An Introduction of Genetic Algorithms for Game Development with an Implementation in Unity3D

Christian Piazzi
course of studies bachelor digital media

August 4, 2019

Contents

Abstract

The content of this project is to provide a collection of C# classes and scripts for the game development platform Unity3D, which supports the developer in the use of genetic algorithms.
For this purpose, this accompanying document is prepared, which provides a brief introduction to the basic biological concepts and mathematical foundations.
In the further course of the document I will deal with the necessary methods for an implementation of genetic algorithms. This will cover four main areas:

- Natural Selection Algorithms
- Artificial Selection Algorithms
- Recobination Procedure
- Mutation Method
- Substitute Schemata

Besides these individual methods, I will also explain the application of genetic algorithms. Afterwards I will give a short outlook on the use cases of genetic algorithms in the context of game development.
As a practical part, necessary classes and functions are implemented. Using this basis, one or two approaches to the use of genetic algorithms in the context of game development are implemented as prototypes.
Finally, there is a conclusion on the topic of genetic algorithms and genetic algorithms in the context of game development.
The aim of this project / document is to provide the game developer with the knowledge and the code to take the first steps towards game development with genetic algortihem. Furthermore, the developer should get an overview of the possible use cases.

Genetic Algorithm

Evolutionary algorithms (EA) are optimization methods that are based on aim's model of biological evolution. In principle, they can be divided into four sub-areas:

- genetic algorithm
- genetic programming
- evolutionary strategy
- evolutionary programming

Genetic algorithms were first introduced in the early 1960s by John Holland and his colleagues at the University of Michigan [1]. About at the same time, Ingo Rechenberg and Hans-Paul Schwefel founded the TU Berlin the evolution strategies [2][3]. The subdivision into the individual Sub-areas are thus historically conditioned mainly, since they show very high similarities.

Nevertheless, there are some differences between the sub-areas in terms of content and focus. In genetic algorithms, for example, individuals are coded as bit strings, while evolutionary strategies and evolutionary programming usually use real vectors.

A further difference between the subareas is the selection of the Individuals. Genetic Algorithms and Evolutionary Programming Take a stochastic selection, i.e. also the individuals with a poor fitness value have a, albeit small, chance of passing on their genes to others. to the next generation. With the evolution strategies, on the other hand, a deterministic selection is carried out, i.e. only those individuals with the best Fitness pass on their genes.

In the further course we will mainly deal with genetic algorithms.

Motvation for Genetic Algorithm

With Genetic Algorithms you can find a "good enough" solution "fast enough". This makes genetic algorithms attractive for use in solving optimization problems. The reasons why genetic algorithms are needed are solving difficult problems, failure of gradient based methods and quickly finding a good solution.

Solving difficult problems

In computer science, there are a number of problems that are NP-hard. This essentially means that even the most powerful computer systems take a very long time to solve this problem. In such a scenario, genetic algorithms prove to be an efficient tool for providing useful, nearly optimal solutions in a short time.

Failure of gradient based methods

Traditional calculus-based methods work by starting at a random point and moving towards the gradient until we reach the top of the hill. This technique is efficient and works very well for single peak target functions such as a cost function in linear regression. But in most real situations we have a very complex problem called landscapes consisting of many peaks and many valleys, which causes such methods to fail because they suffer from an inherent tendency to stick to the local optima, as the following figure shows.

Finding a good solution quickly

Some difficult problems like the Travelling Salesperson Problem (TSP) have real applications like pathfinding and VLSI design. Now imagine you are using your GPS navigation system and it takes a few minutes (or even hours) to calculate the "optimal" route from source to destination. Delays in such real applications are unacceptable and therefore a "good enough" solution that is delivered "fast" is just right.

Basic Terms from Genetics

Every living being carries the characteristic genetic information in its cells, that is generally unchanging throughout his life. On a whole genus, this information changes over time.

As organisms reproduce, new blueprints emerge. Mating for example two animals, then their individual construction plans are combined with each other. and combined into new ones. That's why the conceived children are similar to their two parents, but never identical to them.

Furthermore, spontaneous changes in the construction plans can occur during reproduction, which also lead to new genetic information. Such changes are caused, for example, by external environmental influences.

The Evolutionary Principle "Survival of the fittest" according to Charles Darwin is based on the theory that those living beings of a species survive that are have adapted best to the prevailing external environmental conditions. These organisms therefore have the largest number of offspring and have the most decisive influence on the further development of their genus. This phenomenon is often referred to as "natural selection" because nature, so to speak, makes a selection of organisms whose offspring are increasingly adapted to their environment.

Individuals / Chromosome

An individual in the biological sense is a living organism, whose genetic information is stored in a set of chromosomes. In connection with genetic algorithms, the terms individual and chromosome is usually equated.

An individual is coded as a binary string of the fixed length n, i.e. one can it as an element from $\{0, 1\}^n$ to understand.

Gen

A specific site or sequence of a chromosome is called a gene. Usually, the context determines whether a gene is a understands a single passage or an entire section.

Allelic

The concrete expression of a gene is called an allele. If the gene as a variable, the allele is the value of the variable. If the genes are used to denote individual digits of the binary string, the alleles can only contain the values Assume 0 and 1.

Length of a Chromosome

The length of a chromosome is the length of the binary vector, i.e. the number of genes of an individual.

