GALAPACOS

God Algorithm for Living Artificial Physics-based Adapting Creatures with Omnipotent Scalability

Author:
Peter Ølsted
IT University of Copenhagen
ptoe@itu.dk

Author:
Benjamin Ma
IT University of Copenhagen
benm@itu.dk

Supervisor:
Joseph Roland Kiniry
josr@itu.dk

May 23, 2012

Abstract

This paper describes a system that simulates and evolves virtual creatures. The virtual creatures exist in a simulated 3D world that has similar properties to the real world. Each creature's fitness is evaluated for distance traveled or jump height. The creatures with the highest fitness produce offspring for a new generation of creatures to be evaluated. Iterating this approach dozens or hundreds of times times mimics Darwinian evolution.
Using modern approaches for multithreading the simulation can scale to a large number of cores with little overhead. We present how we utilize modern hardware and off-the-shelf software libraries to simulate up to a million creatures per hour on a consumer level quad core machine.
We present the artificial DNA used to create the creatures, their neural network and how it is used to mutate the creatures. To evolve the creatures we also describe the different breeding and selection techniques used. While completely random evolution did not produce any creatures with good capabilities, specifying some boundary rules of the mutation produced better results.

We would like to thanks our supervisor Joseph Roland Kiniry for his support for this project.

Contents

1. Introduction

1.1 Problem Description

We want to build a simulation where creatures evolve that can react dynamically to its environment. Using genetic algorithms we will evolve the physical shape and behavior of the creatures. The creatures will move by their own decision using simulated physics and using concurrency to scale the simulation. The inspiration for this project includes Karl Sims' Evolved Virtual Creatures[10], DALi World[3] and the Microsoft developed Practical Parallel and Concurrent Programming[7] online course. Using some sort of "survival-of-the-fittest" parameter, to determine the best adapted mutation, the algorithm should pick the best performing creatures of each generation to further breed.
Currently, real-time interactive simulations, namely games, are using concurrency to improve performance, but the majority of systems are unable to scale with the addition of more CPU cores. Many major games use a one-thread-per-subsystem method, with one thread handling physics, one handling audio etc. and manually using mutexes for locking critical sections[13]. Optimally, we would like to create a scalable system that requires little synchronization and make the parallelization unintrusive.
Because the simulation is dynamic and the composition and behavior of each creature is dynamic, the concurrency has to work for every kind of creature, as well as a variable number of cores, making it very difficult to manually handle the concurrency. The dynamic concurrency and parallelization can, in turn, improve the performance of simulation, enabling the system to simulate more creatures at the same time.

Primary goals:

- create a physics-based simulation of an evolved creature
- evolve a creature, able to adapt to given fitness goals
- using parallelization and concurrency to improve the performance of

the simulation

Nice to have extra features:

- evolve a legged creature and give it behavioral feature(s) such as vision and movement
- pluggable creature features like hearing or offensive abilities like claws
- have creatures react to some external inputs (like player presence or injury to the creature).

1.2 Glossary

Concurrency: Methods relating to communication and handling multiple threads.
Parallelism: Methods relating to running the code in parallel to increase performance.
Simulation: The simulation of the world in which creatures evolve.
Creature: An artificial creature with both a physical body and a neural network as a brain.
World: The simulated space within which in which creatures is simulated and evaluated. A world only exists to simulate a single creature, so a world is created for each creature that needs simulation.
Neural network: The creature's artificial brain that gets its input from the creatures body, and determines the movement and behavior of the creature.
Evolved creature: Refers to a creature evolved by one or more algorithms that can react to its environment in its simulation.
Joint: The connection between two parts of a creature, that prevents free movement of the parts, and limits the possible movements.
Race: Definition of creatures with similar physical features. Two almost identical creatures would be referred to as having the same race.
Fitness: A value indicating how good a particular creature is at fulfilling its given task. An example would be: how fast can the creature moves, or how high the creature can jump.
Population: A collection of all the creatures in a generation before selection, breeding and mutation.
Generation: Every time a population has finished simulating and breeding it is called generation.
Mutation: Random alterations that is applied to every creature when building a new generation. It could add or remove limbs, change the neural network or change an existing part.
Crossbreeding/Crossover: Taking the physical and neural network of

two creatures and exchanging DNA parts to create an offspring of the two parents.
Selection: The decision maker of which creatures gets offspring.
Phenotype: The physical appearance of a creatures. Includes the body parts and the joints.
Genotype: The artificial DNA created for this project that describes how the phenotype and neural network is build.

1.3 Technology Utilized

Since we are trying to create virtual evolution with performance as a focus, the choice of technology is critical, especially since our time-frame was very limited, we would not be able to create everything from the ground up. It was therefore necessary for us to create our virtual evolution with as much pre-existing code as possible to meet our requirements.

1.3.1 C#

Undertaking a project like this, where part of the premise was to recreate old A-life simulators with new technology, it would be fitting to also use a modern programming language.
One of the key aspects of our choice was to find a language that did not require too much attention to semantics or memory management. It would be best if it was a familiar language and performance not much slower than C/C++ . Built-in concurrency support was also important.
C# was the straightforward choice for us, since we had previous experiences in programming with the language and it supports all needed features. Microsoft has also developed a simple framework called XNA in C# that abstracts the graphics initialization and other low level initialization and handling needed to make a graphical simulation.

C++ was also a consideration, but since experience with C++, was lacking, being able to attain the required goals in time was questionable.

Java was also considered since the programming language also has built in concurrency support and looks very similar to C# , but since easy utilization of the graphics card was lacking, and a good 3D physics engine was nowhere to be found, this language was not chosen either.

1.3.2 XNA

XNA is a game framework developed by Microsoft, written in C# . It abstracts many tedious things that are required for developing a game. It gives easy access to rendering capabilities, keyboard and mouse input, audio, etc. This allows us to have almost complete control over the rendering

and update aspect of the simulation. Large commercial game engines like Unity3D also gives you the ability to write in C# and gives you advanced rendering methods. However, Unity3D only supports limited multithreading capabilities and thus not chosen for this project.

1.3.3 BEPU

One of the other critical aspects of the program, was simulating the creatures' movements realistically. The best solution would be to create photo-realistic creatures (e.i. a dinosaur would have a dinosaur-like shape and silhouette, and not be a model made up of many boxes) but due to our time constraints this was out of the question.
Large scale commercial physics engines like Havok and PhysX support every feature needed, but only unofficial and outdated C# wrappers existed and writing one ourself would be a major undertaking and take up valuable development time.
Looking through all native C# physics engines we could find, BEPU physics seemed to be the best choice. It is a former commercial physics engine, that is now open source and still in active development. It had all the necessary features and the included demos indicated good performance and it supported multithreading.

1.3.4 NeuralDotNet

There exists a number of open source C# neural networks, among others AForge[1], Encog[2] and NeuronDotNet[3]. All of them supports the needed features, as only a basic neural network is needed for this project. NeuralDotNet was ultimately chosen to be used as it seemed more simple to comprehend than the other projects and easier to modify to meet this projects requirements.

1.4 Inspirational Work

1.4.1 Karl Sims

The main inspiration for this project were Karl Sims' two papers "Evolving Virtual Creatures"[10] and "Evolving 3D Morphology and Behavior by Competition"[9]. These papers outlined how it was possible to create simple life-like creatures via a simulated evolution. Karl Sims' approach was using boxes as the basic physical shape, and have different types of joints connecting them to create features resembling that of a real animal. To control the

[1] `http://code.google.com/p/aforge/`
[2] `Encog`
[3] `http://sourceforge.net/projects/neurondotnet/`

creatures joints, he a neural network as a neural network for brain. Sims was able to evolve creatures that would behave in ways difficult to replicate with manually created programmed AI by simulating natural selection. By randomly mutating the creatures' genes, new features would emerge, and using selection algorithms to eliminate the creatures that could not adapt, the fittest creatures were guaranteed to survive. The surviving creatures would then go on to breed and create offspring, where offsprings is a mix of its parents genes. These offspring would then be the next link in the virtual evolution, where the weakest offspring would be eliminated and the fitter ones would go on to breed.
We also found two videos showing the results of his work [8] with the second also containing an interview [11].

1.4.2 AI Game Programming Wisdom

AI Game Programming Wisdom was a book series containing of four books published between 2002 and 2008. Each book consisted of a series of articles of different subjects all related to AI in games. Articles include papers on game unit tactics, learning, pathfinding and general AI code architecture. We had access to AI Game Programming Wisdom 2 published in 2004. We used three articles on how genetic algorithms work to help build this project. [15], [14] and [1].

2. Parallel Architecture

A simulator can be a fairly easy task, if doing calculations is the only goal. However, writing a good simulator that can make good use of the hardware to increase performance, can vastly decrease the duration of the simulations. Having good visualization of the simulated data can also help with the understanding of the results. But for this project we aimed not just to make a fast simulator with good visualization. We attempted to create the fastest artificial life simulator to date. In the following chapter we describe what went into this attempt.

2.1 Requirements

- **High performance** Must be high performance to improve iteration speed.
- **Interactive & observable simulation** All aspects of the simulation must be interactive and observable if desired by the user. Being able to see the strands of the evolution should make it easier to debug. Increased performance should improve iteration speed as results become available faster. This requirements will simply be referred to as being interactive.
- **Scalability** Must be able to scale close to linear in regards to number of cores on normal consumer hardware. At time of writing you can expect up to 4 cores in a CPU where some CPUs can handle two hardware threads per core.
- **Simple implementation** Being able to easily understand and reason about it should help to avoid normal problems like deadlocks and race conditions.

Most of these things are pretty standard requirements, like high performance and simple implementation. Especially "simple implementation" is something that is always desired by developers, but not always achieved. Parallel programming may also face race conditions and deadlock problems, something we hope to avoid by sidestepping the problem as much as possible.

Locks should only be applied to private data and race conditions be avoided by use of simple concurrency safe collections. See "Implementation", section 2.5, for more details.

2.2 Game Loop

This section is a short introduction to how games are usually structured as the core concept is very similar to this project and uses technology designed for video games.
While the simulation aspect of this project strictly speaking is not a video game, it shares many of the same aspects. Video games and this project share the of needing to get the user input (keyboard & mouse input), updating the simulation, render the result and repeat. Each full iteration of the game loop is referred to as a frame. The state is maintained between frames and updated during each frame iteration.

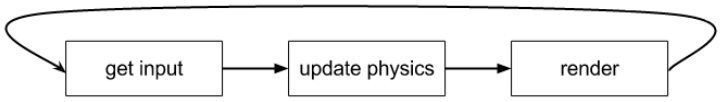


Figure 2.1: Simple overview of update process of a single frame and the looping of the frames

The update process of each frame is also strictly linear. If you switch around the order of the input, physics update or rendering, it will take longer for the game to react to the user input. Input lag is something that should be avoided as it makes the interactive part noticeably more annoying to work with. Having a separate thread to receive user input does not improve recessiveness as the game cannot react to the input before next frame. On current hardware the minimum delay one can expect is three frames multiplied by the length of a single frame [17].
The interactive part of the simulation is mostly in regard to moving and rotating the virtual camera. The mouse is used for rotating the camera and the keyboard for moving the camera, usually in the direction the camera is facing.
The higher the frame rate the smoother the simulation, usually between 30 and 60 frames per second (fps) is required for a smooth simulation. Frame rate stability also matters. If the frame rate continuously jumps between 30 and 60 fps, users will still notice the instability.[1]

[1] `http://en.wikipedia.org/wiki/Frame_rate#Video_games`

To cope with varying frame rates, each frame is usually updated with how long the previous frame took to update. This makes the game seem less jittery than it actually is. Say you want the camera to move 100 units each second. Multiply 100 by 0.016 (if the last frame took around 16ms) and the camera will behave uniformly with varying framerate.
If the computer can run the simulation faster than 60 fps you have to cap the frame rate unless you want the simulation to run faster.
Having a uniform framerate is especially important with the physics engine. The physics engine is the part of the code which handles all the physical interaction and behavior of everything in the simulation. As it is only updated in discrete steps the behavior of the physical objects can change drastically depending on how long a frame takes. This is because a physics engine usually first updates the position of involved objects and then resolve any collision. Say that two high speed objects fly directly at each other and framerate is low, the objects could fly directly through each other as they never collide according to the physics engine. Some engines solve this problem by having the physics engine always update with the same timestep and simply avoiding having high-speed objects. It is one example of how too high or too unstable framerate can change the results of the physics engine.

