Kungliga Tekniska Högskolan (KTH) - DD143X - DKAND12

Silverfish Simulation

Degree Project in Computer Science, First Level

Dennis Johansson
Vadagränd 8, 194 55 Upplands Väsby, Sweden
dennis.j.johansson@gmail.com
Supervisor: Henrik Eriksson

2012-04-11

This document focuses on simulation of silverfish in an application that makes use of a simple method, which is proven to be inadequate. With this method the genes are represented by integers that are used by an automaton to control the silverfish. A foot killing silverfish is proven to be enough to alter the evolution.

Table of Contents

Tables

Figures

1 Introduction

This essay targets simulation of silverfish with focus on an application that I've made for the purpose of this essay. The application is used to discuss the possibility of simulating silverfish through the use of a simple automaton and a few variables representing different genes for each silverfish. These variables may differ for each spawned silverfish to simulate mutations in the genes and allows for some interesting results. To make it further more exciting we introduce a foot that can be controlled either by an artificial intelligence or by interaction, and is used to stomp the silverfish to death.

2 Problem statement

The main problem, simulating evolution, requires natural selection for the silverfish. Usually this requires silverfish that can move around and eat food, and finally reproduce. However to make things a bit less mainstream we introduce a flashlight and a foot, which represent a human trying to kill the silverfish in a dark room. Non-lit silverfish cannot be seen, and are considered safe. The final results should help answering whether it really is possible to simulate evolution with simple tools and if the foot truly alters evolution.

2.1 Application

The purpose of the application is to simulate silverfish evolution. To do this we need an environment that can be controlled by the application in real-time. We also require silverfish and food in this environment where the silverfish neither survive nor reproduce without food. To introduce evolution the silverfish must mutate, which can be done in three different ways brought up in section 3.1. Once we have silverfish in an environment we introduce a stomping foot to alter their evolution.

We further increase the realism of the application with an interface where the user can control both the flashlight and the foot. We all know that evolution requires many generations to make things interesting, which calls for an artificial intelligence to control the flashlight and foot until the user is ready to take over.

3 Background

Simulation of silverfish is not much different from what‘s already been done, however interaction with the evolution is rare to say the least. Therefore the difference comes with the foot that interacts with the simulated evolution. The application (especially the interface) is inspired by a similar application [1] which has critters that reproduce when they have enough energy and mutate in different directions. In that application you can introduce a wall segment that the critters can collide with, which is very similar to the foot.

3.1 Methods

There are three major solutions to this problem; parameter based and module based simulated genes along with artificial neuron networks. I understood the approach with simulated genes best however neuron networks caught my interest, as it's a versatile tool that can be applied to almost anything [2].

3.1.1 Simulated genes

Simulating genes is a simpler technique than neuron networks. While it's easy to implement, it has consequences, mainly you have to define exactly what the genes affect, which can be difficult to know beforehand. Predefined genes cause two problems, it's difficult to get it right, and it may also lower the quality of the end result. Finally a pool of genes may cause problems when different genes coincide in a bad manner.

3.1.1.1 Parameter Based

The parameter based approach is by far the simplest. Each gene is represented by a parameter, which in turn alters the behavior of the objects. Mutation is very easy to implement with parameter based genes as they can be anything between their minimum and maximum value respectively.

3.1.1.2 Module Based

This is the approach used by the similar application [1]. A module represents a gene, which can either control parameters, such as increased lifespan or more directly via behavior. In an example on Wikipedia [3] a module controls a leg of a creature to move it up and down with a frequency. However if this gene makes use of an internal parameter it is useless without another gene to set it. This pool of different modules controls the creatures' behavior. In order for this to have any greater impact we need a lot of modules. Mutations in module based simulated genes describes how we alter the pool of modules, where we can remove or add a module as a mutation.

3.1.2 Artificial Neuron Network

Neuron networks are complex in nature, and therefore very versatile. However the versatility of neuron networks work both in favor of, and against the resulting application. They require a lot of processing time for training [2], which really isn't an option for an interface based application. As emulation is required for a neuron network to function properly [2], along with my inexperience with neuron networks, the end result is hard to determine in advance.

3.1.3 The choice of method

Parameter based simulated genes, the simplest of the three options has one issue; it may be too simple as it's got a finite amount of states, which can be too few with a small amount of parameters. The module based approach gives better results, but may be hard to apply on simple silverfish without implementing a huge amount of modules. Finally artificial neuron networks are just too complicated to work with. The main reason for my choice is time, as both module based simulation and artificial neuron networks may take a significant amount of time to implement they're discarded. So we're left with parameter based simulated genes as they're easy and quick to implement.

4 Approach

As I'm a seasoned C++ programmer it felt like a good choice to implement the application in C++. However the C++ standard does not come with any API for drawing, therefore I used the Windows GDI [4] for drawing, as I'm fairly used to it. The Windows GDI does all the application demands and works well with Windows dialogs. The dialogs are easy to create in visual studio, which is the main reason why the application uses it. Other libraries may not work well with dialog boxes and therefore it's probably a bad idea to use them.

4.1 Application

As seen in Figure 2 there’s only one important hierarchy in the application with *Object* as the base. The purpose of *Object* is to connect the drawing of rectangles with the report list control on the right-hand side of the applications window (see Figure 5 on page 7). The objects have a tag associated with them which is connected to a color in a *ColorList*, see Figure 1. This color is used by the report list control to associate its counts with the drawn simulation on the left-hand side of the applications window, where the silverfish are drawn as colored rectangles. You can only see the silverfish where the flashlight is revealing them; also this is where the foot will stomp.