Genotype

The genotype is the coded vector of the decision variables. From it generally depends on which encoding method is selected.

Phenotype

The phenotype, on the other hand, is the decoded vector of the decision variables. Its expression depends on the genotype and the chosen decoding method.

Population / Generation

A set of structurally similar individuals of a certain genus is called a population. When new living beings of this genus are born or others will die, inevitably also the size changes of the population. If one considers the populations of a genus over several over time, one speaks of generations of living beings.

Mathematical Basics

A few mathematical basics are needed to understand genetic algorithms. Not all of them are used directly in the context of an algorithm.

Minimum, Maximum

A number $a \in M$ means minimum of M, if for all $x \in M$ applies: $a \leq x$. The minimum of M is called $\min(M)$.

A number $a \in M$ means maximum of M, if for all $x \in M$ applies: $a \geq x$. The maximum of M is called $\max(M)$.

Minima and maxima are clearly determined. If two minima or maxima are a_1 and a_2 exist, then $a_1 = a_2$

If a quantity M has a maximum or a minimum, then $\min(M)$ or $\max(M)$ is the smallest or largest element of M.

Random Variables

A variable whose values are determined or changed by random events is called a random variable. If the variable can only contain certain values, one speaks of a discrete, otherwise of a steady random variable.

Warh Probability Distribution

The function that determines the probability for each expression of the random variable. of its occurrence, the probability function or probability distribution f of the random variable X is called:

$$f(x_i) = W(X = x_i)$$

A probability function has the following properties:

$$(x_i) \geq 0$$

$$\sum_i f(x_i) = 1$$

Distribution Function

With the help of the probability function, the so-called distribution function can be defined. It indicates the probability that the random variable X takes at most the value x:

$$F(x) = W(X \leq x) = \sum_{x_i \leq x} f(x_i)$$

If X is a discrete random variable, the graph of the distribution function is a staircase function.

Otherwise the graph is a continuous function with the following properties:

$$0 \leq F(x) \leq 1$$

$$x_i < x_j$$

$$f(x_i) \geq 0$$

$$\sum_i f(x_i) = 1$$

Expected Value

The expected value $E(X)$ of a discrete random variable X is defined as

$$E(X) = \sum_i x_i * W(X = x_i)$$

i.e. the values of the random variable are calculated with their occurrence probabilities multiplied and summed up.
For continuous random variables X the following equation applies:

$$E(X) = \int_{-\infty}^{+\infty} x * f(x) dx$$

where $f(x)$ is the derivative of the distribution function $F(x)$. It is also called density function.

Variance, Standard Deviation

The variance $Var(X)$ of a discrete random variable X is defined as

$$Var(X) = \sum_i (x_i - E(X))^2 * f(x_i)$$

For continuous random variables X the following equation applies:

$$Var(X) = \int\limits_{-\infty}^{+\infty} (x - E(X))^2 * f(x_i)$$

The variance is a measure of the dispersion of the values of the random variable X. Another measure of the dispersion of the values of X is the standard deviation. It is defined as the positive root of the variance:

$$s(X) = \sqrt{Var(X)}$$

Normal Distribution

The normal distribution is the most important statistical distribution, since a great many characteristics occur normally distributed in nature. In addition many other distributions can easily be approximated with the help of the normal distribution.

The density function of the normal distribution has the shape of a bell and is therefore also referred to as Gaussian bell curve. The concrete form of the curve is dependent on three parameters:

- the characteristics of the continuous variable X and its occurrence probabilities
- the expected value m
- the standard deviation s.

The function equation of the Gaussian bell curve is:

$$f(x, m, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} * e^{-\frac{(x-m)^2}{2\sigma^2}}$$

Here m determines the position of the bell curve and s the scatter. The maximum of the density function is $x = m$, the inflection points of the bell curve are $m - s$ and $m + s$.

The integral of the Gaussian bell curve represents the distribution function of the normal distribution. It has an S-shaped shape and its formula is:

$$f(x, m, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} * \int\limits_{-\infty}^{x} e^{-\frac{(t-m)^2}{2\sigma^2}} dt$$

Structure of a genetic algorithm

Genetic algorithms can usually be divided into the following subroutines will be:

1. the problem to be optimized is coded, i.e. it is written to a binary coded chromosome.
2. a population of individuals is generated and randomly initialized. One refers here to the initial population or generation 0.
3. each individual is evaluated with a fitness function, which is assigned to each a real-valued number to a single chromosome.
4. two parents are assigned to each chromosome by means of a selected selection variant is selected.
5. from the genetic information of the parents the offspring are produced by means of a selected crossing variant.
6. the alleles of the offspring can mutate, i.e. their values will be inverted.
7. the newly created offspring are added to the population. Will the population size is exceeded, the system uses a selected replacement schema a lot of individuals who have been affected by the new can be replaced.
8. From step 3, the subroutines are repeated until an abort criterion is fulfilled.