2.3 Parallel Simulations and Rendering

To achieve a high-performance simulation without compromising how observable it is, special considerations must be taken of the observability. If the frame rate dips below 30, reaches above 60 or is generally unstable it becomes unbearable to watch. A solution must be able to stay within that framerate and if requested by the user any simulation must almost instantly be rendered. Each frame is also strictly linear as described in "Game Loop", section 2.2 . Only a single simulation can be rendered at a time.

To pick the best solution, we will propose three possible ones in this section. There are of course more possible solutions and the three discussed here can themselves vary. These these solutions should be simple to implement and must provide good performance. Having simple and effective solutions should make bugs less likely to appear and improve development speed.
A few prerequisites is needed before describing the . With modern CPUs supporting around 2-8 parallel threads, the solution has to scale at least up to 8 threads. There should not be a hard-coded maximum number of threads. It is expected that each thread used for the simulation will be fully utilized, so any more threads than the hardware support will lower performance because of thread switching.

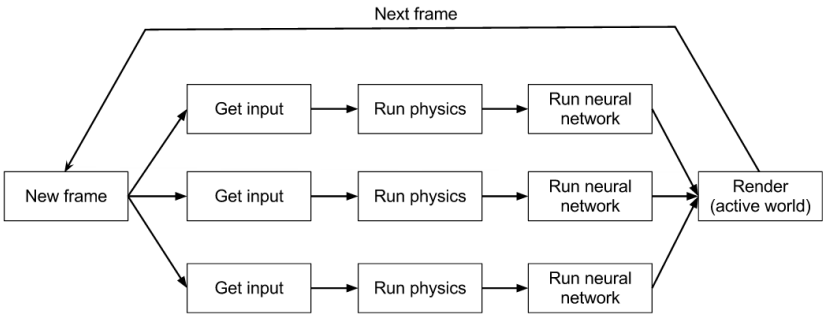


Figure 2.2: Overview of the frame-by-frame synchronization for three parallel threads.

2.3.1 Frame-by-Frame Synchronization

With frame-by-frame synchronization, Figure 2.2, all threads start and end the frame at the same time. Each thread is assigned it is own simulation world. Each world updates its inhabitants' physics and their neural networks. They then have to wait for all threads to finish this frame before rendering starts. Only one thread is used for rendering and it renders only the world the user sees. Also only the active world receives any user input to avoid unwanted behavior.
The primary advantage with this approach is that communication with each world is very easy due to the synchronous approach.
As all simulation threads are suspended while waiting for the renderer to complete, an opportunity is given to manipulate any accessible simulation data in the program with no risk of concurrency problems. Using one of the other solutions, it is only safe to manipulate simulation data when an entire generation is finished simulation.

Frame-by-frame synchronization should also give a stable frame rate with no background threads interrupting the rendering. It also enables you to use all available threads for simulation which the other solutions may not.
However this approach severely limits performance. With all simulations being suspended every time rendering occurs, you loose a lot of cycles that could have been spent on the simulation. Especially if one creature takes longer than the other. Also the more time spent on rendering, the less time available for simulating.

2.3.2 Synchronized Rendering

Synchronized rendering, Figure 2.3, is a solution where only the world the user is observing is synchronized. All threads run with their own isolated

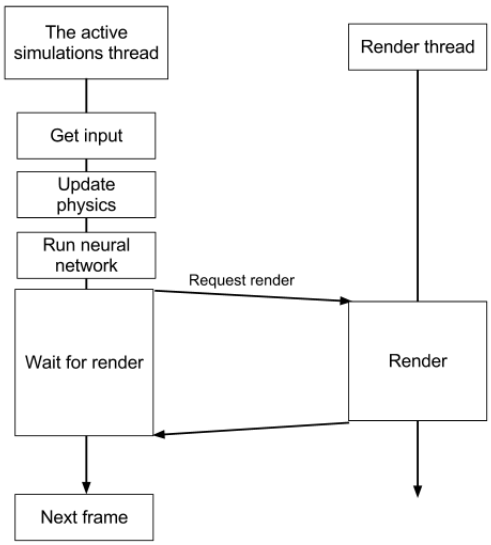


Figure 2.3: Sequence diagram-inspired diagram of synchronized rendering

simulation with only the thread handling the observable simulation, is synchronized.
The thread handling the observable simulation runs the simulation the same way all the other threads do, except for when it should render. There is an extra thread whose only purpose is to render the observable simulation. During rendering the active simulation thread simply waits for the render thread to render and visa-versa when simulating.

This solution should have excellent performance. Only a single simulation thread at a time is synchronized so there will be minimal overhead.
Depending on the implementation and how system handles rendering with almost 100% CPU utilization, this solution can cause problems for the interactive aspect of the program. To avoid this problem, it may be necessary to use a hardware thread only for rendering, loosing a thread that could have been used for simulating.

Not needing a dedicated thread for rendering would be a better solution, but XNA restricts all rendering to the the main thread. Not having to worry about which thread renders the simulation could provide for a simpler solution as less synchronization would be necessary, but that is not possible.

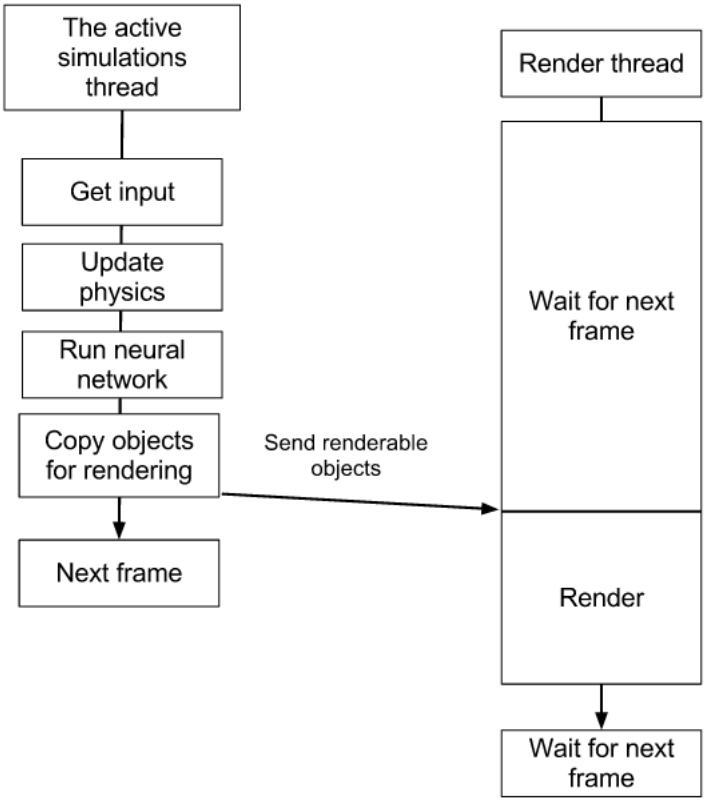


Figure 2.4: Sequence diagram-inspired diagram of unsynchronized rendering

2.3.3 Unsynchronized Rendering

Unsynchronized rendering, Figure 2.4, is similar to synchronized rendering, expect that the rendering, as the name suggests, is unsynchronized. The difference is that the simulation thread, instead of waiting for the renderer to finish, makes a copy of the objects that should be rendered and via some mechanism sends it to the render thread. After the copying the simulation thread starts simulating the next frame.
The problem with this approach becomes very clear when the speed of the simulation does not match the rendering. Say the simulation runs 200 times per second, but the renderer can only render 60 frames pr. second. Rendering only every three frames or letting the rendering fall increasingly behind are both terrible solutions.
The only proper solution is to approach the synchronized solution above. If the renderer falls behind the simulation thread must wait. With this approach the previous frame is rendered while a new is being simulated. However, the closer to 100% CPU utilization the program has, the less do you gain by this approach. If the render thread and some simulation thread continuously steal CPU time from each other there is no benefit over the synchronized rendering solution.

2.4 Selected Method

For this project we have chosen to implement the synchronized rendering approach. It is simpler to implement than unsynchronized rendering as it does not have problems with the renderer being slower than the simulation. There should also be considerably less overhead than the frame-by-frame synchronization, which seems to be the easiest solution to implement.

2.5 Implementation

Synchronized rendering and XNA is not completely compatible. As described in "Synchronized Rendering", subsection 2.3.2, the render thread has to be the same as the main game thread and thus rendering cannot happen on a simulation thread.
Trying to implement it the parallel architecture so simply as possible, the primary focus was on isolating the parallel parts and making the parallelization intrusive transparent as possible. The code handling the simulation should only be aware of its own simulation. It should not know about anything other than itself.

In this context a world refers to the physical world with a creature being simulated in it.

Isolating the code handling the frame-by-frame simulation is the core part of the implementation. The two most important classes are SimGame and NormalWorld. SimGame is the class handling the rendering and initialization of the program. It also knows which world is the one being observed. NormalWorld is the class handling the simulation of a creature. Each simulation thread runs a single NormalWorld and thus a single world.

The code handling the simulation is all contained in a class called NormalWorld. It is given a queue with creatures waiting to be simulated and queue to push the creature to when the creature has been given enough simulation time. How long a creature is given depends on the simulation and can be changed depending on the simulation.
To facility communication between the main thread and the simulation threads we created a service provider. The difference between this and other service providers is that it facilitates that each world is unique and can fetch different services depending on which worlds requests a service.
With rendering being restricted to the main thread because of XNA an interlocking mechanism is needed between the simulation threads and the main thread. Making the NormalWorld oblivious to how it is rendered was also a goal to separate responsibilities.
To accomplish this a handler for the simulation thread and world was created called WorldUpdater. WorldUpdater handles frame duration, rendering requests and is able to suspend the simulation on demand. It is the class that initiates each frame update.
Rendering requests is a method pointer call to SimWorld. Every world requests rendering, but only the world the user is observing is rendered. The thread handling the observable world is yielded until the main thread completes rendering. If XNA is ready to render before the next frame is simulated the main thread simply yields until a frame is ready.
This way the rendering is centralized to one place and NormalWorld is completely oblivious to being suspended.

With a variable number of creatures in each generation and a variable number of threads a work distribution system was needed. With the work units being very coarse grained, we implemented a simple concurrency wrapper around a C# queue, exposing only the minimal needed functionality. It is a simple lock mechanism that is contained within the that class to avoid locking problems.
To avoid problem with the physics engine when the frame rate is not 60 hz, every physics engine update is always done with the same timestep of 16.6ms, emulating 60 hz. This results in reliable physics simulation that behaves consistent no matter the actual speed of execution.
Parallelization of the breeding at first written single threaded to make the program run. No regards was given to performance in its implementation. Before trying to parallelize the breeding we observed the performance. In

general the breeding takes about 2-0.5% of the total run time of each generation. This was observed with both small and large creatures on slow and fast computers. Deeming the performance good enough we choose not to parallize the breeding.
To avoid compromising between how interactive the simulation is and the performance of the it, there are three modes the program can be in: interactive, balanced and performance. In interactive mode rendering is enabled and the frame rate is limited to the refresh rate of the monitor, usually 60. In balanced there is no frame rate limit and rendering is still enabled. In performance mode there is no frame rate limit and no rendering. This is made by simple ignoring any rendering requests and make the main thread draw a blank screen.

