations in a known media file format.

1.2 Glossary

Creature: A solution to the problem, represented as a DNA string, and at fitness value for the solution.

Population: The group of creatures that is simulated and evolved in in a generation.

Simulation: A simulation of a creature in the physics engine that runs it for the predefined amount of time.

Generation: One simulation and evolution of a population.

Batch of generations A series of generations.

DNA string: A sequence of numbers representing a creatures genotype.

Fitness value: A value evaluating a solution. The higher value the better.

Bullet: Short for `Bullet` Physic library or `Bullet` physics engine. `http://www.bulletphysics.org/`

Design 2

2.1 Learning System

The creatures we want to evolve have to be able to move, and to this end they have muscles in each joint. These obviously needs to be controlled by some kins of "brain", or we will never see any movement. We additionally want to evolve this "brain" as well. To do this we use a learning system approach. The model we are using is called a neural network (hereafter called "neural net"). It is a biologically inspired model, that seeks to approximate the neural network of the brain.

Our situation can be viewed as a reinforced learning problem, since we are judging the net on its performance in the simulation (coupled with the creature physiology), instead of trying to fit it to a quantity of input/output pairs.

2.1.1 Neural Network

The normal way to build neural nets is to have a net consisting of layers of nodes, where each layer takes inputs from the previous layer, and sends its outputs to the next layer. This approach does not allow loops, and the net is strictly hierarchical.

The nodes of the neural net, each contains a function which is applied to the input of the node. A basic neural network node contains only few simple functions, which means that a large network is needed for more advanced function, which is need for coordinated movement of the creatures. To avoid the large networks which takes allot of time to optimise, it is possible to expand the array of possible functions with some more advanced, which then act as a predefined large chunk of a basic neural network.

The neural net could consist of a single net or a series of connected nets. A series of connected nets can be viewed as just a normal larger neural net as long as the nets does not form loops, and as such the functionality of the model should be preserved. We divide our creatures' neural net into a main neural net, that gets its inputs from sensors, and the subnets, that takes the output values of the main neural net as inputs. Each joint in a creature have its own subnet. The output values of a subnet are used in the muscles of the respective joint.

The idea behind using subnets as compared to simply using a single neural network is, that when cross-breeding creatures we will copy the subnet as well as a creature part, when the part is copied. This could also take place by simply giving a copied part the same output as the previous part (from a single neural net). Small modifications to the controlling however (such as making asynchronous walking), can be done with just a few mutations with the subnet method, compared to having to evolve an entirely new control output in the net. The downside is that we get more degrees of freedom in the creature DNA, thus making it harder to optimize.

To make it easier to optimize the net, mitigating the downside of the previous feature, we limit the size of the neural nets. By restricting the size of the nets we simplify the neural network part of our DNA, increasing the importance of the remaining parts of the neural net DNA. At most 3 outputs are ever used from each subnet, since a subnet is attached to only one joint which has at most three dimensions of freedom. This means that a subnet with over 3 nodes each layer in most cases equals a lot of nodes without any influence. The main network should have more nodes in each layer. The more nodes each layer however, the less the chances are for two subnets to use the same output, which is one of the points of having the main neural network in the first place. Restricting the amount of layers in addition to this limits the complexity of the resulting function for an output. Since we are just attempting to teach the creatures to walk, and not to solve the problem of world hunger, we do not believe that the creatures will be able to utilize the potential of a larger net.

2.2 Artificial Life

Artificial life or `ALife` is a branch of computer science, that concerns itself with implementing approximations of real life objects into virtual environments. This could be with the objective of examining reactions to different changes in environment, or to simulate growth of an organism [1]. Since our project is inspired by the works of Karl Sims, it falls squarely under the category of `ALife`. What we are essentially doing is attempting to create a very simplified representation of an unknown animal, in order to build the creature, and simulate its behaviour in a virtual environment. Evolving on the creature representation, can also be viewed as an attempt to simplify and virtualize the natural evolution, which also falls under the `ALife` category.

2.2.1 Genotype

One of the questions concerning the `ALife` is how to represent a creature. A possible way to do this is a object oriented approach, where each part of a creature is an object and saved in some data-structure (directed graph, hierarchical objects). Another way could be as a string of numbers or symbols, a DNA string of sorts. That is the approach we are using. A DNA string can be read in several ways, and can simply be viewed as a compressed version of the objects, which makes it easier to send a creature's data. A way to interpret the DNA string could be by the use of context-free grammar and use the Backus–Naur Form notation.

The way the creatures are represented affects how difficult it is to implement an evolution scheme. Our initial approach was to use grammatical evolution since it has a simple evolution scheme, and crossovers[1] (on the DNA string) are exceedingly simple. Doing this when using saved objects as representation would be much more complex which reinforced us in the fact that using a DNA string was the way to go.

[1]Used in crossbreeding. Basically switches a part of one creature out with that of another

2.2.2 Grammatical evolution

Grammatical evolution in our case would consist of reading the DNA string with a grammar, and mutating on the DNA simply by randomly changing the value as some place/places. Crossbreeding would be done by switching a small number of DNA segments with segments from another creature, with random sizes. The implementation of this is extremely simple, and we used this in the early stages of the project.

We realised however that pure grammatical evolution is not the optimal way to go for this project, since our grammar had some flaws when it came to crossbreeding, or evolving parts like the neural net. It has a large tendency to make major uncontrolled changes, and going in a completely different direction. The problems are more precisely that a creature consist of two main parts (body and main neural network) that is structured quite differently. Because of this difference they are defined in opposite ends of the DNA string. Some times when mutating, but more urgently very often when crossbreeding, the changes in the DNA send ripples through the rest of the DNA, completely eradicating any resemblance to the original creature. While such drastic changes in the creatures (or evolution nukes as we call them), can yield interesting results, they are supposed to be at least semi-rare. Otherwise it makes it almost impossible to optimize an existing creature.

We could make two separate DNA strings for the body and the neural net, however it was not a good option since the neural net and body are interlinked. Also the subnets should be placed in the same area of their respective parts, to allow copying of the entire part with the defined movement (which was the entire point of the subnets).

To avert this we instead modify the original grammatical evolution idea, by limiting which parts the crossbreeding are able to access. We ensure that it can only switch parts of DNA that actually defines entire parts of the creature, and that the part/parts we copy fit into the area we paste it into.

2.2.3 Evolution

Our evolution method is based on the principle of survival of the fittest. In nature the theory of survival of the fittest tells us, that the creature most fit to survive inevitably will produce the most offspring, and thus ensure that its species will survive. In our virtual environment, we are the ones that determine whether a creature is fit, and how fit it is. Thus we are simplifying the concept quite a bit. We will use this unfair power over virtual evolution, to evolve creatures with features we deem interesting.

To reach our goals the evolution should strive to get a diverse population, in order to reach as many possible creatures as we can, but if some of them are falling too far behind, they should be eliminated, because they slows the evolution down without contributing. The perk of killing the stranglers is that the population size can be kept down when adding new creatures(solutions) to the fold, thus getting more results without increasing simulation time much.

Our evolution method will divide the population up into several groups, sorted after how well the creatures perform. The worst of the lot will be discarded as the waste of gene pool space they are, while the very best of the creatures will be kept without mutating them. Thus we ensure that we do not mutate our best creatures and risk rendering them useless, which would yielded worse results. The rest of the creatures, those that are neither worst nor best, are crossbred with the elite creatures, and mutated depending on the diversity of the population. The ratio of elite creatures, breeding creatures, and gene pool waste, are defined statically, and are currently set to:

Elites 20% of the creatures

Breeders 30% of the creatures

Gene Pool Waste 50% of the creatures

Then there is the question of choosing a fixed or a dynamic population size. In the real world the a population size is dynamic; changing over time. If we want to make the simulation as realistic as possible this means that we will need dynamic population sizes as well. Having the population size dynamic would be intriguing as well, but will also cause several problems. For instance we are not interested in the population dying out, and if the population gets too large the simulation would begin to take immense amounts of time, creating lots of similar creatures. We could mitigate this by restricting the population size with upper and lower bounds, but the simplest solution is simply to fix the population size from the beginning. When creatures are discarded we instead replace them, by copying some of the middle creatures, and processing them like the others.

Fitness test

To guide the creatures' evolution progress and decide who the stragglers are, a way to evaluate them is needed. To do this we test certain parameters in the simulation, evaluating the performance of the creatures. We use the term "a fitness test" to describe the evaluation. Several different fitness tests could be devised, but common for them all is that in the end we want an absolute score, that is a single number, and always gives the same result for the same creature.

The fitness tests can be devised in several ways and based on several factors. The simplest factors would be size, weight, movement or a combination of them. Weight and size are not suitable for their own fitness test since they are too simple, and the goal can be reached quickly. Also the results yielded by such a test would not be very interesting. On the other hand movement is more complex, and can be split into two categories: movement horizontal and vertical, aka walking and jumping. Weight and size can be used as secondary parameters to create fitness tests that value larger creatures more/less. Movements in one or both plans can also be used as a secondary requirement for more elaborate fitness test.

Due to the randomness of the evolution it is likely, that the creatures will have very varying fitness values, in every generation. The creatures will attempt every way to solve the problem of the fitness test, even ones that make no sense. Unfortunately this means, that the creatures will probably cheat in any possible way, to achieve a higher score, utilizing any loophole in the programming. Because of this the fitness test needs to be very precise to achieve the desired behaviour.

Selective evolution

The evolution need more than just the fitness test to guide them, if we want quicker and more interesting results. While mutation is random by nature, crossbreeding methods can be more or less supervised. There are several ways to do crossbreeding, depending on the implementation of the DNA. The basic idea is that some features of one of the elite creatures (hopefully) has a chance of being transferred to the breeder creature, which could result in a creature superior to both its "parents". While simply copying a random part of the elite DNA into the DNA of the breeder might do the trick, we instead wish to have more control over the process. We switch a part of the breeder creature with a part from the elite, by switching the corresponding DNA parts. This way we ensure that we do not ruin the DNA when crossbreeding. Mutation can of course still ruin a creature completely, just as it can in nature.

Another way of controlling the evolution is to simply watch over it. While it is quite impossible, and not very productive, to view every single creature, it is possible to view several results of simulations with "few" generations. The most interesting of these results can then be chosen for further evolution and the results viewed in the same way. This way we allow ourselves to choose creatures not only on performance, but also depending on how we deem the future chances of a creature developing interesting attributes. its an interesting approach and as shown by Karl Sims [2] it can create some complex results. The only real problem with this approach is that it requires human intervention each x generations.