Workflow of genetic algorithm

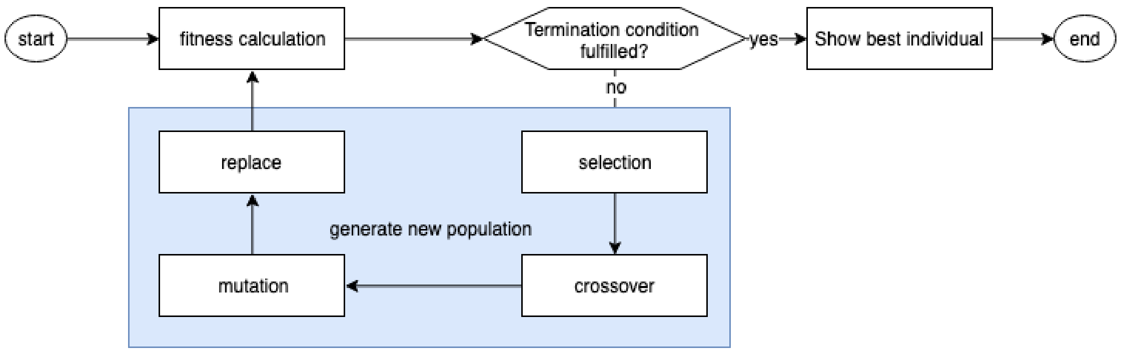

Figure 1: Workflow of genetic algorithm [4]

Advantages and Limitations

Advantages of genetic algorithms

Genetic algorithms have several advantages. These have ensured that genetic algorithms have attained a certain degree of popularity. The advantages are:

- No derived information is required (this may not be present in the real problem).
- genetic algorithms are more efficient and faster than conventional methods
- genetic algorithms can be parallelized well
- It is possible to optimize continuous and continuous functions, as well as to solve multi-critical problems.
- There is not only one solution offered, but a list of "good" solutions.
- Always finds an answer to the problem depending on the time.
- Useful when the search space is very large and a large number of parameters are involved.

Limitations of genetic algorithms

Like any other method, genetic algorithms have certain limitations. These include

- genetic algorithms are not suitable for all problems. Particularly simple problems or those where derived information is available can be better solved by other methods.
- A fitness value is calculated again and again, which can be very computationally intensive.
- Since this is a stochastic solution, there is no guarantee of optimality or quality of the solution.
- In case of improper implementation, the genetic algorithms may not converge to the optimal solution.

Fitness and Evaluation Function

The evaluation function measures the quality of an individual in relation to the task to be optimised, while the fitness function measures his chances of reproduction is evaluated. Both functions can be equated, if the functions at optimum the measured best individuals also have the best chances of reproduction should.

A popular fitness function is the so-called proportional or linear fitness, which represents the fitness of an individual in direct proportionality to the for evaluation:

$$Prop_{fit}(I) := \frac{a * I}{B}$$

I = Evaluation of the individual, B = Sum of the evaluations of all individuals and a = any factor

Natural Selection

The selection can be divided into a selection step and a selection step. The selection algorithm assigns each chromosome to a probability value for its replication. This value is initially an expectation

$$E(I) = \mu * p_s(I)$$

where μ is the population size and $p_s(I)$ is the selection probability of the of a single individual. Thus $E(I)$ gives the expected number of replicas in the so-called "mating pool".

Fitness Proportional Selection

With this selection method, the selection probability $p_s(I)$ is direct proportional to the fitness of the individual:

$$p_s = \frac{\Phi(I_j)}{\sum\limits_{j=1}^{n} \Phi(I_j)}$$

Φ is the evaluation function, I an individual and j the index of the individual.

However, the selection pressure with this procedure is relatively low. The higher the selection pressure, the faster the algorithm converges and the faster a local optimum is found.

A measure of the selection pressure is the so-called "takeover time". It gives the number of generations required to work with a given Selection algorithm for using the selection alone for a population to produce the $n - 1$ copies of the best individual of the initial population where n is the fixed population size.

Rank-Based Selection

With this selection variant, the selection probability p_s is no longer available. in direct proportion to fitness. Instead, the chromosomes of the Population sorted descending by fitness value and numbered. The selection probability is now related to the resulting Rank of a chromosome.

Competition Selection

This selection algorithm is also a selection procedure. Here a set of individuals (at least 2, not more than n) from the population of size n, compared their fitness values and found the best in copied the mating pool. This procedure is repeated n times until n values are available in the mating pool. With the help of the number of the drawn chromosomes, the selection pressure can be adjusted - the more individuals the more the greater the selection pressure. However, it can in this process, the mating pool is finally filled with n identical individuals are present.

Artifical Selection

Roulette Principle

The roulette principle can be illustrated by a roulette wheel, which is divided into n sections, where n is again the fixed population size. The width of a section is directly proportional to the probability of selection. of an individual. Now n times a roulette ball will be placed in the and the individual is copied into the mating pool, in whose section the bullet has come to rest. The higher the fitness of a chromosome, the more likely it is that the bullet will end up in its Section. However, there is also a danger here that n times the same individual is selected.

Stochastic Universal Sampling

In this procedure, the representation is possible by a wheel of fortune, which is represented in n large sections proportional to the selection probability of an individual is subdivided. In addition, n pointers are displayed at even intervals around the wheel. Now the wheel is turned once, and each individual as many copies are placed in the mating pool as pointers to point to the corresponding section. By this procedure can be excluded that n same individuals end up in the mating pool.