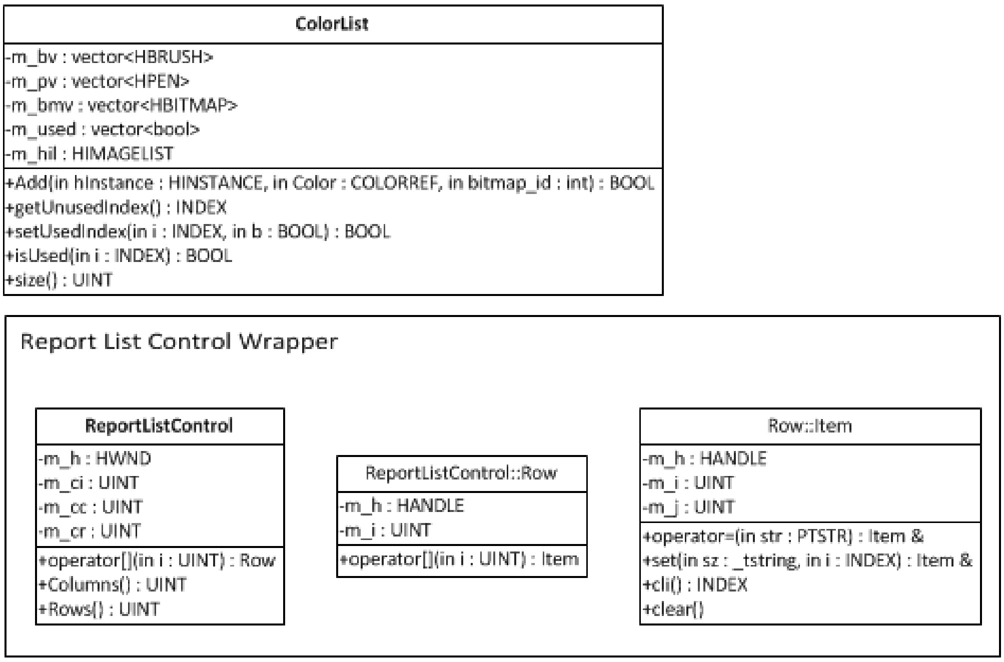

ColorList
-m_bv : vector<HBRUSH>
-m_pv : vector<HPEN>
-m_bmv : vector<HBITMAP>
-m_used : vector<bool>
-m_hil : HIMAGELIST
+Add(in hInstance : HINSTANCE, in Color : COLORREF, in bitmap_id : int) : BOOL
+getUnusedIndex() : INDEX
+setUsedIndex(in i : INDEX, in b : BOOL) : BOOL
+isUsed(in i : INDEX) : BOOL
+size() : UINT
Report List Control Wrapper
ReportListControl
-m_h : HWND
-m_ci : UINT
-m_cc : UINT
-m_cr : UINT
+operator[](in i : UINT) : Row
+Columns() : UINT
+Rows() : UINT
ReportListControl::Row
-m_h : HANDLE
-m_i : UINT
+operator[](in i : UINT) : Item
Row::Item
-m_h : HANDLE
-m_i : UINT
-m_j : UINT
+operator=(in str : PTSTR) : Item &
+set(in sz : _tstring, in i : INDEX) : Item &
+cli() : INDEX
+clear()

Figure 1: Wrappers for the interface objects.

The *Food* class is minimal and only defines the tag for food, logically defined as “Food”; however the *Silverfish* class defines a whole lot more of which is covered in section 4.2. Each of the silverfish owns an automaton described in section 4.3 that controls its actions. Also as the silverfish derives from the *Object* class it must define its own tag; how this is accomplished is described in section 4.1.2.

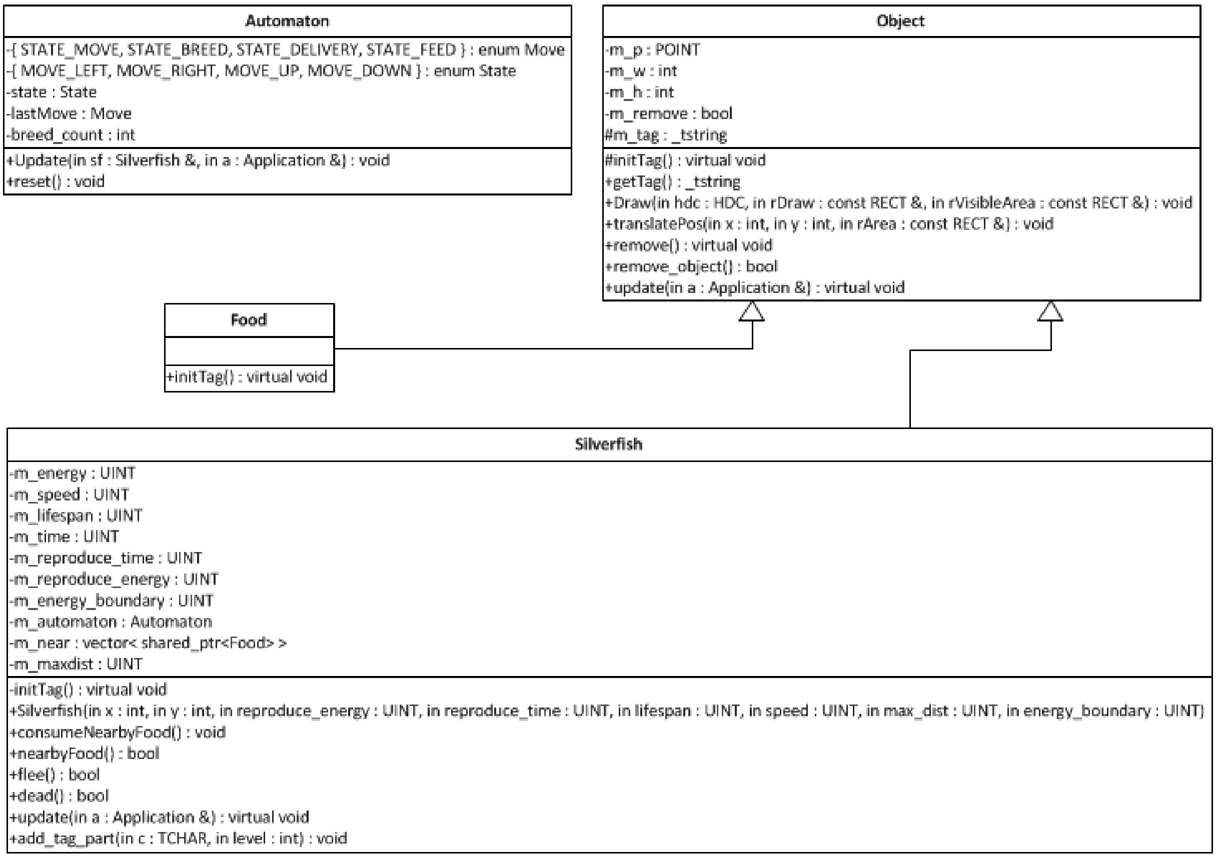
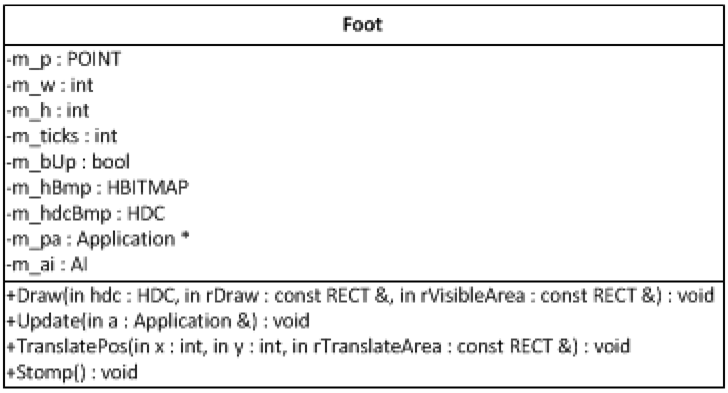
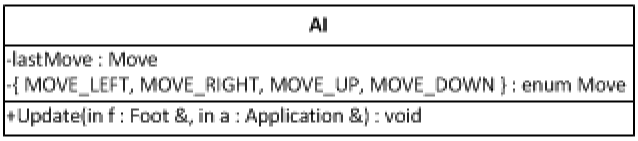