This solution does have some limitations. It is cumbersome to communicate between the main class and the different worlds. All reliable communication has to be via the service provider. Any query of worlds state may be invalid if the world happens to be in the middle of an important method.
It also lends itself to unstable frame rates on some computers. On some computers, when simulating on all available hardware threads, the frame rate dips to non-interactive levels or jitters wildly. If the observer just want to run the simulation it is not a problem, but if trying to debug or observe the evolution, it is a problem. Why this changes so much depending on the computer we do not know. Using one thread less than the available amount avoids the frame rate instability, but this will increase simulation time as the simulation runs one less thread .

When we started on the project we also though it would be a good idea if there were a separate camera for each world. This way each creatures could be observed individually, however this turned out to bad a idea since the camera had to be repositioned for every world.

3. Artificial Life

3.1 Introduction

In our simulations we wish to create artificial creatures using some basic principles of evolution. In this chapter we discuss the implementation of these principles in the program, and creation of artificial life forms capable of developing a brain.

3.2 Neural Network

A artificial neural network is a network inspired by how real biological neural network works. Artificial neural networks will be referred to as neural networks.
A fundamental element of neural networks are their ability to learn. Based on some input and a desired output, they are trained to transform the input to the desired output. Which input is given, what the output is and how the network is trained, depends on its application area. The primary reason for using neural networks is because they adapt to solve a problem.
A network is composed of a set of neurons communicating via connections. The input is gathered by the neuron from the connections to it and transforms the input to an output. The transformation is called an activation function and is usually non-linear. A common shape of a neuron is the shape of a sigmoid function. Each connection is weighted, meaning the value going through the connection is multiplied by the weight. The network can change behavior by changing the weights.

A full cycle of the network is made every frame. Every neuron is given the output of every other neuron connected to it, and the output propagates to its target neuron.

To react to the state of the phenotype of the creature, each part is given two input neurons and two output neurons. The first input neuron is activated if the part touches the ground and the other is continuously given the angle of the attached joint. The output neurons are both used to control the joints

motor that will move between its minimum and maximum angle. Depending on which neuron outputs the largest value, the joint will move towards its minimum or maximum angle.
Using input neurons that reacts to different kinds of input is also a possibility. The implemented neurons are two options among many. Other types of input neurons could be proximity based sensors, speed sensor or perhaps a timer sensor. In combination with different fitness algorithms these could help train a creature to for example move to a static light source or even a moving light source.
While there are many possibilities, we deem the two implemented input neurons the most important ones. Collision reaction and joints awareness are some of the most two fundamental input awareness a living being can have. It allows reaction to its environment, while the other suggested inputs could help evolve more dynamic creatures, we do not deem them essential. They would be interesting to implement in future work.
The choice of output neurons is made from using the same arguments as the input neurons, using the output neurons for the most basic of movement. Controlling joint stretching and retracting is the most fundamental parts of movement control. Additional output neurons could control the speed and strength of a joints movement.

To create more varied behavior of the neural network, four different types of neurons are used that all operate in the 0 to 1 range. While a sigmoid function operate in the -6 to 6 range, we believe that a smaller range would be easier to control and thus give better results. Since changing the operation range will affect the result, we lowered the default weight of the connections to compensate for the smaller range. While we cannot say how it affects the result, we can still confidently say that the neural network works and reacts to the simulated world.

The four different types of neurons are described below with their general shape shown in the pictures below them.

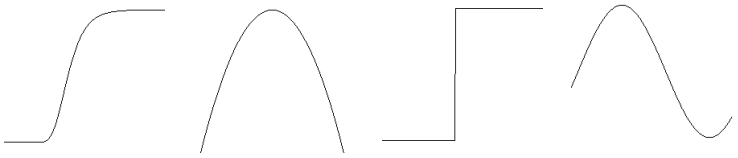
Gompertz is similar to a sigmoid function, but allows more control over growth and range.

Threshold function, activates at 0.5.

Gaussian function is a curve that looks like a bell. The standard form never output negative values and peaks where x = 0.

Wave function is a modified sinus curve that does not use take input, but oscillates based on how long the creatures has lived. ...

To enable the creature to move without external input the wave neuron was added. The behavior of the creatures was rarely affected by the angle of the input and thus only moved when the right part was touching the ground.



(a) Gompertz (b) Gaussian (c) Threshold (d) Wave

Figure 3.1: The shape of the four different activation functions used. They may differ slightly in the implementation.

The addition of the wave neuron gave the creature a more reliable input than just relying on external factors, like touching the ground.

Mutation of the neural network is done without regards to the creation of cycles. Having cycles in a neural network makes it recurrent. Recurrent network contains an internal state and this may result in temporal behaviors[1]. Having a temporal behavior could make the network more unstable with new input being negated by its current state. It could also improve effectiveness by using its state to handle new input better. This is only theoretical as a non-recurrent network has not been implemented.

We compared our neural network to that of Karl Sims. In his simulation of the neurons, the cycle only propagates two steps every frame. In this project the entire network is run to completion for each frame. NeuralDotNet only supports a full run of the network. While it is not possible to compare the two implementations in this project, we think that this projects approach gives a more reactive network, as any input immediately affects the output sensors. It could result in more unstable behavior as any change is immediately reflected in the behavior of the creature.

3.3 Physical Creatures

Opposite the genotype, is what Karl Sims refers to as the "Phenotype". A phenotype is the physical representation of the genotype; the genome. Each creature is made from the description that it inherits. The genotype, like the DNA only holds information on how to build the phenotype, and does not directly contain any information that the creature receives during its lifespan.

[1] `http://en.wikipedia.org/wiki/Recurrent_neural_network`

3.3.1 Boxes

The goal of our phenotype is to have something that looks and preferably acts like a real creature. While it is not currently possible to make living creatures photorealistic[2], it is possible to make models that resemble them a lot. The problem with this is that the models are not easily generated, and take up a lot of computing-power. Creating and simulating a skeleton is a much easier solution, but the collision area of a skeleton is much smaller than a real creature with muscles, meat and body tissue. Adding muscles to the creature would also take up a lot of processing power, and add a lot of work on top of the existing requirements. The goal however was not to create any existing creatures, but to create creatures that were able to adapt to their environment and to a given fitness measurement.

To approximate the shape of a creature, but retaining some shape and have good performance in the physics simulation, we use simply geometry. Using multiple small shapes can approximate more complex appearances while retaining fast performance. It is the same approach large game productions use[2, 19]. An example of a human approximated, from the game Unreal Tournament 3, via simple shapes can be seen in the appendix, Figure D.

3.3.2 Joints

Connecting the boxes is simple enough, but making them lifelike is a different case. Movable joints need to connect them, that allow for different movement limits.

Along with joints are motors, the control mechanism for joints which act as the muscles of the joints. It moves the joint within the specified limits, and by linking these motors to the neural network, we get the creature behavior.

The physics engine supports creation of different types of joints. Among others, it supports universal, ballsocket and revolute joints[6] that all behave similar to their real life counterparts.
For this project we have only used revolute joints. This decision was a combination of the simplicity of using a joint the abundancy of that joint in nature. While revolute joints found in nature, like elbows, we have more unrestricted movement allowing bending up to 180 degrees. We deemed universal joints too unrealistic, but could be used in a future project. Ball sockets joints, like shoulders, are very commonly found in nature, the control mechanism was incompatible with revolute joints as they allow for movement around two axis, while revolute only allows one.

[2]Computed generated pictures may be indistinguishable from photos, but we have yet to see a video that is indistinguishable from a real video.

3.3.3 Phenotypes in Practice

With the above mentioned boxes and joints, we were able to create and evolve creatures in combination with the neural network. The boxes (although maybe a bit unfair) provide a big flat surface that help the creatures balance, even with just one "leg".
All boxes are connected via revolute joints where the genotype controls the features like how strong a joints movement is, how fast it can move, minimum and maximum angle.

3.4 Genotypes

When we are talking about simulating evolution of creatures, it is important that we copy any critical factors. In the case of creating offspring from fit creatures, it is then extremely important that we have some sort of digital DNA.

According to Wikipedia, DNA is a

> "nucleic acid containing the genetic instructions used in the development and functioning of all known living organisms"[18]

While the theory is a lot more complicated, this is the core idea, that we need to copy. Furthermore DNA does not stay intact. Among other reasons, radiation or small chemical malfunctions can cause the DNA to be "corrupted" or mutated. Mutation is thought to be random, and any part of the DNA could potentially be altered. While mutation may have negative side effects, it is believed to be part of the reason that creatures can evolve, since it can also provide better suited genetic features for a given environment.

3.4.1 Representation

While the DNA is easy to replicate (4 base pairs form a code. A base-4 language), it is not a very practical solution when working with as simple creatures as we do. DNA stores an immeasurable amount of data and is very context specific. A strand of DNA moved to another place would produce very different results. Creating a copy of it is not impossible to do, but we deemed it would take too long time to create and opted for a more straight forward solution.

A solution that is similar to real DNA would be a binary code representation. With it being similar to real DNA, it suffers the same drawbacks as a copy of real DNA. Binary code would be very delicate to handle, and would need to be designed in a manner that would allow for totally random mutation, variations and lengths of code. As with real DNA it not impossible, but simply deemed to take too long to implement.

A more readable solution would be using strings and would allow for human readable data and much simpler implementation. We opted for this approach, but it does have some drawbacks. It disallows completely random mutation and crossover. Both can only happen at predetermined points to keep the DNA valid. A class representation was implemented to simplify mutation and crossbreeding. Using these it is impossible to mutate non-legal places in the string.

3.4.2 String DNA

The string representation of the DNA, is a basic description of each subpart of the creature. The four categories of subparts are: *Part, Joint, Neuron* and *Connection*. Each part then has a number of fields, that have been uniquely named (ex. part and joint can not both have a field by the name: "x". One is named "x" and another is named "jx"). A field can contain an integer, floating point number, string or a boolean. The type is converted from a string to its respectable type, at different points during the program execution.

At the beginning of the program, a DNA seed is created, that will be the entire population of the simulator in the first generation. This is created from a string DNA. If a command line is passed to the program that identifies a file with the string DNA, the genotype seed used will be the one loaded from the file.

The advantages of strings are the readability and ease of saving/loading them, regardless of how they are stored.

The inspiration for genotype description came from a similar project called framsticks[4]. It also used a textual DNA representation[5] and seemed to fit the needs for this project. It was used as the inspiration for the DNA used in this project. While similarities definitely still exists, the many variables is changed. Some information in the representation is no longer used, but is kept for compatibility.

A full description can be seen in "Full Genotype", Appendix A .

3.4.3 Class DNA

While string DNA in a sense is close to an actual strand of DNA, it does not behave in any way like a regular DNA. True DNA can be mutated at any place. While our string DNA is open to random mutation, it would cause corruption, that would possible lead to the simulation crashing. It is

easy to split up the string, and then only mutate part of the string, and avoid corrupting the field-names or splitting-symbols. Even if a mutator was written, that could distinguish between what should be integers, floats etc. it would be hard to cross-breed the creatures.

For this reason we also created a class representation of the DNA information. The DNA class contains four lists. One list for each type of DNA where each item contained a representation of a single part of the creature (a joint, part, neuron or connection). These lists would make it easy for us to manage the DNA subparts, mutate them in a way that would not corrupt the program and allowed us to do easy look-ups, which in turn allowed for crossbreeding in a manageable way. This approach however limits crossbreeding to a simple exchange of objects. Single objects cannot be split.

String DNA and class DNA represent the same information, and can therefore be converted between each other without any information-loss.

3.4.4 Mutation

A key point to evolution is the randomness of mutation, that is able to produce new unexpected creatures, with features that are either better suited for the environment, do not matter or produce features that cripples the creature.