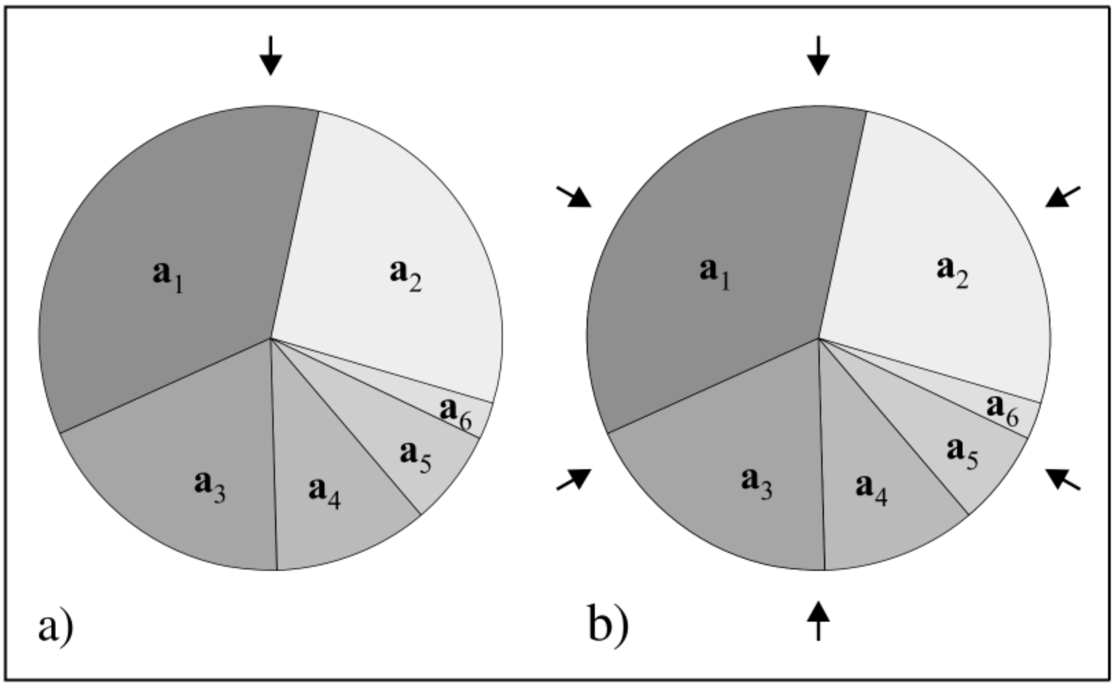

Figure 2: a) Roulette wheel selection b) Stochastic universal sampling. [4]

Recombination

The most important genetic operator in genetic algorithms is the cross-pollinator. Operator (also crossover operator). From two individuals of the parent generation offspring are created by copying genes from the parents and the children are passed on.

The selection of the parents is made stochastically by one of the already mentioned Selection procedure. The crossover probability pc indicates with which probability that there will be a cross between the two parents at all. There are different crossing procedures, to which in the following now should be further elaborated.

One-Point-Crossover

A random number is used to determine the intersection where the two parents are cut. The first of the two offspring produced receives the first part of the chromosomes of the first parent and the part from the crossing point of the second chromosome. The second offspring produced receives the remaining genes, i.e. the first part of the second parent and the second part of the first parent.

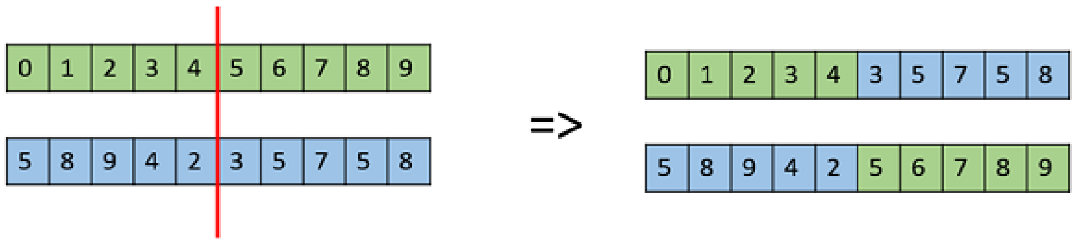

Figure 3: one point crossover [12]

N-Point-Crossover

In contrast to the One-Point-Crossover there are several crossing points. The first child receives the genes of the first parent up to the first crossing point, then the corresponding genes of the second parent up to the next crossing point. Now the chromosomes of the first parent are taken over again and so on. The second child receives the chromosomes of the other parent in the same way.

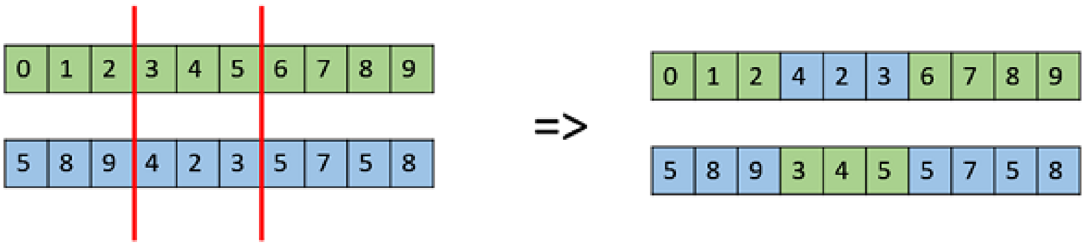

Figure 4: multi point crossover [12]

Template-Crossover

With this method, a template is first randomly generated along the length of the chromosomes. Like the two parents, the template consists of ones and zeros. The first child receives for each 1 in the template the corresponding allele from the first parent and for each 0 from the second. For the second child, the behavior at 0 and 1 is reversed.