Automaton
-{ STATE_MOVE, STATE_BREED, STATE_DELIVERY, STATE_FEED } : enum Move
-{ MOVE_LEFT, MOVE_RIGHT, MOVE_UP, MOVE_DOWN } : enum State
-state : State
-lastMove : Move
-breed_count : int
+Update(in sf : Silverfish &, in a : Application &) : void
+reset() : void
Object
-m_p : POINT
-m_w : int
-m_h : int
-m_remove : bool
#m_tag : _tstring
#initTag() : virtual void
+getTag() : _tstring
+Draw(in hdc : HDC, in rDraw : const RECT &, in rVisibleArea : const RECT &) : void
+translatePos(in x : int, in y : int, in rArea : const RECT &) : void
+remove() : virtual void
+remove_object() : bool
+update(in a : Application &) : virtual void
Food
+initTag() : virtual void
Silverfish
-m_energy : UINT
-m_speed : UINT
-m_lifespan : UINT
-m_time : UINT
-m_reproduce_time : UINT
-m_reproduce_energy : UINT
-m_energy_boundary : UINT
-m_automaton : Automaton
-m_near : vector< shared_ptr<Food> >
-m_maxdist : UINT
-initTag() : virtual void
+Silverfish(in x : int, in y : int, in reproduce_energy : UINT, in reproduce_time : UINT, in lifespan : UINT, in speed : UINT, in max_dist : UINT, in energy_boundary : UINT)
+consumeNearbyFood() : void
+nearbyFood() : bool
+flee() : bool
+dead() : bool
+update(in a : Application &) : virtual void
+add_tag_part(in c : TCHAR, in level : int) : void

Figure 2: The hierarchy and automaton.

The foot is either controlled by the human or by an *AI*, which uses a tiny automaton with a move and a stomp state. It uses the same move state as the automaton explained in section 4.3. The foot is controlled via the applications **OnLeftMouseDown** and **OnMouseMove** methods respectively (see Figure 4). These are either passed into the application from the user (via an external message procedure) or via the artificial intelligence.



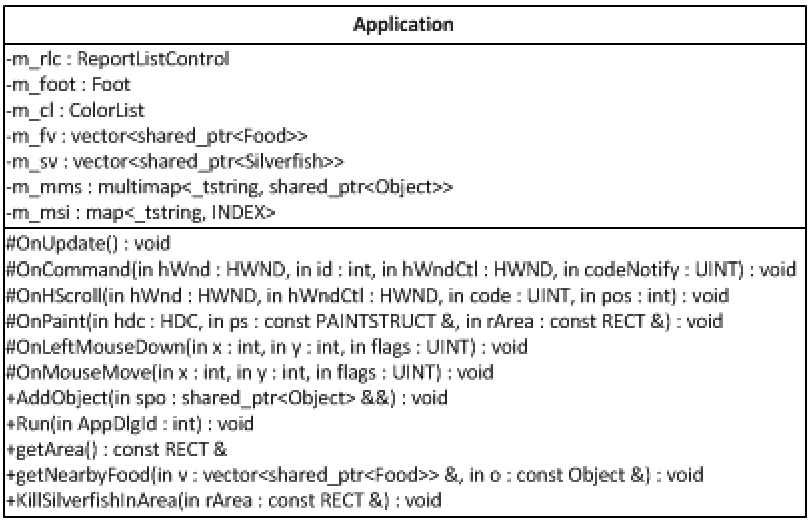
```
Foot
-m_p : POINT
-m_w : int
-m_h : int
-m_ticks : int
-m_bUp : bool
-m_hBmp : HBITMAP
-m_hdcBmp : HDC
-m_pa : Application *
-m_ai : AI
+Draw(in hdc : HDC, in rDraw : const RECT &, in rVisibleArea : const RECT &) : void
+Update(in a : Application &) : void
+TranslatePos(in x : int, in y : int, in rTranslateArea : const RECT &) : void
+Stomp() : void
```



```
AI
-lastMove : Move
-{ MOVE_LEFT, MOVE_RIGHT, MOVE_UP, MOVE_DOWN } : enum Move
+Update(in f : Foot &, in a : Application &) : void
```