As mentioned, total random mutation would not work within our simulator, and a different system for mutation had to be set up. Instead we used the class representation of the DNA to create a subpart-specific mutator, that we could tweak for individual fields. Each field would have either have a random value (positive or negative) added to them, or have an equal chance of mutating into each possible value. (e.g. true/false)

With such a mutator it would be easy to mutate every gene for each creature, but we needed to simulate real life as much as we could in our numbers, so we allowed for more extreme mutations, but only a few across all subparts per creature. A "roulette" would pick the number of fields that should be mutated for each creature. An unlimited number of mutations is possible in theory, but a high number of mutations is highly unlikely, and a low number is often occurring, with zero as a possible amount. Afterwards the random amount would be used to choose parts, joints, neuron and connections. For each subpart a field would then be chosen, and then the mutator for that individual field would provide a new random value.

3.5 Evolving Life

3.5.1 Cross-breeding

Another reason that evolution could be made possible is the process of crossbreeding. Many animals and organisms require breeding before offspring can

be created. Due to low odds of identical DNA and mutation, the offspring will have a chance of have a fitter genotype than its parents (however the opposite is also a risk).

During natural breeding, a new DNA is created from a combination of the parental DNA. To simulate that a number of algorithms exist.

One-point crossover

A one-point crossover is the simplest form of crossover, that is also the least natural one. Two strings of DNA are combined by splitting both DNA at the same position, and then using one end of each DNA to combine the offspring.

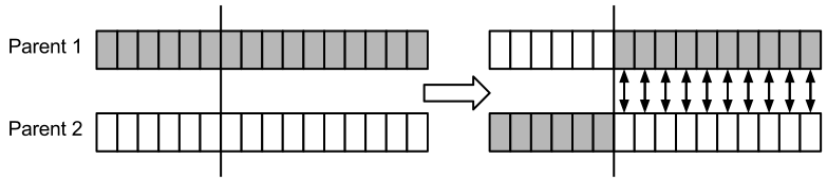


Figure 3.2: Example of a one-point crossover

Two-point Crossover

The two-point crossover is similar to that of the one-point, but allows for both the beginning and end of a DNA to be from a single parent. Instead of one split across the DNA, two are made, and the ends and the middle part is used to produce an offspring.

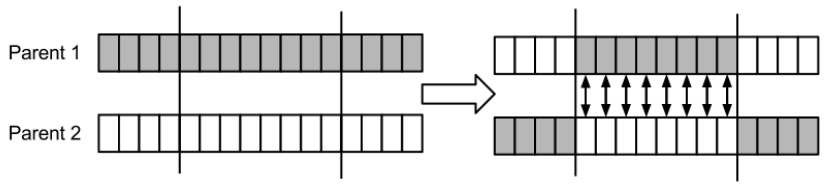


Figure 3.3: Example of a two-point crossover

Multi-point Crossover

The two previous crossover algorithms, while simple and having a good probability of creating working creatures, are not totally random in their crossover, and allow only for big chunks of DNA to be passed along. The multi-point crossover differs in that it allows for total random crossover. each single "step" on the DNA of the offspring has a 50-50 chance of being from

either parent, and that allows for greater variety in the offspring. However with total randomness also comes a greater risk of creating invalid creatures.

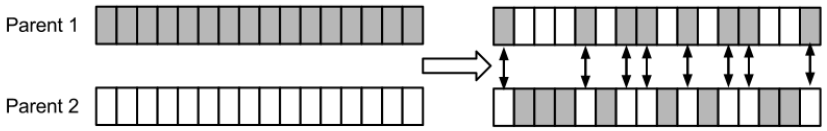


Figure 3.4: Example of a multi-point Crossover

3.5.2 Fitness Explanation

The phrase "survival of the fittest", which is often used to describe the core principle of evolution would also appear to be a reasonable explanation of what happens in our simulation. While due to different selection algorithms (See Section: Selection Algorithms) this may not always be the case. However the principle of a "fittest" creature being a survivor is still what goes on.

However how do we decide what creature is more fit than others? In farming, this is done "by hand" but this process needs to be automated, so we need a measurable number, that we simply call "fitness". The more fitness a creature has, the better it fulfilled a measurable goal.

3.5.3 Use of Different Fitness Algorithms

Measurable fitness targets are almost unlimited, but we need simple tasks that can be measured across most or all creatures that can be developed. The fitness measurement should also indicate what the desired behavior should be, ex. if realistic movement is desired, it isn't simply enough to measure the distance traveled, since the nicer movement in no way is reflected in the distance the creature moved. The more specific the goal is, the harder it is to measure across a number of creatures, how fit they are. Using the "nice movements" example, how is the nice movement defined, and could this algorithm be exposed to create creatures that are fit, but do not move in the desired "nice" way?

To help guide the evolution the goal had to be clearly stated. For example: distance traveled or jump height.

Measuring distance can be done in a number of ways, but our approach was to take the average position of all parts in the creature at the end of the simulation compared to the center of the world. The difference in width and depth (2D distance) was the traveled distance.

Height will be measured as the average distance as well, but along the third axis. The value will also be the maximum amount of height obtained.

Depending on how the creatures are added to the world (when the may experience a small drop, after spawning at a higher position, than the ground) measuring only after a given amount of time would be important. Height can also be measured in how high the creature's average Y value is at the end of the simulation.

Proximity will be a reverse fitness, where the smaller fitness will be preferred. The fitness will be the creature's average distance from a given position on 2 or 3 axis.

3.5.4 Scaling

After measuring the fitness of all creatures, a number of scaling options can be applied, to remove extreme biases towards fitter creatures, or increase the difference the fit and less-fit creatures.

No Scaling

If no scaling is applied after fitness evaluation, the raw fitness value becomes the number fed into the selection algorithm. This has the advantage of being completely unbiased, and leave any creature with extreme fitness, remain with an extreme number, compared to the other creatures. The drawback is however that an extreme fitness can severely dominate the creature population, and take over, which isn't optimal. While fitter creatures should survive and remain, diversity will help prevent the evolution to hone in on a single ultimate gene.

Linear Scaling

One solution to the above mentions problem is the linear scaling method. For a population of X the creatures will have their fitness scaled according to the size of the population. The fittest creature will have the value of the population size: X, and the second fittest: X-1. That way the fittest creature still has the highest fitness, but any great gap in performance is reduced. The final population will thus have fitness ranging from 1 to X[1].

Sigma Scaling

Sigma scaling is a formula that alters the fitness for the population, to have the best creatures dominate, but not let any weak creatures be left with a tiny value[1].

$$NewFitness = \frac{OldFitness \text{ - } AverageFitness}{2\sigma} \tag{3.1}$$

Sigma(σ) is the standard fitness deviation in the population.

$$\sigma = \sqrt{\frac{\sum(f - mf)^2}{PopulationSize}} \tag{3.2}$$

Here f is the fitness of the creature and mf is the average fitness.

3.5.5 Selection

The final algorithm we use to simulate natural evolution is the selection algorithm. This is the key algorithm that determines what creatures get to reproduce, who will reproduce with who, and how many offspring they will get. In our project all selection algorithms needed to output just as many offspring as there would be parents.

No Selection

The easiest solution is to entirely skip the crossbreeding part. After all the mutation is the part of the evolution that gives new features, and the algorithm will easily return as many creatures as is received, since the output is the same as the input. However no crossover will have an effect on how fast fit creatures can appear, and how many different possible solutions will appear from the evolution.

Random Selection

Random selection is exactly what the name suggests. Two creatures are continuously randomly picked and breed until the population size reached a specified number.

Size Tournament

One of the simplest selection algorithms is the selection algorithm. It will pair two creatures, and remove the least fit creatures. This will half the population, and the remaining creatures will cross-breed, until the offspring population will be the same size as the parent population.

Probability Tournament

Probability Tournament is our own version of the Size tournament. Instead if the fittest creature winning, we give the fittest creature a high probability of winning, but not guaranteeing it. This way a less fit creature CAN win over a fitter creature, but the bigger the difference in fitness, the less likely it is.

$$Probability = 0.5 + \frac{(\frac{HighestFitness\ -\ LowestFitness}{HisghestFitness\ +\ LowestFitness})}{2} \tag{3.3}$$

Calculation of winning chances in a
Probability tournament for the fittest creature.

Probability tournament should help increase the variety of creatures in the population, to prevent the tendency to let a single race of creatures dominate or take over the population.

3.6 Elite

Elite selection it used to keep a population from decreasing performance. With this enabled the best percentage of creatures, 10% in this project, is not mutated or crossbreed[1].

3.7 Elite Copying

Elite copying is a term we have introduced for this project. It is a difference to elite selection, the best percentage is not changed or breed for the next generation, but in addition they are also cloned into the rest of the population to mutate and crossbreed. This give fast results, but also quickly converges on a solution. This removes the worst creatures each generation and thus could converge on a similar solutions in only ten generations.

3.8 Staleness

Staleness is used to keep the evolution from reaching a plateau where the elite creatures keep being among the elite. Staleness applies only to elite creatures. Each creatures DNA contains a counter for how many generations in a row generations it has been a part of the elite percentage of creatures. If a creature is not among elite creatures its staleness is set to zero. While staleness removes some of the benefits of elite, it also avoids the solution from reaching a fitness plateau. There is however no guarantee the solution will not simply converge on the same solution again or if a better solution will ever be found.

3.9 Solution Limitations

While the solution allows varied creatures and has good performance, see "Results", chapter 4, it does contain a number of limitations. All of which would be nice to fix in future work.

The first limitation is the use of third party software. Without a physics engine like BEPU we would never have made it as far as we have. It is truly a case of the idiom "you have to take the good with the bad". However it still has a number of problems with unstable joints physics and contains a

number of bugs when running multiple instances in parallel. NeuralDotNet was both a blessing and a curse. A good deal of bad workarounds was needed to implement the needed features like varying type of neurons. It was also unnecessarily complex for our need and does not support partial runs of the network. The good thing is that once it was all set up it ran without any problems.

The mutator is an essential part of the solution, but it is limited due to its implementation as it can only do semi-random mutations. Updating the mutator to be more aware of the the phenotype of the creature would help to create more exciting and life like creatures.
The limitations was in the addition of parts, how the neural network was added to a part and how a new part was formed. When creating a new part a neural network is not attached. Only input and output neurons were added and no connections were made. Until another mutation happens and the neurons are connected to the neural network a new limp is essentially dead weight.
Adding a new part to the creature is only limited by a very basic system. While the notion of head, body and limp to control part placement does exist, it still allows placement of parts inside other parts. If not placed inside another, the position can later be randomly mutated to end up inside another part. Improving the part addition code should allow much more interesting creatures.

With the semi-random placement of new parts, a part could violate the constraints of the joint connected to it. A violation is generally when two connected parts is placed with an angle between them that exceeds the minimum or maximum angle. It can also happen when a box is placed inside another. When a violation of the joint happens the physics engine tries to resolve it. Sometimes it is not possible for the physics engine to resolve it, but it cannot do anything but keep trying. This usually makes the affected parts jitter wildly and can in worst case suddenly fly into the air. Because of these problems collision between the creatures parts are disabled. An improved phenotype creator would help avoiding the problem in the first place.
When calculating a creatures fitness value, it can only be calculated from the creatures final state. This resulted in a limited view of the true performance of a creature. A creature evaluated for its jump height could be the best of its generation, but could be rated zero if the jump starts to late or too soon. A continuous fitness evaluation could result in more varied behavior and more complicated evaluations. One example is a fitness function created to breed legged creatures. This fitness function could punish creatures where the torso touched the ground, forcing the creature to evolve around this limitations. This could possibly produce legs or could evolve some creature we could not even imagine.

Possibly the most serious limitation, or bug if you like, in the program is that the simulation runs differently in debug and release mode. This also affects saving and loading. At time of writing we do not know where this problem lies. We think it is due to how differences in how the program floating points are handled in the different modes. Where the bug lies we do not know at the time of writing.

4. Results

4.1 Performance

When running our GALAPACOS program, the performance lived up to our expectations. In terms of performance utilization with a varying number of cores, the simulation scaled well, and the number of creatures being processed, reached new heights. GALAPACOS refers to this projects simulation program. To fully test the parallelism of the chosen solution, we decided to run the tests without rendering. Rendering is more a benchmark of the computers video card than the CPU, simply chancing the position of the camera can change the framerate drastic.