Uniform-Crossover

The Uniform Crossover tests for each bit individually whether it is exchanged between the two parents or not. This is determined by a probability p_{ux} and a bit-related probability $U_z (z = 1, 2, ..., n)$. If $p_{ux} \geq U_z$, then the bits between the parents at position z are swapped.

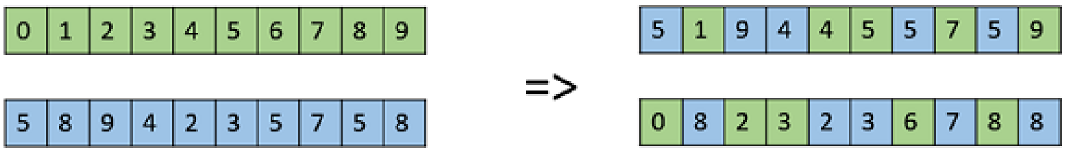

Figure 5: uniform crossover [12]

Schuffle-Crossover

In this crossing procedure, the genes of the parents are first numbered consecutively and then mixed. A one-point or N-point crossover is now performed on the mixed chromosomes. Finally, the genes are again arranged according to their numbering.

Mutation

A mutation is the flipping of a bit, i.e. the transition from one bit to the next. from a 1 to a 0 and vice versa. Alternatively, the mutation can also be as a random redefinition of a bit. In this case, however, it can be it happens that the mutated bit is identical to its predecessor. The mutation is primarily intended to prevent individual alleles from being found in all individuals of the population are identical, which would mean that the search would only be possible in a subspace of the original search room would take place.

The probability that a bit will mutate is usually relatively low. Mutation probabilities of $p_m = 0.01$ and $p_m = 0.001$ are quite common in genetic algorithms.

Replacement Schema

Once new offspring have been produced with the help of the genetic operators mentioned, it must be decided which proportion of parents should be replaced by the children. The population size used in the evolutionary strategies is is designated μ, must be as large as before at the end of the iteration step. There are a number of procedures for this as well.

General Replacement

Here all parents are replaced by the children. However, this procedure is running Danger of losing the best chromosome of the previous generation, and to thereby reduce the average quality of the total population. On the other hand is the chance to commit to a few good individuals (local maximum), relatively small.

Principle of the Elites

If a subset of the best individuals of the parent generation are retained one speaks of the principle of the elites. However, here is the danger of to a local maximum, relatively high.

Weak Elitism

This principle is similar to the previous one, but it does not change the chromosomes into the new generation.

delete-n-latest Schema

In this procedure, the n worst generated children are killed by individuals from the parent generation.

Use Cases in Game Development

In this section I want to give you a small overview of the application areas of genetic algorithms in connection with game development.

Pathfinding

In the context of pathfinding there are quite a few examples where genetic algorithms are used. Not only for pathfinding in game development, but also in the context of autonomous driving. The first question here is where the advantage of genetic algorithms over traditional methods such as the A* algorithm lies. I am of the opinion that both methods have a right to exist. Ryan Leigh, Sushil J. Louis, and Chris Miles summarize this quite well with one sentence:

> A*, the traditional pathfinding algorithm in games is computationally expensive when run for many agents and A* paths quickly lose validity as agents move. Although there is a large literature targeted at making A* implementations faster, we want believability and optimal paths may not be believable. [5]

In the context of pathfinding, genetic algorithms can also be combined with other methods. For example, there is the method of Ulysses O. Santos, Alex F. V. Machado and Esteban W. G. Clua which is described in the paper Pathfinding Based on Pattern Detection Using Genetic Algorithms. [6]

I find that the use of genetic algorithms must be assessed contextually. When it comes to a single object, such as a character in an Shooter, it makes little sense to use genetic algorithms from my point of view.
But if you have a game where several objects have to find the way at runtime, like in an RTS game, then it makes sense to use genetic algorithms.
It also becomes interesting if an object has to move over a map, where there are e.g. areas that cause damage. (Acid in an RPG) here the direct, fast way doesn't always make sense but the one where you get the least damage for example. This is a very good area to use genetic algorithms, because you can realize the damage areas with a penalty function. This damaging function would then, for example, reduce the value of fitness.

Application in the field of AI

A field of application for genetic algorithms is also the area of artificial intelligence. This is called Evolutionary Artificial Intelligence. For example you can adjust the gameplay at runtime so that the longer the game lasts the better the player gets. However, in such a context the traditional methods of crossover and mutation are used somewhat differently. An example can be found in the paper by Siddhesh V. Kolwankar with the title Evolutionary Artificial Intelligence for MOBA / Action-RTS Games using Genetic Algorithms. There Siddhesh wrote:

> On the contrary to the traditional methodology of the Genetic Algorithms, Crossover and Mutations now may happen infrequently and without a definite sequential order. The individual in the population consists of the parameters necessary to drive the AI into the gameplay. The crossover points

> are the interactions of the AI with its allies in the game while the encounters with enemy AI serves as the Mutation points. [10]

With the help of genetic algorithms the evolutitonary AI is defined especially for MOBA and RTS games. This Ki is able to change and optimize itself to meet the current needs of the team. Furthermore, it enables the KI to react multi-reactively to the enemy's behavior.