2.3 Physics

To simulate the physical world a physic engine library is needed, since simulating the physics by ourselves would be an project on its own. What we need is a physics engine covering the following.

- 3D engine
- Friction
- Gravity
- Well enough precision
- Established library with a good renome

2.3.1 Scientific vs. game engine

There are two kinds of physics engines, game engines and scientific engines. A game engines main purpose is to deliver fast results, since they are used for games where lag is not tolerated. The speed comes at the cost of precision. A Scientific engine is in many aspects the opposite of the game engine, since its main purpose is to deliver correct precise results. This usually comes at the cost of speed. We went for a game engine, since a reliable game engine is a lot easier to get by than a scientific one. By restricting a game engine enough it should be possible to make it act enough like a scientific engine and therefore oblige our meagre needs.

There are mainly two game physic engines which suited our needs

Havok

- 3D engine
- Friction
- Gravity
- Large community
- Well established brand with a good renome
- Used for large projects (movies and games)

Bullet Physics

- 3D engine
- Friction
- Gravity
- Open source
- Access to source code
- Large community
- Used for large projects (movies and games)

The physic engines seemed rather similar and both covered our needs for a physic engine. The only thing that distinguishes them is the fact that one is commercial and the other open source. We went for `Bullet` solely on the reason that it is open source.

2.3.2 The Shape of the World

When creating a world it is possible to use a wide array of shapes. More shapes means a higher complexity of the code though. If more complex shapes are used, it would be possible to create more complex creatures, and therefore a wider array of results, but by using a simple shape like a square instead, our program gains a lot of simplicity. This makes the physic calculations less time consuming because of the fewer edges. Simplicity also have the advantage of fewer parameters to optimize and therefore it is easier to achieve results faster. Besides that, we like squares. Squares are good.

2.4 Framework

We want a batch of generations to run as a process with a clearly defined end. Thus the number of generations to run, and the number of creatures each generation will be specified before the process itself has begun. The processing of a batch of generations would have to roughly follow the following algorithm:

1. Read DNA of a creature and build the creature in a simulation world.
2. Check if the creature is legal.
3. Simulate creatures in a physic world for a set amount of time, and calculate the fitness value.
4. Retrieve the final fitness value.
5. Once all the creatures of a generation has been through these steps, sort and evolve the creatures.
6. Repeat until the required number of generations have been run.

2.4.1 Link between GUI and Simulation

While our initial idea was to have two completely separate programs, a simulator and a simulation viewer, we instead opted towards having a graphical user interface, that would handle both tasks. We wanted to be able to start the simulation from this interface, and show the result when it has been completed. By doing it this way instead, we would be able to reuse the simulator to show results, by only slightly modifying it, as opposed to having to save the simulation result, as some kind of video. Alternatively we thought of simply copying some of the simulator to the viewer, however this would just give us redundant code for no good reason. This led to the GUI solution being preferred.

2.4.2 Parallelism

We can use both the CPU and the GPU to parallelize the code. But each is best in its own specific areas.

GPU

The GPUs main purpose is to make many similar simple calculations. It will therefore make most sense to parallelize the physics calculations on the GPU, especially the collision detection. For a single simulation world however, we lack the amount of calculations necessary to overcome the overhead and the slower speed of the GPU cores. Even if the collision detection uses brute force ($15^{15-1} = 210$ comparisons, since we are limited to 15 boxes each world), the amount of calculations is too few. It does not help either, that all shapes used are boxes and therefore fast to compute collisions for.

`Bullet` does not fully support GPU simulations, which means that only a few parts of the simulation can be parallelized, without rewriting large parts of `Bullet`. The `Bullet` documentation is also rather lacking on the subject.

The effort needed to make `Bullet` work correctly on the GPU, and parallelize complete simulations is rather daunting, but will probably result in a speed up. On the other hand only parallelizing what `Bullet` supports will result in only a minor speed up, if any. The gain from this is not worth the effort and time.

Another area which actually would benefit from being parallelized, on the GPU, would be the rendering.

CPU[DONE]

The CPU is more suited for our task since it is a multi purpose device. There are three main ways to parallelize a program: threads, `OpenMP` and IPC- Inter Process Communication.

The easiest thing would be to let each world be paralellized in its own thread. The fastest way to achieve this is to use `OpenMP`, which is a simple, effective and high-level approach. Another way to do it would be to use threads (`WIN32` API, `pthread`s etc.). This would result in more control but also more work, and more complexity.

These two options depend on the fact that the code, including the physics engine, is thread safe. After some study it turns out that this is not the case with our physics engine, since `Bullet` is a game engine, and therefore prioritizes single thread speed over thread safety. This means that we can only use these approaches for the parts not using `Bullet` or we will have to rewrite parts of `Bullet`.

The last approach, and the only which ensures thread safety, is IPC. The main idea is to completely separate the unsafe parts from each other by letting them have their own stack and heap. This is done by separating the code in different executable files. It is more complicated and has more overhead than the other approaches, but it ensures the thread safety. The extra overhead caused by the IPC would be limited by the fact, that we have a rather low amount of communication need. This is the case since every simulation is independent of each other, and only needs to send info in the beginning and end of a simulation.

2.5 GUI

The Bullet demos are already visualized using GLUT and WIN32. The GLUT framework, while useful in showing the results earlier in the project, is not very good for creating a more elaborate GUI. When choosing the API for our graphical user interface to use we discussed WIN32 compared to plug-in libraries such as QT. The main reason WIN32 was considered, is because the Bullet demo framework already contains the basic features, for showing the results in a WIN32 GUI. Since none of us have worked with any of the frameworks we found that could be used, this was a great incentive for simply picking the native solution. Also with the WIN32 API we will not need to worry about linking a library with our project, something that has proven quite difficult when using Bullet. A disadvantage with using WIN32 API is that it is usable on Windows computers only. However since we already decided to go for the Windows platform in Parallelism, the cross platform feature was not possible anyway. Since WIN32 is also a widely used API and that we know it has the necessary capabilities, we ended up choosing it for the GUI.

2.6 Rendering

Since we optioned for using Bullet Physics we have two options on how to handle the rendering. we can make the rendering from scratch using technology's like OpenGL and DirectX, or we could use the basic Bullet's demos rendering (OpenGL) as a starting point. To start with, lets look at the rendering system in the Bullet demo framework.

2.6.1 Bullet's rendering system

As part of Bullet's demo framework, a system for rendering the demos are implemented utilizing OpenGL. The system uses either GLUT or WIN32 to create the window and run the animation loop. The rendering itself uses a kind of 3 pass algorithm. On the first pass the objects are drawn. On the second pass the stencil buffer is enabled and filled with the shadows of the objects (this pass is actually run twice to get contributions from both objects defined in clockwise triangles, and objects defined in counter clockwise triangles). On the last pass the rendering of the objects are repeated with the stencil test enabled. Thus only the parts of the blocks that are in shadow are redrawn. The color used during the last rendering pass, is a darkened version of the normal color of the object. All of the rendering passes runs through the objects saved in the physics engine, and draws for each one of them (the object itself or its shadow depending on the pass). The color of the object alternates between two main colours giving a chequered pattern. Additionally the main colour is given a more red tone if the object is active, and a greenish tone if the object is sleeping. As part of the rendering system, an option to only draw the wire-frame of the objects are also implemented.

2.6.2 Analysis

A disadvantage of choosing the `Bullet` rendering, is that it is just a demo visualisation, and the rendering therefore are not very interesting, since it is just there to show simple demos. For example lightning is almost non-existing, since shadows are made with a simple shadow casting algorithm. Also the colour of the objects are flat, and even determined at random.

It is possible to remedy these fault of course. Lambertian light can be implemented by adding a shader and with relatively few modifications to the rendering. Softer shadows would prove more of a challenge, and could require a major part of the rendering to be rewritten , but is also far from impossible. A few speed-ups are also possible, by sacrificing memory per object for rendering speed. for an example we could save the normals instead of calculating them each pass. In our project though it is not worth the effort, since the rendering only runs when the simulation is not running, and because of the simplicity of our scenes, there should be no speed problem.

If we wanted to do beautiful renderings of our results, we would most likely have to rewrite the rendering from scrap. However since the focus of this project is not solely on the rendering, we went for the `Bullet` demo framework, since it will produce much faster results, and will fit our needs just fine.

2.6.3 Modifications

While the prospect of chequered dinosaurs is intriguing we decided that we would improve the rendering a bit, by modifying the system to be a bit more pleasing to the eyes. The debug colours and the chequered pattern thus obviously had to go. We replaced this with a white base, and added texturing to the objects. The texture is different depending on the type the object have. This means that we are able to use different textures for the ground and the creatures themselves. Additionally the grey background were a bit uninteresting, so we added a simple skybox to give the illusion of a field with mountains and sky in the background.

2.6.4 Skinning

Adding skin to the creatures in the project were discussed. The objects themselves would act as bones, and a skin would be placed over them. This way the creatures not only look more like creatures, but also the simulation would likely get more realistic results. `Bullet` implements a way of simulating soft bodies[2] , that could be used to simulate skin. The complexity of this however led us to drop the feature, coupled with the fact that when simulating evolution the creatures have a tendency to exploit any weakness in the physics engine. The skinning would almost certainly have led to exploits, and for a low priority feature, it would give too much trouble.

[2]See the Bullet demo, App_SoftDemo

Implementation 3

3.1 Grammar

The construction of the creatures are done by reading the DNA string through a grammar. The basic grammar is designed as a context free grammar in Backus Naur form. While the grammar is being read the creature is being constructed simultaneously, each part or constraint being added when the necessary information is obtained. The only exception to this rule is the neural subnets. The information for these are placed next to their respective parts. A subnet cannot be created while the main neural net is not there though, so the subnets are buffered and created after the rest of the creature.

The basic version of our grammar can be seen below. A more elaborate version is available in appendix B page 42. The more advanced version also contains the chooser variables, that are used where necessary to pick the case. As an example the `B` non-terminal contains 6 cases (5 cases and a null case). Since the input is only numbers the next input number has to decide which one is the current case, or the grammar would not be able to decide how many more numbers to read. While this adds more length to the DNA, it also enables the grammar to weight the cases, making some more probable than others (See Neural Nets for an example).