Figure 3: The foot and Artificial Intelligence

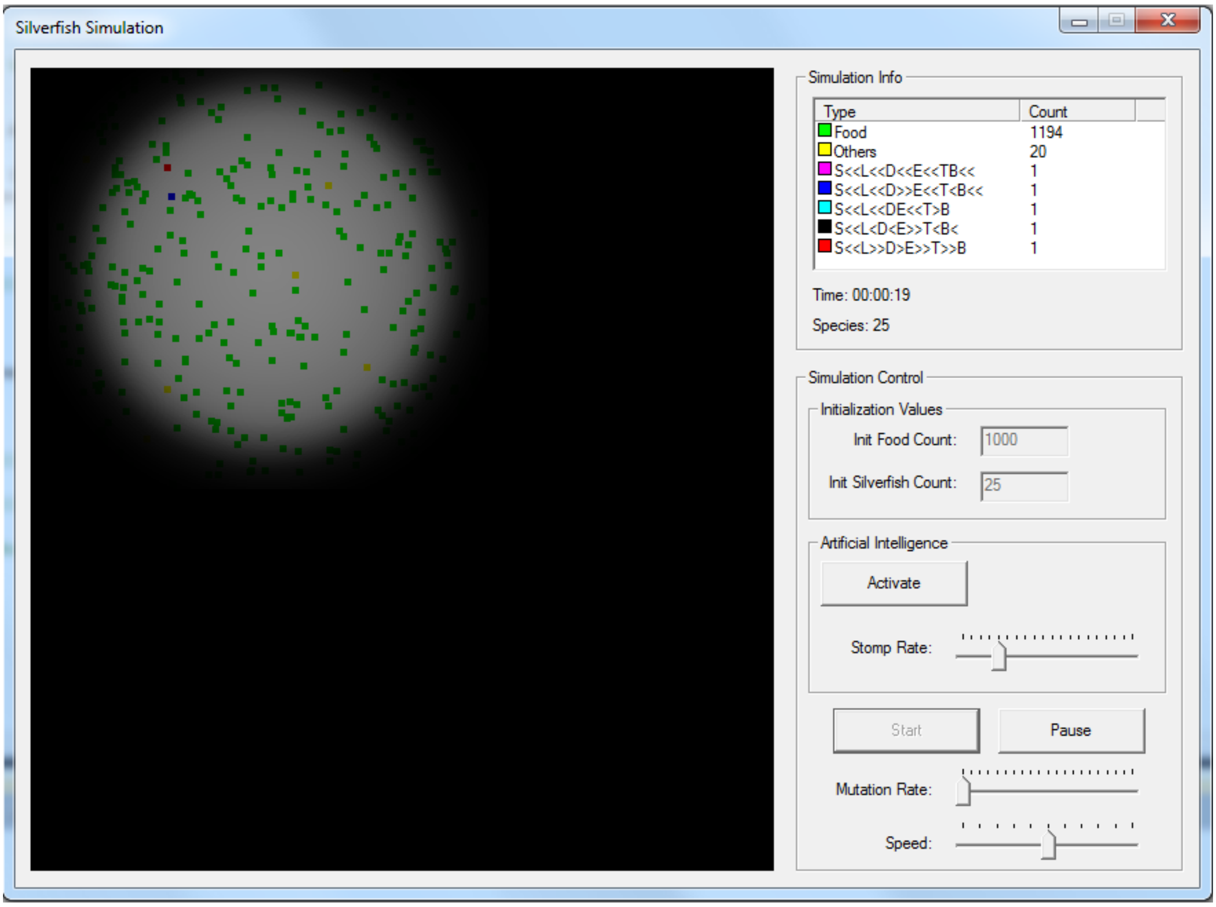
The *Application* class binds all of these components together. The *Application* class handles the controls on the lower right corner of Figure 5 to alter internal variables used by the different classes and also for spawning the initial set of silverfish and food. The speed control defines how often the internals get their update functions called and hence the speed of the simulation. The internals is the three first fields in the *Application* class (see Figure 4) along with the objects in the **m_mms** multimap. The interface is redrawn separately from the updates, which makes the implementation of the pause really easy – stop calling the update functions. A general **AddObject** is defined for adding new objects, as they're shared in three data structures, **m_fv**, **m_sv** and the multimap. Removal of the objects is accomplished in an update via the virtual **remove_object** function defined in *Object* (see Figure 2) which returns true if the object is discardable, or via the **KillSilverfishInArea** method of the *Application* class.



```
Application
-m_rlc : ReportListControl
-m_foot : Foot
-m_cl : ColorList
-m_fv : vector<shared_ptr<Food>>
-m_sv : vector<shared_ptr<Silverfish>>
-m_mms : multimap<_tstring, shared_ptr<Object>>
-m_msi : map<_tstring, INDEX>
#OnUpdate() : void
#OnCommand(in hWnd : HWND, in id : int, in hWndCtl : HWND, in codeNotify : UINT) : void
#OnHScroll(in hWnd : HWND, in hWndCtl : HWND, in code : UINT, in pos : int) : void
#OnPaint(in hdc : HDC, in ps : const PAINTSTRUCT &, in rArea : const RECT &) : void
#OnLeftMouseDown(in x : int, in y : int, in flags : UINT) : void
#OnMouseMove(in x : int, in y : int, in flags : UINT) : void
+AddObject(in spo : shared_ptr<Object> &&) : void
+Run(in AppDlgId : int) : void
+getArea() : const RECT &
+getNearbyFood(in v : vector<shared_ptr<Food>> &, in o : const Object &) : void
+KillSilverfishInArea(in rArea : const RECT &) : void
```

Figure 4: The Application class

The applications interface is split into three separate areas, the drawing, simulation info and the simulation control. The drawing area is used to illustrate the simulation in real-time, and is also used to interact with the silverfish via the foot and flashlight. The flashlight always follows the mouse and the foot stomps wherever the flashlight is when the user hits the left mouse button as described earlier. The simulation info shows a top-7 most common list of objects. If there are more objects than six present in the application the rest will be "summed up" and called "Others". The runtime for the simulation and number of species is also shown below. Simulation control is used to initiate the simulation as well as pausing and/or altering it. The artificial intelligence can be switched on or off on demand if the user wants to interact himself.



Type	Count
🟩 Food	1194
🟨 Others	20
🟪 S<<L<<D<<E<<TB<<	1
🟦 S<<L<<D>>E<<T<B<<	1
🟦 S<<L<<DE<<T>B	1
⬛ S<<L<D<E>>T<B<	1
🟥 S<<L>>D>E>>T>>B	1

Time: 00:00:19

Species: 25

Init Food Count: 1000

Init Silverfish Count: 25

Figure 5: Image of the Application.

4.1.1 Performance

As the application was near completion one could see a bottleneck in the application, where it spent most of its time on handling user timers. It had a massive impact on the performance as more than 90% of the CPU time was spent handling these user timers. User timers aren't designed to be used the way the application demands. There are two ways to solve this issue, either we change the type of timer or we design our own timer. There are two timer choices in the Windows API, waitable timers and thread pool timers. Waitable timers use the asynchronous procedure call (APC) queue on the thread. An APC queue is filled up by the kernel or other threads and freed up by the thread that owns it. As a single thread handles everything when using a waitable timer no thread synchronization is required. Thread pool timers have the advantage of being even more effective as it spreads out the handling on several threads. However as the application wasn't built with thread synchronization in mind, there was but one choice… waitable timers! There is one big problem with waitable timers though. Remember that it's the kernel that is filling up my APC queue with requests. As the application uses four different timers, all firing at given intervals, this APC queue fills up quite rapidly and only the main thread can empty it. However the kernel won't care if the thread can't handle all the requests and continues to spam it with APC requests causing the application's user interface to hang as the thread spends all of its time handling the APC requests instead of user input. As you can see this is a really dangerous solution.

The second choice is to discard the Windows API timers and create my own. The main advantage here is that there's no way to create a deadlock, livelock or hang with this solution. However it does demand a smaller internal rebuild, which is exactly what's been done for this application. This is obviously the best choice given the major flaws of the other solutions.

4.1.2 Tag Explanation and Colors

To understand the test results a thorough explanation of the object tags is necessary. First of all there are two tags that are easily understood, "Food" and "Others", where the tag "Others" is virtual in the sense that it doesn't exist internally, as it's only calculated as simulation info. The tags are connected to colors which are used to draw the rectangles representing objects. Only the top 6 most common objects are assigned a unique color, the rest share the color assigned to "Others". Note that the "Food" tag is not necessarily among these top 6 most common objects and does not always have a unique color.

Each silverfish species has a unique tag assigned to it, which describes the current set of genes. We can split the tag into parts to understand what it tells us. Each part consists of a gene tag (see Table 1) and a representation of the state of the gene, see Table 2.The genes are represented by values internally, so we can split these into five different areas; very small, small, normal, large and very large. So a very fast silverfish would have the tag part "S>>". With this knowledge we can easily see how common a characteristic is among the top 5-6 silverfish just by looking at these tag parts.

Gene Tag Part	Description
S	Speed of the silverfish; the **m_speed** field of the Silverfish class in Figure 2 on page 5.
L	Lifespan of the silverfish; the **m_lifespan** field of the Silverfish class in Figure 2 on page 5.
D	The maximum distance to the center of the light source where the silverfish is scared, see section 4.2 and 4.3 for more information. This is the **m_maxdist** field in the Silverfish class in Figure 2 on page 5.
E	Reproduction energy, the energy required to reproduce, see section 4.2 and 4.3 for more information. This is the **m_reproduce_energy** field in the Silverfish class in Figure 2 on page 5.
B	The energy boundary is the maximum amount of energy for when the silverfish ignores fear; see section 4.2 and 4.3 for more information. This is the **m_energy_boundary** field in the Silverfish class in Figure 2 on page 5.
T	The minimum age before reproduction is possible. See section 4.2 and 4.3 for more information. This is the **m_reproduce_time** field in the Silverfish class in Figure 2 on page 5.

Table 1: A short description of each type of gene, and the letter used by the tags to describe it.

Gene Value	Tag part
Very small	<<
Small	<
Normal	Nothing
Large	>
Very large	>>

Table 2: The tags part that describes the size of a parameter.

4.2 The Silverfish

The silverfish class is quite simple; all the hard work is done for it by the *Application* and *Automaton*. There are three important methods in the *Silverfish* class, the **update**, **nearbyFood** and **consumeNearbyFood**. When the **update** method is called three things are done, first we request the nearby food from the *Application* (see Figure 4 on page 6) to be used by the other two methods. Then we calculate the distance to the center of the light source. Finally we call the **update** method of the internal *Automaton*. **NearbyFood** and **consumeNearbyFood** is used by the automaton when it's in the feed state (explained in section 4.3).

Fear is used quite a lot by the automaton and therefore we have a method **flee** to reflect whether the silverfish is ready to flee because it's scared. Internally this method makes use of two variables, namely the energy boundary and the max distance (B and D in Table 1).

4.3 The Automaton

The following subsection explains how the automaton works, so please refer to Figure 6 when reading.

The automaton is made up of four different states (see Figure 6), where it can move around, feed or breed. By far the most common one is move, which moves the silverfish in any of four directions, namely left, right, up and down. To make the silverfish appear a bit smarter it saves the last direction, which is more likely to be chosen again the next move. These movement rates are constant among all silverfish even though it could change the behavior and success of a species. A silverfish can come upon food when moving around and will move into the feed state as long as it's fearless. Fear is a big factor here, as it will get stuck in the move state if scared. A scared silverfish will always run away from the light source to the safety of darkness and this can be thought of as an extra state. A silverfish that is feeding will continue to do so until there's no more food at its position. With each pass through the feed state it will eat one, and only one bit of food. The final part of a silverfish life is breeding. Once breeding there's no running away, the silverfish will continue until the delivery state. As a shorter breeding period is preferable in all situations it's constant among all silverfish. After the fixed amount of breeding the silverfish enters the delivery state and will produce two new silverfish with or without mutations depending on the mutation rate. These children start in the move state and will share the energy given from its parent. Finally after the delivery the automaton will continue to move if it's scared or can't find food, otherwise to the feed state.

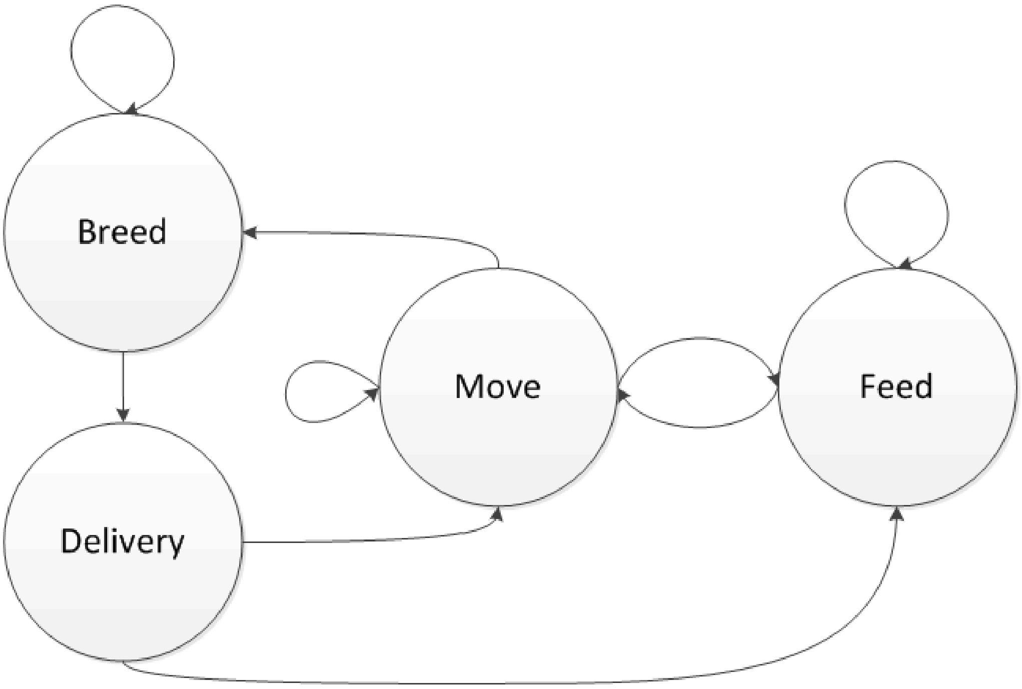


Figure 6: The four states of the automaton used by the silverfish.

4.3.1 The genes impact on the automaton

We have five genes that directly impact the automaton. Even if the lifespan doesn't impact the automaton itself, it controls whether the automaton should update the silverfish at all. When the age exceeds the silverfish's lifespan it will die and hence the automaton is not used anymore.