Since we based this project on Karl Sims' research, we also used him as a benchmark. It should however be kept in mind that his project was running on state of the art hardware, though this was in 1994. While Karl Sims had access to supercomputers, we only had access to home computers. Other programs for evolving virtual creatures exist, but we do not have as much detailed information regarding them, and the ones that we were able to try our selves, did not match the speed of GALAPACOS. The latest similar program [4], did not support the utilization of multiple cores, and thus had a very slow evolution, that was limited to what the program could render. With each creature having a default lifespan of 20 seconds, the simulation-time we measured was 10 seconds on the i7 processor per creature at its fastest. A comment on a video of the program on Youtube gave us an indication of the performance over a longer stretch of time:

> "It is a real shame this isn't multi-threaded, especially since my CPU has 12 threads so less than 10% of the potential computing power is available.I get about 300-1000 fps when it is running the simulation, so I get about 100 generations done in 24 hours, depending on the settings of course."[1]

We think the CPU is a 6-core i7 processor as all i7 processors have two hardware threads per core. It could also be a server CPU, but we deem it more likely to be an i7. Bear in mind that the default generation size is 50,

[1] `http://www.youtube.com/all_comments?v=01rsTBJMmQo`

Processor	Cores	Runtime	Max time pr. generation
Intel i5 2500	4	4:20	
Intel i7 Q720	8	6:43	4.7 seconds
Intel Core 2 Duo T6400	2	40:24	28 seconds
CM-5	32	3 hours	?

Table 4.1: Runtime of the simulator (in minutes and seconds), with similar variables in terms of simulated time, population and amount of generations. The CM-5 is Karl Sims' simulator, added as a comparison.

so this program is very slow compared to the performance of this project. This is by no means a reliable source of information, we will take this as real information as any slight misinformation does not seriously affect this projects results.

Sims' simulations did have some missing information about simulation time, but we knew that he had

> "an evolution with population size 300, run for 100 generations, might take around three hours to complete on a 32 processor CM-5."[10]

for each generation. Combining this with the information from a lecture, where video recording of his simulations are shown, we estimated that the simulations of a creature probably lasted for 10 seconds, but to be sure we assume that all creatures underwent 20 seconds of simulation.

A different test was also run with reducing number of world threads, which was done to measure whether or not the simulation scaled fully to the amount of cores, and give an indication to how much performance was gained for each subsequent thread.

When judging the performance, we take into consideration Amdahl's law[2]. This is a formula made by the computer architect Gene Amdahl, which should predict the maximum speedup achievable by adding any number of processors. This law also takes into consideration how parallelized the code is. We will attempt to calculate how parallel our simulation code is, and from that determine how well our multi-threaded performance is.

4.1.1 Performance Test

The test was run with similar time and population size, and was able to reach the 100th generation in 6 minutes and 43 seconds on the Intel i7 processor. The same benchmark was run on the Intel Core 2 Duo with a runtime of 40 minutes and on the Intel i5 with a runtime of 4 minutes and 18 seconds.

As we can see in Table 4.1, the larger desktop processor i5 performed the best. The two other processors, designed for use by laptops, showed

[2] `http://en.wikipedia.org/wiki/Amdahl%27s_law`

Processor/Cores	1	2	3	4	5	6	7	8
Core i5 2500	2:58	1:31	1:05	0:54				
Core i7 Q720	5:25	3:37	2:51	2:20	2:00	1:52	1:50	1:44
Core 2 Quad Q6600	5:09	2:44	1:58	1:28				
Core 2 Duo T6400	7:40	3:57						
Core i7 Q720*	7:31	4:25	3:00	2:07	2:00	1:50	1:49	1:41

Table 4.2: Runtime of the GALAPACOS program (in minutes and seconds), with varying amounts of simulation threads.

Processor/Cores	1	2	3	4	5	6	7	8
Core i5 2500	1	1.96	2.74	3.3				
Core i7 Q720	1	1.5	1.9	2.32	2.71	2.90	2.95	3.13
Core 2 Quad Q6600	1.88	2.76	3.47					
Core 2 Duo T6400	1	1.94						
Core i7 Q720*	1	1.77	2.51	3.55	3.73	4.10	4.14	4.47

Table 4.3: Amount of time faster, than the comparison. Higher is faster and one is the comparison.

a lower performance. Although the i7 technically had more cores, it only operated on 4 physical cores, and they each ran at a lower clock rate, which resulted in slower performance than the i5. The overall conclusion that we can draw from these measurements are that more powerful processors provide significant faster simulation. We were also able to exceed Karl Sims' runtimes by a large factor. One thing that we discovered, was the fact that the performance of a 32-processor CM-5 [16] was equal to that of our Core 2 Duo T6400[12], in terms of gigaflops. Both systems had a maximum of 1.9 GFlops.

4.1.2 Core Variation Test

The core variance test was measured at the completion of the 50th generations, with 10 seconds of simulation per creature, and a population of 300 creatures per generation.

A graph, displaying this data can be found in C.1

As can be seen in Table 4.3 the performance always increases as the number of cores increase. However, what we are looking for is whether or not the increase is linear, or there is a significant overhead. When we look at the graph, we can see that the performance constantly increases, and does seem to show steady growth, but not linearity. The results also show that there is no one-to-one core/performance scaling. Generally all

the processors have 3.5 times the performance, running on 4 cores, than they do running on 1. Some factors as to why this is not so, could be the single-threaded nature of the generation builder, which runs in between the simulation of the generations. While the time appears insignificant (usually between 0.09 0.2 seconds per generation) this is a potential bottleneck, and the runtime of this will not scale at all. In percentage it is usually between 0.5% and 2%. Of the total time this could add anywhere from 4.5 seconds to 10 seconds to a simulations total time. This results a growing bottleneck, the faster the simulations are completed. Other small non-threaded actions in the program, like the update-loop of the SimGame class, among others, could also add to the overhead. This however is a very small part of the overall runtime.

One measurement that sticks out is the i7, which afterwards had to be remeasured without the built in Turbo Boost enabled, to prevent the automatic over-clocking from occurring. Turbo Boost will over-clock a single core, in the case that the other three are not being used, and thus skewed our results. 3.5 times performance would not be achievable, since the results of a single core had the advantageous result from re-routing of the other cores' current. An interesting observation from this measurement is that, when utilizing the virtual cores (5 though 8) the level of performance exceeds a factor of 4, which is the number of physical cores in the processor.

Lastly we looked at Amdahl's law, which is an indicator as to how much performance can be gained, compared to how parallelized the program is, and the amount of cores the program utilizes. We compared our results with the limits that Amdahl's law provides. If we add the limitations that Amdahl describes, to our graph, we can see that with the exception of the i7 with Turbo Boost, our scaling is to that of 97% parallel code. Achieving this high a parallel code percentage fits well with our perception of the system. Building a new generation takes in general .5% to 2% of the full time for a generation. Adding a little synchronization overhead because of the the creature queue it adds up to almost 3% overhead. This also shows that the synchronized rendering approach is a very high performing solution.

The conclusion we came to after this test, is that there is a measurable overhead, so 100% utilization of the cores was not achieved. However when considering the predictions of Amdahl's law, we saw that our program reached a scaling that was very close to the theoretical maximum performance considering every part of the program is not parallel. Had the remaining parts of the program also been parallelized the overhead could be reduced.

4.2 Creatures

Because of the element of randomness in evolution, creatures not only will develop unique features, but the same creature can develop differently without any outside influence. Therefore true comparable scientific data is hard to produce. We can however present different creatures, and analyze how well they adapted over a number of generations.

4.2.1 Creature Appearance

Judging the look and ability of the creatures, in relation to how natural they are, is very subjective. Here we will show certain creatures and reflect upon their looks, but not provide any scientific measurements or tests.
The phenotypes of GALAPACOS, are only boxes connected by thin lines, that are supposed to represent real creatures. While imagination enabled us to see something resembling real creatures in the models, in reality, the creatures are not very realistic in terms of phenotypes. It could be tempting to say that the boxes and joints instead provide a skeleton, which real creatures pretty much always has. These skeletons would however then be missing a big part of their collision, since boxes are only able to interact with the ground via the boxes, and not the joints. The creatures also don't have collision internally, so boxes are allowed to overlap. This however did not subtract much from the believability of the creatures, since it provided for less boxy silhouettes, and more freedom of movement for the creature. Another reason for removing the internal collision, was the glitches in the BEPU engine, where boxes would fly away at great speed, if they were overlapping. Another bug we encountered, was that of very short joints (near zero length) where the connected boxes would be able to turn move outside the limitations that the joints provided. While this resulted in new interesting creatures, the glitchy nature of the joints, made the movements uncontrollable to the creature. Their movement ended up being random, even after extensive evolution.

One observation that was made, was that increasing the amount of parts on the creature, also resulted in more complex creatures, that was harder to identify as living creatures, but though evolution, and luck some creatures could be made, that had remarkable resemblances. The downside of a creature of many parts is that the movement becomes very complex, and the neural network seems to have difficulties keeping up with the growing number of joints. As a result movement becomes jaggier and more jittering. Due to cross-breeding creatures were also able to develop parts, with unnaturally long joints.

To summarize: the final version of the program has no discovered bugs in terms of creature appearance and physical behavior. Therefore the creatures now act closer to that of natural creatures, but the natural appearance is

very limited. Life-like creatures are possible to create, but requires that the creature consists of many parts. Oppositely movement became less fluid and natural with the addition of more joints.

4.2.2 Creature Evolution

The evolution of the creature population is a tricky thing to monitor, since chance can drastically alter the course of a population, and observing every single creature is simply out of the question. Instead we give an impression from our observations, as well as any measurements we can make in the population.

Each selection, fitness and cross-breeding algorithm implemented will be tested, and compared with each other.

Fitness Algorithms

The fitness algorithms need freedom, in order to truly allow the resulting behavior, to shine though. Ex. the minimum number of parts needs to be removed, and the simulation-time needs to take into consideration what fitness we are looking for, since the fitness algorithms at present only measure the end state of the creature.

First a test was run for each fitness algorithm, with varying amounts time.

The observation made was that creatures acted and evolved differently depending on the active fitness function. In the case of Average Distances, we observed a tenancy for the creatures to gradually move in a straight line, from moving in circles, as phenotypes began to dominate the population. This was especially visible in the case of Z Distance-fitness, where the creatures are only measured in one axis, and movement in a straight line provides a greater benefit. In the case of Total Distance and Average Distance w/ Weight Bias, the behavior was quite surprising. While Total Distance was first created as the intended Average Distance, but due to the distances of parts being added instead of averaged, creatures were created, that simply grew in size and parts, and saw no need to develop any intelligent movement. Instead of this behavior, we noticed a tendency to create creatures with two parts, and just like Average Distance develop efficient movement. This same tendency was also noticed in Average Distance w/ Weight Bias. Reducing the simulation time to 10 seconds showed behavior closer to our expectations, but rarely showed creatures with more than 3 parts.

Ground Touching also had the same habit of creating creatures with just 2 parts. These creatures however evolved to move forward, with a higher jumping-motion than other creatures.

A very surprising result we found, was the simulation of the Height fitness. When the program started, an unusually high performance seemed to

Fitness Function	Explanation	Simulation Time (seconds)
Average Distance	Measures the average distance of the creature's parts from the start position (X and Z axis)	20
Average Distance w/ Weight Bias	Measures the average distance, and multiplies by the amount of parts on the creature	20
Ground Touching	Measures average distance, and reduces fitness to 0 if the any part of the creature is touching the ground	10
Total Distance	Measures the combined distance of the creature's parts from the start position	20
Z Distance	Measures the average Z distance (depth) of the creature's parts from the start position	20
Height	Measures the average Y distance (height) of the creature's from the ground	5

be present in the non-rendering mode, and when we switch to a rendering mode, we could see that simulations either lasted a fraction of a second, or the frame-rate would have to have gone though the roof (yet the frame-rate counter said otherwise). Once we were able to move the camera to point at the creature, we we struck by how simply our system was abused. To increase performance, simulation of a creature will terminated, of it is detected that the creature only consists of a single part. Since all creatures will be spawned at a fixed height, the algorithm exploited this, and the one-parted creatures gained an advantage over the creatures that had fallen to the ground. This was a great example of exploiting bugs and loopholes to gain a higher fitness.