A slightly different approach is case-injected genetic algorithms. With this kind the playing of the computer game can be learned. Sushil J Louis and Chris Miles describe this as follows:

> Case-injected genetic algorithms combine genetic algorithm search with a case-based memory of past problem solving attempts to improve performance on subsequent similar problems. The case-injected genetic algorithm improves performance on later problems in the sequence by learning from cases recorded earlier in the sequence.[11]

In order for the case-injected genetic algorithms to work better, game recordings from human games can also be used to support optimization of individual problems.

Optimization of strategy game with genetic algorithm und swarm intelligence

Since genetic algorithem is an optimization method, it can of course also be used to optimize strategy games. In a strategy game there are a number of factors that have to be considered. Basic building, resource procurement, technology, attack and defense coordination and all this at an extremely high speed. After each of these components you can also optimize or balance between the areas. In addition, in most strategy games there are also different unit types per race. This means that the individual races also balance each other out, so that no race is too strong.
On this topic there is an interesting paper by Thierry Fayard with the title Using a Planner to Balance Real Time Strategy Video Game[8]. This paper looks at the balancing between the races of Starcraft. The aim here is to examine the action sequences in relation to certain parameters. Thus it can be determined which parameters of a race still have to be adapted. An interesting application case with the same topic is the use of genetic algorithms in combination with swarm intelligence. The accompanying paper Combining genetic algorithm and swarm intelligence for task allocation in a real time strategy game [9] by Anderson R. Tavares, Gianlucca Lodron Zuin, Héctor Azpúrua and Luiz Chaimowicz examines the interaction of genetic algorithms and swarm intelligence in connection with RTS games. Genetic algorithms and swarm intelligence are used to coordinate the distribution of tasks.

Generative Level Design

The use of genetic algorithms is also possible in the area of generative level design. Existing processes for creating levels in computer games are time-consuming and expensive and lead to a static environment that cannot be easily adapted. Clearly, the use of a creative system that automatically designs levels would allow an independent game. With an appropriate procedure it is also possible for small teams to design large and creative levels that would otherwise require a large development studio and team. This can also enhance the gaming experience. A comparable statement can also be found in The Evolution of Fun: Automatic Level Design through Challenge Modeling by Nathan Sorenson and Philippe Pasquier:

> The creation process is driven by generic models of challenge-based fun which are derived from existing theories of game design. These models are used as fitness functions in a genetic algorithm to produce new levels that maximize the amount of player fun, and the results are compared with existing levels from the classic video game Super Mario Bros.[7]

Also conceivable would be an application in a Jump & Run in which the levels are generated in such a way that the difficulty degree of the level corresponds to the abilities of the player. This way an individual gaming experience can be created for each player and frustration caused by too difficult levels can be avoided.
I think, however, that for an analysis of the game behavior, you need an additional component such as a pattern recognition.

Implementation

Before we go into the implementation in more detail, we should first consider that a more extensive implementation was planned. For example, a second use case was to be implemented in addition to the pathfinding. However, since the implementation proved to be more complex and extensive than expected, the second prototype was not needed.

Furthermore, only a selected set of algorithms were implemented to demonstrate the approach. The structure of the scripts allows an extension of further algorithms without any problems. This also applies to special algorithms not mentioned in this document.

The complete source code with the prototype and a PDF version of this document will be made available to the community via GitHub. `https://github.com/Modius22/genetic-algorithm-and-unity3d`

Overview of implemented procedures

For the implementation, it was first necessary to identify everything that needed to be done. This resulted in 5 scripts / classes to be implemented for the genetic algorithm and another one for the player behavior (allele):

- Player.cs
- Population.cs
- NaturalSelection.cs
- ArtificalSelection.cs
- Recombination.cs
- Replacement.cs

Player.cs

All information and functions relevant to the player are stored in this class. In the case of pathfinding, the information is on movement, collision behaviour and respawn in a new round. The player reflects an allele in the context of genetic algorithms, of which several form the population. For this reason, individual-specific information is also stored in the class.

Overview Variables Player.cs

There are some variables in this class. For this reason I will only deal with the most important variables from my point of view:

Vector3 spawn	Spawnpoint for Allel
float moveSpeed = 40	motion speed of an allele
float maxSpeed = 8.0f	Maximum speed of the allele
int brainSize = 300	Initial brain size
Vector3[] brain = new Vector3[brainSize]	Generate the brain array
int lifespan = 15	Life span of an allele
bool reachedTheGoal = false	Has the goal been achieved?
bool dead = false	Is allel dead
float fitness = 0	Fitness of an Allele
float rating = 0	Rating of an Allele
float relativeFitness = 0	Relative fitness of an Allele
float distToGround	Distanc between Allel and Ground
float distToGoalFromSpawn	Distance from start to finish point

Overview Functions Player.cs

A set of functions has been implemented in this class. I just want to give a general overview of the functions I created.