Grammar:

S	::= Bi B NN
BI	::= h w d Se
B	::=
	| J
	| J J
	| J J J
	| J J J J
	| J J J J J
NN	::= NNL
	| NNL NNL
	| NNL NNL NNL
	| NNL NNL NNL NNL
Se	::= $type$
J	::= NN E $screw$ PRE POST DOF Bi B
NNL	::= Ni
	| Ni Ni
	| Ni Ni Ni
	| Ni Ni Ni Ni
	| Ni Ni Ni Ni Ni
	| Ni Ni Ni Ni Ni Ni
	| Ni Ni Ni Ni Ni Ni Ni
	| Ni Ni Ni Ni Ni Ni Ni Ni
	| Ni Ni Ni Ni Ni Ni Ni Ni Ni
E	::= $i1$ $i2$ $i3$
PRE	::= x y $side$

POST::= $x\ y\ side$
DOF ::= $x\ y\ z$
Ni ::= $value$
 | $f\ in1\ w1$
 | $f\ in1\ in2\ w1\ w2$
 | $f\ in1\ in2\ in3\ w1\ w2\ w3$

Description

S = The start of the grammar
BI = Contents of a box
 h = Height of the box
 w = Width of the box
 d = Depth of the box
B = Box (a part of the creature)
NN = Neural net
Se = Sensor
 type = Type of sensor
J = Joint
 screw = Rotation of the joint
NNL = Single layer in a neural net
E = Effector
 i1 = index in neural net for axis 1 of the effector
 i2 = index in neural net for axis 2 of the effector
 i3 = index in neural net for axis 3 of the effector
PRE = Location on the box to attach the joint
 x = displacement along x-axis of the face
 y = displacement along y-axis of the face
 side = the face index on the box
POST= Location on the new box to attach the joint
 x = displacement along x-axis of the face
 y = displacement along y-axis of the face
 side = the face index on the box
DOF = Degrees of freedom in the coordinate system of the joint
 x = Degree of freedom in the x-axis
 y = Degree of freedom in the y-axis
 z = Degree of freedom in the z-axis
NI = Neural net node
 value = Value of a constant value node
 f = Function type
 in1 = Index of input 1 in previous layer
 in2 = Index of input 2 in previous layer
 in3 = Index of input 3 in previous layer
 w1 = Weight of input 1
 w2 = Weight of input 2
 w3 = Weight of input 3

3.2 Neural Nets

Our neural nets are consists of layers with nodes in them. When the net is to be calculated, which happens on each simulation step, we calculate each layer in the net, starting from the first layer. This is opposed to only calculating part of the net as Karl Sims did, however we find that doing that would discriminate against larger neural nets. If we later wish to change the frequency that the neural nets are calculated with, we could instead only calculate the neural net every second simulation step etc. This should yield the same results since only the output values of the last layer are actually used in the physics simulation, and thus calculating a single other layer would not be noticed in the rest of the simulation, before the last layer was recalculated.

When calculating a layer, we simply calculate all the nodes in it. Each node consists of a function, and values defining where it finds its input in the previous layer. Each input can also be weighted according to values defined in the node. Calculating a node is as simple as running its function on its weighted inputs, and saving the result in the output value of the node. A node can be defined as having 1, 2, or 3 inputs. If its function takes more inputs than it owns, the missing inputs will be defaulted to values that has the least impact on the result as possible. The following functions are defined for the nodes:

Function -1: Sensor
While not really a function, this node returns the value of the pointer defined as its sensor input. When it is calculated, we assume that the sensor in question has already defined the value.

Function 0: Sum
Returns the sum of the weighted inputs 1 and 2.

Function 1: Product
Returns the product of the weighted inputs 1 and 2.

Function 2: Divide
Returns the quotient of the weighted inputs 1 and 2.

Function 3: Sum Threshold
Return 1 if the sum of the weighted inputs 1 and 2, are greater than the weighted input 3. Returns 0 otherwise.

Function 4: Greater than
Return 1 if the weighted input 1 is greater than the weighted input 2. Returns 0 otherwise.

Function 5: Sign of
Returns 1 if the weighted input 1 is greater than zero, -1 if it is smaller than zero, zero if it is zero.

Function 6: Minimum
Returns the smallest of the weighted inputs 1, 2, and 3.

Function 7: Maximum
Returns the greatest of the weighted inputs 1, 2, and 3.

Function 8: Absolute Value
Returns the absolute value of the weighted input 1.

Function 9: If
Returns the weighted input 1, if the weighted input 2 is not zero. Returns zero otherwise.

Function 10: Interpolate
Returns the average of the weighted inputs 1, 2, and 3.

Function 11: Sine
Returns the sine of the weighted input 1.

Function 12: Cosine
Returns the cosine of the weighted input 1.

Function 13: Inverse tangent
Returns the inverse tangent of the weighted input 1.

Function 14: Logarithmic
Returns the natural logarithmic of the weighted input 1, if it is positive. Simply returns the weighted input 1 otherwise.

Function 15: Exponential
Returns the weighted input 1 to the power of the weighted input 2.

Function 16: Sigmoidal
Returns the sigmoidal function value of the weighted input 1. ($1/(1+exp(-input1 * weight1))$)

Function 17: Wave
The node keeps track of how many times it has been calculated. It ignores its inputs and returns a value calculated from this by the formula $sin(stepsRun * 2. * 3.1415926/360)$. The `sin` function in the formula takes a radian value as input, so this formula will return a wave function, where the amount of steps run are viewed as degrees in a circle, giving us a smooth wave function as output.

Function 18: Saw
This function works much in the same way as the wave function except that the formula is $sin(stepsRun * 2. * 3.1415926/4)$. Thus we get the values 0,1,0,-1 in rapid succession. This gives us a jagged wave function.

As mentioned we are limiting the size of our neural nets. We do not however wish to fix them to a certain size, only make sure that they stay within a certain range of sizes, and most importantly does not exceed the maximum size that we choose. To ensure this we have implemented a kind of pseudo-probability based size choosing system. The choosing takes part in the grammar that creates the creature. Instead of random values (which would have one DNA string give several different creatures), we use values from the DNA instead. We first get a value between 0 and 99. This we use to determine how many layers will be in the net as shown underneath:

- 0-14 (15% Chance): 1 layer.

- 15-39 (25% Chance): 2 layers.
- 40-84 (45% Chance): 3 layers.
- 85-99 (15% Chance): 4 layers.

For each layer we then determine how many nodes are in said layer. For this we get a value from 0 to 999. We have a default value (currently set to 5) that determines the most probable number of nodes in a layer. Our wish is to ensure that the amount of nodes are close to this, but not necessary always equal to it. Thus we use the following algorithm:

1. Let NN be the number of nodes in the layer. We first examine the cases that are not the most probable case. A value "I" is set to 1, and a value `usedChance` is initialized to 0.
2. A value `dChance` depicting the chance the cases will get this round is set to 300 (30%) and are repeatedly divided by 2 I times.
3. `usedChance` is incremented with `dChance`. If our choosing value (from 0 to 999) is lower than `usedChance` the function ends and the number of nodes in the layer is set to default-I.
4. the previous step is repeated for the case: default+I.
5. "I" is incremented
6. The steps 2 through 5 are repeated until a case has been chosen or default-I equals 0.
7. If no cases are selected the most probable case (number of nodes in the layer = default) is chosen.

Thus the chance of the different cases being selected is $300/2^I$, which gets lower the further the value gets from the most probable case. We can additionally calculate the chance that the most probable case is selected, by taking the limit of the function:

$$1000 - \sum_{i=1}^{\infty}(300/2^I) * 2 = 400 \tag{1}$$

a graph of the can be found at 18 figure **??**

This means that the most probable case always will have at least 40% chance of being selected. The actual chance will be a bit higher, since all of the very improbable cases from $I = default..\infty$ that we do not allow have their chance added to the most probable case.

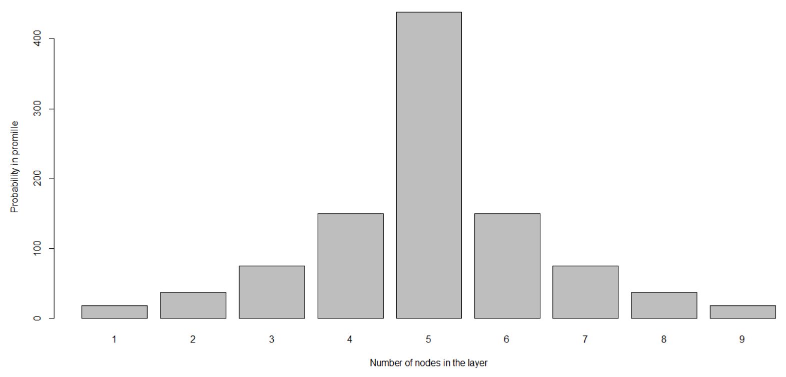


Figure 1: Probability of the different number of nodes each layer

3.3 Evolution

The first part is to evaluate and calculate the statistic for the creatures. the main purpose of this is to find the deviation, find the the elites (the top 20%) and breeders (the following 30% after the elites).

The population that does not belong to elite creatures are filled with newly bred creatures. Each new creature is bred (section 3.3.1) from an elite and a breeder. All new bred creatures are then mutated (section 3.3.2). This means that the elites stay in the population and will not have to be simulated again, where the 80% new creatures has to be simulated.

3.3.1 Breeding

The breeding is done by a two point crossover. The crossover simply selects a part from one creature, copying it and replacing a part on a different creature with it. When copying a part, if it is a body part, the part is rotated 180 degrees. This is in order to have a greater chance of creating a mirroring part(which in turn has larger chance of giving better movement). It is far from given that it will result in this, but rotating in this way has no negative effects to speak of.

The crossover differs from a normal two point crossover, by the fact the the two points are not chosen completely random. They are instead chosen from a series of predefined points, to ensure only complete parts are transferred and thereby avoiding more destructive breeding.

The predefined points are found by splitting the DNA up into its sub-parts, using the grammar. The points are stored for later use in a data structure called a `Mtree`.

3.3.2 Mutation

The mutations are simply just creating a random value and replacing another in the DNA with it. The chance of a mutation happening at any place in the DNA is 1/n, where n is the length of the DNA. A mutation can cause the DNA string to become too short by adding nodes or boxes. This is handled, when the DNA is read, by padding with zeros, to end the DNA sequence if needed. An interesting effect, of the combination of padding with zeroes and the breeding, is that the DNA grows by itself, if it is not kept in check by the evolution.

Deviation

To make it more unlikely that the evolutions stagnates at a local maxima, the deviation of the fitness values is calculated in each generation. The deviation is then used to influence the mutation rates, so the less the diversity in the results is the more it mutates to compensate for it, and the opposite.

Before calculating the deviation the fitness values are normalized. The main idea, about normalizing the values before calculating the deviation, is to make sure the deviation acts similar, no matter which fitness test it is applied to. This is needed since different test can give results in very different scales.