After the loophole was corrected we observed a the intended behavior. Creatures that would appear to jump, were created. Surprising was it however that these jumping creatures preferred 3 pars, and never developed any efficient 2-part creatures. Apparently the a 3-part creature can reach higher, by executing backflips, while pulling the lower bodyparts with it, to a higher area. Due to the gravity, the creatures would also have to execute two jumps in the 5 second lifespan, since the height is measured at the end of the simulation.

In conclusion, we have observed that creatures are able to create behavior though our virtual evolution, that changes depending on what fitness is being measured. The behavior also follows a somewhat predictable pattern for each fitness function. Furthermore creatures have a tendency to be smaller, and have fewer parts. Creatures with many parts are difficult to produce, even with fitness algorithms that try to favor these. To increase the number of parts, some sort of manual intervention, or tampering with mutation needs to be implemented.

Selection Algorithms

Three selection algorithms were implemented in GALAPACOS: Probability Tournament, Size Tournament and Random Selection. Each will be run 3 times in the simulator with similar settings for 40 generations, where the population fitness will be measured at the 3rd, 10th, 20th and 40th generation on different statistics.

Figure C.3 As can be seen from the results in the table, the highest fitness obtained from these three methods of selection, do not deviate much from each other. However Size Tournament did receive the highest results after 40 generations, and random the lowest, but the difference seems small. But Size Tournament does separate itself, by quickly having reached the same solution, that also dominates after 40 generation.

Even more interesting is the data for Size Tournament in the average and median. We can see that the median and average quickly approaches the same as the best result, which is an indication that the population is

"overrun" by that creature. Domination often leads to less stimulation for further development, and it is probable that there after 40 more generations would not be much more development.

Opposite Size Tournament, Random Tournament has massive deviance in its population. Even after 40 generations. Here further development is more likely to occur, but the random nature of the selection (only saved by the implementation of Elitism "Elite", section 4.2.2) makes the further development very unpredictable, and as we can see from the median, the population is rather low in fitness, even after 40 generations.

In early development, as a response to these extremes of selection, we created the Probability Tournament, which was supposed to lead the population on a path, between the two extremes. Further evolution should still be stimulated, but the population had to follow with the best creature to a degree. The results show that Probability Tournament is a middle way between the other two selection algorithms, but that the results also seem to suffer in terms of best fitness.

In conclusion we can see that Random Tournament is able to produce good results, but lacks in terms of good fitness in its diverse gene pool. Size Tournament ends up with the best fitness results, but lacks a diverse gene pool. Probability tournament has a diverse gene pool, with high fitness, but does not end up producing fitter creatures than the Size Tournament.

Elite

Elitism covers two areas in terms of selection. First of all elite in our program refers to the saving of the best creature, but also quarantine. We think of it as the memory of a record holder. The record holder may have had children, but if those children do not carry on the good genes, they are pretty much lost. Elite copy however is more like knowledge that is stored. Instead of quarantining the best creatures, they are copied for each generation, so the good genes will always stay alive. To test the effectiveness of these two additions to the selection algorithms, we tested each selection algorithm with 5% of the population reserved for the elite, but without the elite copy. Then another test with the elite copy, and finally a test with neither elite selections.

As we can see in Figure C.2 the lack of memory in the generation, except that of the DNA, shows its downsides, since no steady development in the population is visible. Even with the size tournament, no offspring is guaranteed to have efficiently assembled genes, and otherwise amazing results can quickly get lost in the following generation.

With the addition the elite, but without the elite copy, we can see that evolution now is guaranteed Figure C.2. in terms of best fitness. Keep in mind that the general population still follows the same trends, as the simulations without any form of elite. The observed average and median values was ranging from 5-12, and never grew any further.

The final test had median and average results similar to that of the selection algorithm results in "Selection Algorithms", section 4.2.2. Here the population also climbed steadily, but the average and median values also climbed as the top fitness values grew. This was the result of good fitness genes being forced to stay in the population in later generations. As we can also see from Figure C.2 improvements to the creatures were found faster than the simulations without elite copy. This was most likely due to the saved DNA, that was being preserved in the population.

To conclude on our selection algorithms in combination with elites, we have seen that the choice of selection algorithm can have an impact on the best fitness of a population, but that the fitness needs some sort of undying elite to remain among the creature population until fitter creatures are produced, and take over as the elite.

5. Evolved Creatures

As an extension to our results chapter, we present the results that we did not find appropriate to include as tests, since they were all the outcome of chance. This is chapter devoted to showing and analyzing the creatures we have developed during the production of GALAPACOS.
A collection of creature videos, including those mentioned here, can be viewed on the YouTube channel: "galapacosproject"[1].

5.1 The Evolved Creatures

5.1.1 Runners

Here we will discuss the best creatures that were the result of fitness algorithms designed to create creatures adapted to running.

The first race is an example of what we call a "motorcycle creature" due to the long shape, and method of propulsion. It has two touching points, and uses the back end to push itself forward. This creature has been very common throughout the project period, and has appeared in many variations, but always has the above mention properties. Examples of this creature can be seen in the "Motorcycle.avi" video supplied on the CD, and Figure B.2 in the appendix. One observation about this race is that it always develops a longer joint for the back-part, and keeps all other parts relatively close. This design is very efficient since the longer joint provides a farther push, and the front shape helps the creature not suddenly standing up and jumping backwards.

Other evolutions of the motorcycle creature can be seen on the YouTube channel.

A second very abundant race is the "head-flipper". This race is made up of creatures that have developed a concave body, with a head at one end, that tilts back and forth, and makes the creature jump in that direction. Examples of the head-flipper can be found on the CD as "head flipper 2.avi" and Figure B.1. The head-flipper race also has a tendency to have a long "neck" between the body and head, the development of which provides a

[1] `http://www.youtube.com/user/galapacosproject/videos`

higher and further jump. The Concave body is used to both tilt the body forward on landing as well as providing a good timing for the head flip, as a special part of the creature touches the ground. A rather odd headflipper was also observed, with a head actually resembling a flipper. This is the video included on the CD.

The "backflipper" is a creature that was created via first developing a creature suited for jumping. By changing the fitness function to instead measure distance, the creature quickly learned to jump in a direction rather than upwards. The resulting creature is one of extraordinary balance and timing. Two large "legs" prevents the creature from falling over, and the middle-body is built to support in the flip. One of the feet is also used at the contact sensor that starts the execution of a flip. This creature can be observed in the "backflipper 1.avi" file included.

This next, rather inefficient, walking creature is mentioned due to its appearance. Most people who has watched it in action seem to identify it as a cute creature with two big feet, a body and head. Once it is spawned, it corrects itself to an upright position, and proceeds to jump forward, and uses its two feet to not lose balance. The head and neck provide the jumping motion. We have called this creature "cutie". "Cutie.avi" is included.

5.1.2 Jumpers

Usually though the evolution of a population, one race of creatures seemed to emerge every time. This the jumping version of our "backflipper". This creature uses its long body/neck to create a whiplash that provides upward momentum. Once the creature is in the air, is curls its body, to maximize the average height. This motion also provides a backwards roll. If the creature lands, it will straighten back out and execute more backflips. This creature is included as a video on the CD, and Figure B.4 in the appendix.

One creature that diverged from this trend was the "spin-jumper" ("Spin-jumper 3.avi" on the CD). The creature had a circular shape of boxes, and joints that appeared to meet on the middle. To execute a jump it would spin most of the body around the center part of the creature and stop them, to leave the ground. Since one of the parts being swung around was very heavy, the stop would provide a catapult-like effect, and pull the creature off the ground. Another "leg" of the creature would also be used as part of the catapulting, by hitting the ground near the end of the catapult, to also provide a push off the ground, in a addition to the upwards pull.

5.1.3 Special Creatures

This section is reserved for creatures that require special mentions, or simply blew our minds.

The first creature on the list is the "worm" which we have seen in two

different versions ("Rolling worm.avi" and "Worm.avi"). This creature owes its shape to the restraint of only producing new parts along a single axis. Once the creature spawns, the joints contract, and provides the worm like S-shape. This shape then proceeds to move forward, by producing wave-like motions.

A different worm evolved with a different motion for forward momentum. This creature Closed itself into a C-shape upon spawning, and once it contacts the ground it opens up into a 3-shape. During the jump it closes upon itself again, and proceeds to land and roll using its now round body. When the time is right, is straightens itself out again, and repeats the motion, to jump and roll forward.

The third special creature we would like to present, is the jumping horse. This is the result of the creatures being forced a minimum of 10 shapes. The resulting creature is what we refer to as the "mad horse". The creature has two legs pointing to the sides that provide support. Two other legs point perpendicular outwards, compared to the to first legs, and these have movement. They rock up and down like a see-saw, and this motion we see as resembling that of an angry horse, jumping and kicking. While the physical shape probably looks more like some other creatures, we find this resemblance rather amusing, and hence why we included it and its peculiar motions. This creature can be found in the video "Mad-horse 1.avi" on the CD, and Figure B.5 in the appendix.

The final creature is our prized trophy. At the beginning of the project, we were told that getting a four-legged creature to stand up was a marvelous feat, and we immediately set ourselves the goal of doing just that. This creature is not a result of that goal, but a result of chance and our program creating a creature that appears to be walking on four legs. The creature is included as a .crit file on the CD, that can be opened though the GALAPACOS .exe executable. The creature that has been triumphantly been named "4-legged" even though actual 4-legged movement is open to discussion. The creature has a flat crab-like appearance, with a small core, as its central body, and six boxes surrounding it. Two of the boxes are big, and being held off the ground, at the "front" of the creature, and are used for rocking the creature from side to side. These two "antlers" very rarely touch the ground, and are therefore not considered as legs. The remaining 4 legs are two thin front legs, and two cubes as hind legs. These legs move in sync with the rocking motion, and are used for moving the creature forwards. While this is nowhere near the 4-legged creature we had in mind, it was an example of very complex movement. Videos of the creature can be found on the YouTube channel, and a screenshot is provided in part Figure B.6 in the appendix.

6. Future Work

Future work for this project would in large part be to remove the limitations described in *Solution Limitations*. The most important limitation to overcome would be the creation of the phenotype. Symmetric limps, like elbow joints, feet and legs would be interesting to experiment with. Experimenting with creatures with features similar to a four legged creature or perhaps a centipede could provide interesting work.

Manually creating a phenotype and disallowing mutating of it, but still allowing mutation of the neural network could give insight into how now extinct creatures walked. Creating the phenotype similar to a Tyrannosaurus rex or Brontosaurus and evolving how they walk, could give interesting insight into how they walked millions of years ago.

Evaluating the fitness creatures fitness continuously could give a more balanced view of the creatures. I.e. a creature moving in circles could be given evaluated for the sum of the movement, not just the final result.

A simple improvement would be the addition of some reference points in the world. Implementing shadows could improve the observability of the program. Adding reference points to the world could also help.

7. Conclusion

To summarize, we created a simulation of Darwinian evolution with use of parallelism to speed up the simulation. We reached all our required goals, but did not accomplish any of the additional goals.
The created program's performance was close to linear scaling up to a least four cores, no higher amount of CPU cores were tested. Extra hardware threads per core scaled poorly, as expected. We were able to create virtual creatures in the simulated world through mutation, and different selection-, fitness- and crossover algorithms. Thanks to a virtual brain, simulated using a neural network, these creatures achieved both simple and complex movement. We evolved creatures that successfully adapted to moving and jumping. Totally random evolution did not produce creatures with both good capabilities and complex appearance, but limiting the boundary of the mutation produced better results.