MovePlayer()	Used to move the allele
OnCollisionEnter()	Checks for collision
Respawn()	Spawn the allele in the next generation
Die()	Lets an allele die
SetAsChampion	Sets an allele as champion of his generation
IsGrounded()	Determines if the allele is on the ground
GenerateVectors()	Generates the start vectors for the brain in the 0th generation

Population.cs

This is probably the most important class of the project. Population.cs controls the genetic algorithm. Everything concerning the population is managed here. Above all, which algorithms should be used for natural and artificial selection, crossover and recombination,

Overview Variables Population.cs

GameObject player	Game object for allel
GameObject goal	Game object for goal
GameObject champion	Game object for champion
int playerNum	Number of allel per generation
GameObject[] Players	Arral with all player objects
GameObject[] MartingPool	All those selected allel for crossing by artificial selection
int MartingPoolSize = 40	Size of marting pool
float fitnessSum	Sum of fitness of all Allel of an population
float mutationRate = 0.02f	Rate of mutation by recombination (2%)
int minStep = Player.brainSize	minimum of steps taken to reach the goal
int generation = 0	Number of the current generation
bool noWinnerBefore = true	Was there already a winner in the last generation?
long k = 0	Counter for update

Overview Functions Population.cs

controlGA()	This function controls the algorithms for artificial selection and replacement. Furthermore the fitness functions are called here.
PauseAndRespawn()	Respawn new genereation
AllDead()	All allel dead?
ReachedTheGoal()	Checks if the goal has been reached
RespawnAll()	Next Generation Respawn
SetChampion()	Set the champion of the generation
IncreaseLifespan()	Increases the lifespan of a allel
SpawnPlayers()	Spawn of a single player object
CalculateFitness()	Calculation of fitness
CalculateFitnessSum()	calculation of the total fitness
Mutate()	testing and carrying out mutations

NaturalSelection.cs

In this class all algorithms concerning the Natural Selection are defined. So far only one algorithm has been implemented. Other algorithms that can be implemented are Ranked-Base and Competition Selection.

Overview Functions NaturalSelection.cs

FitnessProportional()	Implementation of the Fitness Proportional Selection Procedure

ArtificalSelection.cs

In this class the artificial selection algorithms are realized.

Overview Variables ArtificalSelection.cs

GameObject[] sorted	Sorted list of allel of one generation by rating

Overview Functions ArtificalSelection.cs

RoulettePrinciple(int num)	Implementation of roulette principle
StochasticUniversalSampling(int num)	Implementation of stochastic universal sampling

The parameter *num* specifies the size of the marting pool.

Recombination.cs

In this class the recombination algorithms are implemented. I have implemented two algorithms here. a one point crossover and an n-point-crossover (two-point-crossover). Not implemented were template-crossover, uniform-crossover, schuffle-crossover. If these algorithms are needed for a problem, the class could be extended accordingly.

Overview Functions Recombination.cs

OnePointCrossover(GameObject P1 , GameObject P2, int brainsize)	Implementation of one point crossover.
TwoPointCrossover(GameObject P1 , GameObject P2, int brainsize)	Implementation of two point crossover.

Two alleles ($p1$ and $p2$) and the size of the brain (*brainsize*) are given as parameters.

Replacement.cs

In this class, the replacement scheme is implemented. From here, a corresponding recombination algorithm is called. Three schemes have been implemented (general replacement, principle of the elites, weak elitism). The delete-n-latest schema was not implemented.

Overview Functions Replacement.cs

GeneralReplacement(int recombination, int brainsize)	Implementation of general replacement
PrincipleOfTheElites(int elite, int recombination, int brainsize)	Implementation of principle of the elites
WeakElitism(int elite, int recombination, int brainsize)	Implementation of weak elitism

The parameter *recombination* transfers the recombination algorithm, *brainsize* transfers the brain size and *elite* transfers the number of elites for the population.

Demo: Pathfinding

The implementation was based on a project existing on GitHub (https://github.com/hobogalaxy/EvOLuTIoN). In this project there are already the basics for the visualization of the pathfinding path and the single allele of a generation. In the project, three example routes were implemented. For my prototype I only used the second track.

In the project the pathfinding was also realized with genetic algorithms, but not with the general algorithms but rather with a very special approach. The components of the genetic algorithm were replaced by my own classes. Thus one has the possibility to select the procedures for selection, replacement and recombination by means of corresponding variables. This allows us to test the algorithms implemented in each area.

I tested the whole thing with all the algorithms I implemented in several runs. This allowed me to identify the best combination of algorithms for the existing pathfinding problem. I summarized the whole thing in the following table:

Algorithm Set			Generation Information										
Selection	Replacement	Recombination	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	Avg.
RP	GR	OPC	41	26	30	43	66	43	43	30	34	47	40,3
RP	GR	TPC	39	40	40	30	37	33	38	42	42	33	37,4
RP	POTE	OPC	23	35	40	32	36	26	33	31	32	33	32,1
RP	POTE	TPC	30	31	32	39	30	36	34	30	31	32	32,5
RP	WE	OPC	28	29	39	25	25	32	26	27	32	27	**29**
RP	WE	TPC	30	30	27	34	30	26	25	29	30	36	29,7
SUS	GR	OPC	62	58	53	56	61	59	43	41	82	51	56,6
SUS	GR	TPC	47	72	53	53	40	70	48	101	61	49	59,7
SUS	POTE	OPC	29	32	35	31	25	27	33	29	29	30	30
SUS	POTE	TPC	35	34	36	34	37	48	35	31	41	38	36,9
SUS	WE	OPC	20	35	26	40	29	37	30	28	28	21	29,4
SUS	WE	TPC	30	40	29	37	48	33	29	33	37	30	34,6

RP: RoulettePrinciple, SUS: StochasticUniversalSampling, GR: GeneralReplacement, POTE: PrincipleOfTheElites, WE: WeakElitism, OPC: OnePointCrossover, TPC: TwoPointCrossover

The table shows that the combinations of the different algorithms behave differently. In the case of finding the way, the Descending three would be the best.