3.3.3 Fitness Tests

The main purpose of the fitness test is to encourage a certain behaviour from the creatures. The different fitness tests will give us a numerical evaluation of the creature's performance. The values of the different fitness tests are not necessarily on the same scale. This means that one fitness test might return a much larger number than another. Since we are evaluating all creatures in the same way, while doing a simulation, this will not be an issue. The following is an explanation of the different fitness tests we have implemented.

Walking

The desired effect is a continuous movement away from the starting point. The general approach is just to measure the creatures' distance from the center when the simulation ends. We used center of mass as its position, to make it more favourable to move the whole body instead of just parts.

A more advanced approach to the problem (`Iterative move`) also takes the time into consideration, since distance gained late in the process is more often caused by continuous movement, than distance gained early, because of falls and events like that appears early. This will also favour creatures that move a certain distance continuous but slow compared to one doing the same distance fasts and the stopping.

Jumping

We want the creature to jump up into the air. Jumping could be defined in several ways, but our definition of a high jump is close to the definition used in athletics. We define the height of a jump as the distance between the lowest point of the creature and the surface of the ground. That way a creature is not allowed to touch the ground. and does not gain an advantage simply by being larger.

Dwarf slaying

To encourage more complex creatures this test kills creatures with fewer than 3 parts. Otherwise the test works just as the `Iterative move` test. We attempt in this way to mitigate the fact that it is easier to optimize a creature of lower complexity.

Another attempt at this (Box loving), instead try the "nice guy" approach, by scaling the fitness value of an `Iterative move` test with the amount of parts the creature consists of.

Fat loving

To encourage creatures of larger physical size this test scales the fitness values of an `Iterative move` test by the creatures weight. We attempt in this way to get creatures with a more logical volume, if a creature were to have inner organs and the like.

Combination

By using several tests at the same time, we can attempt to get creatures that fulfil not only one objective. For instance using the jump test, together with a movement test, should logicically result in a creature moving forward in small jumps.

3.4 Physics

The main idea of the physics part is to regulate and control the physics engine. The physics is implemented using `Bullet Physics`. It is based on `Bullets` `basicDemo`. By basing it on the demo a lot of the basic set up is achieved without too much rewriting.

3.4.1 Bullet Limitations

Using `Bullet` has its disadvantage's, since it is a open source project with the problems often following these kind of projects. These include issues such as lacking documentation, inconsistency in the implementation, incomplete implementations and lack of robustness.

The main problem with this is that `Bullet` has a tendency to undocumented behaviour, especially when its supplied with a large range of random numbers as with the DNA. Therefore it has tendencies to generate energy out of nothing during deep penetrations, and other non natural events.

The implementation of the physics thus has to be able to handle all such events. This is done by limiting all parameters for functions in `Bullet`. This is not enough in itself though and it is therefore necessary to make short test simulations where a creature is evaluated on whether or not it obey the laws of physics (section 3.5.1 Tests and Checks).

3.4.2 Stepping the simulation

To run a simulation it is necessary to continuously calculate the physics and move the object accordingly. This is done by `Bullet`s function `stepSimulation` which takes as parameter the step size, such that running the simulation can be done in small steps. To get a consistent result each time a simulation is run, it is necessary for the steps to be of a fixed size. The size of these steps therefore has to be decided. To reach the a precision high enough for our purpose and to avoid stuttering in the visual part (both due to performance and large instant movement), the simulation is stepped with 1/1000 each millisecond. The steps size also effects the likelihood of deep penetrations happening in bullet. So if the precision is lower, strange behaviour will appear in the simulation.

3.4.3 Boxes

Everything in the world is build out of boxes. Since a world of single boxes is rather boring, the boxes can be can be connected to each other through joints, to make more complex figures. Not only are the shapes in the world limited to that of boxes, but the moving/dynamic boxes has also been limited in size, since they have to be within bullets specifications 0.05 to 10 for moving objects. Each box is designated a weight proportional to its volume.

To make a wider array of movements possible the boxes will not collide with each other. The only exception is with the box designated as the ground. Even though there are no collisions between boxes, no boxes are not allowed to start in a simulation where they are overlapping from the beginning. This is to avoid the creatures from breaking the rules of physics.

3.4.4 Joints

The joints are implemented using `Bullets` `btGeneric6DofConstraint` joint. This constraint has 3 degrees of freedom in translation(stretch) and 3 degrees of freedom in rotation. We limits the degrees of freedom to match that of only rotation, since it fits normal creature abilities. The rotation is further limited to be in the range of $\pm 85°$in each direction. The full 90°is not possible due to the specifications of the constraint. The degrees of freedom is symmetrical. If degrees of freedom in the y axis is max +40°the the min in the y axis is -40°

When creating a joint it is not only necessary to create the joint itself, but also to relocate and rotate the boxes the joints connects to, since else they will fly towards each other as soon the simulation starts. To allow more complex figures with a series of joints it is only one of the boxes which is moved. For this concept to work one of the boxes must not be connected to anything. That could create a lot of problems, with moving everything that box is connected to, and the next box, and so on. A strict hierarchical build up is needed which fits perfectly with the way the grammar is build.

Strength

The strength of both tensile[3] and muscle[4] of any given joint is linear proportional to the cross sectional area between the two boxes. The cross sectional area is calculated as the smallest contact area achieved between the boxes when they are rotated around the joint. Tensiles are 5% stronger than muscle to avoid unintended self mutilation. When the the force applied to the joint exceeds the tensile strength the joint will automatically be disabled by bullet which results in the part falling of.

3.4.5 Effectors

The effectors are the parts that controls the forces applied to the joints. The effectors are based on the motors already attached to the joints. There are six motors, one for each degree of freedom. Only the three are used, since the joint should not be able to extend.

The value the motor receives is simply equivalent with the force it exerts at the next simulation step. The max force, exerted by the three motors combined, however is limited to that of the muscle strength in the joint.

3.4.6 Sensors

A sensor is a value reflecting a part of what is happening in the simulation. Thus the sensors are basically just a series of values updated after each simulation step, depending of the state of the creature and simulation.

There are two kind of sensors: pressure and angle sensors.

[3]The amount of force needed before the joint disables
[4]Force a joint can apply to its effectors

Pressure Sensor

A pressure sensor is a sensor attached to a box. The sensor registers collision between the box and any other object. In our cases, this is only collisions with the ground (section 3.4.3). The sensor is set with a special flag in the box it belongs to. The pressure values is found by checking all the collision/contacts that occurred during a simulation step for the pressure flag. The value is either 1 for contact or 0 for none.

Angle Sensor

The angle sensor shows the position of the joint its attached to. There are three angle sensors for each joint, since there are 3 degrees of freedom. The angle sensor's values are retrieved from the constraints, one value for each degree of freedom. The three angle sensors will be created whenever a joint is created.

3.4.7 Calculation of the size

Calculation of a creature's height, and the auxiliary points needed to calculate it, is done with vector calculations. We run through all the blocks the creature consist of and find the highest and lowest points of them. Then we can just compare the height values of them to find the highest and lowest point of the creature. With these we can trivially find the height of the creature. To find these points the 3 half sizes of the box is rotated by the current rotation of the box. The size of each of the resulting vectors then have their height parameter added together, to give us the extra height due to the size of the box. Finally this size is added to the box position to get the maximum height and subtracted to get the minimum height.

3.4.8 Calculation of the position

All positions used to evaluate a creature, are based on the creature's center of mass. The calculation of the center of mass is trivial since we have both the center position, and the mass, for all parts of a creature. The creature parts are all made of a single homogeneous material, which further simplifies the calculation.

3.5 Framework

The main point of our program is to run a batch of generations using a single creature as a starting point. This section explains the general outline and flow of the program. A flowchart of the program can be viewed in figure 11.

When beginning a new batch of generations the starting creature DNA are copied to the population, and the rest of the population is filled with slightly mutated versions of it. The chosen number of generations are then run, and for each generation the creatures in the population of that generation are simulated. At the end of a generation the evolution algorithm is run, creating the new population for the next generation. So far this still roughly follows the algorithm from the design section 2.4. For performance however, creatures that were chosen as elites in the evolution step are not simulated, since the DNA of those has not been changed.

Creatures are not actually displayed during the processing of a batch, since rendering takes dreadfully long, and there's no way of predicting what's worth looking at in real time.

3.5.1 The Simulation

The simulation of a creature is in many ways the cornerstone of our program, and as such we wish to explain properly, how it works chronologically. The first part of the simulation is to initialise everything. A world is created in the physics engine, and the creature's DNA is read to insert parts, and to create constraints that join them together.

Next it is checked whether or not the creature is legal. Several different tests and checks are needed in order to ensure that the simulation has the right starting point, and that the creature does not cheat. The tests and checks are described in the Tests and Checks paragraph.

If something is amiss, the creature gets a predefined negative fitness value (-999999), and no further simulation is run. Otherwise the main simulation loop is started, which in normal cases will continue for 10000 steps, each representing a single millisecond. In each step the following actions are taken.

- All neural net nodes are updated, beginning with the first layer of the main net, and going hierarchically through the layers, ending with the last layer of the last subnet[5].
- Muscle forces are added to the motors of the joints. The saved effector indexes of a joint determines where in its subnet the force value should be taken from. These values must be set each step since `Bullet` resets them after they are used.
- The fitness value is calculated using the appropriate fitness test.
- A simulation step of 1/1000 (1 milliseconds in our program) is run in the physics engine, doing collision checking and applying forces where it is necessary.
- The sensor values are calculated, saving the values for use in the neural net the next step.

Finally as the last part of the simulation of a creature, the "Dismemberment" check is run, as is explained in the Tests and Checks paragraph.

[5]The subnets are not really ordered, the "last subnet" thus referring to the subnet created last, which has the largest index

Tests and Checks

Ground fix Because the creature parts are moved around depending on their connections, some parts can end up inside the ground at the beginning of the simulation. This results in the creature flying upwards like a spring, as the collision handler solves the problem. To avoid this the lowest point of the creature is calculated, and the ground is moved down to the same vertical level as that.

Relaxation The creature evolution will utilize any opportunity to cheat the physics engine. Thus it is necessary to "relax" the creatures before any simulation. This keeps creatures from building up a lot of potential energy and gain fitness points use it in a fall. It also deters creatures from moving without the use of effectors. The creature is simulated on a frictionless surface with no effectors enabled, until it reaches a neutral state, from where the simulation can begin. Neutral state is here defined as a state, where the vertical position of the center of mass has not moved for a certain amount of simulation steps. The actual amount has been found empirically, and may not catch all creatures that move unlawfully. Because this can cause infinite loops, we have to restrict the time that the relaxation can occur. If this time is exceeded we discard the creature. While this can in rare cases remove working creatures, the alternative (simulation livelock) is unacceptable.

Height test Even with the relaxation the creatures have a tendency to end up as giant towers that use joint motors to fall when the simulation starts. This test ensures that no creature is allowed a height greater than 15 meters.

Internal Collision To avoid overlapping boxes as described in the Boxes section. We test for each box if it collides with any of the other (excluding the ground). Any collision means overlapping boxes, and thus an illegal creature. While this is indeed a brute force approach, it can be defended, since the amount of boxes in a simulation is at most 15, and most often 2 or 3.

Dismemberment This is the only check that is run after the simulation. The joint of the creatures are all checked to make sure they are not disabled. If one or more of them does not check out, it means that the part has fallen off, and the creature should be discarded.

3.6 Parallelism

The program is parallelized in two ways, by threads and IPC. Threads are used when starting a batch of simulations. The batch is separated from the UI, by giving the batch a new thread. The purpose of this is to ensure a responsive UI, even though the batch of simulations are hogging as much process time as it can get.

When the batch thread is started it is given all relevant user inputs (no. population, no. generation, ancestor, and fitness test). All further communication is done by shared variables, since the only interesting communication is a one way communication from server to GUI with the results. The server simply writes its result to a destination given by the GUI.

For further paralellization, IPC is used.

3.6.1 Inter Process Communication

All the simulations are independent of each other (if you do not look at bullet), and can therefore be run independently. The best way to do this is by dividing it out, so each core have one thread. A single thread per core should be optimal since there will not be any relevant idle time, where it would be an advantage to switch threads. This means that the paralellization is limited by the number of cores and there is therefore no need to parallelize more of the simulation.

There are not much knowledge of the simulations ("random" generated DNA) and they are therefore evenly divided between the cores. A problem with just evenly dividing the work, is that the simulations do not necessarily take the same time. This can be avoid by distributing smaller chunks and only giving new chucks to idle clients. This on the other hand requires more communication between the server and its clients. Since the work is randomly generated it would most of the time theoretically be distributed evenly, if the population is large enough. It would most of the time result in more or less the same performance, with the guided approach often a bit slower because of the overhead. Only on the rare occasion the the work is distributed very unevenly the guided approach will have an advantage.

Client-Server

The batch of simulations is divided into two programs, one program, the "server", running the serial code, and one program, the client, running the parallel part. Thus what normally would have been a thread is simply just a program on its own (the client), and when more threads are needed a new client is started. That way the number of active clients equals the number of threads.

The server part distributes the work and handles the evolution. Only the simulations are run on the clients, which then just sends the results back to the server part.

The communication between these parts are done in several ways. The first communication happens when the server starts the client. The client receives command line arguments, which are used to inform the client on how it should start a pipe connection to the server, and which fitness test should be used. This pipe is then used for sending commands and acknowledges, to tell the client when to start/stop simulating and to ensure synchronization. The raw data (creatures) needed for the simulations are sent through a file in binary format.

3.7 GUI

As mentioned in the design section, the GUI is implemented using the `WIN32` API. The interface is comprised of a single main window that is only created once, and several subwindows and dialogues. The dialogues are created and deleted multiple times during runtime.

The simple windows, the popup dialogues were created by a graphical interface designer tool, and saved as a resource file. The relevant part of the resource is simply loaded each time a dialogue should be shown. This very much simplifies the creation of dialogues.

The main window however, needed to be customised more precisely for our needs and had to be created by hand programmatically. It consists of:

- A list of creatures. The user can select a creature here, and can delete, and rename it as well.
- A parameter window. Values for number of generations to run, and population size, as well as a run, and a reset simulation button are located here.
- A menu. Users can create new randomized creatures here, capture a video of the selected creature, save the creatures or exit the program.
- The `OpenGL` rendering window. Here a simulation of the currently selected creature is shown. It can be reset using the reset button in the parameter window. The rendering itself is done by OpenGL which renders into a frame buffer. We switch the buffer into this window, which we are able to, since it has been initialised in a special way.

Once the start button has been pressed, a new batch of generations will be run, using the currently selected creature as a starting point, and with the parameters set in the parameter window.

The `WIN32` API functions by the principle that each window has its own controller that handles messages sent to that window. When a user inputs something a message is send announcing the event. The main window controller distinguish between which of its subwindows has focus, and can use it to decide how it should react to a message. Keys pressed while the rendering window has focus, are just passed on to the simulation, which handles it the way the `Bullet` framework normally handles inputs. Thus a user is still able to rotate the camera around creatures, zoom etc.

The message loop

The complete GUI runs in a single main thread, and everything is run in serial. The thread will continue as long as a quitting command has not been issued. Normally this thread simply keeps checking for messages and handling them, however in this program the thread takes turns between handling messages and simulating/rendering the simulation in the rendering window. This approach is viable since the program have few user inputs, and the handling these are rather short. Thus the simulation should not have any visible delays because of the message handling. The interface will be slowed down a bit, however it has not given any trouble so far, since the interface is very lightweight, and the simulation steps are not overly demanding.

3.8 Rendering

As mentioned in the design section (2.6) the rendering engine in `Bullet`'s demo framework has been used but modified. The modifications to the `Bullet` rendering engine, were implemented by changing the `Bullet OpenGL` plug-in, that is responsible for the rendering of the `Bullet` demos.

3.8.1 Colours

The rendering engine of `Bullet` uses `GL_COLOR_MATERIAL` to set the base colour of an object. This color will be multiplied with the texture colour, giving the illusion that the object is being lighted by a light with that color. While `Bullet` uses it to set the debug colours and the chequered pattern (figure 9, page 44), this is changed simply by replace it with a white base, so the textures correctly can be seen correctly. When drawing the shadows they are set to a dark grey, which lets the shadows be drawn, not as simple black areas, but as darkened areas of the textures underneath them.

3.8.2 Textures

The textures are loaded from the hard disc and bound to a type of objects. When an object is drawn the texture of the object's type is used. there are only ever drawn cubes since they are the only shape our creatures are allowed to use. The cubes are drawn one face at a time, split into two triangles (Quads might also have been used, but two triangles were what `Bullet` had implemented). Each texture coordinate must be set directly before sending its corresponding vector to the graphics card. This means that the texture coordinate of a vector in a triangle must be set corresponding to the position of the vector in the face of the cube. This however is impossible to calculate with that information, so we instead inverse the basic texture coordinates every time we draw a triangle. The order in which to draw the vectors had to be found empirically. The texture coordinates are also scaled with the size of the face, repeating the texture if the face is larger than the texture image, and pinching the texture together if the face is smaller.

3.8.3 Skybox

With grass texturing and scaly creatures, the scene were looking better. The grey background however, no matter how classic and timeless it is, kind of ruined the picture. With that in mind a skybox was acquired[6]. The skybox is drawn after the rest of the objects (and shadows). It is made up of six large faces enveloping the entire scene, and pointing inwards. Each one of them is textured with one of the skybox images. Again most of the texture/vertex coordinate pairs were found empirically, since it quite simply were faster than calculating the positions. A comparison of before and after the skybox can be seen in figure 10 page 45.

[6]`http://www.3delyvisions.com/skf1.htm`

3.8.4 Frame Skipping

The resolution of the simulation is 1 ms, and generating and showing 1000 images each second is not a particularly good idea for performance reasons. Since the human eye can see a fluent film in only 24 images a second, 1000 fps also is quite the overkill. The obvious solution to this is only rendering 1 frame out of x steps in the simulation. 24 fps roughly translates to rendering 1 out of 44 steps. In films they use several techniques to make 24 fps seem fluent such as motion blur. We do not however have these at our disposal, so we have to compensate instead with a higher frame rate.

Graphic Card

The graphic card will slow the simulation down if `V-sync` is activated on it. This is the case since `V-Sync` ensures that the frame rate does not exceed the refresh rate of the screen, by blocking all writing to the frame buffer. It presents a problem since our logic is linked to the frame rate and a delay of the rendering also will delay the logic behind. To minimize the effects of this the frame rate is locked so it cannot exceed 62.5 fps (rendering 1 out of each 16 simulation steps). This helps since the refresh rate of a normal screen is 60 Hz. In the end it means that only three frames will be delayed each second, which is not noticeable.

A possible solution to this problem could be to run the rendering and logic in a separate thread each. This way the graphic card will only suspend the rendering thread. Another solution is simply to manually make an exception in the graphic card settings, that disables `V-sync` for the particular program.

3.8.5 Video Capture

A video capture feature has been implemented using Microsoft's VFW (Video for Windows) library. This makes it possible to save a video of a simulation for any given creature. The length of the captured video has been defined to 10 seconds since that is the length of a simulation. The Library will furthermore check which compression codecs are already installed on the pc, and gives the user the opportunity to choose between them. The user is also allowed to configure the parameters used in the codec if any exists.

The frame rate of the video is set to match the optimal frame rate of our rendering (63 fps). The images used for the video is taken from the GUI rendering, and supplied to the library. Since the rendering does not skip frames with dynamic intervals, there will always be the same amount of frames for the 10 second videos. Thus even if the rendering of the frames are slowed by the extra work of saving the frame to a video file, the resulting video will have the correct frame rate.

Testing 4

We have use two kind of testing, black box and white box, to ensure the robustness of the program. The reason we chose to test like this is that we have many randomized parts of the program. Thus we have to take into account the random values while we test.

Our white box testing consisted of pausing the program at several spots, and examining if the values checked out. The calculation of the box sizes and height are good examples of functions that were better off white box tested. The debug breakpoints of `Visual Studio` were usually the method used for pausing the program.

To cover the areas not suited for white box testing, we used black box testing. Mostly this involved just evolving some creatures, and see if they acted according to physics. Since the creature evolution has a tendency to usurp every physic engine fault, this kind of testing tended to expose such faults like Karl Sims also explained in his report [2]. Other ways of black box testing included slowing down the rendering of the result to a crawl, and checking if the boxes were placed, and rotated, correctly at startup.

Memory leak detection

Memory leaks have been a serious issue at some stages of the project. The constant creation and deletion of simulation worlds, have meant that even small memory leaks could crash the program eventually. Apart from simply manually watching when the memory consumption rose and fell, we have had need of an automated way of telling us if, and preferably where, a memory leak was present.

For this purpose we have implemented the `Visual Leak Detector` library [3]. By including this library in our project, the program will automatically generate a report over any present memory leaks, and a call stack of where they originated. This report is of course not perfect. Some call stacks are unreadable, and we even had a very severe memory leak that were not found. Even so it has been a most helpful tool for testing against memory leaks.

Visual Studio Performance Analysis

To find the areas, where the code was lacking in terms of performance, we have used the `Visual Studio Performance Analysis` tool, to measure the performance of the code. The tool samples the code while it is running and creates a report which shows where the time is spent in the code.

When using this tool we have focused on the areas, where the performance actually has an effect. More precisely the part of the code where the batch of simulations is run.

Results 5

5.1 Performance

We have done a series of test to test the performance of the program. These test will tell the speed of the program but it is worth keeping in mind that it does not tell how effective the algorithm is.

5.1.1 Speed

Program	CPU	starting point	time
Karl Sims [2]	CM-5	?	around 3 hours
GALAPACOS [4]	intel core i7 720-QM 1,6 GHz	?	6.43
WWD	intel core i7 3610-QM 2,3 GHz	super evolved thingy	4.06
WWD	intel core i7 3610-QM 2,3 GHz	block	2.03

The tests are run with population: 300, generations: 100 in 20 sec

Table 1: Run time

The first test we have done is a repetition of the performance test the GALAPACOS group has done. It is a simple brute force test comparing the execution times of the program. We have chosen to compare our results with that of the GALAPACOS results for the i7 processor since it is the one closest to our own hardware specs (a newer generation of i7).

Since the starting point is unknown for Karl Sims and GALAPACOS, we tested WWD both with the most basic starting point a Block (appendix A.II), and with a more complex which moves during the whole simulation (Super evolved thingy, appendix A.IV).

The test shows that the speeds varies much depending on the starting point. Also the effectiveness of the evolution algorithm may have an influence on the results, since more effective results will result in more advanced creatures. These generate a higher work load, because of the more movement and number of boxes.

The comparison is further hampered by the fact that the hardware is not equivalent. The most major differences between the two i7 can be seen in table 2 on page 31

cpu	i7 3610-QM [5]	i7 720-QM [5]
clock speed	2,3 GHz	1,6 Ghz
system bus	5 GT/s	2.5 GT/s

Table 2: I7 Comparison

Even when we take the difference in hardware into consideration, it is clear that our code runs faster than the GALAPACOS. It is not possible to say anything meaningful about the speed of Karl Sims' code since it runs on a completely different architecture.

To explain why our code is faster, it is interesting to look at Amdahls law. This can show if the speed is caused by a higher percentage parallelization, or if the serial parts are just faster. To use Amdahls law we have to test the scaling of the code.

Threads

Estimating the scaling of the code is done by taking time on the program running with various numbers of threads, and using Amdahls law on the results. To make the tests more precise, by minimizing the effect of the start up time, a fair amount of computations is needed, so the start up time becomes negligible. Another way used to make the test more precise is choosing a population size where the population is divisible by the number of threads, which means that the work always will be evenly distributed, no matter the number of threads.

no. Threads	1	2	4	8
time 1	451	243	162	105
time 2	429	268	159	156
time 3	449	254	153	142
time 4	407	226	181	113
time 5	416	265	221	140
average time	430.4	251.2	175,2	131,2
speed up	1x	1.71x	2.46x	3.28x
seriel part	100%	17.0%	20.9%	20.6%
max speed up	5.13x (19.5% serial)			

250 generations, 64 population, 10 sec, starting point Defdino appendix A.I

Table 3: Thread speed up

Amdahls law

$$s(n) = \frac{T(1)}{T(n)} = \frac{1}{B + \frac{1-b}{n}}$$

n= no. of threads

s=speed up

B=fraction of the code which are binary

Figure 2: Amdahls law

The results show a slightly larger serial part for a higher number of threads. An explanation to this would be that with a larger number of threads the clients are more sensitive to delays (relax, high work load etc.) since there are fewer worlds for each thread. Also the uncertainty in the measurements could be the cause.

Interestingly enough our test shows that our code is faster than the GALAPACOS code even though their code scales better. This means, according to Amdahls law, that the GALAPACOS code will be faster when using more than a certain number of threads. This is under the assumption that both programs keep scaling in the same manner.

What this means is that the reason for our speed lies in the fact that our code is faster compared to theirs, if both are run completely in serial. The reason for this can be credited to the fact that the time it takes to simulate a single world is faster than theirs. That our evolution part should be some of the cause is not likely, because our algorithm is more complex, and therefore most likely will take longer time.

The serial part that slows our code down mainly consist of the evolution and the IPC communication. That the evolution takes time is not necessary a bad thing if it means that the creatures are evolving more each generation. Also, parts of the evolution needs to be serial. The IPC communication on the other hand could be done more in parallel, and the overhead on transfer between server end clients is to blame.

5.1.2 The effect of population and generations

The effect various population and generation sizes have on the performance and on the scaling.

x	gen=x pop=100	fitness	gen=100 pop=x	fitness
3000	31:30	4.585	11:31	1.8251
2000	21:19	10.0494	08:14	0.257
1000	09:12	1.0081	05.31	0.3112

Tested with the starting point `Block`, and with the fitness test `Iterative move`.

Table 4: Different population and generation sizes

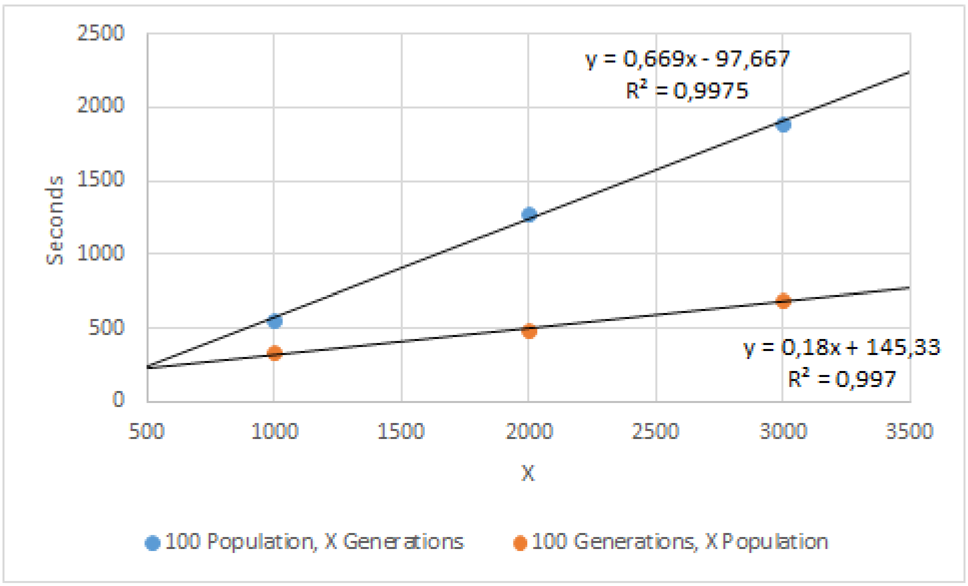


Figure 3: Population vs. Generation

Even with the fitness value taken into account, there is a clear difference in speed, even though they are running the same amount of simulations. The slower speed can most likely be blamed on the more frequent IPC communication where some overhead is known to exist.

5.2 Simulation length

The simulation length affects our solution, for an example the creature Super evolved thingy shows this clearly. It moves more or less away from the center in a straight line, until the exact time when the fitness test ends. Then it turns 180 °and moves back where it came from. This example is not unique in terms of the creatures behaving oddly as soon as the time runs out.

The only fitness test which is not effected by this event is the jump test, since the creatures have a tendency to make their jump early in the simulation, and just waste the rest of the time.

Another way of looking at simulation length could be to look at the amount of simulations we have to run in order to get interesting results. This is of course a very abstract notion, and much is decided at random. We tried to run a few tests while observing this however, and we found an interesting pattern.

generations	50	100	150	200	250	300
with stops	0.0004		0.5127	1.7098	3.2329	3.3071
without stops	0.2917					

population size 300, fitness test Jump

Table 5: Comparison between short generation intervals and large ones

The above table lists the fitness value of a creature after the shown amount of generations run. The test shows a tendency in our evolution, that we have also experienced ourselves while using the program. By running many small batches of generations, results are achieved faster compared to running a single large batch.

Large leaps in the fitness value often happen in the first 10-20 generations, if the the starting creature have already achieved some kind of movement. When a large leap happens the rest of the population has to catch up to the best, which will take several generations. By stopping and starting the batch of generation the new population would be created solely from the best creature. This means that if the best creature recently have performed a large leap in fitness value, the rest of the population will catch up instantaneously. This especially has a great effect when the leap goes from fitness value 0 to a positive value. On the other hand this also locks the evolution on a specific path. Then again if we just continued the batch of generations, we would not see the lost evolutionary paths anyway.

5.3 Fitness tests

The choice of a fitness test affects the outcome of a batch of generations a lot. This section describes the creature evolution tendencies, when using different fitness tests.

5.3.1 Moving

In the first couple of generations the creatures tend to develop falling/rolling techniques, since its requires few changes from most starting points to reach a result. Rolling/falling is mostly an evolutionary dead end, since it encourage large bodies (to gain height) which gives a great mass. The creatures mostly use a small appendage part to tip itself over, starting the fall.

If the simulation is not quickly stopped[7] however, the falling creatures will in most cases be overtaken by creatures actually moving, since the falling ones are limited by the height test.

Iterative move VS. Move

The creatures seem to be more inclined to evolve continuous movement instead of creatures doing a single forward jump (or push). The falling creatures are still present in some batches however. With the `Iterative move` test, the falling creatures tend to optimize their falls so they are timed well with the ending of the simulation. This way they gain the maximum amount of points for their falls, which can make it harder for the creatures actually moving to overtake them. While this is indeed a negative thing, the higher probability of continuous movement outweighs the problem. Thus we have used the `Iterative move` as a basis the conditioned fitness tests (`Dwarfslayer`, `Fat loving`, `Box loving`).

Conditioned tests

The conditioned fitness tests designed for nurturing certain attributes, mostly seems to fulfil their objectives when run from the default starting point (Defdino).

Fat lover This fitness test keeps the boxes large, but in most cases results in stranded gigantic creatures, that simply wag a tail-like appendage to gain them a few points. It is however good at modifying existing moving creatures, since the fitness value bonus for a creatures mass is quite large, and thus have a good chance of giving a winning creature, even though it moves slower. This sounds like a bad thing, but is in fact great for escaping out of evolutionary dead ends.

Dwarfslayer While this fitness test indeed does keep creatures at 3 boxes or more, the case of a 3-box creature is by far the most common. Since creatures are not rewarded for being any more complex than that, they simply attempt to optimize with 3 boxes instead.

Box lover This fitness test actually do its job. Creatures have a tendency for more boxes. It is unlikely that we ever get more than 4 boxes though, and despite the effort the test seems to end up with 2-boxed creatures as well. This is because if a creature of 2 boxes get to fill most of the population, it is very hard to evolve more boxes without ruining the movement.

[7]Stopping the simulation will in many cases ensure that the evolution gets stuck with the falling creature

5.3.2 Jump

The creatures attempting to jump will in most cases do so by moving a part upwards at great speed. This generates a small acceleration as the part is stopped at the limit. The creatures also tend to rotate when "jumping" to get extra points for getting their lower parts into the air. We have also had long thin creatures, that rotated around themselves, instead of jumping, in order to heave their long leg-like parts into the air. An example of a jump evolution chain can be viewed in the appendix **??**.

5.3.3 Multiple fitness test

Testing in sequence

By sequence testing we mean running a batch with one fitness test and then running a new batch with another fitness test, using the previous result as starting point. Doing this can lead to interesting results, and using the conditioned tests one can attempt to guide the evolution according to aesthetic sense. We found that if the scheme developed in the first test used is liable, the creature will stick to it. For example jumpers will continue to jump when movement tested. They will in most cases be moving forward in small jumps, using the same jumping scheme as before (this is of course not the case with "jumpers" that just rotate). Moving creatures on the other hand, when made to jump, will likely start again from scratch if they are not jumping forward as a movement scheme already. They will however quickly evolve jumping capabilities, compared to the default starting point. We consider the already existing connected neural nets in the movers as the cause of this.

Combi

Combined testing simultaneously of two fitness tests, has so far only been attempted with the simple `Movement` test and the `Jump` test. The creatures tend to only evolve movement capabilities, since those are much easier to evolve. The jumps the creature make if any even exists, are almost impossible to see, and the creatures rarely evolve any jumping at all even when the batch is quite extensive. This is because any jumping in many cases ruins the movement scheme, and thus leaves the creature with a lower fitness value. this can also be seen when we tried to apply the Combi test on Super evolved thingy but nothing changed at all, the creature kept doing exactly the same.

5.4 Turtle evolution chain

One of our more interesting results is the evolution of the turtle-like[8] creature. It is a great example of how a chain of evolutions can progress.

[8]or seal-like apparently, depending on the observer

The first step of the evolution were a long fallen pole-like creature (see figure 6), with a small box appendage at the far end. It moved by vibrating the box from side to side, which rotated the pole part around its other side. While this creature moved in circles, it moved continuously and not exactly slow, which gave it a decent fitness value. It was evolved with the `Iterative move` test.

Evolving from the pole like creature, the result were a creature that used the corners of its two parts to clumsily push itself forward. While the movement were in no way perfect, it was quite a lot faster than the vibrating movement of the pole, and it moved in an almost straight line. This creature were further evolved and arrived at a evolutionary dead end (see figure 7, when its movement were optimised. By that point it could push itself determinedly forward at great speed, and the movement only became erratic at the end of the simulation. Both these evolutionary steps used the `Iterative move` test.

Wishing to use this result for other things, the `Fat loving` fitness test were used to evolve the creature further. This resulted in a turtle/seal like creature that used paddling motions to move itself forward (see figure 8). While in no way near as fast as the previous creature, this result is far more usable for further evolution, and is quite interesting on its own as well.

Future Work 6

Future work for this project would mainly be around rewriting the genotype, so it is easier and faster to manipulate. A possibility could be to do a more object oriented approach. This change would also make it possible to implement a more complex breeding/evolution algorithm, which for example inspired by nature could be more focused on symmetric creatures.

Another area which could use some work would be the scalability of the program, both in terms of more parallelization and in terms of handling different population sizes better. To achieve this we would have to go into making the IPC communication faster by for an example using shared memory. Also more of the serial parts of the program (evolution) could be parallelized.

A way to make these changes and others in the future easier, could be by separating the program into the model-view-control pattern. Besides a better structure and maintainability of the program, this would also be one way to go for fixing the `V-sync` problem (section 3.8.4). In the more extreme end of changes that would make things easier, we could change to a different physics engine. This would hopefully remove a series of obstacles that slows the development process down (section 3.4.1). Changing the physics engine could also speed the program up since a series of checks could be removed. The checks removed would be for events that should not even be possible e.g. energy out of nowhere.

Conclusion 7

We have successfully implemented a system that evolves virtual creatures. The evolved creatures, while not exactly resembling dinosaurs, have shown several interesting methods of propulsion. The creatures have furthermore shown varying results, depending on the chosen fitness test. This has enabled us to guide the evolution to the point where we are able to successfully cause certain features to appear. Multiple short batches of generations have shown to give a more effective evolution than long running large batches.

The creatures are simulated in a 3D-environment created with the help of the `Bullet` physics engine, which simulates such forces as gravity, impact and friction.

When evolved a creature can be saved as a video file, showing a simulation of the creature. It can also be saved as a data file, that can be loaded and used as a starting point for a new batch of generations, or rendered in the main window of the program.

Our simulation is being run in parallel using interprocess communication methods. This has given us a speed-up of 3.28 times the normal speed, when using 8 cores. Furthermore we have found that a great amount of generations, is more time consuming than a similar amount of creatures per population.

References I

[1] Demetri Terzopoulos. Artificial life for computer graphics. 8 1999. Communications of the ACM.

[2] Karl Sims. Evolving virtual creatures. Technical report, 1994. `http://www.karlsims.com/papers/siggraph94.pdf`.

[3] Dan Moulding and the VLD group. Visual leak detector. `https://vld.codeplex.com/`.

[4] Peter Ølsted and Bejamin Ma. Galapacos. Technical report, 5 2012. `http://www.kindsoftware.com/documents/reports/OlstedMa12.pdf`.

[5] Intel. Intel. `http://ark.intel.com/compare/64899,43122`.

[6] Karl Sims. Evolving 3d morphology and behavior by competition. Technical report, 7 1994. `http://karlsims.com/papers/alife94.pdf`.

Appendix II

A Creatures

A.I Defdino

The "dinosaur" like creature, that dosnt move and have no neural net.

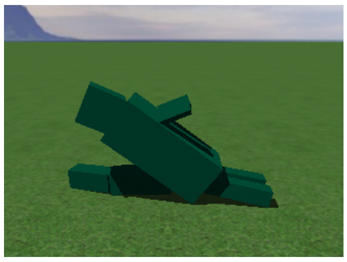


Figure 4: defdino

A.II Block

Simple block that dosnt move and have no neural net.

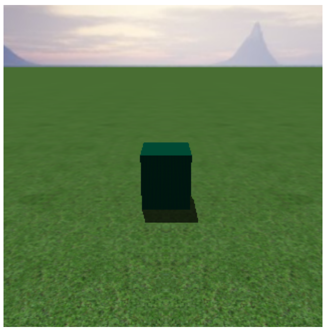


Figure 5: Block

A.III Pole

Moves by vibrating the small part outermost on the edge of the pole. Has a tendency to make circular movements

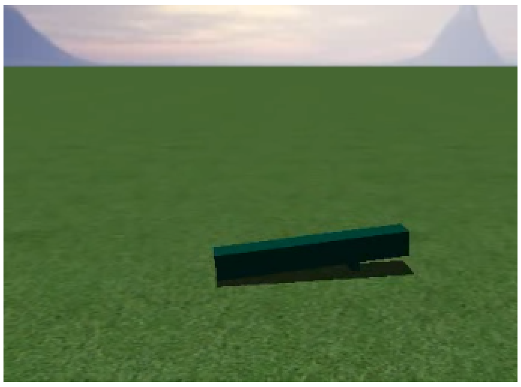


Figure 6: Pole

A.IV Super evolved thingy

Moves forward by using each block as a legs and paddling forward. The movement is similar to a run where the body does not touch the ground all the time, by lifting slightly from the ground.

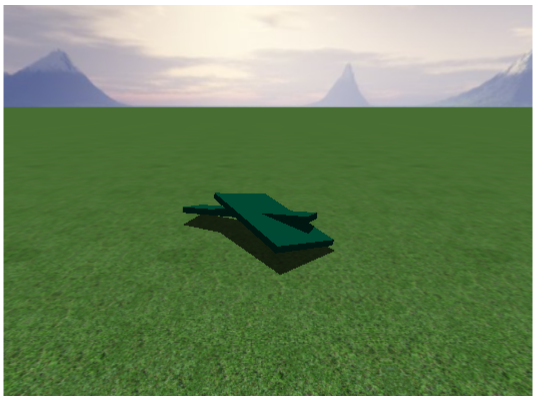


Figure 7: Super evolved thingy

A.V Turtle

Paddles slowly forward with the flat lower part.

B Grammar

A version of our grammar that is closer to the final result we implemented:

Grammar:

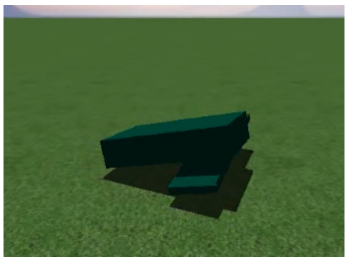


Figure 8: Turtle

S	::= Bi B NN
BI	::= $h\ w\ d$ Se
B	::= $count$ J^{count}
NN	::= cv NNL$^{1..4}$
Se	::= $type$
J	::= NN E $screw$ PRE POST DOF Bi B
NNL	::= lcv Ni$^{1..([NETSIZE]*2-1)}$
E	::= $i1\ i2\ i3$
PRE	::= $x\ y\ side$
POST	::= $x\ y\ side$
DOF	::= $x\ y\ z$
Ni	::= $ncv\ value$
	| $ncv\ f\ in1\ w1$
	| $ncv\ f\ in1\ in2\ w1\ w2$
	| $ncv\ f\ in1\ in2\ in3\ w1\ w2\ w3$

Description

S = The start of the grammar

BI = Contents of a box
- h = Height of the box
- w = Width of the box
- d = Depth of the box

B = Box (a single part of the creature)
- count = Amount of boxes attached to this box

NN = Neural net
- cv = Chosing value determining how many layers are in the neural net

Se = Sensor
- type = Type of sensor

J = Joint
- screw = Rotation of the joint

NNL = Single layer in a neural net
- lcv = Choosing value determining how many nodes are in this layer

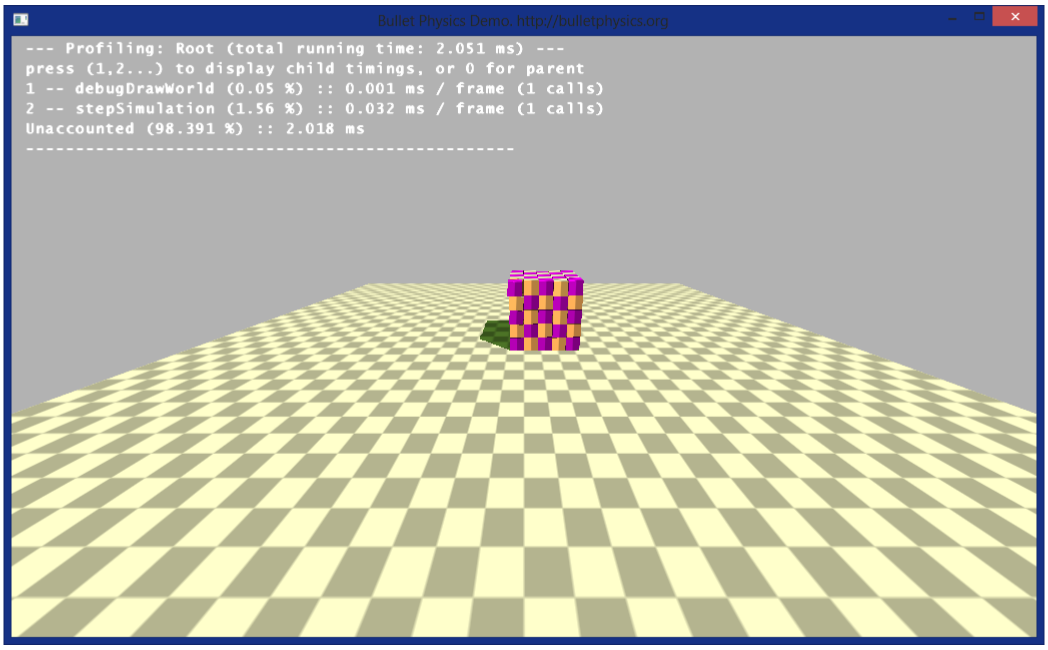
E = Effector
- i1 = index in neural net for axis 1 of the effector
- i2 = index in neural net for axis 2 of the effector
- i3 = index in neural net for axis 3 of the effector

PRE = Location on the box to attach the joint
- x = displacement along x-axis of the face
- y = displacement along y-axis of the face

side = the face index on the box

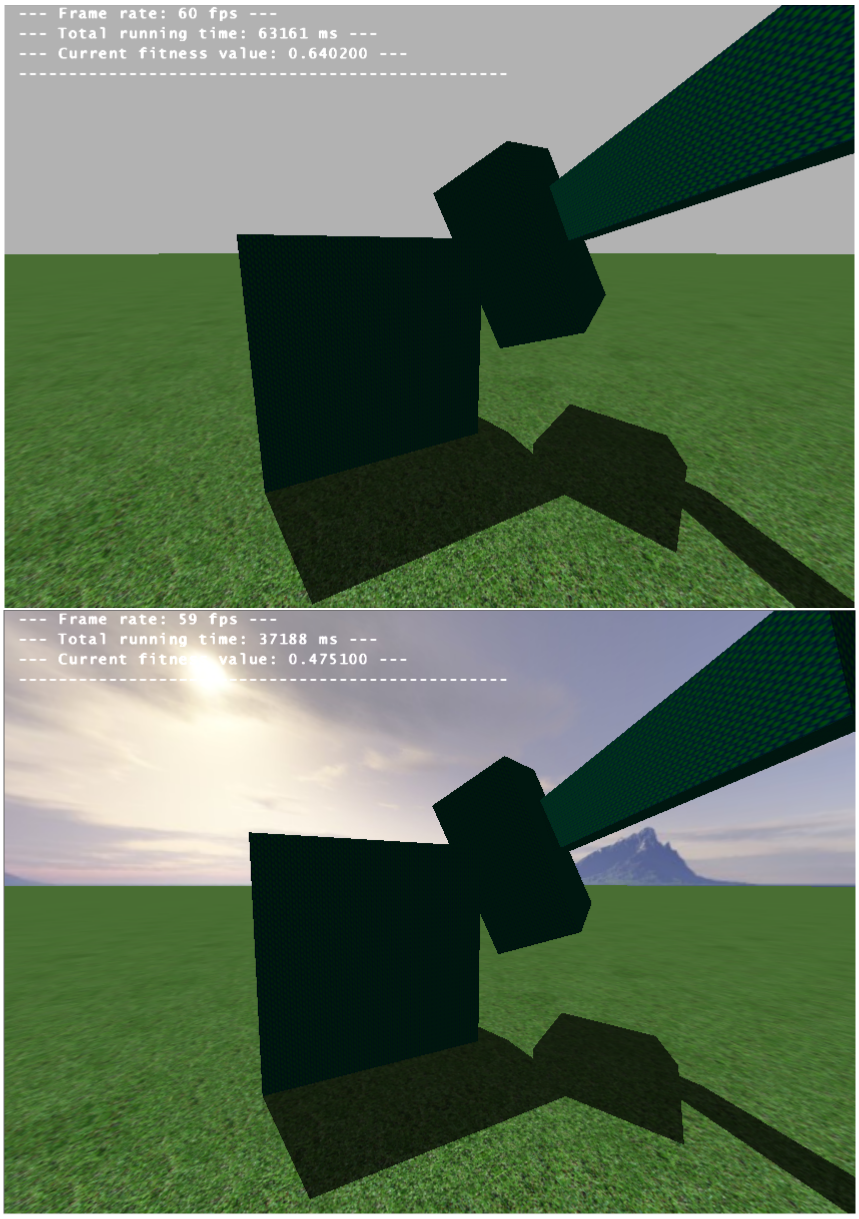
POST	= Location on the new box to attach the joint
	x = displacement along x-axis of the face
	y = displacement along y-axis of the face
	side = the face index on the box
DOF	= Degrees of freedom in the coordinate system of the joint
	x = Degree of freedom in the x-axis
	y = Degree of freedom in the y-axis
	z = Degree of freedom in the z-axis
NI	= Neural net node
	ncv = Choosing value determining the amount of inputs
	value = Value of a constant value node
	f = Function type
	in1 = Index of input 1 in previous layer
	in2 = Index of input 2 in previous layer
	in3 = Index of input 3 in previous layer
	w1 = Weight of input 1
	w2 = Weight of input 2
	w3 = Weight of input 3

C Figures



```
--- Profiling: Root (total running time: 2.051 ms) ---
press (1,2...) to display child timings, or 0 for parent
1 -- debugDrawWorld (0.05 %) :: 0.001 ms / frame (1 calls)
2 -- stepSimulation (1.56 %) :: 0.032 ms / frame (1 calls)
Unaccounted (98.391 %) :: 2.018 ms
-------------------------------------------------
```

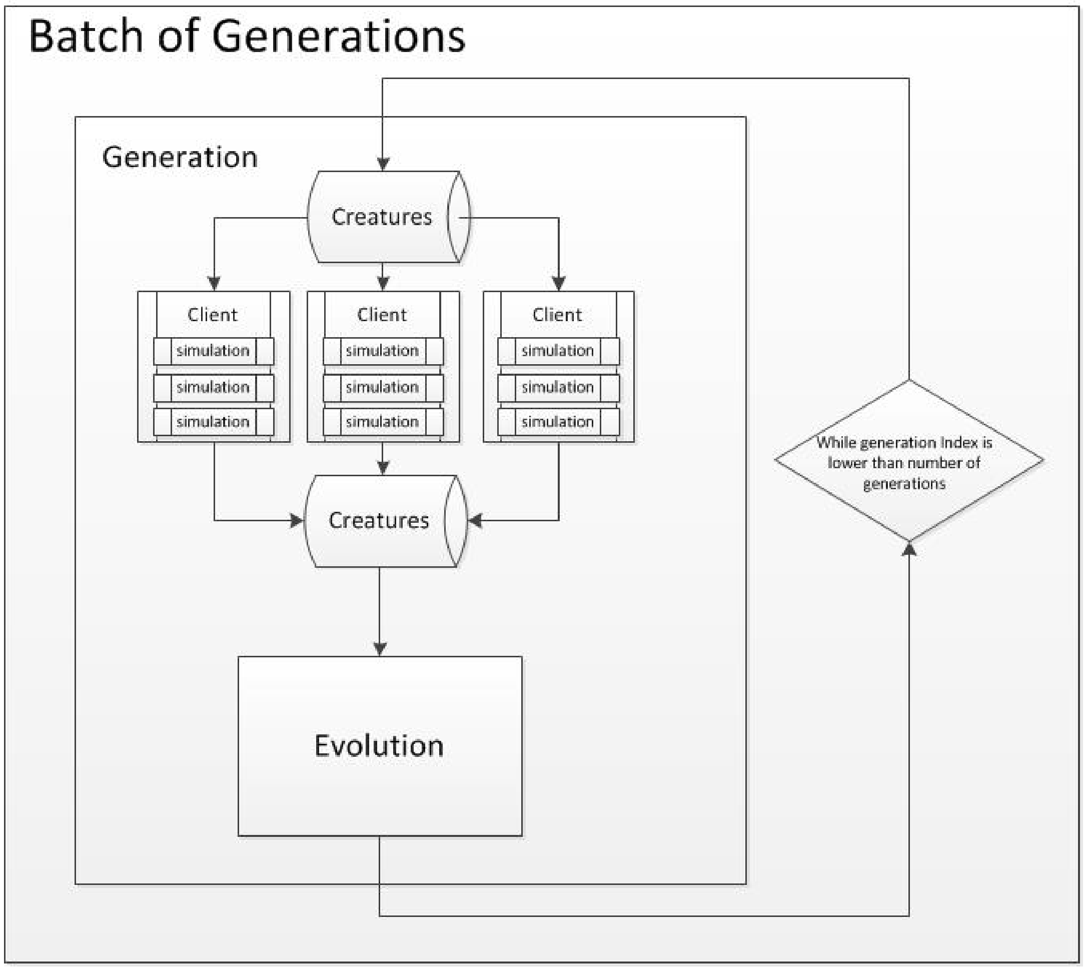
Figure 9: Bullet standard image



```
--- Frame rate: 60 fps ---
--- Total running time: 63161 ms ---
--- Current fitness value: 0.640200 ---
--------------------------------------------------
```

```
--- Frame rate: 59 fps ---
--- Total running time: 37188 ms ---
--- Current fitness value: 0.475100 ---
--------------------------------------------------
```

Figure 10: WWD skybox comparison



The example is shown for 3 cores and 9 simulations

Figure 11: Flowchart of the program