We also showed the impact that the selection algorithms and elite selection had on the population, and the diversity of creatures. Some limitations were met during the development and some desired features did not end up being implemented, but would be interesting to improve in future work.

Some videos can be seen on the included CD, but due to the limited CD size all videos have been uploaded to Youtube.
`www.youtube.com/user/galapacosproject/videos?view=1`
Short link: `goo.gl/qZHqL`

Bibliography

[1] Mat Buckland. Building better genetic algorithms. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 649–660. Charles River Media, 2004.

[2] Epic Games. Unreal engine 3 collision reference. `http://udn.epicgames.com/Three/CollisionReference.html`, 2012.

[3] Joe Kiniry. DALi: Distributed Artificial Life. `http://kindsoftware.com/documents/talks/DALi_talk_ITU.pdf`, 2001.

[4] Maciej Komosinski and Szymon Ulatowski. framsticks. `http://www.framsticks.com`, 2009.

[5] Maciej Komosinski and Szymon Ulatowski. framsticks. `http://www.framsticks.com/a/al_geno_f0.html`, 2009. the artificial DNA used for their project.

[6] BEPU physics. Bepu physics documentation - joints and constraints. `http://bepuphysics.codeplex.com/wikipage?title=Joints%20and%20Constraints&referringTitle=Documentation`.

[7] Microsoft Research. Practical parallel and concurrent programming. `http://ppcp.codeplex.com/`, 2010.

[8] Karl Sims. Evolved virtual creatures video. `http://www.youtube.com/watch?v=JBgG_VSP7f8`, 1994. Video showing paper results.

[9] Karl Sims. Evolving 3D Morphology and Behavior by Competition. `http://karlsims.com/papers/alife94.pdf`, 1994.

[10] Karl Sims. Evolving Virtual Creatures. `http://www.karlsims.com/papers/siggraph94.pdf`, 1994.

[11] Karl Sims. Evolved virtual creatures interview. `http://www.youtube.com/watch?v=b1rHS3R0llU`, 2008. Interview video showing paper results.

[12] SiSoftware. Power efficiency fpu benchmark. http://www.sisoftware.eu/rank2011d/top_device.php?q=c2ffc6e092af9eb8d1ecdef890ad9cbac2ffcee88de8d5e3c5b68bba89&l=en, 2009.

[13] Tim Sweeney. The next mainstream programming language: A game developers perspective. http://www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/docs/sweeny.pdf, 2005. presentation.

[14] Penny Sweetser. How to build evolutionary algorithms for games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 627–639. Charles River Media, 2004.

[15] Penny Sweetser. How to build neural networks for games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 615–625. Charles River Media, 2004.

[16] Top500. Cm-5/32. http://i.top500.org/system/167041, 1994. Showing GFlops of the supercomputer Karl Sims used in 1994.

[17] Mick West. Programming responsiveness. http://www.gamasutra.com/view/feature/1942/programming_responsiveness.php, 2008.

[18] Wikipedia. Dna. http://en.wikipedia.org/wiki/Dna.

[19] Wikipedia. Physics engine. http://en.wikipedia.org/wiki/Physics_engine.

Full Genotype

Part definition

Example:
p:0,t=h,x=0,y=-35,z=0,sx=4,sy=4,sz=4,ox=0,oy=0,oz=0,m=20,kfr=9.636603,sfr=10

Type=p:
id=-1 - Unique id
t=h,b,l - Part Type (head, body, limb)
x=0,y=0,z=0 - position in 3D space
sx=1,sy=1,sz=1 - sizes for each axis
ox=1,oy=1,oz=1 - orientation axis - Not used m=1 - physical mass
kfr=1 - kinetic friction
sfr=0.4 - static friction

Joint definition

Example:
j:s,id=0,p1=0,p2=1,jx=0,jy=0,jz=0,rx=0,ry=0,rz=0,fmin=-1.256637,fmax=1.256637,hforce=10000,s

Type=j:
id=-1
t=s - Join type - only swivel hinge is used (revolute like with slightly more freedomg)
p1=-1,p2=-1 - id of the two connected parts.
jx=0,jy=0,jz=0 - joint offset - not used
rx=0,rx=0,rz=0 - rotation-axis: Not used
fmin=-1 - Joint min angular freedom.
fmax=1 - Joint max angular freedom.
hforce=10000 - strength for the joint
speed - how fast a joint can move
fa=false - should the joint be flip? Meaning min angle becomes max and visa versa

Neuron definition

Example:
n:1,c=input,t=Sigmoid,p=0,j=0

Type=n:
type=Sigmoid - neuron type. Could also be wave, gaussian or threshold
c=category - normal, input or output neuron
id=-1
p=-1 - id to the attached part - not used.
j=-1 - id of the joint attached to

Connection definition

Example:
c:sid=9,tid=15,w=1.086872

Type=c:
sid - id of the source neuron
tid - id of the target neuron
w=1.0 - connection weight

Creature Snapshots

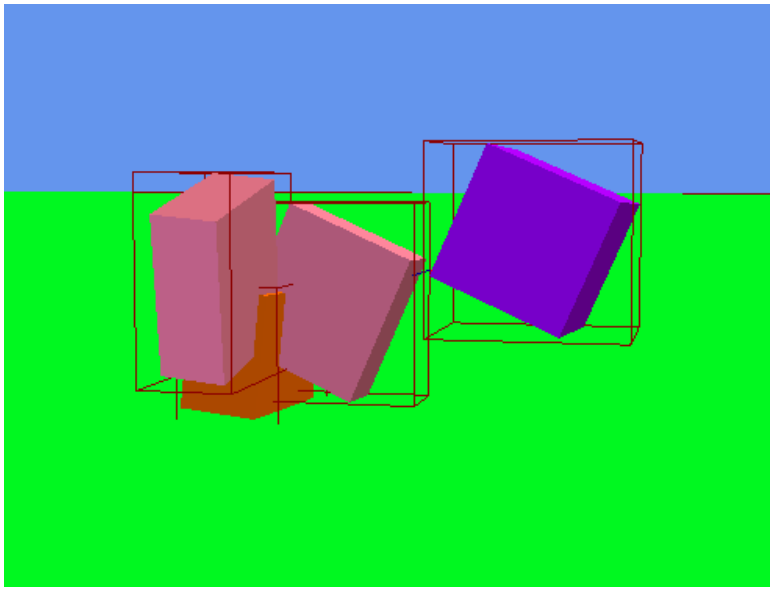


Figure B.1: The head flipper creature. The head is the rightmost box.

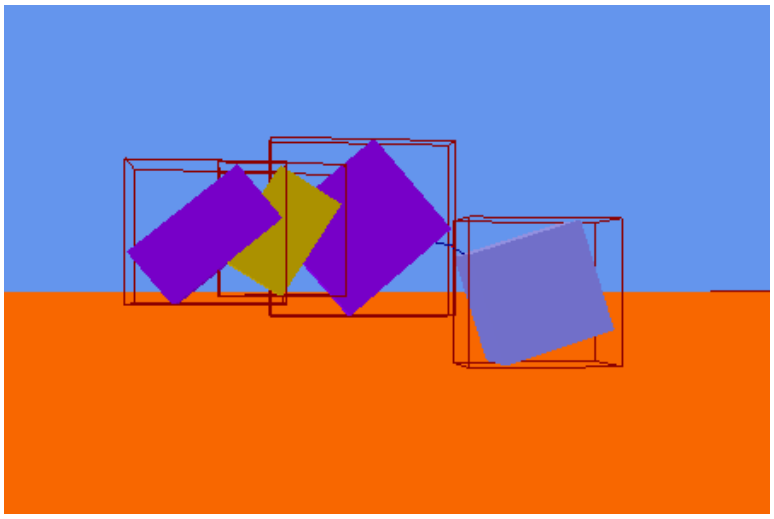


Figure B.2: The motorcycle creature. The rightmost part provides propulsion.

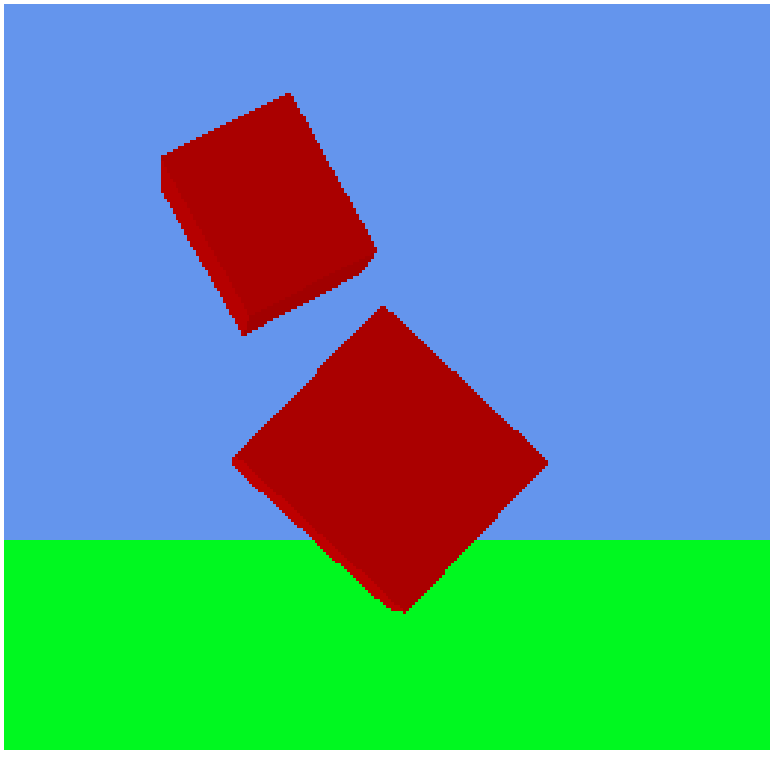


Figure B.3: Two-parted walking/jumping creature. Top part nods up and down to jump/move.

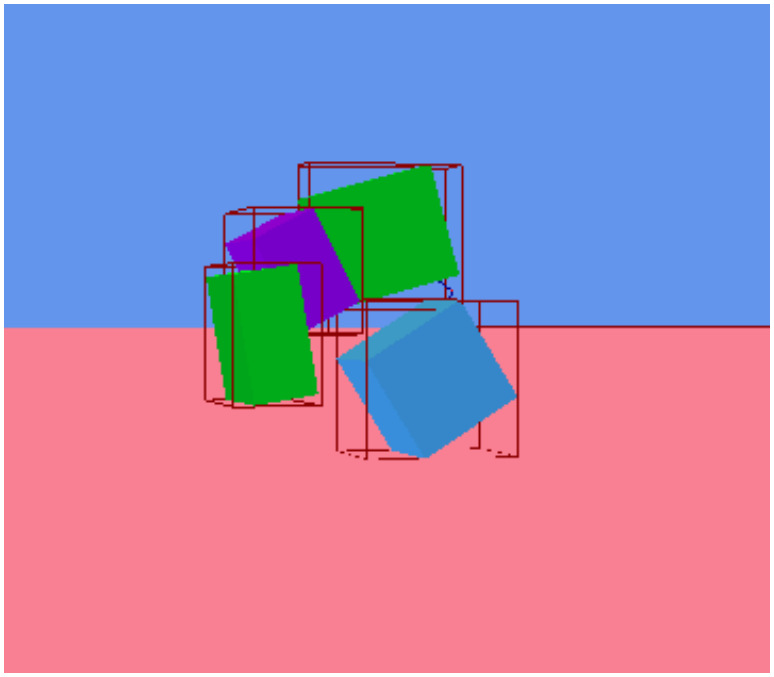


Figure B.4: Backflip-jumper. Stretches the into a straight line to jump upwards.