- Roulett Principle, Weak Elitism, One Poin Crossover (Average 29 Generations)
- Stochastic Universal Sampling, Weak Elitism, One Point Crossover (average 29.4 generations)
- Roulett Principle, Weak Elitism, Two Point Crossover (average 29.7 generations)

It is remarkable that all three combinations contain the Weak Elitism. So this seems to be a procedure that works well in the context of pathfinding.

Conclusion

In general, the use of genetic algorithms in connection with game development is very interesting. As you can see in the Use Cases section, there are some use cases where genetic algorithms can be used. In general, this has to be decided from situation to situation.
My goal with this work was to give an introduction to genetic algorithms and to provide classes to integrate a genetic algorithm within a game project.
During the implementation I have to realize that this is a rather complex topic, which I underestimated at the beginning. This has the consequence that not all methods in the context of pathfinding make sense and therefore were not implemented by me.
Furthermore a second prototype was planned. Due to the complexity I didn't have the time to implement this prototype. Nevertheless I would like to briefly describe in two sentences what this prototype would be capable of.

In the prototype it should be a matter of letting a project target, e.g. a shot from an artillery vehicle, hit a target. In an extended version, the artillery vehicle could also move. A realization would have been in 2d.

My conclusion is, it is exciting to use genetic algorithms in games. I think I have given a good introduction to the subject even if it is a complex and extensive topic and not every single procedure has been considered or implemented.
As an extension of this work, the future implementation of further prototypes and algorithms would be conceivable, as well as the combination of genetic algorithms with neural networks through the Unity Machine Learning Agents Toolkit.

List of Figures

General Ressources

1 Bianca Selzam: Genetische Algorithmen, `http://ls11-www.cs.tu-dortmund.de/lehre/SoSe03/PG431/Ausarbeitungen/GA_Selzam.pdf`

2 Maik Buttelmann und Boris Lohmann, Optimierung mit Genetischen Algorithmen und eine Anwendung zur Modellreduktion, `https://www.rt.mw.tum.de/fileadmin/w00bhf/www/publikationen/2004_Buttelmann_at.pdf`

3 Frances Buontemo, Genetci Algorithems and Machine Leraning for Programmers, Pragmatic Bookshelf, Book Version 1

References

[1] Holland, John: Adaptation in Natural and Artificial Systems; The University of Michigan Press, 1975

[2] Rechenberg, Ingo: Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution; Frommann-Holzboog-Verlag, Stuttgart 1973

[3] Schwefel, Hans-Paul: Evolutionsstrategie und numerische Optimierung. Dissertation; TU Berlin, 1975

[4] Buttelmann, Maik und Lohmann, Boris: Optimierung mit Genetischen Algorithmen und eine Anwendung zur Modellreduktion,https://www.rt.mw.tum.de/fileadmin/w00bhf/www/publikationen/2004_Buttelmann_at.pdf, 2004, Seite 153/154

[5] Ryan Leigh, Sushil J. Louis, and Chris Miles: Using a Genetic Algorithm to Explore A*-like Pathfinding Algorithms,http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.666.4409&rep=rep1&type=pdf, Abstract

[6] Ulysses O. Santos, Alex F. V. Machado and Esteban W. G. Clua: Pathfinding Based on Pattern Detection Using Genetic Algorithms , http://sbgames.org/sbgames2012/proceedings/papers/computacao/comp-full_09.pdf

[7] Nathan Sorenson and Philippe Pasquier: The Evolution of Fun: Automatic Level Design through Challenge Modeling , http://axon.cs.byu.edu/Dan/673/papers/sorenson.pdf

[8] Thierry Fayard: Using a Planner to Balance Real Time Strategy Video Game, http://icaps07-satellite.icaps-conference.org/workshop8/Using%20a%20Planner%20to%20Balance%20Real%20Time%20Strategy%20Video%20Game.pdf

[9] Anderson R. Tavares, Gianlucca Lodron Zuin, Héctor Azpúrua und Luiz Chaimowicz: Combining genetic algorithm and swarm intelligence for task allocation in a real time strategy game, https://seer.ufrgs.br/jis/article/view/56146/43683

[10] Siddhesh V. Kolwankar: Evolutionary Artificial Intelligence for MOBA / Action-RTS Games using Genetic Algorithms, https://pdfs.semanticscholar.org/c93b/c141a1d4e8d4f90c0e9bf8dc766cee294945.pdf

[11] Sushil J Louis and Chris Miles: Combining Case-Based Memory with Genetic Algorithm Search for Competent Game AI, https://pdfs.semanticscholar.org/6c4a/512739378cec374b96887def4a5327bc621a.pdf?_ga=2.216196566.1165910270.1564300046-2089667493.1564300046

[12] https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover