Evolution Number 9

or another

Fast'n Crazy Breeder (with a lot of digital fallout)

Everybody is talking of artificial intelligence, artificial life etc. And all that talk is inspired by a big rhetoric boost, nourished by a profound misunderstanding of the actual simplicity of these undertakings. It's just mumbo-jumbo philosophy, tricky and sophisticated.

Genetic algorithms are a part of that – and this is what Evolution Number Nine is all about. It provides the Gamestudio user with a dll – and a very easy interface: Just one function he has to manipulate. [In the current state all that is very rude, just fit enough to demonstrate that it works. In the next releases this will be refined]

I will try – very short - to describe what genetic algorithms can do and why it could be interesting to use them in game development.

One might read that genetic algorithms are a kind of evolution system – and if you want to be a natural philosopher, you might put it that way. There are mechanism like selection, crossover breeding, elite heredity etc. – but all these mechanisms are metaphors, because it's digital evolution and the things that evolve are the one you have foreseen. I would prefer looking at them as a sorting mechanism which allows to go through hundreds of possibilities.

Compare it with chess. Imagine you had a mechanism which allows you to evaluate every position in the game. This evaluation would turn out into a number (some fitness value). Now you have a lot of possible moves – and the genetic algorithm would do nothing else than tracking the various combinations. Most of them will be crap – and if your evaluation system work properly the machine will analyse this is crap.

The bad moves are now sorted out – and only the promising ones are pursued. In a way that's what a genetic algorithm does.

a) it creates a population (possible moves) which consists of

b) various attributes (could be the colours of the model – which is the actual demo)

c) At the beginning the population are filled with random values

d) Now the evaluation begins

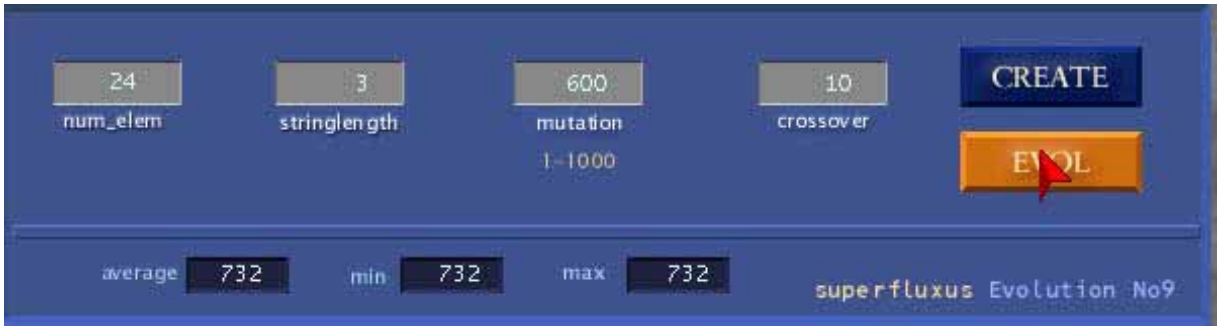
e) After evaluation the »good« specimen may reproduce, the bad ones will disappear

f) the process begins once again (new game, new generation)

The play where you will come into play is the step number d). You will have to say what you want – and make some fitness rules (but this comes a little bit later).

The interface

As you will see with a fast glance, the interface of *Evolution No 9* is extremely simple.



Let's have a look at the parameters.

num_elem stands for the number of elements the population consists of. It should be a twofold of 2 (in the evolution it will be split into a male and female group - and therefor the number should be equal)

stringlength stands for the attributes eachg element has. In our exampe we have chosen a stringlength of 3 because it codes color information (RGB). The stringlength can be much higher, this depends on the problem you want to solve.

mutation stands for the mutation possibility. A value of 1000 means that the chance for a mutation equals 1 - so each generation one element is subject of a unforeseen mutation. - This might be a desirable effect because the population usually tends to homgenity - which is a loss of information. Mutation brings new elements into the game - therefor a gain of information.

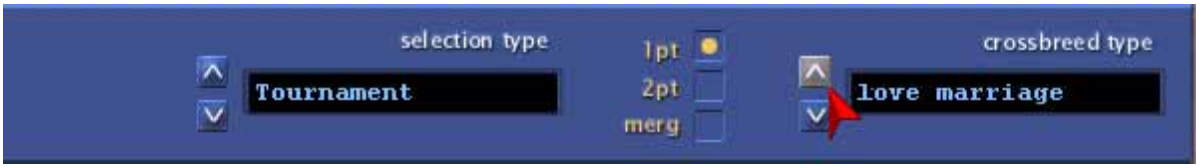
crossover stands for the number of elements that will couple in each generation. This produces some interesting mixer effects.

The values in the lower row average, min and max are there just for feedback reasons. The first gives the overall fitness of all the elements, min gives back the minimum, max the maximum fitness.

Just play around with. If you press the CREATE will see that the function generates as much colored cubes that you have defined. If you press EVOL the evolution process beginns.
The fitness function I described is not a very intellgent one. It sorts out all the elements which have a brightness value higher than 70 - and it promotes (to hinder the population to turn all grey) the pronounced blue and red elements. To make the upgade visible the x-value of the respective element is position according to its fitness.

Evolution Strategies

Besides just creating a population and controlling the initial setting you have the chance to control the evolution process itself. This is what you do with the second panel. It looks like this:



to exlain what you can do with these setting goes deep into the theory of genetic algorithms. Anyway: I'll try to explain and to keep it simple.

You can manipulate two basic processes:

a) selection
b) reproduction

Since genetic algorithms do not deal with nature, but just with the symbolisation of nature, you might look at these processes as pure metaphores. This is recommendable because it takes away the myth and the false respect one might feel towards a subject like *evolution strategies*. Let's start simple. Selection can be understood as the mechanism which sorts out the weakest individuals. But how does the genetic algorithm know about the various garades of fitness? In fact: it does not know. There is some humane instance which has described a fitness function, that means: an evalutation system. According to this fitness fuction the genetic algorithm keeps record of the *fitness* of each indidual. The selection process now tells the algorithm which individuals shall be sorted out and be replaced by other. The simplest solution: look for the one with the lowest fitness and replace it with the fittest one. This is min/maximal substitution

This is a very easy way of upgrading the fitness of the population, but it has it's disadvantages. The population will be uniform, like an insect state after a while. Everybody goes mainstream - which is a gain in overall fitness, but a loss in diversity. Although this strategy does not seem to be too promising it helps to understand the problem of genetic algorithms, which is a double and often conflictuous task:

a) upgrading the fitness
b) preserving diversity of information

One way out of the selection dilemma (which diminuishes the richness of a population) is the random mutation which we have already mentioned, another one is the process of rerpoduction, or crossbreeding. Imagine the population is a society which consists of two rows, males and females. If two individuals form a couple the offspring will inherit attributes from both. The simplest way to thinks of this process would be to assume the sum of to numbers a + b and attribute the sum/2 to the offspring. This would be simple merging.

You will see three basic techniques on the panel

1pnt – which is 1 point croosover
2pnt – which is 2point crossover

merge –

The crossover is in fact very simple. Remember that an individual consists of a string of attributes, like

A A A B B B

and another one

C C C D D D

The crossover would now look for a random split point. For simplicites sake it will be the middle of the string, Now the string will be broken up into two pieses and the two of them will be recombined to:

A A A D D D
C C C B B B

These two individuals will replace their parent ones. This is the 1point crossover solution. The 2 point crossover solution would split the string at two point - and arrange the crossover accordingly. Actually the number of reproduction techniques could be multiplied. Generally this process can be understood as more or less intelligent way of merging to individuals. But the aim is the very same: preserving the richness of diversity by bringing in recombined elements.

There is another setting you can regualte (which is mainly my contribution to genetic algorithm and by no means *conventional*). You will not find *evolution*, but *social* statregies here, like:

love marriage

elitism

enrichment

plebs

hedonism

What is meant by these parameters is that the coupling process is regulated not mechanically but by some coupling mechanism. Take love marriage. Love, as a social invention, ignores social hierachies. And that means: the fitness range is not respected very thoruoghly. (So what I have done is that I left the male hierarchy intact, but mixed up the female by some random factor - with the effect that there is a increased diversity)

The best (and in a way self explaining mode) is just to play around with the parameters. You will see that the results may differ to a broad extent - and that some stategies promote diversity and others a fast but homogenous, more narrowminded solution.

Using your own model

In the example you are provided with the dll creates a cube. There's no big deal to choose another model. - Since it is not implemented in the interface you will have to do it manually. Just open the „fitness.wdl“ file and modify the string modelname with your model (actually it is „cubetest.mdl“);

Using your own action

In the file "actionfile.wdl" you will find a prototype of an action that you can apply a genetic algorithm on. You do not have to take mines for granted, just use your own. But beware that you keep the synonym obj_ptr and the assign_id() command.
Why are these lines important?
The assign_id() command stores the handler of the object in a array. When a command is sent back from the dll, you will get the index of this array - and this allows you to manipulate your objekt with the obj_ptr synonym.

```
action prototype
{
obj_ptr = me;
assign_id();

while(1)
  {

  ////////////////////////// just for demonstration reasons
  my.z = my.skill10;        // delete this
  ////////////////////////////////////////////////////////

  wait(1);
  }
}
```

Use the above action as a template - and you won't have problems.
There is one thing to do though. You have to modify the actionstring string.
Open open the „fitness.wdl" file and modify the string actionstring that it holds the name of your action (actually it is „redden");

Writing your own fitness functions

Given that the predefined fitness function does not do anything intelligent, you might be inclined to write your own function. There is a special file for that - the fitness.wdl file. Here you find the fitness function which determines what is going on, and it's form is:

```
claculateFitness(identity);
```

But before I do this I have to describe the main interface elements:

The values you want to evaluate are stored in a particular array, which is called smallarray

Since we perform a genetic algorithm on color information, the array consists of 3 elements red green and blue

red is stored in smallyarray[0]
green is stored in smallyarray[1]
blue is stored in smallyarray[2]

Usuallay we do not only want to evalute these values but also make changes on the respective object. For that puprose we have a handle_array which stores the handles of the objects and we have an object pointer obj_ptr which

```
function CalculateFitness(identity)    // here the WDL gets the object id
{
obj_ptr  =  ptr_for_handle(handle_array[identity]  );
// now the pointer points to the respective object
...
}
```

If you want to modify the fitness functions according to your need you must leave these lines intact. The next line is

```
chromosom = 0;
```

What does it mean? The variable chromosom stores the fitness value of the string. And to avoid errors it is highly recommended to reset it weach call.

Let's say we want to favorite combinations which are very dark. So we could do the following:

Now we can start to think about evaluating our color values. Let's assume that we want to make the average brightness a criterion. Everything higher than an average brightness than 70 should be sorted out.

```
var  temp;
temp  =  smallarray[0]  +  smallarray[1]  +  smallarray[2];   // the addition

if  (temp  >  90)
{
chromosom  =  0;  // this is the fitness value for the string – it's zero
}

else  {
      // and here we could differentiate
      }
```

If you evaluate something as 0 you can be sure that this version will be omitted in the evolution and selection process. Only the fittest will survive - and you will have to describe what you think is fit. This description is the basic thing – and it is the one and only function you will have to bother about. The whole mechanism itself (with selection, reproduction, mutation and crossover mechanisms) is done in the background.

The values should be positioned in a reasonable range. Myself I concentrated on the range between 0 and 1 (so 555 on the screen is actually 0.55), but there is no objective limit for that.

Now we have evaluated an element above 90 as zero, but what about a positive evaluation. I could write the following:

```
chromosom  =  temporal/90;
```

That means when the brightness value (which is stored in the temporal var) is 90, fitness is 1. When it is less, the fitness value will be accordinglyless. The results of such a rule would be that the values with the brightness of 90 will be the fitest. But this is not very pronounced - and I can asume that there will be a lot of various grey colors.

To proomote a sharper contrast I could modify the formula, stating that a pronounced red (>80) has a very fitness.

```
 if (smallarray[0] > 80)
      {
      chromosom = 0.9;
      }
else     {
        chromosom = temporal/90;
         }
```

I have to admit beforehand that my example is not telling that much. I could fancy a whole bunch of more intriguing possibilities.

Instead of using the colour information it would be much more appropriate to evaluate the fitness of an object towards it's environment.

So we cold do some motion studies and investigate the fitness of a moving entity to circumvent others. We needed for that a scan command, an overall velocity, a variable containing the frame rate – and we would ask: how oft can we refresh the scan command without bringing the frame rate down and produce optimal recognition result at the same time.

I hope the example shows that the *Evolution Number 9* dll will work as a general problem solver. (It may optimise a parameter room – where simple trial and error would cost too much time). But in general: It's up to you to define where you want to apply it.

So one of my reasons to publish the program in this early state is to find out what people might use it for.

The future of this program (version 0.93 and above)

Refinement of the interface.
Output of a logfile with the best results of the evolution process.

If you happen to know "Tiny Things" (the particle generator I wrote) the concept in general is quite similar. There is a production environment and there is a playback environment. So one could use the dll also as a strange random generator, which do the genetic algorithm calculations on the fly. The other option (the studio) will end up in a printout of desirable values. Maybe it will be more, and frankly I have to say: I don't know what kind of ideas may enter my mind.

So have fun.

Martin Burckhardt

Ps. if you like to contact me directly, getting into a more profound discussion, please mail to
mb@superfluxus.de