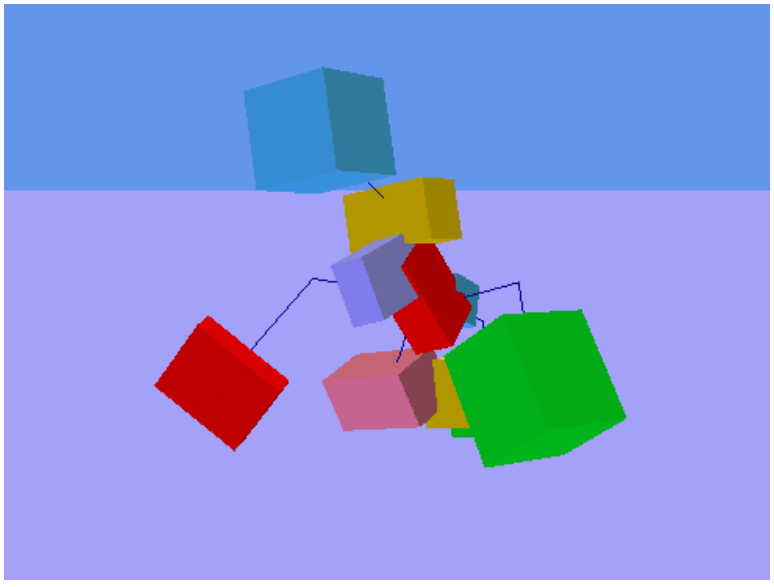


Figure B.5: The mad horse. Leftmost and rightmost boxes are supporting legs. Highest box is head, that moves up and down with the rest of the body.

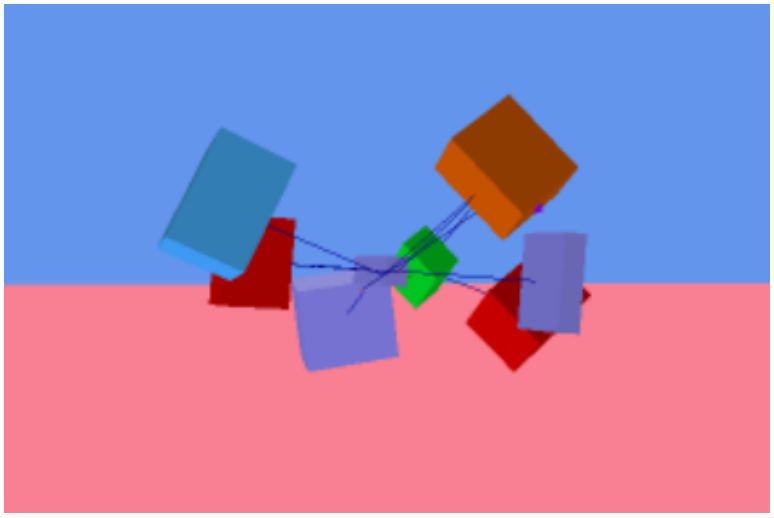


Figure B.6: 4-legged creature. Two highest boxes are the antlers for rocking from side to side. Creature is here "looking at the camera". Two smallest boxes make up the core body.

Performance Results

C.1 Core Variation Results Graph

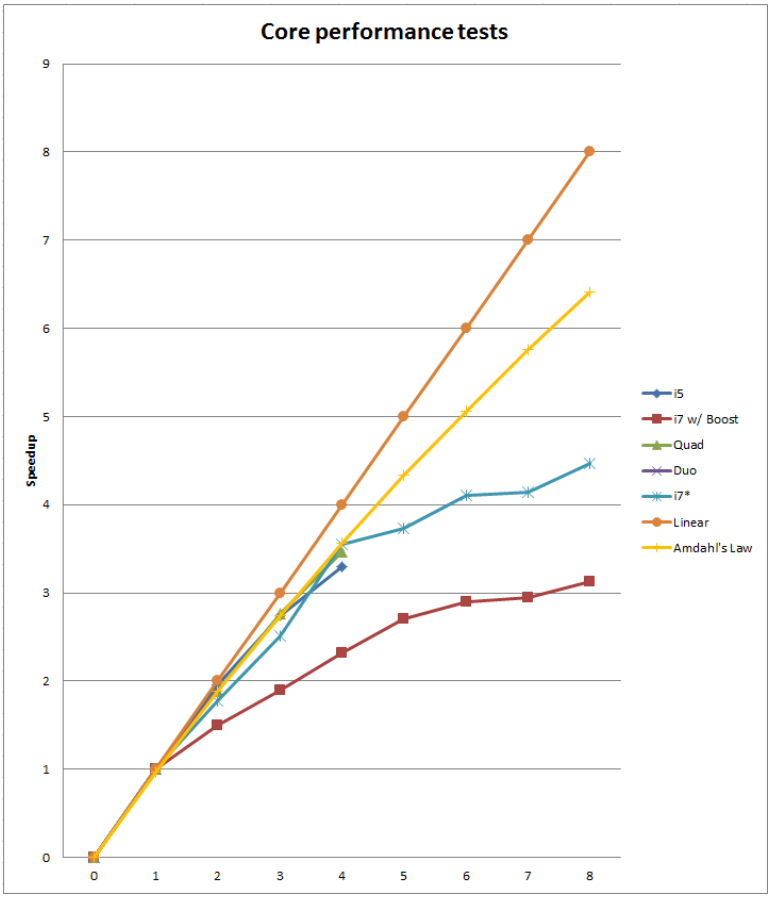


Figure C.1: Results of the core variation test. i7* = i7 with no Turbo Boost.

C.2 Selection Test Graphs

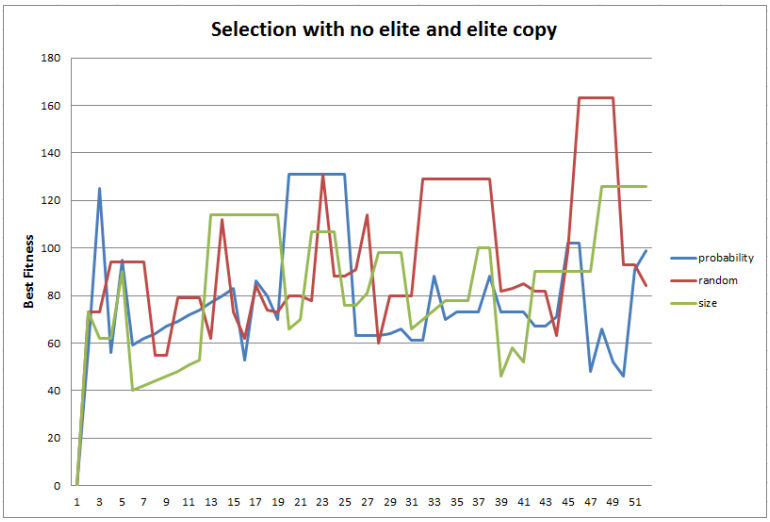


Figure C.2: Results of test with 0 elite percentage.

C.3 Selection Comparison Results

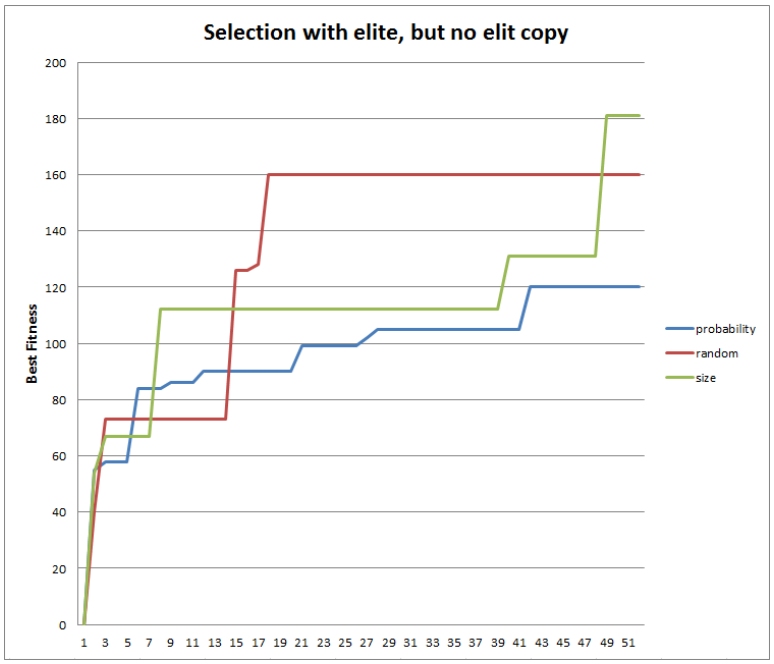


Figure C.3: Results of test with 5 percentage of 5, but no elite copying.

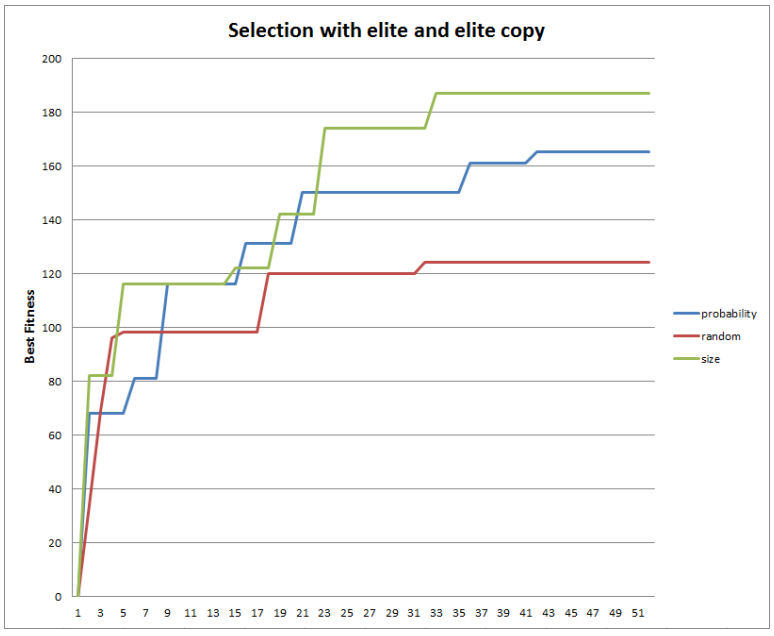


Figure C.4: Results of test with 5 percent elite, and elite copy.

	Best				Average				Median			
	3	10	20	40	3	10	20	40	3	10	20	40
Random	63	108	130	152	4,88	20,01	32,18	41,46	5,3	6,11	8,76	15,36
Random	101	108	133	163	6,1	18,94	33,69	57,1	5,03	6,2	6,74	22,68
Random	91	91	118	157	4,74	12,59	17,57	44,99	5,14	5,96	6,02	8,28
	85	102,3333	127	157,3333	5,24	17,18	27,81333	47,85	5,156667	6,09	7,173333	15,44
Size	70	123	155	188	7,71	25,92	69,99	109,22	5,52	6,64	70,61	155,47
Size	153	153	153	162	9,54	35,68	86,48	143,09	5,32	8,31	153	153
Size	67	158	158	158	8,12	34	81,81	152,79	4,49	6,59	52,89	158
	96,66667	144,6667	155,3333	169,3333	8,456667	31,86667	79,42667	135,0333	5,11	7,18	92,16667	155,49
Probability	57	93	123	178	6,22	22,14	49,01	115,6	5,28	6,43	41,13	123,72
Probability	62	87	119	161	7,04	22,15	48,14	82,32	5,46	6,38	25,75	119,75
Probability	91	93	154	160	7,88	28,82	63,17	124,29	5,32	6,54	61,91	154,88
	70	91	132	166,3333	7,046667	24,37	53,44	107,4033	5,353333	6,45	42,93	132,7833

Figure C.5: Results of test, comparing different selection algorithms.

Miscellaneous

Figure D.1: Human approximated via simple shapes. `http://udn.epicgames.com/Three/PhysicalAnimation.html`