The gene with most impact is the speed, as it controls the amount of passes through the automaton's update function for each tick. As described earlier in section 4.3 a silverfish will get stuck in the move state if scared. If a silverfish is closer to the light source than a minimum distance it's scared and will run away, though there's one exception to this rule as it's willing to take the risk when it's running out of energy. That is when the lower boundary for energy is crossed. The last two genes impact when the silverfish can reach the breed state. The first gene describes when the silverfish is mature enough to produce children and the second one the energy required. All of these genes are briefly described in Table 1 on page 8.

4.4 Mutation

As explained in section 4.1 each gene is represented by a value internally. This value has a maximum and minimum value. With an easy function (see Equation 1) we can calculate a new value for the mutated gene given the parents gene. In the equation we use the parent value, a minimum and maximum value for the gene along with a maximum difference between the parent's and the mutated gene value. The function *rand(a, b)* gives us a random value between (a-b) and (a+b). The **mod** (modulo) operation always returns the smallest positive result.

Equation 1: $\boldsymbol{Mutate(parent, min, max, maxdiff) = rand(parent - maxdiff, 2 * maxdiff)\ mod\ (max - min) + min}$

Depending on the value set with the slider control for mutation rate, Equation 1 will be used to alter any of the genes. See Figure 7 for an algorithm describing how the mutation function is applied.

```
spawn_children(parent, given_energy):
for i: 1 to 2
  child = copy(parent)
  child.energy = given_energy / 2
  child.time = 0
  child.automaton.reset()

  x = -1
  c = rand(1, MAX_MUT_COUNT + 1)
  for j: 1 to c
    x = rand(1, mutation_value * 2)
    if x == 1
      speed = Mutate(speed,MIN_SPEED,MAX_SPEED,SPEED_DIFF)
    elseif x == 2
      lifespan = Mutate(lifespan,MIN_LIFESPAN,MAX_LIFESPAN,LIFESPAN_DIFF)
```

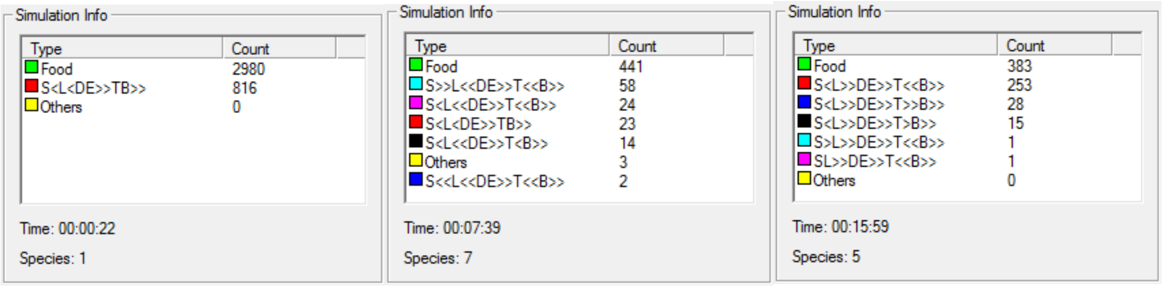
Figure 7: The algorithm used when applying mutations to the children where the mutation_value is controlled by the slider control for mutation rate.

5 Results

The results are shown as snapshots of the simulation info part of the simulation window (see Figure 5 on page 7); where the characteristics of each species among the most common ones can be deduced via their tags (refer to section 4.1.2 for a detailed description of the tags). The AI was used with the same settings in the first four simulations, where it stomped franticly. Note though that it's hard to get the exact same settings in the simulations. Also the AI moves around in a random fashion which might produce different results.

5.1 Test results

The first test (see Table 3) started out with a completely randomized silverfish that reproduced without any interference for a few seconds. When the numbers reached above 800 the AI was switched on and mutation introduced. At first the mutation rate was set very low in an attempt to discover how even tiny changes can create completely new species. Due to these low settings it took a few minutes before any truly different species started gaining in numbers. Finally, after 16 minutes of simulation, the results started to coincide again where similar species appeared as most common.



Simulation Info

Type	Count
🟩 Food	2980
🟥 S<L<DE>>TB>>	816
🟨 Others	0

Time: 00:00:22

Species: 1

Simulation Info

Type	Count
🟩 Food	441
▣ S>>L<<DE>>T<<B>>	58
🟪 S<L<<DE>>T<<B>>	24
🟥 S<L<DE>>TB>>	23
⬛ S<L<<DE>>T<B>>	14
🟨 Others	3
🟦 S<<L<<DE>>T<<B>>	2

Time: 00:07:39

Species: 7

Simulation Info

Type	Count
🟩 Food	383
🟥 S<L>>DE>>T<<B>>	253
🟦 S<L>>DE>>T>>B>>	28
⬛ S<L>>DE>>T>B>>	15
▣ S>L>>DE>>T<<B>>	1
🟪 SL>>DE>>T<<B>>	1
🟨 Others	0

Time: 00:15:59

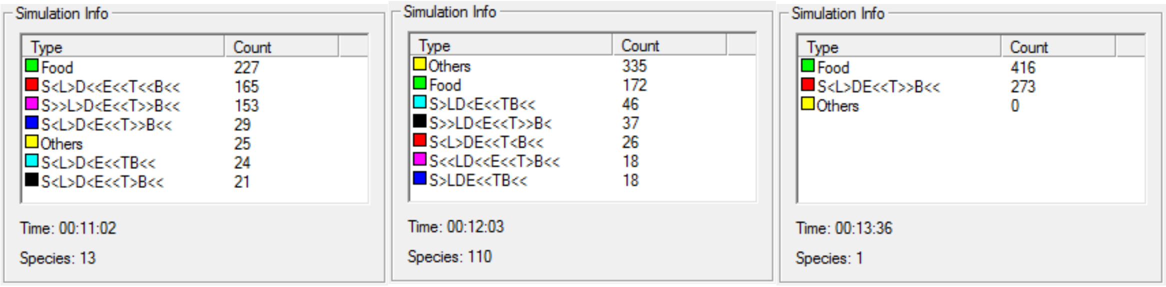
Species: 5

Table 3: Simulation results of the first test.

The second test (see Table 4) let a species start out with all gene values set to their minimum. Again the initial species gained in numbers before mutation was introduced. Also, a low mutation rate was used for the first 11 minutes. As the species hadn’t changed much with these settings the mutation rate was maxed out for one minute, which clearly gave results. To see which of these species were the strongest, the mutation rate was decreased to zero again to see a final victor in the last snapshot of Table 4.

An important thing to notice here is the fact that the **reproduction energy** and **energy boundary** never increased drastically from the lowest values, while the **speed** and **reproduce time** changed constantly throughout the simulation. Also the **lifespan** and **distance** genes remained steady around the normal values. There are two reasons why a value stays at the minimum; either the silverfish that gain from the increased gene value cannot survive or the gene isn’t viable at all at higher values.

At the 11 minute mark of the simulation two silverfish outnumbered the others. These had profoundly different tactics to survive. The species tagged with a red color relies on reproducing faster than the foot can kill, as they mature early and require little energy to reproduce. On the other hand the pink species relies on survival by being fast enough to escape the foot while reproducing less often.



Simulation Info

Type	Count
🟩 Food	227
🟥 S<L>D<<E<<T<<B<<	165
🟪 S>>L>D<E<<T>>B<<	153
🟦 S<L>D<E<<T>>B<<	29
🟨 Others	25
⬜ S<L>D<E<<TB<<	24
⬛ S<L>D<E<<T>B<<	21

Time: 00:11:02

Species: 13

Simulation Info

Type	Count
🟨 Others	335
🟩 Food	172
⬜ S>LD<E<<TB<<	46
⬛ S>>LD<E<<T>>B<	37
🟥 S<L>DE<<T<B<<	26
🟪 S<<LD<<E<<T>B<<	18
🟦 S>LDE<<TB<<	18

Time: 00:12:03

Species: 110

Simulation Info

Type	Count
🟩 Food	416
🟥 S<L>DE<<T>>B<<	273
🟨 Others	0

Time: 00:13:36

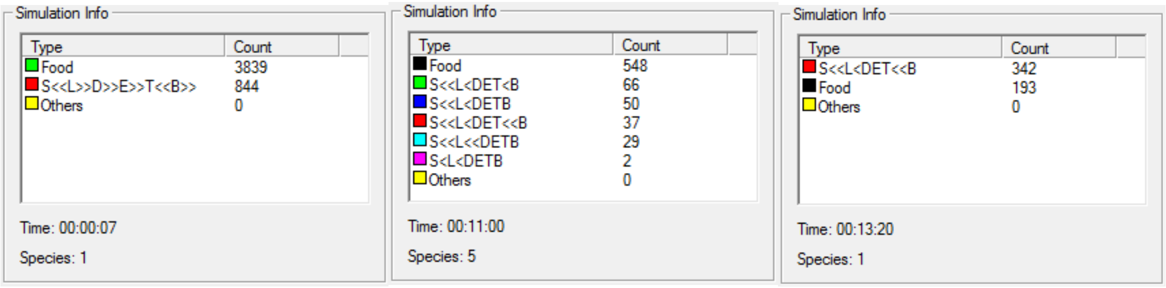
Species: 1

Table 4: Simulation results of the second test.

The third test (see Table 5) is an attempt to prove or disprove the theories from test two. To repeat the important steps in test one, the simulation ran for 11 minutes with little mutation. Again the mutation rate was increased temporarily for a minute, to finally be switched off until only one species was left.

In this test the slower species had a clear advantage, which is caused either by inexact settings on the AI, other genes coinciding or pure chance. However we can still see that the **distance** is in the normal region, which is the same result we saw in test two while the **reproduction time** and **energy** differed in this simulation, which is a possible reason for the change in the **speed** gene.

An interesting result that sadly went unseen in Table 5 is that the blue and red species were the only survivors less than half a minute after the mutation was switched off. Finally the red species won, even though the only difference here was the **energy boundary**. This is the same gene that appeared useless in the earlier tests. Here the red species didn’t care about dying of starvation, but rather dying to the foot, which saved it in the end as the other species got stomped to extinction.



Simulation Info

Type	Count
🟩 Food	3839
🟥 S<<L>>D>>E>>T<<B>>	844
🟨 Others	0

Time: 00:00:07

Species: 1

Simulation Info

Type	Count
⬛ Food	548
🟩 S<<L<DET<B	66
🟦 S<<L<DETB	50
🟥 S<<L<DET<<B	37
🟦 S<<L<<DETB	29
🟪 S<L<DETB	2
🟨 Others	0

Time: 00:11:00

Species: 5

Simulation Info

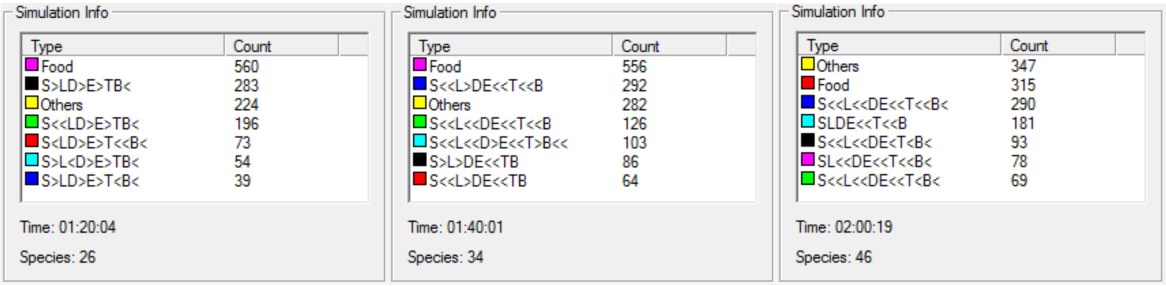
Type	Count
🟥 S<<L<DET<<B	342
⬛ Food	193
🟨 Others	0

Time: 00:13:20

Species: 1

Table 5: Simulation results of the third test.

An upper boundary for the amount of silverfish can clearly be seen in all tests. However, after more than one hour of simulation with 5 times as much food spawning only roughly 2.5 times as many silverfish appeared, as seen in Table 6. As the silverfish count can only be limited by the foot in this case, we can see that it does have a clear impact on the amount of silverfish; however does it affect the evolution?



Simulation Info

Type	Count
🟪 Food	560
⬛ S>LD>E>TB<	283
🟨 Others	224
🟩 S<<LD>E>TB<	196
🟥 S<LD>E>T<<B<	73
🟦 S>L<D>E>TB<	54
🟦 S>LD>E>T<B<	39

Time: 01:20:04

Species: 26

Simulation Info

Type	Count
🟪 Food	556
🟦 S<<L>DE<<T<<B	292
🟨 Others	282
🟩 S<<L<<DE<<T<<B	126
🟦 S<<L<<D>E<<T>B<<	103
⬛ S>L>DE<<TB	86
🟥 S<<L>DE<<TB	64

Time: 01:40:01

Species: 34

Simulation Info

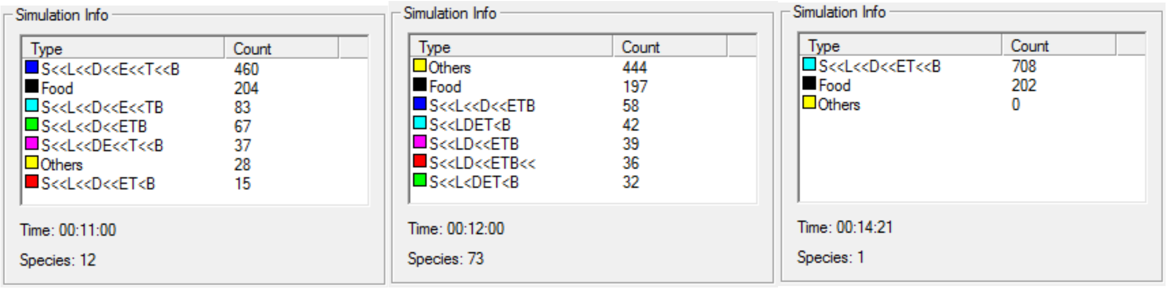
Type	Count
🟨 Others	347
🟥 Food	315
🟦 S<<L<<DE<<T<<B<	290
🟦 SLDE<<T<<B	181
⬛ S<<L<<DE<T<B<	93
🟪 SL<<DE<<T<<B<	78
🟩 S<<L<<DE<<T<B<	69

Time: 02:00:19

Species: 46

Table 6: Results from a longer simulation at max speed.

To see how the foot impacts the evolution, test two was repeated without any AI (see Table 7). This time the top 1 silverfish never changed – it was permanently the initial silverfish with the lowest values on the genes. However maxing out the **mutation rate** again clearly changed this even though a very similar species to the first one appeared as the winner. These results suggest that the lowest values on the genes are preferable without the foot, which indicates that the foot does affect the evolution!



Simulation Info

Type	Count
🟦 S<<L<<D<<E<<T<<B	460
⬛ Food	204
🟦 S<<L<<D<<E<<TB	83
🟩 S<<L<<D<<ETB	67
🟪 S<<L<<DE<<T<<B	37
🟨 Others	28
🟥 S<<L<<D<<ET<B	15

Time: 00:11:00

Species: 12

Simulation Info

Type	Count
🟨 Others	444
⬛ Food	197
🟦 S<<L<<D<<ETB	58
🟦 S<<LDET<B	42
🟪 S<<LD<<ETB	39
🟥 S<<LD<<ETB<<	36
🟩 S<<L<DET<B	32

Time: 00:12:00

Species: 73

Simulation Info

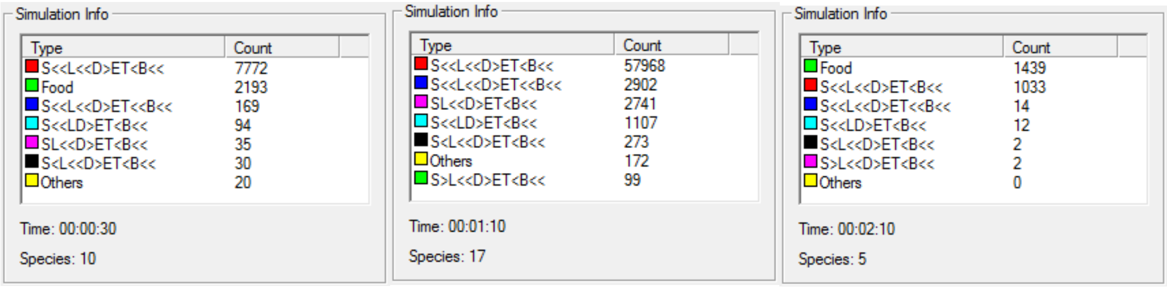
Type	Count
🟦 S<<L<<D<<ET<<B	708
⬛ Food	202
🟨 Others	0

Time: 00:14:21

Species: 1

Table 7: Test two repeated with neither AI nor stomping.

When the previous test was repeated with more food, some unexpected results appeared as shown in Table 8. One species (with the red color) spawned during the first few seconds with mutation and instantly started growing in numbers, very rapidly. The population reached past 60000 to suddenly drop down below a thousand again. Even though the simulation went on for another 30 minutes the phenomena didn’t appear again as the initial large amount of food (1500) enabled lots of silverfish to gain enough energy to reproduce quickly to finally die out again once the rich food supply disappeared.



Simulation Info

Type	Count
🟥 S<<L<<D>ET<B<<	7772
🟩 Food	2193
🟦 S<<L<<D>ET<<B<<	169
🟦 S<<LD>ET<B<<	94
🟪 SL<<D>ET<B<<	35
⬛ S<L<<D>ET<B<<	30
🟨 Others	20

Time: 00:00:30

Species: 10

Simulation Info

Type	Count
🟥 S<<L<<D>ET<B<<	57968
🟦 S<<L<<D>ET<<B<<	2902
🟪 SL<<D>ET<B<<	2741
🟦 S<<LD>ET<B<<	1107
⬛ S<L<<D>ET<B<<	273
🟨 Others	172
🟩 S>L<<D>ET<B<<	99

Time: 00:01:10

Species: 17

Simulation Info

Type	Count
🟩 Food	1439
🟥 S<<L<<D>ET<B<<	1033
🟦 S<<L<<D>ET<<B<<	14
🟦 S<<LD>ET<B<<	12
⬛ S<L<<D>ET<B<<	2
🟪 S>L<<D>ET<B<<	2
🟨 Others	0

Time: 00:02:10

Species: 5

Table 8: Test two repeated with neither AI nor stomping and with more food spawning, leading to an interesting population boom early on.

6 Conclusion

The results reflect what is seen in biological evolution; different species survive via different characteristics as seen in Table 4 on page 12. The application could clearly produce many different species, even with a simple tactic like parameter based simulated genes. However the results without the foot clearly lacked profoundly different characteristics of the top species, as only four genes were involved. This makes me draw the conclusion that the tactic is too simple with few genes as it's unable to create a well-functioning simulation of evolution.

Altering evolution with the stomping foot was a success; we clearly saw a difference with and without the foot. Therefore implementing a feature like the foot into an application with module based simulated genes or artificial neuron networks calls for some interesting results!

7 References

[1] scotchfaster, "Chaos is Creation: evolution in real time," 10 April 2008. [Online]. Available: http://www.youtube.com/watch?v=GuWJ1ZiwO2g. [Accessed 4 March 2012].

[2] D. Rios, "NeuroAI Artificial Neural Networks," 2010. [Online]. Available: http://www.learnartificialneuralnetworks.com/. [Accessed 7 March 2012].

[3] T. T. "Artificial life," Wikipedia, 2 April 2012. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Artificial_life&oldid=485074045. [Accessed 7 April 2012].

[4] Microsoft, "Windows GDI," Microsoft Developer Network, 7 March 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/dd145203(v=vs.85).aspx. [Accessed 7 April 2012].