

Gheorghe Asachi Technical University of Iași
Faculty of Automatic Control and Computer Engineering
Department of Computer Science and Engineering
Master: Distributed Systems and Web Technologies

Development and Integration of Genetic Algorithms in Unity

Dissertation Thesis

Scientific Coordinator
Assist. Prof. Cristian Buțincu

Graduate
Robert-Ilie Vicol

Iași, 2020

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII LUCRĂRII DE DIZERTAȚIE

Subsemnatul VICOL ROBERT-ILIE,

legitimăt cu seria MZ nr. 320825 , CNP 1960701226791

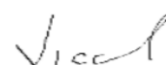
autorul lucrării DEVELOPMENT AND INTEGRATION OF GENETIC ALGORITHMS IN UNITY

elaborată în vederea susținerii examenului de finalizare a studiilor de masterat organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi” din Iași, sesiunea TOAMNĂ a anului universitar 2020, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi” din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data

04.09.2020

Semnătura



Iași, 2020

Contents

Introduction.....	1
Chapter 1.Fundamentals and Theoretical Considerations	3
1.1 Genetic Algorithms	3
1.2 Previous applications	5
1.3 Achieving motion.....	6
1.3.1 Animation driven movement	6
1.3.2 Physics driven movement.....	7
1.4 Application requirements	8
Chapter 2. Application Design	11
2.1 Hardware specific details	11
2.2 Main application modules	11
2.2.1 Population Component.....	11
2.2.2 Evaluation Component.....	11
2.2.3 Genetic Operations Component.....	12
2.2.4. Replay Component.....	12
2.2.5 Menu Component.....	12
2.2.6 Genetic Algorithm component.....	12
2.2.7 General Character Component.....	12
2.3 Design Advantages and Disadvantages.....	13
2.4 Limits and calibration	16
2.5 Software details	17
2.5.1 Technologies	17
2.5.2 Classes. Modularity. Diagrams	18
Chapter 3. Implementation.....	23
3.1 Implementation of the Genetic Algorithm	23
3.1.1 Genetic Algorithm – Representation.....	23
3.1.2 Genetic Algorithm – Mutation.....	25
3.1.3 Genetic Algorithm – CrossOver	26
3.1.4 Genetic Algorithm – Elitism.....	26
3.1.5 Genetic Algorithm – Selection.....	26
3.2 Issues encountered	26
3.3 Original ideas and solutions	30
3.4 Application look	31
3.5 Communication between components	33
3.6 Data storage.....	33

3.7 User Interface	34
3.8 Software calibration.....	34
Chapter 4. Testing and Experimental Results	36
4.1 Application setup	36
4.2 Observing the genetic oprators	36
4.3 Determinism of the Physics Engine.....	37
4.4 Experimental results.....	37
Conclusion	41
Bibliography.....	43
Annex.....	48
Annex 1	48
Annex 2.....	52
Annex 3	57
Annex 4.....	61
Annex 5.....	66
Annex 6.....	70
Annex 7	71
Annex 8.....	72
Annex 9.....	73

Development and Integration of Genetic Algorithms in Unity

Robert-Ilie Vicol

Abstract

Animation's importance, as a component in the context of developing applications, simulations or video games, has increased drastically recently. Perhaps this is partly owed to the significant increase in computational power available, but also due to the high expectations for the interaction with complex systems, that are able to immerse the users in their created environments. Nowadays some applications have moved away from the esthetic role of animation and started providing more practical purposes to this concept. We can take smartphones as an example, with their visual feedback mechanism on touch, scroll and other similar events. For this study however, we choose to focus on another purpose fulfilled by animation – replicating natural behavior, met in our day to day lives, in order to give users a deep, close to reality experience. Thus, considering animation as means to achieve motion in a visual simulation, we notice the following aspect: the closer to reality a representation truly is, the higher the animation complexity and therefore, the computational power and effort needed to build it. In this context, a certain question arises: Can we use a different method for building motion in an environment, without losing the realism of the designed level? This thesis analyzes the efficiency of animation – as it is defined and built in game development and simulations – and the possibility of having it replaced with other means of generating natural motion.

As such, we created a simple environment in which we attempt to move a single individual in a natural way. For this purpose, we chose to implement genetic algorithms, which control the individual's (a human model, in this case) motion. By completely removing the animation component, the only way to generate motion (without forcing the translation or rotation of the objects), is by applying forces on the individual. These fall into two distinct categories: there are external forces applied by the environment (such as gravity) and there are internal forces, generated by the individual (such as torque, which is regularly used by humans in reality to be able to move, walk, pull and push objects around). Out of the two categories, the implemented genetic algorithm controls the forces generated by the individual, modifying them in order for the candidates to get closer to the desired goal.

The results of our analysis show the algorithm's capacity to simulate the individual's movement for fulfilling basic tasks, such as traversing space from a start point to a destination point. The successful imitation of these movements reveals that by exploiting the genetic algorithms, we could, in theory, achieve replication of complex behaviors, such as the human walking ability, without using animation for the motion control. Another advantage to this approach consists of the ability to naturally react with other elements in the environment, without needing the developer's involvement (foreseeing and handling the potential interaction). A simple example here would be the individual's collision with a foreign object. There are, however, side effects to this approach too: The population size, as well as the fitness function complexity, will increase together with the complexity of the desired behavior, if we are to obtain quality solutions.

Therefore, the use of genetic algorithms can be a valid alternative to animation, worthy to be considered in the areas of visual simulation and game development.

Introduction

Realism, as an artistic movement, has been discovered in the XIXth century, as an esthetic ideology that focuses on the connection between art and reality. As the name implies, the unique aspect about this movement lies in the creator's tendency to be an objective observer, attempting to obtain a true and clear reflection of reality through his art. As such, people have realized the importance of representing details accurately in their creation almost two hundred years ago. Even if we do not see this in art as frequently anymore, the influence of this tendency can still be felt nowadays, since there are a lot of areas which make use of representing accurately different aspects of our day to day lives. Out of these areas, we will focus on video games and visual simulations.

In this context, realism is desired because the accuracy of the representation helps the application offer credibility and deliver immersion: as long as the created world feels real, users are able to experience a sensation of being spatially located in the generated environment" [1]. The actual notion is called Spatial Presence and basically revolves around the user feeling like he is "there", in the created world. Therefore, we can understand that sensory perception plays an important part in this phenomenon. This is easily noticeable if we look at some recent titles that have been released. These put an increased focus on refining computer generated imagery and sound effects, in order to maintain this illusion: "What you have is you, standing on the edge, looking around and just being in awe of what you're seeing. In games, I think we can do that", as Cory Barlog stated [2]. Even the tactile sense has started being valuable in this context, if we consider virtual reality or basic controller functionalities (for instance, vibrating while the user is arming a bow in a game). All this considered, sensory information, be it sight, hearing or any of the other, is not the only factor that needs to be taken into consideration when attempting to represent a corner of reality. For the reflection to actually be completely accurate, it is just as important to be able to correctly capture the behavior or interaction between the elements which are creating the environment as a whole.

As such, the most frequently used way of mapping an element's behavior is through use of animation. Being a technique that simulates movement, animation is extremely useful for defining the actions an element can perform, be it alive or simply an object. Moreover, since the animations can be mapped over user input, the actions they simulate can be controlled and called by the user directly. There are, however, certain side effects or limitations to this approach too. One of these would be that for each action a simulated individual can perform, we need a corresponding animation. Thus, the complexity of the behavior impacts the complexity of the animation component, which will, in turn, be limited by the hardware capabilities. Moreover, the effort needed to create the simulated actions scales with the complexity of the animation component. Another aspect, worth mentioning, is that animations are not able to adapt when the individual interacts with foreign objects, if the said interaction has not been foreseen and handled by the developer. This basically means that the creator is tasked with finding and fully understanding every possible interaction use case in his designed environment. Because of that, any "forgotten" interaction will render the individual unable to behave naturally, thus hurting the level of realism of the simulation.

This thesis analyzes and tries to find a way to represent an individual's behavior, without placing this large responsibility on the creator, of predicting every single type of interaction with the environment. In order to find a suitable match for our solution, we need to recap what we have analyzed so far. This helps us elaborate the reasoning behind our algorithm choice and also, at the same time, it shines a light on the further ideas and concepts, which will be further detailed in this paper.

Thus far, we have deduced the following requirements for our potential solution to the presented problem:

- We need a way to make sure the simulated individual is able to adapt to interactions, even when those interactions were not foreseen by the developer.
- We are looking for a potential solution that can replace animation or at least accompany it, but it is important for this solution to be better than animation in the sense that we need its difficulty to not equally scale with the complexity of the behavior that is desired to be imitated.
- Ultimately, this solution would be required to accurately represent movements which are found in realistic scenarios. Therefore, it could help to work on a model based on real life patterns.

By analyzing all these requirements, it becomes clear that genetic algorithms could be a potential match for our solution. If we look into their definition, we find out that these are a subset of evolutionary algorithms which specialize on attempting to solve optimization problems by traversing a solution search space and finding the best candidates [3]. The concept is based on the real-life model detailed by Darwin's theory of evolution, in which it is explained that only the best individuals that can adapt are able to survive. Thus, the search for solutions is not performed randomly, but in a manner that is facilitated by the usage of genetic operators such as selection, mutation, and cross-over. The best candidates are chosen based on a fitness function, which is basically the function that one desires to optimize.

Looking back on the presented problem, we can further notice that the potential candidates found by genetic algorithms could be able to adapt to a certain extent, even when the developer doesn't take some interactions into consideration, since the populations are evaluated directly in the same environment that they are a part of. Thus, the responsibility of finding interactions shifts from the developer to the actual solution. Moreover, since the genetic algorithm remains basically the same for most problems that we try to solve, the only effort needed is to find a corresponding fitness function for each action that needs to be represented. While this can prove difficult in some cases, generally good results can be obtained, if there is a clear connection between the problem and the fitness function. Lastly, we motivate our algorithm choice by considering the real-life model it is based on. With a population large enough and plenty of iterations, there are chances we could obtain an accurate behavior which could accompany or even replace animations.

For the creation of the environment, as well as the individuals, we used the Unity 3d game engine and for the actual implementation of the algorithm we used C#, being an object-oriented programming language that boasts several templated containers which helped with handling and processing the large data sets. The internal Profiler tool was used to mark delays in the components, track down the most time-consuming tasks and – as the end result – help reduce the computing power needed to run the algorithm.

The following chapter will provide a theoretical basis for the thesis, revealing the starting point of our work and the concepts around which it revolves. Afterwards, the design of the application will be presented, together with the main modules and functionalities, certain advantages and limits of the implementation we used to illustrate the concepts that the thesis elaborates. Furthermore, we will detail the main software components and some hardware requirements, followed by the actual implementation logic presentation. There, we will also elaborate the issues we faced during development and how we worked around them. Lastly, we will show some test results and present our conclusion, together with a few directions in which our research could be continued.

Chapter 1. Fundamentals and Theoretical Considerations

This chapter takes an in-depth look into the genetic algorithm's structure and reveals some of the previous ingenious ways they have been applied in our presented areas of interest. Furthermore, it will shed a light upon a few currently available applications and how these applications compare to our ideas and interests. Finally we will elaborate upon what our research and our application are trying to achieve.

1.1 Genetic Algorithms

In this subchapter we shall define the genetic algorithms, try to explain why they were needed and reveal what kind of problems they solve best and what limitations we can encounter while using them.

Genetic Algorithms were introduced by John Holland as search and optimization techniques with basis formed around Darwin's theory, detailing the principles of survival of the fittest individuals and natural selection. Since then, multiple ideas have been refined and adopted, to the point where this method has become a viable solution, in specific areas. Thus, while being very different from classic algorithms, that give an answer to a problem based on input, these techniques have proven to be an excellent way to solve search and optimization problems by finding good candidates in a discrete space of potential solutions. While, at times, this approach will not provide the best answer (due to several side effects, such as converging towards a local optimal point), it has several benefits over traditional algorithms. Just a few of these advantages consist of:

- Always able to provide a solution, which gets better and better in subsequent iterations.
- Modular, easy to apply in different areas and easily modifiable in order to find solutions to different problems. As explained already, as long as we are able to find a mapping between the real world input and the actual data that can be processed using genetic operators, the only adjustment (that is mandatory to be done) is the fitness function, which we desire to get an optimal solution for.
- Not prone to getting stuck on local optimal points (compared to other heuristics in the same class, like hill-climbing [4]), due to the variation that can be applied on the recombination and selection procedures, thus increasing exploration over exploitation. This also explains why these algorithms work fine in "noisy" environments [5].
- Certain steps of the algorithm (several of the known selection procedures, fitness computations) can be easily parallelized [6], thus the algorithm can work on distributed systems.
- Lastly, it can find solutions even with problems of multi-object optimization, being able to minimize or maximize more than one function at once (possibly by creating a fitness function by combining all the functions that are needed to be optimized) [6].

There are variations regarding the structure of a genetic algorithm. However, there seems to be a consensus that uses the following general steps:

Representation – a mapping needs to be done between the input data (phenotype) and the processable data (genotype). Here we also need to define certain genetic notions, in order to better understand this stage.

- Genes, that contain the actual data, exchangeable information
- Chromosomes, which represent the candidate solution and are formed by a group of genes.
- Populations, comprised of a group of chromosomes.

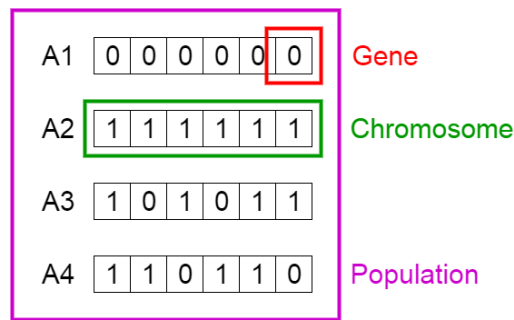


Figure 1. Genetic Representation [7].

Selection – at this stage, the parents which will generate offspring are being chosen. There are several ways of achieving this, but every single way needs a fitness method, in order to measure the efficiency of the chromosomes – their ability to adapt. The variety of selection methods appeared due to two big factors. On one hand, execution time – some selection methods are complex (such as the rank selection, which requires a sort for every run), while others are straightforward (such as the tournament selection or elitism). On the other hand, different methods of selection have been developed in order to strike the right balance between exploration (focus on searching the solution space) and exploitation (focus on selecting only the best individuals in order to find an optimal solution). This balance has to be shifted, depending on how big the search space is and how many local optimal points the fitness function has.

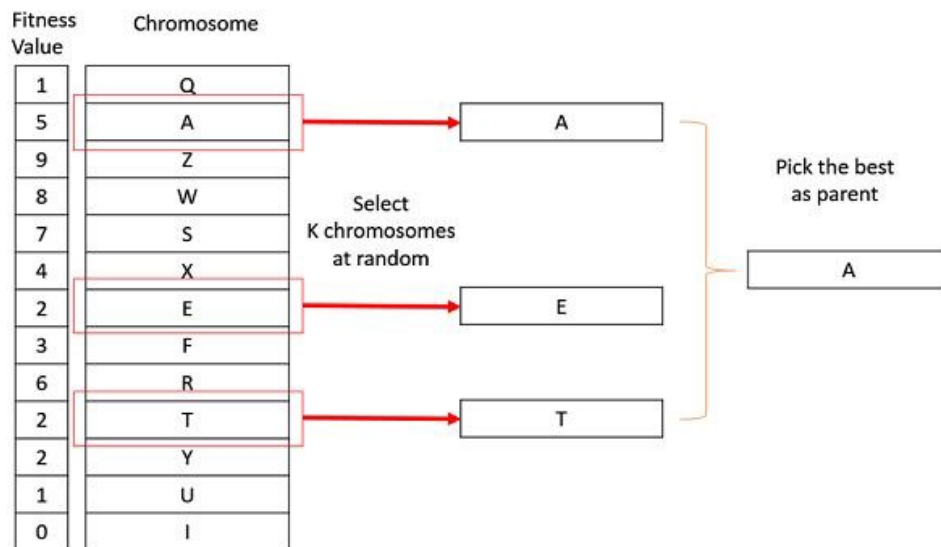


Figure 2. Tournament selection [8]

Recombination – this step mixes the genetic information from the selected individuals, in order to generate offspring. The crossing over procedure occurs by swapping genes between chromosomes. The resulting candidates will form the new population, which will perform better, if the exploitation has more weight than exploration, Otherwise, the evaluation will have an unknown result, since the generation would be focused on covering a more diverse portion of the solution space.

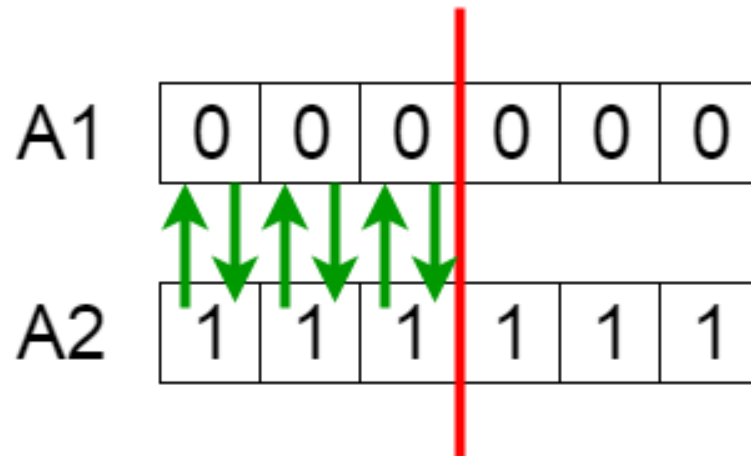


Figure 3. CrossingOver with one cut point [7]

Mutation – this step is highly important, since it aids the algorithm in getting out of the local optimal points. It will also cause a shift in balance towards exploration (as opposed to exploitation), based on how high the frequency parameter is set. This can have direct consequence on the diversity of the population, but setting the frequency too high will amount to similar results as if we would be using random individuals. The mutation chance is usually a lot smaller than the recombination chance, being considered a secondary genetic operator as opposed to crossover.

Before Mutation

A5

1	1	1	0	0	0
---	---	---	---	---	---

After Mutation

A5

1	1	0	1	1	0
---	---	---	---	---	---

Figure 4. Mutation [7]

Another important (albeit optional) step in the algorithm consists of Elitism: the most adapted individual is preserved through generations, in order to have a higher chance of achieving convergence to the optimal solution starting from any arbitrary initial population [9]. This technique, of keeping the best fitted candidates alive, has a noticeable effect on the optimization speed of the genetic algorithm [9].

1.2 Previous applications

Genetic algorithms have been used, for the past few years, with great effect and influence over the quality of life of some video games and simulations.

In a simulator that manages the development of civilizations, Watson et al came up with the idea of using genetic algorithms for optimizing the in-game process of building a city, believing that it would lower the effort and need of micro-managing the diverse aspects of this activity [10]. They used an interesting representation for this, mapping town building factors to genes, actual

towns to chromosomes and the entire civilization to the population. The purpose of the genetic algorithm was to find the most efficient and easy way of rising the cities using all the in-game available town building tools.

Genetic algorithms were also used by Cole et al, in order to fine tune the behavior of a first-person shooter bots [11]. Realizing that the development time increases together with the number of parametrized rules, they went ahead and automated this process by having the genetic algorithm tweaking bot settings, such as weapon preference, aggression and path preference. The end result was that the bots had behavior comparable in complexity to the ones obtained by having experts manually tweak the strategy in a trial and error process.

A closer application to our area of interest was found while researching Karl Sims' paper on creating virtual creatures and behaviors in a 3d world environment [12]. By having the genotype mapped to directed graphs in which nodes are body parts and edges are simply connectors between these body parts, he managed to create a huge number of creature designs and select out of those, only the ones which are best suited for certain tasks, such as swimming, jumping, following and walking.

Through our research and application, however, we try to achieve an accurate representation of some of these actions when the body structure of the individual is already established. Since we try to replace or accompany animation, we cannot change the character's aspect or structure (for instance, adding an extra limb). Taking that into consideration, our representation stage doesn't need to include an encoding for the individual's body structure, since that is not subject to change. What we are trying to adapt is the way the characters move in the environment and as such, the focus of our representation needs to be connected to motion and means to achieve it.

1.3 Achieving motion

Therefore, in order to figure out how to map our genes and chromosomes to motion, we need to understand the means through which we can achieve basic motion.

1.3.1 Animation driven movement

Like we already mentioned, animation is one of ways we can simulate movement and probably the one most used today in the industry.

We can define animation as being a sequence of taken snapshots that are manipulated by showing them in quick succession, creating the illusion that they are moving [13]. When referring to the area of video games, animation becomes a trickier topic, as opposed to other areas, such as movies. In films, the watcher is always limited to the viewpoint of the camera. Thus, the animator is allowed some breathing room, meaning he knows exactly from what point his product is going to be looked at and evaluated. This aspect means that the creator needs to focus only on the details which are observable from the camera viewpoint. In a video game, this whole use case is changed. There are certain difficulties – specific to this area – that arise. The easiest one to notice is that the animation must look good from every angle, simply because the camera component, in a video game, can be usually moved around by the player [13], making it easier to spot flaws that break the immersion. Especially when combined with methods of processing real time movements, such as motion capturing the actor's performance, animations are able to aid in the delivery of an immersive and accurate virtual environment [14]. It would be a shame not to give credit to the tremendous progress done in this area. There are even recent projects, such as God of War (2018), in which the whole movement animation content is modeled starting from mocap data sets [15].

Thus, when used together with a control component, usually some form of state machine,

animations can simulate complex behavior in motion using a limited amount of computing resources. Memory usage can even be reduced by storing less data and by generating more data on the fly, albeit this does increase the number of operations that the processor needs to perform. Therefore, a balance has to be found between these two resources, while using real time animation [16].

Considering examples of complex behaviors in movement, we can take the basic act of human walking. Albeit looking basic and simple for us, real people, walking is an incredibly complex, diverse and subtle action, which can convey a large amount of information [17], if represented accurately.

However, there are some drawbacks to this approach, which made us wonder whether there was any way of conveying such movement realism while relying on animation as little as possible. The most immediate issue that comes to mind when relying on animation is the complexity involved in creating them. This process is not easy or routine, it is quite unique for every action, requiring a deep understanding of the knowledge to be conveyed, as well as understanding the users of that knowledge [18]. This suggests that the creator needs to be specialized in the category of motion he is trying to represent. He needs a deeper understanding of the movement's mechanics than a basic observer, normally would have. Another flaw in using just animation for controlling movement can be noticed in data driven methods. Motion capture or specific animation styles can be created and provide a smooth and accurate movement representation, but this technique is restrictive, in the sense that only the precomputed actions are able to be achieved [19].

Thus, we set out to find an answer for our question and try to find a way to represent motion without using animation. Keeping this in mind, there are two other ways through which we can achieve what we set out to do. One option is to directly modify the position and the rotation of our character, thus achieving motion either directly (teleportation like), or continuously (by incrementing the position of the individual frame by frame, or by directly setting a velocity to it). While this approach seems simple and could be easily mapped over the genetic data types, it doesn't seem to fit our purpose because of two main reasons:

- 1) Updates will occur frame by frame, which means we are relying on a stable framerate to achieve deterministic results during a simulation. This can hurt the effectiveness of the genetic algorithm, since the same candidate can perform differently in separate simulations.
- 2) We do not wish to apply motion manually, from an external source. This beats the purpose of what we are trying to achieve. The whole point of it is to have the character rely on his own ability to move by using his own mass and forces.

1.3.2 Physics driven movement

The second option relies on a more physics-based approach and is the opposite technique to data-driven approaches, named controller approach [19]. The idea is to give the individual means to make use of properties such as mass, resistance, inertia and take into consideration external forces from the environment too (such as gravity), while attempting to move. This movement would then be created by means of internal forces (torque). This would be a more accurate representation of the real-world scenario and also, it would restrict the individual's movement within its own precomputed limits. While this has the benefit of allowing, in theory, any action to be performed, the downside is that, usually, the actions represented in this manner are not as realistic as the ones from data-driven approaches [19]. However, that is exactly what we will be trying to experiment with.

Other new ideas are trying to somewhat combine the two methods, by having the physics-

driven motion controller learn certain actions from the animation data-driven approach [22]. As we can see in the video, by using a multi-term reward focused on important features (such as movement and heading direction, speed and locomotion style) and deep reinforcement learning, the physics-controlled character is able to successfully replicate actions in a realistic manner.

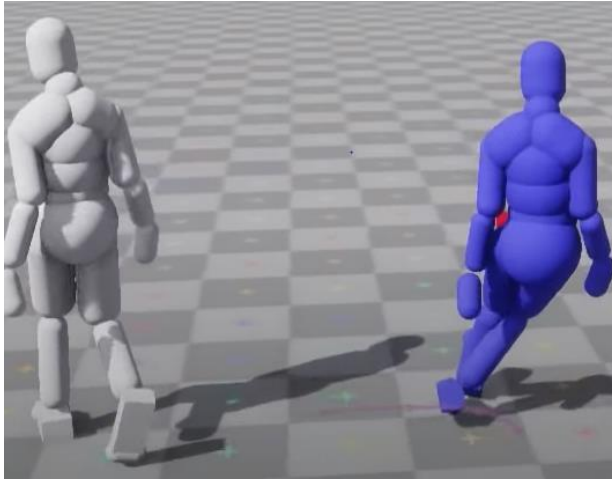


Figure 5. Ubisoft reinforcement learning example [20].

The white character is kinematically driven and is being maneuvered by controller-based input. Direction and movement are the two features being controlled [20].

The purple character is being controlled simply by physics, trying to replicate the actions of the white character. [20]

After 30 hours of calibration and running the deep reinforcement learning, the purple character is able to mimic the actions of the white one in a realistic manner, while also being able to keep its balance when hit by other objects. [20]

This ingenious approach was an inspiration to us, since it actually managed to achieve what we are trying to, by different means: the end result (the purple character) is able to simulate realistic behavior, while being controlled by physics. This means the character is able to adapt to any interaction from the environment it is placed in, even if the said interaction is not programmed or foreseen by the development team involved.

1.4 Application requirements

As elaborated in our introduction, we plan to implement a genetic algorithm for our application to be able to provide a simulation for realistic movements of a character in a 3d world environment.

As such, since we have decided upon the means which will be used to have the character move, we need to establish how to map the movement to the genetic data types. We know that without a proper representation of a candidate's attributes, we will not be able to obtain good results from the genetic algorithm [21]. While doing so, we also need to take into consideration how the genetic operators would affect our solution candidates. Our mapping has to be done in a way that simplifies the process of mutation and crossing-over, since the operations need to be performed in real time, while the simulation is rendering the scene. The more complex these computations are, the more noticeable the staggering effect on rendering will be.

We also need a simple environment, on which we can perform the simulation. The character needs to be anchored in a virtual realistic world, in order to hope to achieve realistic motion. Being controlled by physics, the character needs to be able to respond to external forces such as gravity, or forces applied by collision with other objects from the environment. It is also required for the candidates to not be able to influence each other during the evaluation phase of the simulation, in order to make sure their fitness does not depend on the population's size. This can be done easily by removing the candidate's ability to collide with the objects (from the environment) which are of the same type as it is.

We also require an evaluation component, in order to be able to compute the general and

individual fitness, to measure the adaptability of each candidate, and to create composite fitness functions with weight values on the involved parameters, based on what features we consider more relevant in achieving the desired representation. This component needs to be implemented separately, since this has the highest probability of being continuously changed and tweaked, especially when we want to simulate different behaviors. Also, it would be useful to consider profiling this particular component in the early stages, since the evaluation of the individuals can become a complex task, especially if we consider having the computations done every frame of the simulation.

One component is needed to perform the genetic operations on the chromosomes. It will select the parents for producing offspring and also keep some of the best candidates alive for the next generation. This component also needs to be decoupled, in order to allow us to consider a parallel approach for implementing the genetic operations (some types of selection and crossing over can be implemented in a distributed manner).

We also need a population component, which would be able to run a cycle of instantiating and destroying our characters every generation. This component would also be responsible for providing means to check features of interest in our characters in real time, mainly for the evaluation component. These will be used together to create a fitness function with diverse and different variables, hoping to reach the end result of finding a solution for our desired representation.

Another useful feature that would help our analysis is a replay mechanism. This will aid us in finding flaws in our obtained behaviors, and in case some of the better candidates reach their predefined goal, we would need to store them and perhaps compare the resulting representation to real life actions. Lastly, we would need a way to serialize the data we want to store persistently, since the data structure that we need to preserve the forces applied can get complex and large in size quite fast, impacting not only the memory usage, but also the processing time needed for saving a population or replaying it.

A camera that is able to follow and focus on the character's movements is also needed, in order to closely observe the represented behavior in any frame. Be it through animation or any other means, we need more than numbers in order to evaluate the method we use to achieve motion. In order to be able to compare the two techniques (animation and physics motion), we would need to be able to visually observe our character's behavior from different perspectives.

For quality of life purposes, we also need to consider a menu, to be able to access all the different functionalities of the project. This is less important in our case, since the application does not focus on achieving a smooth user interface/experience, but rather on the experiment of replacing the animation component with a physics-based controller. However, the menu still needs to be present, to allow some degree of navigation and selection of functionalities.

Lastly, the application needs a manager component, to be able to synchronize all the other components and reload the environment when one of the simulations ends. This component will also be used to inject dependencies into all the other components, as well as coordinate the activities of all the genetic algorithm related controllers.

Beyond all that, for practical purposes, we need to consider the complex nature of our problem. Finding a solution might take a variable amount of time, based on the stop criteria we choose to implement. Most genetic algorithms stop either when a predetermined upper limit is reached for the number of iterations, or when there is a low chance of achieving any significant changes in subsequent iterations [22]. Either way, it would help to discover ways to speed up the algorithm, particularly the candidate evaluation phase during the simulation. A method of doing that consists of trying to speed up the time component of the environment, such that virtual seconds can be compressed to a much shorter duration. Also, a considerable benefit would come from

starting the simulation without the graphical component. Being able to detach from the rendering phase allows us to run simulations in parallel and perhaps without the need of human interaction to configure and start the simulations. We would also spare the processor a considerable amount of cycles, which would otherwise be spent on feeding the GPU information about the objects and the environments that need to be rendered [23]. Thus, the load put on the processor would consist solely of the physics computations and collisions between objects in the environment.

An example is provided in the following image, where we can see that in a few particular spikes, the CPU load that handles rendering almost halves the number of frames that are shown in a second. The profiling is done on an earlier draft of our application, but it is still relevant for our analysis.

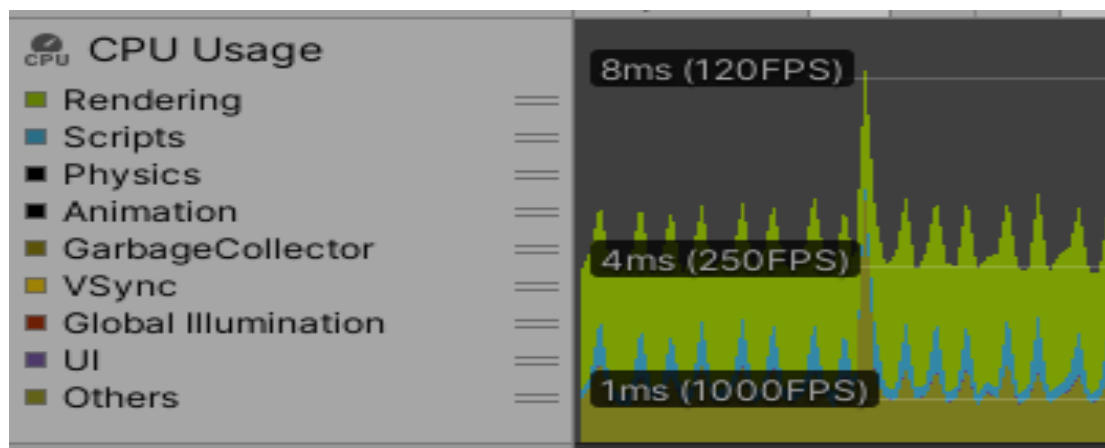


Figure 6. Profiling tool - CPU usage types

In this particular version, we can see that the rendering takes up most processing done by the CPU. Of course, we are sure that in the later versions of our application, the Scripts and Physics parts of the CPU usage will increase, due to the genetic algorithm implementation and the physics computations done in the evaluation phase of the candidates. However, the rendering load will also increase, together with the number of individuals in a simulation, or with the complexity of the environment. Thus, it is obvious that running the application detached from the rendering component will provide noticeable benefits (for instance, beyond the increase in performance, this can allow us to run the simulation without having a GPU in our machine).

Chapter 2. Application Design

This chapter details some hardware requirements needed to run the application, followed by a general presentation on the main modules of the application and how they communicate with one another. We will also focus more on some advantages and disadvantages for our chosen design approach and moreover, we will show the limits in which the application can smoothly run on our configuration. Lastly, we will present in detail the software component of our application.

2.1 Hardware specific details

Being a simulation developed on the Unity Platform, there is a starting requirement of having a supported GPU with DX10, DX11, or DX12 capable. The amount of RAM needed depends from project to project, and even in our case a precise approximation cannot be made, since we will be able to use fitness functions of diverse complexities and variable population sizes, from generation to generation. While the memory consumption will be lower due to lack of animations, we are making use of the physics engine to simulate motion. That means the application will be performing more CPU intensive tasks, since the physics engine and all its floating point mathematical computations run entirely on the CPU. Therefore, the minimum requirement can be set somewhere around an i5-7500 model, in order to support a somewhat smooth experience of the simulation. The processor load will also increase based on how complex the environment is, because, as we noticed already using the profiler, the CPU plays a part in rendering too, alongside the GPU.

2.2 Main application modules

Our application needs to cover a list of diverse requirements, ranging from the actual simulation and the genetic operations performed, to data persistency and replayability of the best candidates in between runs. As such, our design contains several components, each taking care of one of the functionalities.

2.2.1 Population Component

This component addresses the simulation functionality, instantiating and initializing the characters and assigning them the data set containing the forces they need to apply during the simulation. It is also the only component which has direct access to the characters, thus being charged with notifying them when the actual simulation starts. The population component also does the following tasks:

- Creates the first random population.
- Configures the individuals to not be able to collide with each other (at the start of the simulation)
- Stores the genetic information belonging to each character, during the whole run of the application
- Feeds the genetic information to the Genetic Operations Component at the start of every run.
- Gives information about relevant features from each individual to the Evaluation Component in real time (for fitness computation purposes).

2.2.2 Evaluation Component

The purpose of this component lies solely in the fitness computation operations, being an actual part of the genetic algorithm. Some of its responsibilities are:

- Gathers the relevant features from the Population Component, in order to be able to compute the fitness for every solution candidate.
- Once the simulation is over, provides the fitness results for each candidate to the Genetic Operations Component.
- Normalizes the values obtained in the functions (in case we are using a composite fitness function) and assigns weight to them.

2.2.3 Genetic Operations Component

This component handles data processing and provides methods for the genetic algorithm's operators. As such, it is tasked with:

- Parents selection, needed for generating offspring for the next simulation.
- Mutation, aiding in the exploration of the possible solution space.
- Crossover, as means of providing an exchange of genetic information between two solution candidates.
- Elitism, in charge of preserving the most optimal solutions found.
- Sets the frequency of mutation and crossover.
- Provides the newly formed population DNA, to be used by the Population Component at the start of the next simulation.

2.2.4. Replay Component

This component handles the replay functionalities. It can focus the view on any particular candidate and show its behavior and fitness score values. It is also tasked with:

- Serializing and Deserializing DNA information and the fitness score values.
- Storing DNA information and the fitness score values persistently.
- Loading DNA information and the fitness score values that have been previously saved in current or past runs.
- Replace older save files with newer ones.

2.2.5 Menu Component

This component handles the user interaction with the application. It contains several user interface elements that can be accessed to:

- Pause or speedup the simulation (speeding up the simulation puts more load on the cpu, since it needs to do all the computation X times faster, or simply perform more operations between two consecutive rendered frames).
- Enter replay mode.
- Load save files, if available.
- Change the focus on the next or previous candidate, inside the replay mode.

2.2.6 Genetic Algorithm component

This component acts as a manager to all the other components, coordinating their steps and setting the required references needed for them to function properly. It is also in charge of resetting the global state and reloading the scene, giving start to the next simulation.

2.2.7 General Character Component

This component acts as an interface between the genetic algorithm and the actual character inside the simulation. They can provide any information about the properties of the character model. Also, they are able to provide data about the spatial properties of the game object. Such

information can relate to positioning, rotation, or scale of the said game object. This can be useful in measuring the fitness of a solution candidate (since we are in a 3D environment, most evaluations will involve spatial computation of distances, positioning or collision).

2.3 Design Advantages and Disadvantages

Our custom design was done in a manner that facilitates the development process and reflects the main functionalities we are trying to achieve. All defined components are encapsulated and built in a modular fashion.

Although it required more initial effort, splitting the genetic algorithm in separate modules has proven effective for us in the long run, since it enabled us to work in an incremental fashion. Additionally, having smaller modularised components helped debugging potential issues met along the way. Another advantage here, regarding separating the genetic algorithm steps, consists of being able to consider a parallel approach to implementing genetic operators, in the future. The side effect to this separation, however, is represented by the need of having a manager component, in order to inject dependencies and coordinate the genetic algorithm steps.

Another valuable aspect about the design came from the choice of having a single point of interaction, for the candidate, with the game objects in the environment. This allowed for a concept separation between on one hand, the algorithm used and on the other hand, the simulation which contains the actual character and game objects. Thus, for example, any information about the candidates, which the evaluation component would need, can be obtained from the general character behavior script.

Using a platform as developed as Unity allowed us to perform task, which would have, otherwise, proven difficult. Having a default architecture for our project provided us with an advanced starting point. Moreover, its ease of use helped in designing the environment in a quick manner and the free, open source, online assets domain was useful for finding a good model for our solution candidate. The toolchain also facilitated the creation of a ragdoll system for the characters, which gave us means of applying physics attributes to the model, like mass, collision and forces. The separated modules (as shown in diagram [F], with the data provided from unity docs) exposed the possibility of starting the whole simulation using a terminal or a shell script, without the graphical interface attached or involved in any manner.

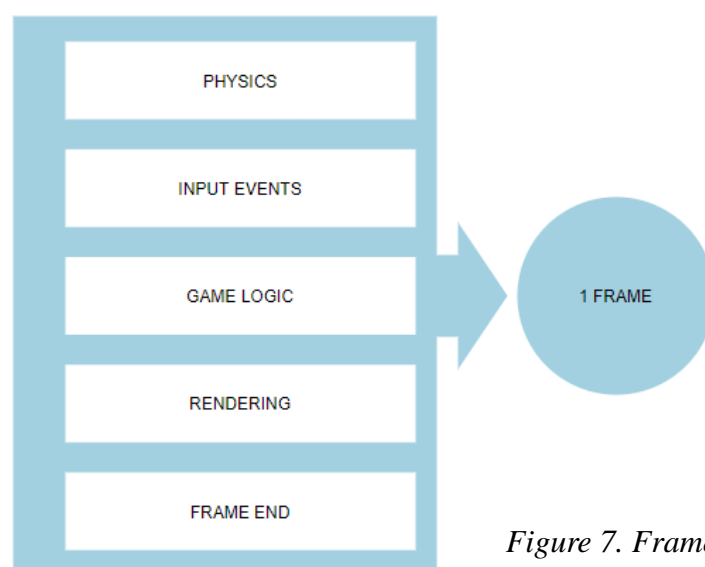


Figure 7. Frame Composition

The Game Logic component of the Unity architecture was helpful to us, although we used it for another reason than the name would imply. It is basically the part which allowed us to provide our custom functionality.

Therefore, as illustrated in diagram [H], our Game Logic segment contains scripts which deal with populating the environment with the solution candidates, linking the new chromosomes to the created individuals, performing the simulation, evaluating said individuals' new behavior and generating new offspring chromosomes, which will further be used in the next iteration of the genetic algorithm.

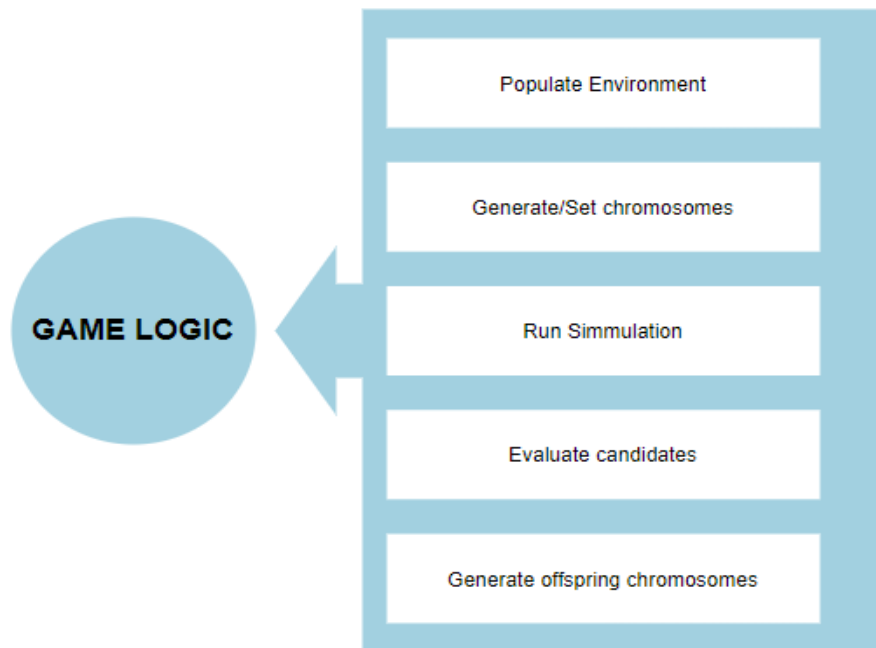


Figure 8. Custom Game Logic

Having a preset architecture from the Unity framework had its downfalls, too. While it did present a good stepping stone for our implementation, the architecture comes with certain limitations, which we needed to work around.

The first and most obvious limitation comes from the understanding that the Unity application needs to run in frames [24]. This makes it mandatory for all our scripts to follow a pattern where a procedure is provided, which is going to be executed in a cyclic manner (every frame). While this works for most tasks, which can be executed rapidly and provide intermediate results, some of our requirements revealed usecases where it is necessary for the whole output to be provided on the spot (or all instructions to be ran sequentially, in a classic manner). One example can be the creation of the chromosomes and their linkage to the instantiated individuals from the environment. The simulation cannot proceed/start until this previous step is completed, which means the execution of the steps will need to be paused for this duration. Certain side effects arise from this: having a more complex function being executed in a single frame means we will have the whole application stop – the steps in the next frame cannot be performed until the current frame finishes execution (goes through all the steps illustrated in [Figure 7]).

Even if we ignore the aspect that the application will look as if it froze, the more relevant drawback is that there will be no new frame to react to new input events. In our case, the input events are mostly related to the user interface (the menu functionality). Therefore, there will be moments during the application execution lifetime, when the menu will become unresponsive, due

to some particularly complex tasks running in the background. Also whenever the frames drop, due to any other reasons, we will encounter the same behavior. For methods that require a long execution time, usually we can employ built-in Unity features such as Coroutines, which allow us to suspend the execution of a function at a point decided by the developer, and have it continue in the next frame [25]. As a result, the frames would keep running at a stable pace and all the steps from [Figure 7] would occur more frequently. The actual function that we want to execute would be prolonged over a variable number of frames, due to this feature. It also comes with some overhead.

This fault is only noticeable when running the application in rendering mode. If we exclude the rendering component from the build and allow the application to run in batch mode (from the terminal, without graphics), the architecture still enforces the same per frame execution, although we will not mind the occasional frame drop. We would also not benefit from the inclusion of workarounds such as coroutines in that case, since we wouldn't notice or have any need for frame stability. Therefore we decided not to take any measures that would hurt the detached build's performance, since that was the most important reason why we considered the usecase to begin with.

We found a better usecase in our design for coroutines, as implementation solution for tasks which we didn't need to run every single frame. One example consists of the evaluation functions, which are particularly complex mathematical operations, that would cause performance issues if they ran each frame. While we do want, in some cases, to have said functions run in a cyclical manner, the accuracy gain if we run them every frame is trivial, compared to a case where the frequency would be, for instance, 0.25 seconds. The performance increase, however, in this case, is important. Another simple coroutine usecase for us was event listeners. It is way less taxing for the cpu, to check, for instance, if the simulation for the current iteration has ended, once every second, as opposed to checking once every frame.

Lastly, for the replay component, we had to choose between two main design paths. Either we could have saved the simulations frame by frame and end up with some form of a recording (that we would later be able to replay), or we could have just recorded the state of the application. The state is presumed to be all the data necessary in order to successfully replicate the behavior seen in the application up until a given point in time. We chose, for our design to revolve around this idea, since it enables to save files to be of reasonable size. By serializing and storing the data used from the beginning of the simulation to the end of it, we require a much smaller amount of space than what would be needed to record every frame of the simulation. After this stage, we would only need to restart the application using the given parameter, in order to achieve a replay mechanism. While this approach has the benefit of a much smaller size required, it does have some disadvantages to it too. In order to reproduce the exact same behavior we need to do the following:

1. Reload the physics scene [26].
2. Recreate the scene elements in exactly the same order [26].
3. Preserve the sequence of events [26].
4. Keep the same frequency for the fixed time update (consistent time-stepping scheme) [26].

Thus, the implementation required to achieve a replay feature using a stored state has an increased complexity level. Additionally, even if all these steps are followed, there are still several factors which can cause the physics simulation to diverge – for instance, not having the API calls on the same frames during the simulation [26]. Another limitation, as picked up from documentation, is that deterministic behavior is not assured if the same simulation runs on different platforms, due to compiler and hardware differences [26]. This last behavior has not really affected us, though, since all our simulations were running on the same platform and on the same machine.

2.4 Limits and calibration

The presented solution will produce results that differ in quality, based on the provided parameters and the complexity of the used environment. Our empirical analysis helped us, on one side, to better understand the limits of our approach and on the other hand, to optimize the application in order to achieve a good performance/quality ratio. The calibration was required between the two possibilities of launching the application. The default mode is more suitable for providing visual evaluation, presentation and user interaction, while the “detached” mode aims at achieving maximum performance and solution quality, by removing any unnecessary overhead. The factors which contribute to the differences in the solution quality can be placed in the following categories:

- Genetic operator related parameters, which modify the frequency of the genetic operations applied during a simulation. These probabilities are generally subject to modifications, or even dynamic changes during runtime [27]. This happens mostly because there is no standard value known, which can match any problem (just like there is not a standard representation available, that can be used for any problem and provide good results). As such, we tweaked these chances to better suit our search for a good candidate in the given solution space. We paid attention, in particular, to the mutation and elitism rates. A high mutation chance makes the algorithm behave like a random search in the solution space, while a low value hurts the exploration property of the genetic algorithm. Considering elitism, a percentage of the saved individuals that is too high will cause the population behavior to hastily converge during consecutive iterations, since most candidates would be preserved from one population to another. At the same time, a percentage too low will fail to preserve important candidates, that could provide high quality genes for future offspring. The number of iterations can also be considered a relevant factor here, since the stopping criteria (be it a fixed number or a point in time where no additional noticeable improvement can be found) will directly impact the final chosen solutions.
- Environment and population size parameters, which are used to maximize the amount of candidates that can be used in a simulation, while still maintaining the determinism of the physics engine and solid performance. The more objects are added to a physics scene, the more operations the CPU needs to do to compute their movement and positions. Additionally, as discussed in subchapter 2.3, the complexity of a scene can cause divergence in behavior during a physics simulation. It is important to try to avoid this potential situation, since not having a deterministic behavior in the computations for positions and collisions will hurt the quality of the final solution. The reason behind this is the evaluation component directly depending on the character’s position for computing the score of any given chromosome.
- Hardware capabilities, especially when running the application detached from the rendering component. Here we can include any optimization done with the purpose of increasing the performance of the application by reducing the CPU load, from limiting the information output every frame, to removing graphics from the simulation. These tweaks were done in order to obtain a satisfactory performance benefit from running multiple simulations in a parallel manner. Of course, we planned these optimizations with the CPU’s number of available cores/threads in mind.

2.5 Software details

We will further provide an in depth analysis to the reasoning behind software related design and choices concerning our application. Some of our diagrams will focus on elaborating the purpose of each class and how the components are generally coupled. It is also interesting to see how we were able to integrate the genetic algorithm and merge it with the simulation code.

2.5.1 Technologies

There is a large variety of available frameworks and engines which can be used to develop and run a simulation. Some of those are simple graphic modules which can be run in any web browser that supports the Canvas API. An example can be Phaser, a web game framework which can use both Canvas and WebGL renderers internally, being able to automatically swap between them based on browser support [28]. The benefits of choosing this as a framework for helping the development process consists of it being extremely lightweight and easy to use. Also, applications made using these tools don't have particularly high requirements and can run fine on most machines. However, on the negative side, we can argue that the functionality they offer is rather limited, compared to other full-fledged game engines. In this category we can place Unity and Unreal Engine, since those are the most popular choices in the game development area. Both of these tools provide great means to develop graphic simulations. More importantly, both of them use reliable physics engines, which we need in order to simplify our work of controlling characters using forces. Unreal Engine, however, is more complex than Unity and has a more abrupt learning curve. It is mostly created to aid in implementing complex environments with full photorealistic imagery. As such, we considered it was a bit too much for our requirements. This, combined with the fact that both Unity and Unreal Engine come with the same physics engine integrated (as mentioned in their documentations – Nvidia's PhysX [29, 30]), made us realise that Unity is the better choice for our application and a better fit, for what we are trying to accomplish.

After deciding the framework which we were going to use, we had to consider the programming languages available for use with Unity. There are two options here, either C# or UnityScript, which is a language with a similar syntax to JavaScript. They both show comparable levels of performance, as long as we are not using the dynamic typing feature of UnityScript, which could slow it down [31]. The UnityScript, however, has less documentation available overall and its support was also pulled out of the latest Unity versions. UnityScript has also been going through a deprecation process [32]. We have, thus, decided to focus on implementing our application using C#, since it wouldn't be reasonable to use an older Unity version (just to have a simpler language available). An important aspect of being able to use C# is also the variety of data structures and containers available (collections). This is better suited for our needs of being able to structure and process our chromosomes, which could potentially become considerably large amounts of complex data.

The data needed for the replayability of the best solutions is being stored in simple text files, after being serialized. We did not need a more complex solution and besides that, it is advised to store only the state of a scene, in order to be able to reproduce it (as opposed to saving the whole scene). The serialization helps reduce the file size and, thus, also reduces the time needed to store and retrieve the state data.

Lastly, after discovering the possibility of running the Unity application build using command line, in a detached mode (without the rendering component), we decided to use a shell script, in order to be able to run simulations in parallel and obtain more solutions for our optimization problem.

2.5.2 Classes. Modularity. Diagrams

We can place our classes in two major categories. The first category includes all the classes which inherit from the base class `MonoBehavior`, this being the main way we can integrate our scripts with Unity [33]. The reason behind this is that it allows us to interact and give input to the tasks Unity will be doing every frame. Therefore, all our custom classes that implement any form of frame by frame features will make use of the methods this base class provides. This category is represented by the following elements:

- `GeneralCharacter` class – direct link to the environment. Instances of this class will appear as characters which can be moved around in the environment. It also provides methods to get information about the character such as positioning, rotation or velocity. Lastly, it has the job of announcing that it has finished all the movements issued based on the DNA during an iteration [Annex 1].
- `PopulationController` class – instantiates and manages the characters during the simulation. It also contains the data structure which holds the DNA for every character in a population [Annex 2].
- `EvaluationController` class – purposed with measuring the efficiency and with the fitness computation of the candidates. Here we have different fitness evaluation methods, which we can use individually or combine in order to create the fitness function. This component will be active the whole time while the simulation runs [Annex 3].
- `GeneticOperationsController` class – contains methods for the selection and recombination of the chromosomes. Methods like mutation and crossover are defined here. The end result produced by this component is the new DNA which will be assigned to the next iteration [Annex 4].
- `MenuController` class – provides means for the user to interact with the application. Has methods to trigger the start of a replay run, where each candidate can be viewed separately. This is not available while running the application detached from the rendering component [Annex 5].
- `ReplayController` class – contains methods to save and load DNA. Can start a replay run with the loaded data as DNA [Annex 6].
- `CameraFollow` class – responsible with having the camera of the environment mirror the character's movements [Annex 7].
- `GeneticAlg` class – manager component. Coordonates all the other classes to obtain solutions for the optimization problem. Sets the needed references for all the objects and also starts (and restarts) the simulation [Annex 8].

The second category has a more specific use case, mainly related to operations which need to be done in a classic manner, on method call (as opposed to frame by frame). Here we can include:

- `ReplUtils` class – static class used to modularize the save, load and the file/folder related methods out of the `ReplayController` class [Annex 9].

Each of the classes we designed and developed serve a defined purpose throughout the main requirements that were set for the application. Moreover, we have designed our application's components with modularity in mind, meaning we focused on having each class perform tasks related to individual functionalities.

This does not mean, however, that a single class can only be part of one large use case in our application. On the contrary, through our design, we have sought to be able to reuse components, as long as the functionality fits the use case. For instance, the `PopulationController` class is able to issue a control group over all instantiated characters. From that point, it does not

matter where the motion input comes from: as long as we can provide, for it, forces and/or rotations as parameters, the PopulationController class will be able to apply them and move each character, frame by frame. This makes it suitable for usage in both the genetic algorithm, as well as in the replay scenarios.

We can use the previous example to elaborate on a different aspect. While the application is designed in a modularised way and this does mean most components fulfill simple and related tasks, the communication between said components is more complex. Because of that, we chose to focus our design on diagrams which can better illustrate collaboration – the way our application’s components interact with one another. It would also be helpful to have a way to show the timing constraints of our modules, since at one given time some of them might be inactive. All things considered, we found that sequence diagrams would be the most suitable for what software details we need to put accent on.

Since sequence diagrams work best by defining workflows around single use cases [34], we need to establish our major functionalities. Looking back at subchapter 1.4, we can distinguish two large scenarios to which the requirements are related.

1. The genetic algorithm use case – consists of iterations in which we see all the solution candidates attempting to reach their predefined purpose over a certain duration. This would also contain the behaviors’ evaluation and the recombination of the genes.
2. The replay use case – triggered by the user, this comprises all operations needing to be done in order to be able to review the behavior of an individual. Here we include the persistent save and the load operations, as well as the actual scene settings and preparations to be able to run character behaviors one by one.

The first use case is implemented by the interaction of the following modules: Genetic Algorithm Controller, Population Controller, Evaluation Controller and the Genetic Operations Controller. If the current run is an iteration, the Genetic Algorithm Controller checks if the DNA already exists, meaning if it had been set in a previous iteration. If not, then the Population Controller creates it. Next, the same controller instantiates the population of characters and assigns each one of them their own chromosome. Once this step is completed, the Genetic Algorithm Controller notifies the Population Controller and the Evaluation Controller when the simulation starts. This is followed by a frame by frame procedure in which each character moves in the environment, according to its own current iteration DNA, while the Evaluation Controller measures their fitness using one of the available defined methods (each behavior needs to have its own fitness function). After the simulation completes (signaled by the first frame in which none of the characters have any movement left to apply), the Evaluation Controller has a total fitness computed for each individual. Next, the Genetic Algorithm Controller sends the fitness values to the Genetic Operations Controller, which uses it to perform selection and recombination and thus, create the new DNA, which can be used in the following iteration. In case elitism is marked as usable, this controller will also save a percentage of the best candidates from the current simulation, by storing their chromosomes in the DNA to be used for the next population. Furthermore, the Genetic Algorithm Controller saves the newly computed DNA in the Population Controller data structure and lastly, uses the Replay Controller to persistently store the current generation’s serialized DNA. Then the whole process is restarted and the next iteration occurs.

This use case is better illustrated in the sequence diagram from [Figure 9]. It is not visible in the diagram, since it is not theoretically related to the scenario, but the user can speed up the simulation using a slider, up to 8x times. Beyond this point, with the available hardware, the simulation is no longer fluently rendered.

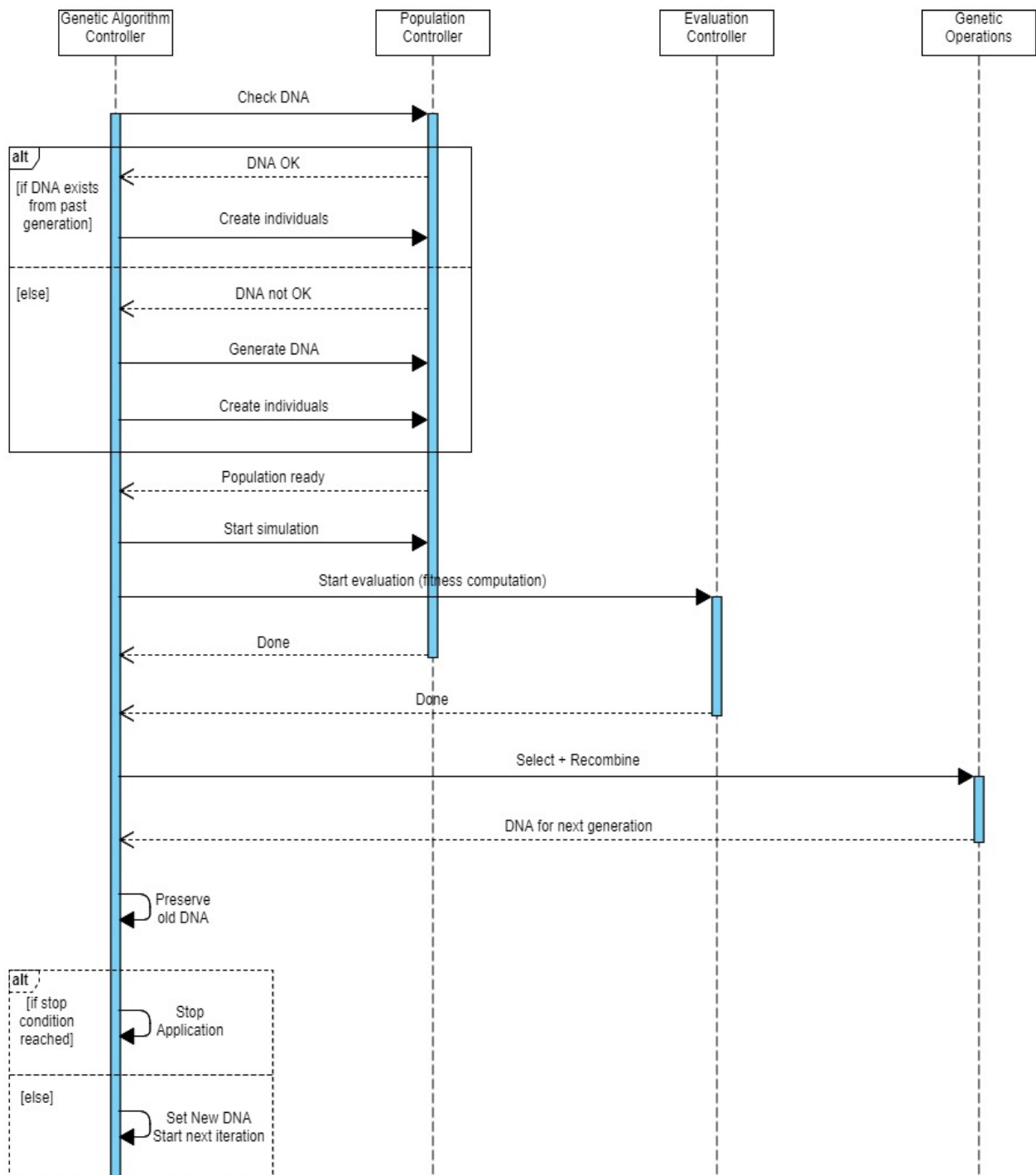


Figure 9. Genetic algorithm sequence diagram

The second use case is only reached with the help of user input. The flow of the first scenario can be interrupted at any time by interacting with the GUI. This will open a window in which the user must additionally choose a folder and a file, belonging to the simulation desired to be replayed. The selected filename is further provided by the GUI input handler to the Replay Component, which retrieves the file data and sends it as DNA to the Population Component. This one, in turn, will instantiate the correct number of individuals and assign each one of them their

own chromosome. The Replay Component will then hide the GUI menu and will start the replay run. The Population Component, lastly, orders each one of the individuals (turn by turn) to move according to the DNA which had been previously set.

This flow is better clarified through the sequence diagram at [Figure 10]. The folder and file operations are actually done using the ReplUtils static class, but we have decided to combine it with the Replay Component, for simplicity and better understanding on the general flow.

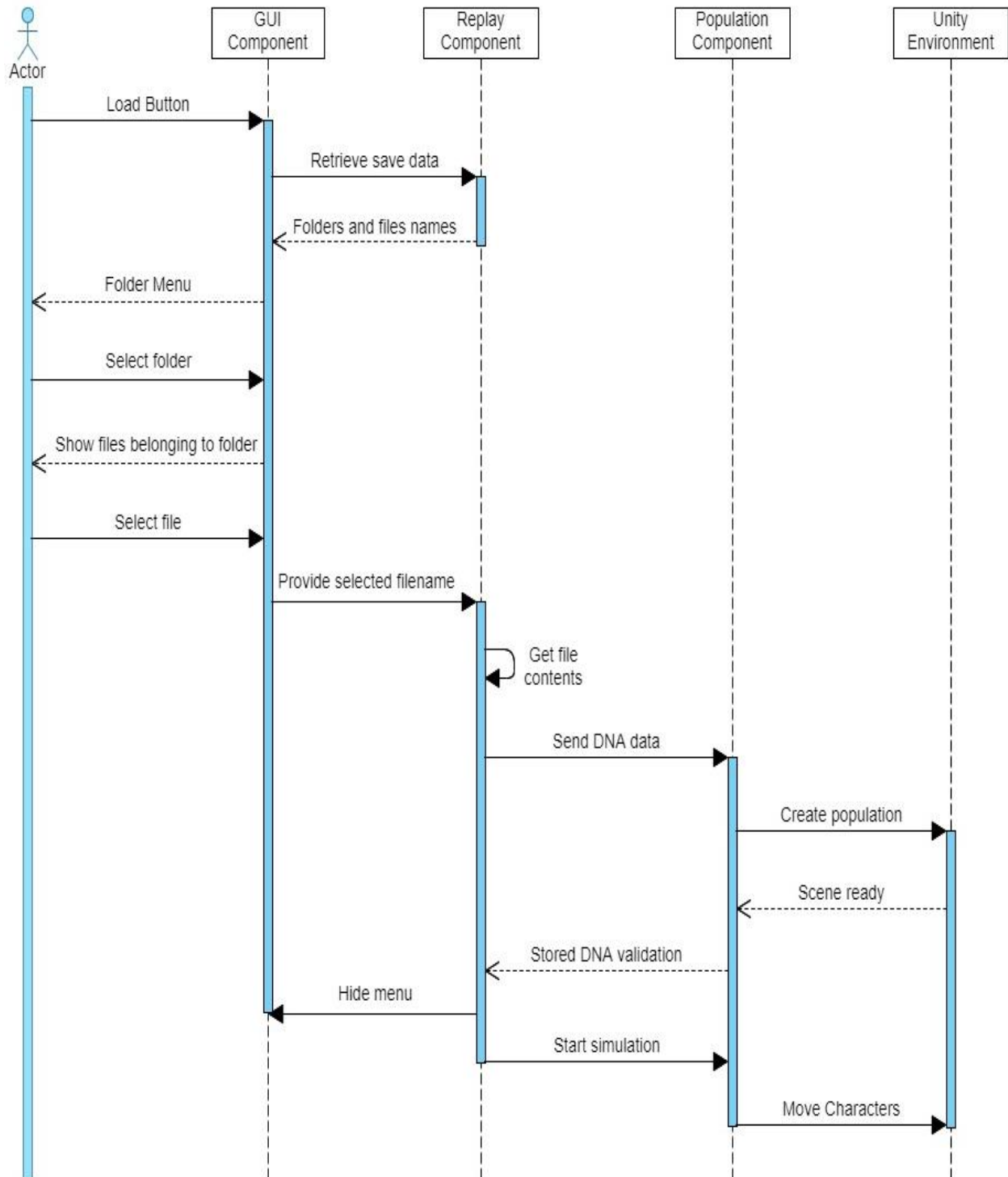


Figure 10. Replay Sequence Diagram

There are several other settings which the user can access, inside the replay mode. In order to have a better view on the whole population, each character can be observed individually. In order to achieve that, the GUI component provides buttons to control the current index of the character being replayed. Once the user decides upon the candidate, the GUI handler sends the index to the Replay Component, which tells the Population Component which DNA needs to be updated. Once the Population Component sets the correct chromosome for the candidate, the replay run is started and the character starts moving, as illustrated in Figure .

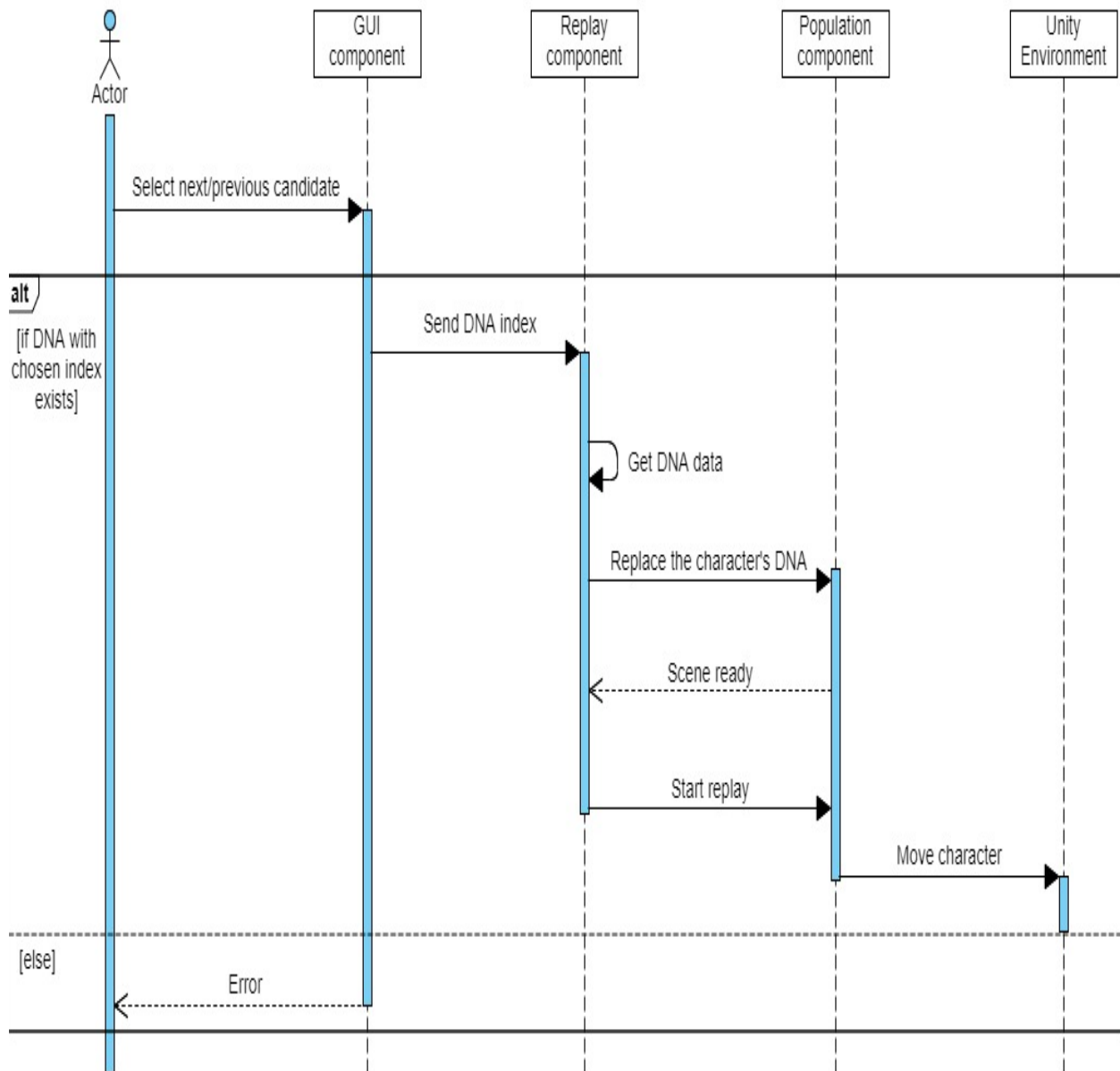


Figure 11. Replay next/previous candidate sequence diagram

The secondary settings, like increasing the speed of the simulation (up to 8x) are available to be used here as well. It is also important to note that, since the replay use case needs the GUI interaction in order to work, running the application in detached mode (without graphics) makes it impossible for the user to arrive at this scenario.

Chapter 3. Implementation

We have previously described, in subchapter 2.5, how each of our modules interacts with one another, in order to achieve our required functionalities. What is still left to describe is how we integrated the genetic algorithm with Unity and what the representation for our problem looks like. Thus, this chapter will focus on how the genetic operators were implemented and on what the chromosomes look like. Furthermore, we will present some issues we encountered and ways we tried to work around them. New ideas applied for optimizing our application will also be exposed and lastly, we will show, using a few screenshots, what the application looks like and how it works.

3.1 Implementation of the Genetic Algorithm

The most interesting aspects of any genetic algorithm revolve around the modelling of the representation, on how the genetic operators are developed and on how the evaluation is performed. These components present interest especially because of the variety of possible implementations that exists, given a certain problem. Beyond the implementation details, the frequency with which we applied the operators during an iteration can improve or hurt the end result.

3.1.1 Genetic Algorithm – Representation

Our given problem consists of finding a way to give natural movements and behaviors to characters using physics, instead of animations. Therefore, the first thing we need is a model with some kinetic system, that can be interacted with using physics. To achieve this, we chose a free open source model [35], which we imbued with 17 structures that can be affected by external factors from the environment. Each of those contain:

- A RigidBody – component which can react and adapt to forces, having properties like mass and acceleration.
- A Collider – important for detecting collisions with the environment (can be noticed by the green outlines in [Figure 13])
- A Joint – naturally connects two rigidbodies together. Without this component, rigidbodies would be unable to affect one another [36].



Figure 12. Unity Model



Figure 13. Unity Model with colliders attached

From this point onward, it is clear that our representation needs to somehow convey information about applying physics forces to characters – more specifically, to the rigidbodies contained by the characters.

Unity allows to set either a force or a torque to a rigidbody. The forces are applied in a general direction given as a parameter by the developer, while torque is added as a rotational force applied around an axis, which again, is given as a parameter. Thus, a general mapping of motion comes into mind: the whole DNA is formed of chromosomes, each character has a chromosome, each chromosome has a number of genes, and each gene must contain 17 forces (one for each rigidbody in our character). During the simulation, every frame we will use the information from one gene to apply forces to all the rigidbodies from a character. Knowing this, we focused on two particular DNA structures:

Our first representation relies on a combination of both types of applying forces. Therefore, we consider our chromosome to be a sequence of forces being applied frame by frame, on a character. These forces are of the following types:

- Horizontal, meaning the general direction is parallel to the plane.
- Vertical, meaning the general direction is perpendicular to the plane. Also, this type of force can only be applied when the character is touching an external fixed surface from the environment. Using this, we are trying to emulate a character being able to push itself upward when it has support.
- Rotational, where the given axis is computed randomly for every gene (frame). The rotation is done in the limits allowed by the connected joint. This minimizes the possibility of a character rotating his limbs in an unnatural way.

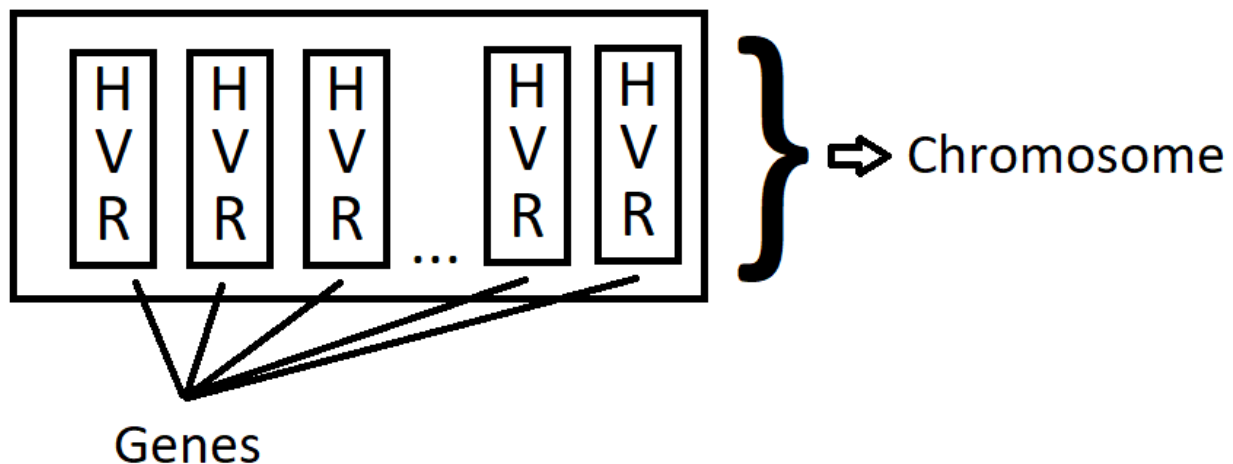


Figure 14. First representation

The data structure used to store this DNA is:

List<Tuple<List<List<Vector3>>, List<List<float>>, List<List<Vector3>>>>

While this representation allows the character to move further away, it is not entirely realistic. That is mostly because the human body naturally uses only torque to achieve motion. Due to this, our second representation is less complex and closer to reality, since we keep only the rotational force in the gene.

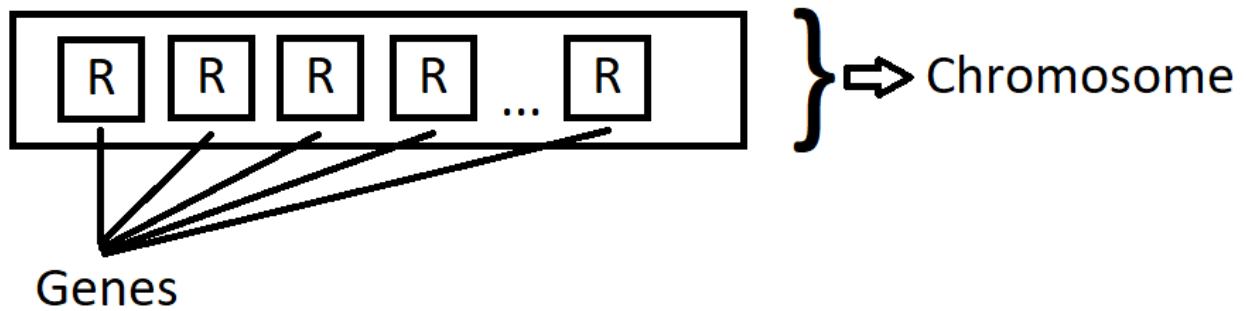


Figure 15. Second representation

The data structure used to store this DNA is:

List <List<List<Vector3>>>

This approach also has the benefit of significantly lowering the size of the DNA data, since we reduce almost two thirds of a gene's size.

3.1.2 Genetic Algorithm – Mutation

The classic example of mutation (negating the bit from the chose gene) would not work here, since our representation does not contain bitstrings in any of the genes. For this particular problem, we experimented with two types of mutation, both done on gene level.

We have already detailed that in our representation, the genes contain force data that can be applied on all 17 rigidbodies of a character, in one frame. Therefore, our mutation implementation will need to alter this gene information. This can be done by randomizing the axis and the power of the rotation. This is the common part of the mutation types that we implemented. The difference between them is that in one case, we chose to have the mutation affect only one of the rigidbodies' force data in the gene, while in the other case, we alter the data for all the rigidbodies. While this would not seem to have a big impact at first sight, the factor that changes is the operator's reach in the population. Applying the mutation only on part of the rigidbodies from a gene means more genes will be affected by the operator. The alteration of the gene will, however, be on a smaller level than in the other case.

The following example can explain this effect better. Note that while the operator has the same chance of being applied (1/6), more genes are affected:

<p>1 gene affected</p>	<p>4 genes affected</p>
Mutation that completely alters genes	Mutation that only affects genes partially

3.1.3 Genetic Algorithm – CrossOver

In this case, it is easier to apply the classic implementation for the operator. We have chosen to implement the crossover with variable cutting points. Therefore, between two chromosomes, genes are exchanged based on a randomly generated number of cutting point indexes. The value of this number is between 1 and $(n-1)$, where n is the number of genes in a chromosome.

Considering our representation, this means that the two individuals participating in the crossover process will exchange the force data from sequential frames (based on the indexes).

3.1.4 Genetic Algorithm – Elitism

Elitism has proven to be an efficient way to improve the quality of the end solutions provided by the genetic algorithm [37]. If the operator is enabled, by saving a percentage of the top candidates, we can actually observe the same behavior being replicated in consecutive populations, for some of the characters (during the simulation phase).

This operator is implemented by simply replacing the first x chromosomes from the new population with the best x chromosomes from the previous population (where x is the defined number of candidates that we wish to preserve).

3.1.5 Genetic Algorithm – Selection

The selection operator we chose to implement is the roulette wheel. This allows us to have an increased pressure of selection, based on the fitness of the candidates. It is also one of the faster methods of selection, together with the tournament selection method. This was also important, in the simulation context.

The implementation follows a procedure that creates intervals and assigns them each to a chromosome. Afterwards, a number is generated randomly and the selection is done based on its placement (in one of the intervals). The intervals are created starting from the selection probabilities, which are computed by dividing the individual fitness by the total population fitness. The mapping between an interval and a chromosome is made using the following rule:

- $C_1 \Rightarrow [0, 0+P_1)$
 - $C_2 \Rightarrow [0+P_1, 0+P_1+P_2)$
 - $C_3 \Rightarrow [0+P_1+P_2, 0+P_1+P_2+P_3)$
 - ...
 - $C_n \Rightarrow [0+P_1+P_2 + \dots + P_{n-1}, 0+P_1+P_2+P_3+\dots+P_{n-1}+P_n]$
- (where C_n is the n th chromosome and P_n is the selection probability of the n th chromosome)

The formula clearly favors chromosomes with increased selection probabilities. The higher the probability the larger the interval.

Once the random number is generated, a version of binary search is used to find the interval to which the number belongs. This implementation was chosen in order to take advantage of the fact that the elements (interval bounds) are already sorted in the "roulette wheel" vector.

3.2 Issues encountered

We have found workarounds for most of the issues we faced. The most important ones that needed fixing were the ones that hindered the performance of the genetic algorithm.

Some blocking points were discovered during the planning phase of the project. While designing the application, we realised that the structure of the genetic algorithm must be slightly

changed, in order to acomodate our requirements. Most algorithms of this kind have an evaluation phase which can be performed on the spot, being based on simply computing the value of a function or something similar. In our case, however, the evaluation phase directly depends on the simulation phase. In almost all scenarios, they even have the same time window as the simulation phase. This leads to a massive slowdown of the genetic algorithm, since now an iteration should wait for the whole simulation phase to be over in realtime, in order to be able to start the selection and recombination. We tried solving this issue by speeding up the actual simulation (for instance, by making the simulation's virtual time pass twice as fast), which in turn, increased the speed of the evaluation process as well. As a consequence, however, the cpu load increased. Thus, we had to work out a balance between the speed of the simulation and the performance of the rendering process. Some other factors contributed to this balancing as well, such as the map complexity or the number of individuals active at a time. Another fix considered for this was starting the application multiple times and having them run in parallel. Unity provided the option of running the build of the application without graphics, so the only thing left to do was to write a simple shell script in which the executable was ran multiple times, in a parallel manner.

This last optimization now caused an issue in the replay mechanism, since all the running instances were trying to save the DNA data in the same files. This caused data corruption and crashes, since following such an event, the replay component would no longer be able to deserialize and make sense of the loaded information. Once we became aware that this was a potential problem, we refactored the save mechanism to have each run of the genetic algorithm create its own folder and store the data inside it. The folders have unique names based on GUIDs. After implementing this fix, different application instances could no longer access the same files to save or load data.

Another problem we faced, discovered mid-implementation this time, was caused by not being able to fully disable collisions for game objects. Collisions are CPU intensive, especially when a multitude of objects are in very close proximity of one another. This causes stutters and sporadic frame drops during the simulation phase. More importantly though, having the candidates collide with eachother would break the evaluation's determinism. This hinders the results of the produced solution, since the genetic algorithm relies on the evaluation to always produce similar results for the same candidate. Unity provides two methods of removing collisions: using project settings or using code.

1. The Project Settings can be accessed to create layers. These are groups of items that usually contain objects which share the same properties, or for which we want to define a common behavior. Any game object can be assigned to one particular layer. This helps us stop the interactions between individuals, by removing collisions at layer level: when collisions between layer A and layer B are disabled, for instance, elements belonging to A will not interact with elements belonging to B. The only drawback to this approach is caused by the upper limit set for the number of layers in a project. We can have at most 32 layers, a limit which can be easily reached, since some of the layers are created by the default setup of the Unity project. The disabling of the collisions is done using a collision matrix:

	Default	TransparentFX	Ignore Raycast	Water	UI	Smiths	Smiths_bodies	Limbs
Default	>							
TransparentFX								
Ignore Raycast								
Water								
UI								
Smiths								
Smiths_bodies								
Limbs								

Figure 16. Collision matrix for all layers

2. The API provides the functionality of creating callbacks which are hooked to the collision events. We can use one of these methods (eg. `OnCollisionEnter`) to stop the chain of methods called for the processing of the collision. Another option for removing the collisions is explicitly calling the `IgnoreCollisions` method, which needs two parameters given (the two game objects for which we want to have the collisions removed). While this method seems simpler, it needs to be invoked for all the objects which are contained by separate individuals. The approach has its particular benefits too. It allows fine grained control over the collision of all objects and can also reenale the collisions, not only disable them (by the use of a third, boolean parameter).

While both of these options appeared to work (the individuals no longer collided with eachother), none of them actually provided a noticeable benefit to the execution time. This is due to the fact that while the computations for collisions are avoided, the events and the hooks are still being fired. Thus, we tried a different approach, which aimed at spreading the individuals further apart. We accomplished this by duplicating the environment in separate locations, either on the horizontal plane or on the vertical one. After this, we split the candidates in equally sized groups and placed each group in a separate environment.

The replay component had to be adapted to fit this approach as well. The camera, that follows the characters during the replay simulation, needs to be aware of the level ("floor") the character was on during its original run. This offset data is needed, in order to track and successfully follow an individual's movement with the camera.

Another useful benefit to this structuring of the environment is that we don't really need to implement the collisions for all the individuals from a simulation. Since they are split in groups and two characters from different groups are impossible to meet with one another, we only need to take care of the collisions from individuals belonging to the same groups.

As a result of this approach, only a part of candidates, those that belonged to the same group, shared the exact same position at the beginning of the run and more importantly, we reduced the chances of having a lot of individuals being in close proximity of eachother during the simulation.

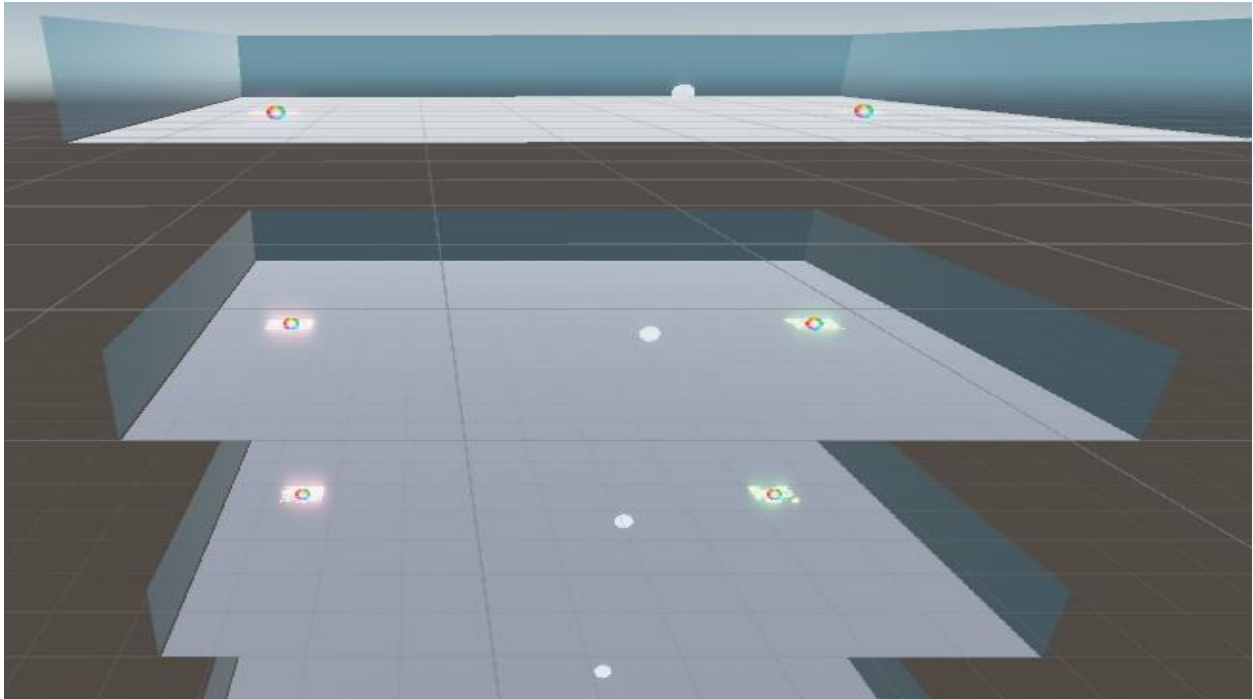


Figure 17. Duplication of the scene on the vertical plane

While this helped with the performance of the simulation, it revealed another issue that heavily influenced the results of the genetic algorithm, in a negative way.

The most problematic issue that was encountered is related to the determinism of the physics simulation. The smallest difference in the starting condition of the simulation, or in any of the physics updates, can change the behavior of the individuals. With the change in behavior, the evaluation can no longer accurately rank every chromozoma and because of that, the genetic algorithm is not able to provide relevant results. Therefore, it was in our best interest to keep the state of the simulation stable over its duration, so that there are as little inconsistencies as possible. To better understand the issue and clarify the cause, we can use as an example the duplicated environment scenario [Figure 17]. If we were to use, in a simulation, the same chromosomes on each level of the environment, we would expect that the individuals behave in the same way and end up being in the same positions (except for the Y coordinate, due to the levels being placed on top of one another). This, however, does not happen: since the candidates don't start in the same positions, they don't share the same state and their behavior will diverge. Thus, the simulation is not deterministic. And because of this, the same chromosome can have different fitness values at the end of a simulation.

Unfortunately, since there is no way to solve this issue completely, what we are able to do is simply limit the side effects, by trying to have the shared state deviate as little as possible. That is why we removed the implementation that duplicated the environment. This helped during the simulation process – the behavior was not visibly different anymore and the candidates arrived in the same end spot. The replay mechanism was also affected by this issue, because we had implemented a separate class that instantiated the characters. Realizing that this influenced the replay, we thought about reusing the Population Component, in order to keep the same initialization code. This proved to be effective and only required a short refactoring of the PopulationController class – we needed to provide a method that changed the DNA of the character dynamically, to match the use case of swapping the focus of the replay on different characters.

The same issue caused us to take other measures, in order to keep the preconditions as

identical as possible. It was important to have the iterations start under the same shared state. To achieve this, we tried first to reset the positions and rotations (all the spatial properties) of the characters. Seeing this did not help preserve determinism, we went a step further, destroying the objects and instantiating them exactly in the same way as did before. Lastly, having no other option, we chose to have the whole scene reloaded at the beginning of each iteration, preserving only the new DNA, which would be used to create the character's behavior. Albeit more costly (performance wise), this was the only implementation that worked in resetting the characters without disrupting the determinism.

3.3 Original ideas and solutions

New elements and takes on the genetic operators were necessary in order to facilitate the integration of the algorithm with the Unity engine.

Since using genetic algorithms with high physics details visual simulations does not have many practical uses or applications (so far), we did not have much inspiration to go on for the implementation. Thus we had to come up with our own design and try to achieve results while not using an extreme amount of resources (so that the simulation can run in a normal state (minimum 20 – 30 frames per second)).

The representation which correlates the chromosome gene with the data used to move a character during individual physics frames was the one that felt the most natural and easy to understand. Together with an evaluation method that computes the fitness as an average over the whole duration of the simulation, the genetic algorithm will select the data from the frames that improve the overall score of the candidates. Due to this design, we can directly see, during sequential iterations, how candidates appear to be "borrowing" behavior from one another. In the latter stages of the algorithm, we can observe how the best solutions share a similar behavior most of the cases, revealing a dominant strategy in achieving the desired result.



Figure 18. Similar behavior of the best candidates (green outlined)

The mutation operator which affects part of the gene, instead of the whole gene, is also something newly implemented. The reason behind that is that we tried to put more accent on the exploration of the solution space, without necessarily modifying the rates at which the operators are applied. It came as an experimental implementation, done after noticing that changing one physics frame (as result of applying mutation on the whole gene) does not have a noticeable

enough effect on the total behavior. It also paid off: we can see that it takes a lot more iterations to arrive at a dominant strategy, when we use the mutation that partly changes genes, but affects a larger amount of DNA.

The replay component also has its level of originality in design. Most applications store replays altogether as sequence of frames, which they can later directly render. Our system uses a more modern approach, which only stores the necessary data for reproducing the scene that we want to save persistently. This obviously has performance and memory related benefits (saves and loads take less time and use a lot less memory). It can also be considered a double edged sword. If the determinism is not assured by the engine, the replay can have deviations from the original run.

3.4 Application look

Once the application is started, the DNA is randomly generated and the simulation iterations start running. The green outlined individuals are the best performing ones from the previous iteration (those preserved by elitism):

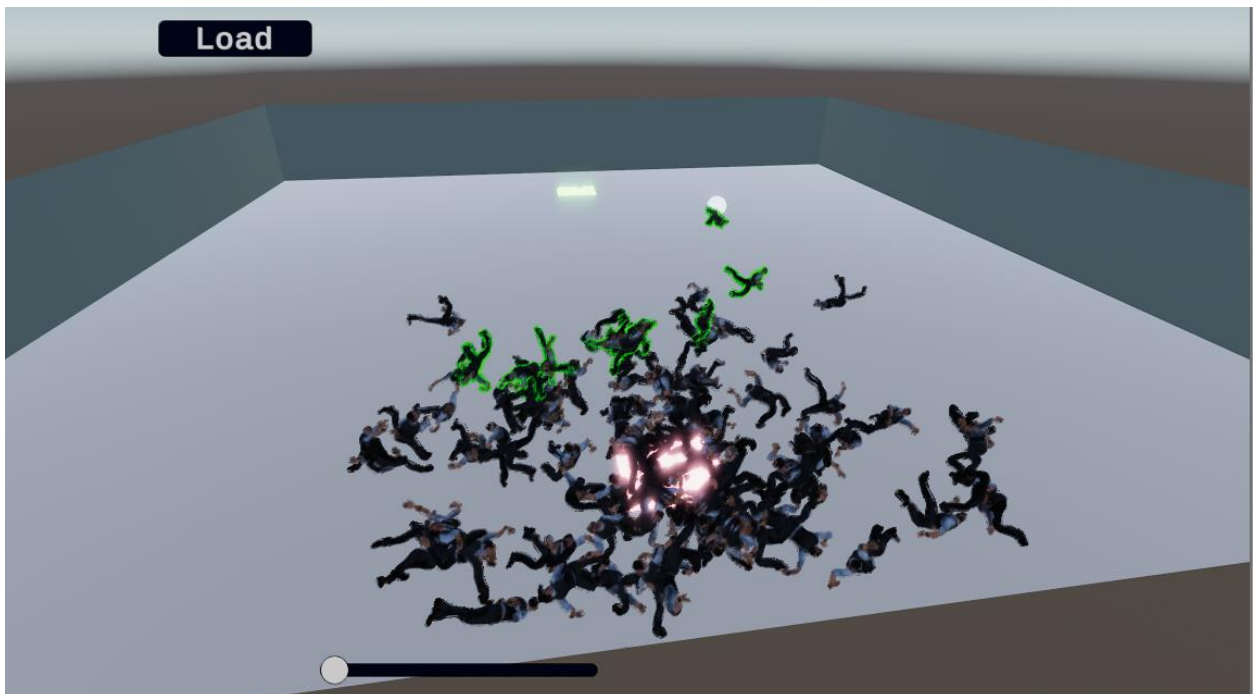


Figure 19. Best candidates from the previous run

At any given moment, the user can increase the speed of the simulations using the slider. Also, one can choose to load a previous simulation using the top button. This will reveal a menu window, where the user needs to pick one of the available folders and of course, the save file inside. The folder contains all the data related to the simulations done during a genetic algorithm run, while the files inside it store the data needed for each of the iterations.

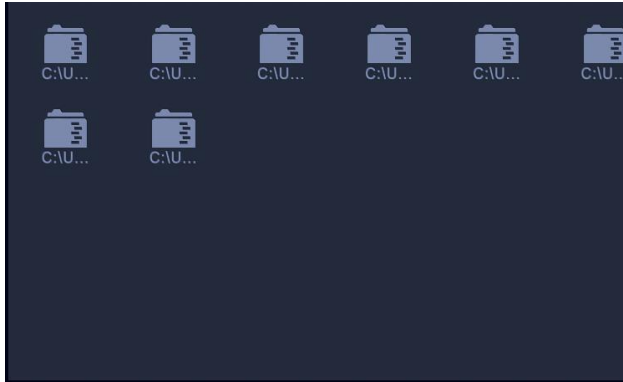


Figure 20. Folder menu

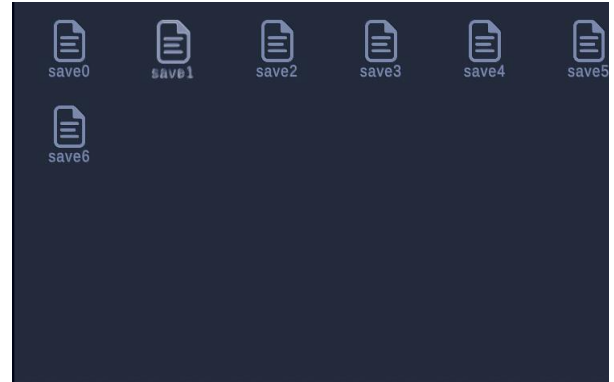


Figure 21. File menu

Once this is done, the current simulation is stopped, the system is restarted and the application enters the replay mode. In this mode, the character number is reduced to one and the camera focuses up close on the single individual existing in the scene. The user can swap the character with the next one (or the previous one) in the population, in order to visualize the behavior of every candidate.



Figure 22. Replay view

All this, however, is only available in the default run of the application. If we run in detached mode (without the rendering component), we are not able to use the replay menu or to visualize the simulations. This mode is mostly used to run more simulations in parallel, at the maximum speed available, so that good candidates are obtained faster. The command used to run the application without rendering is:

```
./GenLearn.exe -batchmode -nographics > "../Logs/log$i.txt" &
```

(**GenLearn.exe** is the build name, **batchmode** removes the restrictions of only having one Unity

instance running at a time, **nographics** removes the rendering component and the string is a file to which the logs are written).

3.5 Communication between components

The application has several modules, each of them in charge of a functionality. In order to achieve it, however, they need to provide data to one another. For instance, the evaluation component has to know the positions of the characters in order to compute their fitness, during a simulation. The component that does the genetic operations will have a new DNA set, which needs to be transferred to the population component, in order to be used during the next simulation.

Knowing the cases where C# does shallow copies helped transfer data faster using references. As long as we copy objects, the language only passes references. Thus, whenever we need to pass parameters to functions or data to objects, we have two choices:

1. Pass data in objects – useful for when we need the transfer to be done as quickly as possible or when we need the passed data to be automatically updated in the original (caller) component as well. The fitting scenario here is when we want the DNA data to be transferred to the population component, in order for it to be ready for the next simulation. The genetic operator component receives a reference to the DNA data structure and any modifications done to it will be visible to the population component (which created and owns the original DNA list).
2. Pass data in basic types or structs – useful in cases where we specifically do not want to have the original data altered. We can consider here the crossover operator use case. We want the crossover to create two child chromosomes from two parent chromosomes. The parent chromosomes must remain unaffected, otherwise future crossovers will not be able to be done with the data from the two previous parents. A viable alternative here is to implement methods that perform a deep copy of the data container (a "true" copy, so that modifications will not be reflected back to the original data).

Apart from this, we have the use case of transferring data between scenes, since we know that we are restarting the scene at the beginning of each iteration. Normally all old objects are destroyed when a new scene is loaded. However, Unity does not delete static objects. Therefore, the simplest way to preserve the state from one iteration to the next is to save it in static variables. This approach is useful when preserving the DNA across iterations. Also the flag that is used to check if we are in a replay run is also a good example.

3.6 Data storage

We needed to implement some form of persistent storage in order to accommodate the replay requirements. The data that we needed to have stored is the DNA from each iteration. This was enough, to have the replay component be able to assign the data to the population component and then start the simulation.

At the start of the application, a folder with a unique name is generated at the Persistent data path owned by the Unity engine. This folder will be used to save all the data for the current genetic algorithm runtime (we use separate folders for each run, to not run into a data corruption scenario when two instances try to access the same file in parallel). The DNA, for each iteration, is stored in files, numbered from 0 to 99. In order to know in which iteration we are currently in, the counter is transferred between scenes. So is the case for the application folder path.

The data which is being stored is of type **List <List<List<Vector3>>>**. We want to store it in a serialized form, to save up space and make the process faster. However the above data container is not marked as serializable. In order to be able to use the C# serializer method from

BinaryFormatter, we created a simple class which contains a list of floats, constructed with numbers from all the Vector3s in the DNA. Additionally we placed in the class a list of fitness scores, to be displayed for each chromosome during the replay run. The data from this class is then able to be serialized and stored in files.

Whenever the user tries to access the replay functionality, a specific file, corresponding to a single iteration, must be chosen. The names of the files are (eagerly) loaded at once. However, the actual content of the files are loaded in a lazy manner, only when the user actually chooses one for the replay run.

3.7 User Interface

The user interface can only be accessed when running the application in the default mode. Thus, when active, it provides the functionality of speeding up the simulation up to 8 times, both during genetic algorithm runs and in replay runs. The simulations during the genetic algorithm runs can be paused at any time and a user can choose to go into a replay run. In this state, each character's behavior can be viewed up close, individually.

To enter a replay state, one must navigate through a menu for selecting a folder (belonging to a particular application run) and a file (containing the DNA data and the scores for each individual from a single population). Closing this menu will resume the simulation from the current iteration of the genetic algorithm.

The folders are created at the start of the application, at the path where Unity stores persistent data. This path is different depending on the OS and it is automatically chosen by the engine. Here, Unity has rights for creating and deleting files and folders. Once the folder is created, files will be added inside it, at the end of each simulation from a genetic algorithm.

The folder names are mostly unique, being generated from different GUID instances.

It is important to note that there are moments when the interface is not responsive. This happens due to large fps drops, that occur during certain CPU intensive tasks. An example can be the loading of a new scene. We have tried to limit the time windows in which this event occurs. The removal of those CPU intensive tasks were however, in some cases, not favorable. While it is possible to provide an implementation that is much quicker for restarting the state of the genetic algorithm and reverting the positions of the candidates, we chose the complete reload of the scene in order to keep the state as consistent as possible. The physics engine determinism issue was a lot more critical than the user interface one, at least for the sake of the genetic algorithm's performance.

3.8 Software calibration

The calibration done on the software depends on the type of launch we consider. The detached mode of the application was optimized with the thought of squeezing as much performance as possible out of the CPU, to be used for the genetic algorithm iterations only. Therefore, a build for this type will have the following modifications:

- Set the optimal number of chromosomes, necessary to provide good solution candidates.
- Maximum speedup of the simulation (we are not able to visualize it, so the frame drops are not important).
- Removal of logs which were being outputted in a cyclic manner, every single frame (surprisingly this put a lot of stress on the CPU).
- Removal of graphics and running in batch mode, to be able to run the application in parallel.

As opposed to this, the default run of the application needed to balance visualization with performance. Thus, in this case we used the Unity profiler to check the factors which were causing large fps drops during the simulations and we tried to tweak those factors. While not providing solutions as quickly and as good as the detached mode, this scenario is useful for debugging and for visual evaluation or demonstration purposes. Some of these tweaks consist of:

- Providing logging information about the modified DNA during the recombination phase.
- Outputting the fitness scores for the individuals every frame or every time the new computation was done.
- Increasing or decreasing the number of individuals from populations, to lower the computations needed to be performed and therefore stabilise the fps level.
- Allowing the user to set the speed of the evaluation/simulation phase
- Lower the physics update frequency (increase the time between physics updates) – this feature improves the simulation quality but also degrades the determinism of the physics engine. Therefore, in the end, we chose not to use it, since we already discussed about how important the determinism is for the correct evaluation of the chromosomes.

Chapter 4. Testing and Experimental Results

This section presents ways to verify the functionalities our application offers. We will also include information about the determinism issue and how it affected our application. Lastly, we will detail some experiments we tried and a few of the results we obtained in the best runs.

But first, let us have a look through the configuration of the application and how we can tweak its target goal.

4.1 Application setup

Our application depends on the Unity environment in order to function normally. Therefore, in order to tweak its settings and have external additional features like pausing the simulation, running it frame by frame or displaying resource consumption level information, one needs to install the Unity editor platform [v]. Once this is accomplished, the project can be imported and started inside the editor. Cameras can be moved manually in this mode. If activated, the internal profiler can show information about the CPU and GPU load, as well as what procedures or steps take the most time to complete during frames. The tool can also be used to visually highlight important details of the environment in which the simulation occurs. For instance, we can choose to color the physics components (rigidbodies) that build up an individual.

Moreover, in the editor we can also tweak genetic algorithm settings such as what selection type to use, or whether to use elitism in the simulations or not. Lastly, one should choose the evaluation method that will be used during the genetic algorithm. This simply refers to what fitness computation function will be active and based on that, the solution candidates will attempt to get better at a particular behavior.

Alternatively, a specific build can be used. Builds can be generated from the Unity editor as an export functionality and in order to run them, one must only meet the hardware requirements (the build is simply an executable). We have used two types of builds so far, based on what we were trying to achieve. One is purposed as a demonstration application, where the user can interact with the interface and view the current generation behavior and visually evaluate the progress live (or perhaps see an older iteration behavior). The second one is the build run in detached mode, where all the components which are not necessary are stripped off and the cyclic logs are removed. This type of build is designed to be run multiple times in parallel, yielding the best chances to obtain high quality individuals for a particular behavior. No interface is available in this mode and the simulations cannot be viewed live. Therefore we lose flexibility in this scenario, since the previous configurations that were described in the editor mode run, or in the case of the default build run, are not available.

4.2 Observing the genetic operators

Most of the operators employed by the genetic operators can be directly observed during the simulation phase, especially while watching populations from consecutive iterations.

Elitism is the simplest one to notice, since when activated, we can see some of the best candidates will have the exact same behavior as in the previous iterations. We have outlined them with a green color, just to eliminate a potential confusion, appearing when one of the newly generated offspring does just as good as the preserved candidates. In those cases, it is more difficult to tell which candidate belongs to the preserved group (and which does not).

The crossover's results can, as well, be observed while watching candidates from consecutive iterations. If a solution candidate has a higher fitness value than the population medium (or when he is closer to reaching the desired behavior than the other candidates), in the

next iterations we will be seeing different candidates borrowing from its behavior. This is the primary reason why, after a number of generations, a dominant strategy for the candidates is being formed, and most of the characters will – more or less – follow a similar pattern in their movement.

The mutation operator is harder to notice, mostly because by design it changes the DNA in a small manner and the frequency of it usually is also quite low. However, if we were to separate a character and just simply apply the mutation operator on its DNA, we would notice a slight change in its behavior, in some of the frames. While the difference is usually small, applying mutation is, in most cases, enough to change a character's end position.

The selection process is not directly noticeable during simulations, but it does, indirectly, affect the population's behavior as a whole. This can be explained by the fact that dominant strategies are always formed around behaviors which have some of the highest fitness values in the populations.

4.3 Determinism of the Physics Engine

The determinism issue has already been presented. Whenever we run a simulation, we must assure the preconditions for the candidates are the same. This minimizes the chances of the behavior diverging when the same forces are being applied on a character. If we do not reload the scene before starting the simulation and choose to just reposition the characters to their initial state, we can notice a change in movements. The most obvious difference is that the end position of the characters is different then in the case of reloading the scene.

Once we implemented the approach of reloading the scene before starting the simulation, the frequency of divergences lowered. The occurrence rate increases with the complexity of the fitness functions and with the number of candidates in the population. Generally, it seems the determinism is no longer assured if the performance of the simulation drops. Even in these situations, however, the divergences were no longer noticeable by just viewing the simulations. We are able to discover them using the fitness functions, because any slight difference in movement, on a candidate, will cause a change in its evaluation score. To avoid this, we tried to tweak the complexity of the algorithm in order to keep the performance level consistent.

4.4 Experimental results

The results can be evaluated based on the fitness obtained by the best individuals from the final population of the genetic algorithm. Since the values are normalized, it is easy to observe the progress of the solution candidates: one must simply compare the fitness of a population to the one that came before it. Additionally, in order to check whether the desired behavior is achieved we can check the fitnesses of the best candidates and see how close to the optimal value (1.0) they are.

Therefore, depending on the behavior we tried to emulate we achieved different results:

The first action we tested was basic pathfinding: the action of finding a way to get from point A to point B. We did this by highlighting a start zone, where all the characters were instantiated, and an end zone, where the characters are supposed to arrive. The fitness function valued the distance between the candidate and the end zone, either after the simulation finished or every 0.5 seconds during the simulation. In the second case a mean is computed, so that we keep the fitness values between 0 and 1.

Both of the fitness functions help the candidates arrive at the target behavior. The following chart shows the evolution of the populations throughout the execution of the genetic algorithm, when the fitness value increases when the distance between a candidate and its destination point decreases – measured once, at the end of the simulation:

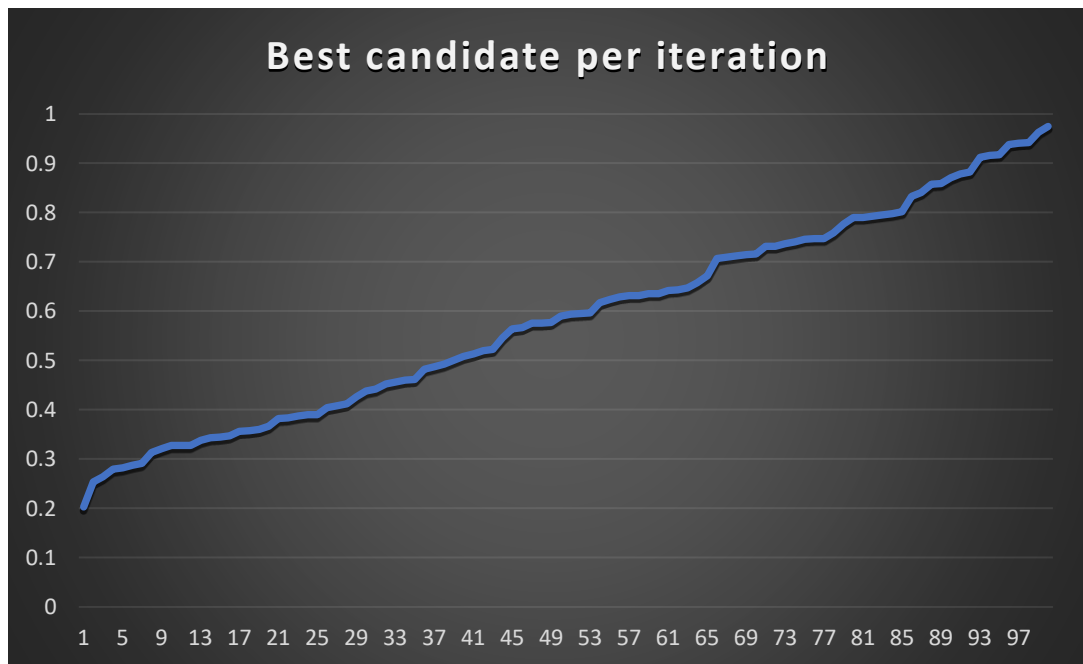


Figure 23. Pathfinding fitness values

Most of the times, the algorithm runs up to a point where the best candidates arrive at the destination area (or very close to it). The mean for the fitness values from the last populations, however, do not always get close to the optimal value. The result can be explained generally by the fact that crossover does not always yield a better chromosome than its parents, even when those are some of the best selected from the previous population. This furthermore proves the importance of elitism as a genetic operator. Without it we may lose some of the best solution candidates, especially when those are created in the earlier iterations of the algorithm.

While the candidates seem to reach their set objective, visually speaking, they are far from achieving a realist movement behavior. That is to be expected, since we did not use a fitness function that would set priority on any human like action. The characters merely crawled and jumped around towards their destination area. Therefore, we tried next to achieve a more complex behavior.

Once we knew the algorithm was suitable to finding solutions for basic actions, we went ahead and created evaluation methods that would allow the algorithm to find solutions for more advanced behaviors, such as walking. Thus, our fitness function valued body stances that would facilitate the human walking. Also, this evaluation needed to be done in a cyclic manner, since we were prioritizing the amount of time a character can stay on its feet and walk around. Our first measurement attempt relied on having the head rigidbody of the character at approximately the same height as the character's, when standing in a T position. For this, we simply compared the Y coordinate of the head rigidbody with the height of the character, every 0.25 seconds. We then ran a mean of all the values obtained while the simulation was running.

The results were not as good as in the previous test. The best solutions which are preserved at the end of the algorithm usually struggle to reach even 0.6 fitness values for their behaviors. The following diagram presents a best case scenario encountered in one of the runs:

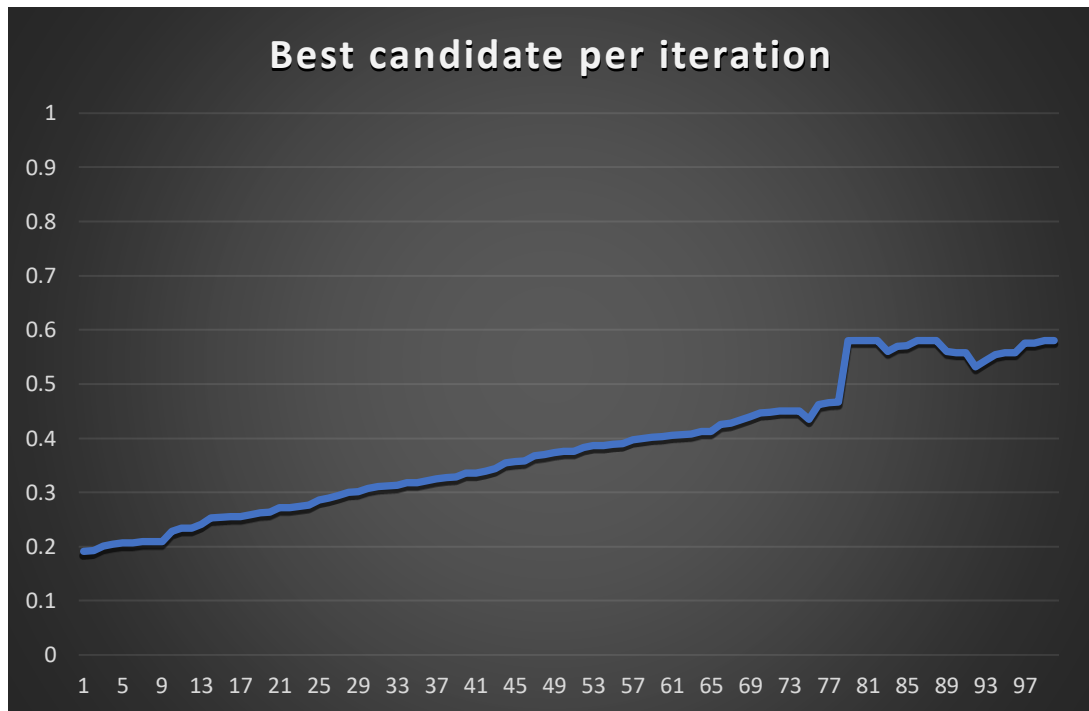


Figure 24. Walking fitness values

An interesting aspect to be mentioned here is that we can notice in the diagram certain iterations where the best solution is worse than the ones in the previous populations. This is usually not possible in a genetic algorithm implementation. However, it did happen in some of the runs, even when having the elitism operator activated. This sporadic issue can be attributed to the lack of determinism of the physics engine, since candidates with the same DNA were present in consecutive populations, but sometimes, their fitness function computed differently. This hurts the results of the genetic algorithm during those iterations and goes to show again, how important it is that the same chromosomes are always evaluated to the same fitness values.

Nevertheless, to improve the results, we tried to make modifications to the fitness function, in order to give more hints to the algorithm as to what the standard walking action should look like. Therefore, here are some of the additional tweaks we attempted:

- Prioritize the feet's Y coordinate value as close to 0 as possible, which simply translates to having the character's feet on the ground during the simulation.
- Shorten the duration between two consecutive evaluations, during the simulation. Thus, we achieved a more accurate mapping between the desired behavior and the evaluation
- Consider the head's position as factor that influences the function value only if at least one of the feet is touching the ground.
- Prioritize having the head and the feet aligned vertically.
- Have the function give 0 values in cases where the feet are positioned on a higher level than the head.

These measures were taken also due to flaws or workarounds, discovered by the genetic algorithm, that allowed the candidates to score points during the evaluation while not improving towards the desired behavior. For instance, the last tweak we described was added due to some candidates having large scores by abusing a behavior that allowed them to jump and perform rotations at the same time. Thus, the head was close to its desired height coordinate, but the

character was not trying to walk, it was simply jumping high in the air with the feet above its head.

Unfortunately the desired behavior was not achieved. The additional complexity (added to the fitness function) did improve the scores and got us closer to the action of walking, but not by a noticeable amount.

In terms of visual evaluation, the candidates appear to try to stand up, but cannot maintain their position for more than a few frames, after which they fall down and try to lift themselves up again. The process continues until the simulation is over.

Having tried all this without achieving the desired results for complex behaviors, we can say that obtaining solutions that satisfy our set requirements is more difficult than we had, initially, considered.

Conclusion

The purpose of our work was to provide an analysis on the possibility of replacing animation, as means of movement in video games or simulations, with a physics based approach for motion.

An application which successfully achieved this [22] implemented a copy mechanism based on reinforced learning, where the character moved by physics was purposed with replicating the actions done by the one controlled by animation.

Throughout our paper we proposed a potential solution that relies on the implementation of a genetic algorithm, which would try to replicate certain human behaviors, while also gaining the ability of adapting to the environment.

In the past, genetic algorithms have been used in this area to achieve other functionalities, such as finding out the optimal frequency of a certain action for a bot character [13].

The algorithm presented in this paper has been integrated with the Unity engine in an application and the evaluation step takes place during the simulation. The representation is done in a suggestive way: each gene that builds up the chromosome consists of the physics forces applied on the character in a physics frame. Thus, the chain of genes builds up the complete movement set of the character.

The desired behavior is selected by implementing a corresponding fitness function, which can be computed on the spot, at the end of a simulation, or in a cyclic manner, while the simulation runs. Either way, the fitness function needs to be designed in a manner that would encourage the selection and recombination of those chromosomes that generate a behavior resembling (or getting closer to) the desired one.

We have met several issues that were solvable during our implementation process, such as finding a way to speed up the evaluation phase and running the simulations in parallel. This increased the initial data set, thus creating a better chance of obtaining higher quality solution candidates in the end. One problem, in particular, was discovered in the later stages of development and it concerned the nondeterministic nature of the physics engine [k]. This hurt the performance of the genetic algorithm by making the evaluation process sporadically score the same chromosome differently in consecutive iterations. While we minimized the impact of this problem, we were not able to fully fix it.

For testing the algorithm, we started by choosing a simple behavior which implied getting from a start area to a destination area over the course of the simulation (by moving in any way possible). This proved to be easy to accomplish and the best solution candidates from the last populations can consistently arrive at a behavior that accomplishes this.

Next, we tried to see if the application was able of achieving more complex behaviors. The selected action was the human walk. Unfortunately this proved to be harder to accomplish and no matter the fitness function we created, the solutions were not able to reach a score close to the maximum. Therefore, the characters were not able to achieve a behavior that mirrored the human walking ability. There are several factors which proved to be impediments, but the most relevant one was the determinism issue. Appearing more frequently as the complexity of the evaluation phase increased, the lack of the physics engine determinism caused the computation of false scores for the candidates, which often led to weaker individuals being selected and recombined.

Compared to the reinforced learning example, which proved to be successful, our project uses a simpler character model, built of 17 components which can be moved by forces. We believe that this, together with the fact that the reinforced learning project had a more advanced starting point from which the behavior was derived (animation driven movement, as opposed to our case, which only uses a fitness function that encourages a given behavior) are the other main factors that caused the difference in results.

Our project can be extended, in the future, in two main areas. First of all, the user interface could be upgraded so that it provides a way to change the fitness function from within the application. This would make it easier to try out the genetic algorithm with different behaviors, since we would not need to create an application build for every new fitness function anymore. This would also benefit us since the Unity Editor would no longer be needed to tweak the project settings. The second aspect, which can be further worked on, is related to the possibility of adopting an island based model for our genetic algorithm. Since we can run multiple instances of the project in parallel, it would be interesting to see how exchanging a small portion of chromosomes between the application instances, every few iterations, could influence the final quality of the solutions.

In the end, the proposed solution was not able to reach a point where animation driven movement can be totally replaced, especially in the cases of complex behaviors. However, we still believe that, given a deterministic environment and a character model that more closely resembles the human body, the genetic algorithm could provide reliable solutions, even for complex movements.

Bibliography

- [1]. J. Madigan, Ed., "Analysis: The Psychology of Immersion in Video Games", in *Gamasutra*, Aug. 2010. [Online]. Available: https://www.gamasutra.com/view/news/120720/Analysis_The_Psychology_of_Immersion_in_Video_Games.php
- [2]. C. Bratt, Ed., "God of War director explains why entire game has no camera cuts", in *Eurogamer*, June 2017. [Online]. Available: <https://www.eurogamer.net/articles/2017-06-21-god-of-war-director-explains-why-the-entire-game-has-no-camera-cuts>
- [3]. C. R. Reeves, "Genetic Algorithms," in *Handbook of Metaheuristics*, M Gendreau and J-Y Potvin, Eds., Boston, Massachusetts: Springer, 2010, pp. 109-139
- [4]. A.W. Johnsona and S.H. Jacobsonb, "On the convergence of generalized hill climbing algorithms," in *Discrete Applied Mathematics 119*, Amsterdam, Netherlands: Elsevier, 2002, pp. 38-40;
- [5]. E. Bouzid. "Benefits of using genetic algorithm" stats.stackexchange.com
<https://stats.stackexchange.com/questions/23106/benefits-of-using-genetic-algorithm> (accessed July 9, 2020)
- [6]. S. E. Bouzid. "What are the advantages of genetic algorithms?" quora.com
<https://www.quora.com/What-are-the-advantages-of-genetic-algorithms> (accessed July 10, 2020)
- [7]. V. Mallawaarachchi. "Introduction to Genetic Algorithms — Including Example Code." towardsdatascience.com
<https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3> (accessed July 9, 2020)
- [8]. C. Shyalika. "An Insight to Genetic Algorithms — Part III." medium.com
<https://medium.com/datadriveninvestor/genetic-algorithms-selection-5634cfc45d78> (accessed July 8, 2020)
- [9]. J. Koljonen and J. T. Alander, "Effects of population size and relative elitism on optimization speed and reliability of genetic algorithms," paper, Department of Electrical Engineering and Automation, University of Vaasa, Vasa, Finland, 2006

- [10]. I. Watson et al., Eds., "Optimization in Strategy Games: Using Genetic Algorithms to Optimize City Development in FreeCiv", 2009. [Online] Available: <https://www.cs.auckland.ac.nz/research/gameai/projects/GA%20in%20FreeCiv.pdf>
- [11]. N. Cole et al., "Using a genetic algorithm to tune first-person shooter bots," presented at the Proceedings of the 2004 Congress on Evolutionary Computation, Portland, OR, USA, June 19, 2004.
- [12]. K Sims, "Evolving Virtual Creatures," presented at the SIGGRAPH94: 21st International ACM Conference on Computer Graphics and Interactive Techniques, New York, NY, USA, July, 1994.
- [13]. Pluralsight. "How Animation for Games is Different from Animation for Movies." pluralsight.com
<https://www.pluralsight.com/blog/film-games/how-animation-for-games-is-different-from-animation-for-movies>
- [14]. D. Kade, O. Ozcan and R Lindell, Eds., "An Immersive Motion Capture Environment," in *World Academy of Science, Engineering and Technology* 73, Jan. 2013. [Online]. Available:
https://www.researchgate.net/publication/268741732_An_Immersive_Motion_Capture_Environment
- [15]. PlayStation. *God of War - How to Fight Like Kratos / PS4*. (Mar. 28, 2018). Accessed: Jul. 25, 2020. [Online Video]. Available: <https://www.youtube.com/watch?v=rKuc73DSscs>
- [16]. E. F. Anderson, Ed., "Real-Time Character Animation for Computer Games," in *The National Centre for Computer Animation*, Jan. 2001. [Online]. Available:
www.researchgate.net/publication/250362465_Real-Time_Character_Animation_for_Computer_Games
- [17]. M. L. Gleicher, "More Motion Capture in Games — Can We Make Example-Based Approaches Scale?," presented at the Motion in Games, First International Workshop, Utrecht, The Netherlands, Jun. 14-17, 2008.
- [18]. B. Tversky and M. Bétrancourt, Eds., "Effect of computer animation on users' performance: A review," in *Le travail humain*, Dec. 2000. [Online]. Available:
https://www.researchgate.net/publication/232606237_Effect_of_computer_animation_on_users'_performance_A_review

- [19]. B. Kenwright, R. Davison, and G. Morgan, "Dynamic Balancing and Walking for Real-Time 3D Characters," presented at the Motion in Games - 4th International Conference, Edinburgh, UK, Nov 13-15, 2011.
- [20]. Ubisoft La Forge. *[SIGGRAPH Asia 2019] DReCon: Data-Driven Responsive Control of Physics-Based Characters*. (Mar. 28, 2018). Accessed: Jul. 25, 2020. [Online Video]. Available: <https://www.youtube.com/watch?v=tsdzwmEGL2o>
- [21]. A. Hassanat et al., "Choosing Mutation and Crossover Ratios for Genetic Algorithms-A Review with a New Dynamic Approach," *Information*, vol. 10, no. 12, pp. 390, Dec. 2019, doi: [org/10.3390/info10120390](https://doi.org/10.3390/info10120390).
- [22]. M. D. Safe, J. Carballido, I. Ponzoni and N. B. Brignole, "On Stopping Criteria for Genetic Algorithms," presented at Advances in Artificial Intelligence, 17th Brazilian Symposium on Artificial Intelligence, São Luis, Maranhão, Brazil, Sep 29 - Oct 1, 2004.
- [23]. Unity Manual. "Optimizing graphics performance." docs.unity3d.com.
<https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html#CPU-optimization> (accessed 21 Jul. 2020)
- [24]. Unity Manual. "Time and Framerate Management." docs.unity3d.com.
<https://docs.unity3d.com/Manual/TimeFrameManagement.html> (accessed 21 Jul. 2020)
- [25]. Unity Manual. "Coroutines." docs.unity3d.com.
<https://docs.unity3d.com/Manual/Coroutines.html> (accessed 22 Jul. 2020)
- [26]. Nvidia Docs. "Best Practices Guide." docs.nvidia.com.
docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/BestPractices.html (accessed 22 Jul. 2020)
- [27]. W. Lin, W. Lee and T. Hong, Eds., "Adapting Crossover and Mutation Rates in Genetic Algorithms." In *Journal of Information Science and Engineering*, Sep 2003. [Online]. Available: https://www.researchgate.net/publication/220587952_Adapting_Crossover_and_Mutation_Rates_in_Genetic_Algorithms

- [28]. Phaser Docs. "Getting Started with Phaser 3" phaser.io.
<https://phaser.io/tutorials/getting-started-phaser3> (accessed 12 Jul. 2020)
- [29]. Unreal Engine Docs. "Physics Simulation | Unreal Engine Documentation" docs.unrealengine.com.
<https://docs.unrealengine.com/en-US/Engine/Physics/index.html> (accessed 14 Jul. 2020)
- [30]. Unity Manual. "Physics" docs.unity3d.com.
<https://docs.unity3d.com/Manual/PhysicsSection.html> (accessed 14 Jul. 2020)
- [31]. L. Meijer and Burnumd. "Is there a performance difference between Unity's Javascript and C#? - Unity Answers" answers.unity.com.
<https://answers.unity.com/questions/7567/is-there-a-performance-difference-between-unitys-j.html> (accessed 20 Jul. 2020)
- [32]. R. Fine. "UnityScript's long ride off into the sunset" blogs.unity3d.com.
<http://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset> (accessed 21 Jul. 2020)
- [33]. Unity Docs. "Unity - Scripting API: MonoBehaviour" docs.unity3d.com.
<https://docs.unity3d.com/ScriptReference/MonoBehaviour.html> (accessed 29 Jul. 2020)
- [34]. M. Fowler, "UML Sequence Diagrams," in *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd Edition, Boston, Massachusetts: Addison-Wesley, 2003.
- [35]. Batewar. "Bodyguards | Characters | Unity Asset Store" assetstore.unity.com
<https://assetstore.unity.com/packages/3d/characters/humanoids/humans/bodyguards-31711> (accessed 5 Mar. 2020)
- [36]. Unity Manual. "Joints" docs.unity3d.com.
<https://docs.unity3d.com/Manual/Joints.html> (accessed 05 Aug. 2020)

- [37]. B. Chakraborty, Ed., "On The Use of Genetic Algorithm with Elitism in Robust and Nonparametric Multivariate Analysis", in *Probal.*, Jan. 2003. [Online] Available: https://www.researchgate.net/publication/244332347_On_The_Use_of_Genetic_Algorithm_with_Elitism_in_Robust_and_Nonparametric_Multivariate_Analysis

Annex

Annex 1

```
using System.Collections;
using System.Collections.Generic;
using System.Xml.XPath;
using TMPro;
using UnityEngine;

[System.Serializable]
public class GeneralCharacter : MonoBehaviour
{
    //17 movable rigidbodies

    //private List<List<Vector3>> HorizontalMovementDNA = new List<List<Vector3>>();
    private List<List<Vector3>> torqueDNA = new List<List<Vector3>>();
    //private List<List<float>> VerticalMovementDNA = new List<List<float>>();

    private List<Rigidbody> allrigidbodies = new List<Rigidbody>();
    public int stage = 0;
    public float Power = 5f;
    private int ChromosomeLength = 500;

    private bool grounded = true;
    // horizontal,vertical,torque
    // vector3 * DNAlength
    [SerializeField]
    public List<string> TerrainList = new List<string>();

    public bool started = false, finished = false;

    private Rigidbody headrb;
    Transform[] allchildren = { };
    private List<Collider> allcolliders = new List<Collider>();
    float distToGround;
    List<Rigidbody> limbs = new List<Rigidbody>();

    public void SetDNA(List<List<Vector3>> torque)
    {
        torqueDNA.AddRange(torque);
    }
    public void SetChromosomeLength(int length)
    {
        ChromosomeLength = length;
    }
    public Transform GetSpine()
    {
        var children = GetChildren();
        var index = -1;
        /*for (int i = 0 ; i < children.Length;i++)
        {
            if (children[i].name == "Spine")
            {
                index = i;
                break;
            }
        }
        */
        //Debug.LogError(index);
    }
}
```

```

        return children[22];
    }
    void Start()
    {
        var rbs = GetRigidbody();
        /*GetChildren();
        foreach(var x in allchildren)
        {
            var renderer = x.GetComponent<Renderer>();
            if(renderer)
            {
                renderer.enabled = false;
            }
        }*/
        /* foreach (Rigidbody obj in rbs)
        {
            obj.maxAngularVelocity = 100f;
        }*/
    }
    private void Awake()
    {
        bool foundhead = false;
        var children = GetChildren();
        var x = new List<int>();
        for (int i = 0; i < children.Length; i++)
        {
            if (!foundhead && allchildren[i].CompareTag("CHead"))
            {
                foundhead = true;
                headrb = allchildren[i].GetComponent<Rigidbody>();
                // Debug.Log(name + allchildren[i].name);
            }
            if (allchildren[i].CompareTag("limb"))
            {
                limbs.Add(allchildren[i].GetComponent<Rigidbody>());
                /*if (limbs[limbs.Count - 1].gameObject.name == "LeftFoot" ||
                limbs[limbs.Count - 1].gameObject.name == "RightToeBase" || limbs[limbs.Count -
                1].gameObject.name == "LeftToeBase" || limbs[limbs.Count - 1].gameObject.name == "Right-
                Foot")
                {
                    x.Add(limbs.Count - 1);
                    Debug.Log(limbs[limbs.Count - 1].gameObject.name + " " +
                    (limbs.Count-1));
                }*/
            }
        }
        /*string y = "";
        foreach (var p in x)
        {
            y = y + p + ", ";
        }
        Debug.Log(y);
        Debug.LogError(1);*/
        //Debug.Log("Limbs layer count: " + limbs.Count);
    }
}

```

```

// Update is called once per frame
void Update()
{

}
void FixedUpdate()
{
    if (started && !finished)
    {
        for (int i = 0; i < allrigidbodies.Count; i++)
        {
            //Debug.Log(name + " " + allrigidbodies[i].name + " " + "Velocity = " +
allrigidbodies[i].velocity);
            //Debug.Log(name + " " + allrigidbodies[i].name + " " + "Ang Velocity = "
+ allrigidbodies[i].angularVelocity);

            //HorizontalMovement(allrigidbodies[i], HorizontalMovementDNA[stage][i],
torqueDNA[stage][i]);
            //VerticalMovement(allrigidbodies[i], VerticalMovementDNA[stage][i]);
            Move(allrigidbodies[i], torqueDNA[stage][i]);
        }
        stage++;
        if (stage == ChromosomeLength)
            finished = true;
    }
    //RandomMovement();
    //Debug.Log(GetHeadYcoordinate());
}

/*void RandomMovement()
{
    int count = 0;
    for (int i = 1; i < allchildren.Length; i++)
    {

        var rb = allchildren[i].GetComponent<Rigidbody>();

        if (rb)
        {
            //Debug.Log(allchildren[i].name);
            count++;

            HorizontalMovement(rb);
            VerticalMovement(rb);
        }

        //child is your child transform
    }
}*/

public float GetHeadY()
{
    // Debug.Log(name +headrb);
    return headrb.transform.position.y;
}
public float GetFootY()
{

```



```

        return limbs[1].transform.position.y;
    }

    public List<Rigidbody> GetRigidBodies()
    {
        if (allrigidbodies.Count == 0)
        {
            for (int i = 1; i < GetChildren().Length; i++)
            {
                var rb = allchildren[i].GetComponent<Rigidbody>();
                if (rb)
                {
                    allrigidbodies.Add(rb);
                }
            }
        }
        return allrigidbodies;
    }

    public Transform[] GetChildren()
    {
        if (allchildren.Length == 0)
        {
            allchildren = transform.GetComponentsInChildren<Transform>(true);
        }
        return allchildren;
    }

    /*bool IsGrounded()
    {
        bool result = false;
        for (var i = 0; i < limbs.Count; i++)
        {
            if (limbs[i].grounded)
            {
                result = true;
                break;
            }
        }
        return result;
    }*/

    /*void HorizontalMovement(Rigidbody rb, Vector3 horizontalforces, Vector3 torque)
    {
        rb.AddTorque(torque*Time.fixedDeltaTime,ForceMode.Impulse);
        if (IsGrounded())
        {
            rb.AddForce(Vector3.Scale(new Vector3(1 * Time.fixedDeltaTime, 1 *
Time.fixedDeltaTime, 1 * Time.fixedDeltaTime), horizontalforces), ForceMode.Impulse);
        }
    }*/

    void Move(Rigidbody rb, Vector3 torque)
    {
        //rb.AddTorque(torque, ForceMode.Impulse);
        rb.AddTorque(torque * Power, ForceMode.Impulse);
    }

    /*void VerticalMovement(Rigidbody rb, float verticalforce)
    {
        if(IsGrounded())
        {

```

```
        rb.AddForce(new Vector3(0, 1*Time.fixedDeltaTime, 0) * verticalforce, Force-
Mode.Impulse);
    }
}*/
float GetHeadYcoordinate()
{
    return headrb.position.y;
}

public List<Collider> GetAllColliders()
{
    List<Collider> result = new List<Collider>();
    if(allcolliders.Count == 0)
    {
        for (int j = 0; j < GetChildren().Length; j++)
        {
            if (allchildren[j].GetComponent<Collider>() != null)
            {
                result.Add(allchildren[j].GetComponent<Collider>());
            }
        }
        allcolliders = result;
    }
    else
    {
        result = allcolliders;
    }
    return result;
}
public void Act()
{
    finished = false;
    started = true;
}

}
```

Annex 2

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using Random = UnityEngine.Random;

public class PopulationControlller : MonoBehaviour
{
    public GameObject CharacterType = null;
    public int PopulationSize;
    public int MaxRunTime = 20;
    public int AppliedStimulusCount = 500;
    private List<GeneralCharacter> Population = new List<GeneralCharacter>();
    public Transform InitialLocation;
```

```

public List<Vector3> Positions;
public List<Quaternion> Rotations;
private GameObject origin = null;
private int EliteCount;
private int SimCounter;
public bool ready = false;

/*
 * List of tuples, each tuple belongs to an individual and contains:
 * 1) All the horizontal motion for the current simulation - as a List of Lists of
Vector3s;
 * 2) All the vertical motion for the current simulation - as a List of Lists of
floats;
 * 3) All the torque motion for the current simulation - as a List of Lists of Vec-
tor3s;
 * Size - x individuals(tuples) * y frames * z rigidbodies * 1 force
 */
//private static List<Tuple<List<List<Vector3>>, List<List<float>>, List<List<Vec-
tor3>>>> CurrentPopulationDNA = new List<Tuple<List<List<Vector3>>, List<List<float>>,
List<List<Vector3>>>>();

/*
 * List 1 - Contains individuals - 50,100,etc
 * List 2 - Contains each frame - 500, etc
 * List 3 - Contains each Rigidbody torque applied - 17 rbs per individual.
 */
private static List< List<List<Vector3>>> CurrentPopulationDNA = new List<
List<List<Vector3>>>();

// -- GeneticAlg
public bool done = false;
// --

public void SetEliteCount(int elites)
{
    EliteCount = elites;
}
public void SetSimCounter(int simcounter)
{
    SimCounter = simcounter;
}
private void InitPopulation()
{
    //Debug.LogError(ReplUtils.GetFolderNames()[0]) ;
    if (CurrentPopulationDNA.Count < PopulationSize)
    {
        for (int i = 0; i < PopulationSize; i++)
        {
            List<List<Vector3>> torque = new List<List<Vector3>>();
            //Debug.Log("Entered");
            for (int j = 0; j < AppliedStimulusCount; j++)
            {
                List<Vector3> allrbTorque = new List<Vector3>();
                for (int k = 0; k < 17; k++)
                {
                    allrbTorque.Add(new Vector3(
                        UnityEngine.Random.Range(-1, 1f),
                        UnityEngine.Random.Range(-1f, 1f),
                        UnityEngine.Random.Range(-1f, 1f)));
                }
            }
        }
    }
}

```

```

        torque.Add(allrbTorque);
    }
    CurrentPopulationDNA.Add(torque);
}
SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
else
{
    for (int i = 0; i < PopulationSize; i++)
    {
        origin = Instantiate(CharacterType, InitialLocation.position, Quaternion.identity);
        origin.SetActive(false);
        GameObject obj = CreateAndPrepare(CurrentPopulationDNA[i]);
        obj.name = "Smith" + i;
        obj.transform.SetParent(transform);

        Population.Add(obj.GetComponent<GeneralCharacter>());
    }
    Debug.Log("InitPopulation Done");
    ready = true;
}
}
private string PrintDNA()
{
    string result = "";
    int i = 0;
    foreach (var list in CurrentPopulationDNA)
    {
        result += ("\n" + "Smith" + i + "\n");
        string h = "";
        for (int j = 0; j < AppliedStimulusCount; j++)
        {
            h += "[" + String.Join(",", list[j]) + "]";
        }
        result += (h + "\n");
        i++;
    }
    return result;
}
private GameObject CreateAndPrepare( List<List<Vector3>> t)
{
    GameObject obj = Instantiate(origin, InitialLocation.position, Quaternion.identity);
    GeneralCharacter individual = obj.GetComponent<GeneralCharacter>();
    individual.SetDNA(t);
    individual.SetChromosomeLength(AppliedStimulusCount);
    individual.stage = 0;
    return obj;
}
// Start is called before the first frame update
private void Awake()
{
    //Random.InitState(42);
    InitPopulation();
    IgnoreRagdollCollisions();
}
void Start()
{
    if (SimCounter > 0)

```

```

for(var i = 0; i < EliteCount; i++)
{
    var outline1 = Population[i].gameObject.AddComponent<Outline>();

    outline1.OutlineMode = Outline.Mode.OutlineAll;
    outline1.OutlineColor = Color.green;
    outline1.OutlineWidth = 3f;
}

// Update is called once per frame
void Update()
{
}

public void ResetState()
{
    //Debug.Break();
    /* Destroy(origin);
    origin = null;
    origin = Instantiate(CharacterType, InitialLocation.position, Quaternion.identity);
    origin.SetActive(false);
    for (int i = 0; i < Population.Count; i++)
    {
        Destroy(Population[i].gameObject);
        Population[i] = null;
        Population[i] = CreateAndPrepare(CurrentPopulationDNA[i].Item1, CurrentPopulationDNA[i].Item2, CurrentPopulationDNA[i].Item3).GetComponent<GeneralCharacter>();
        Population[i].name = "Smith" + i;
    }
    IgnoreRagdollCollisions();
    /* foreach (var candidate in Population)
    {
        var i = 0;
        var x = candidate.transform.GetComponentsInChildren<Transform>(true);
        foreach (Transform child in x)
        {
            var rb = child.GetComponent<Rigidbody>();
            if (rb)
            {
                rb.velocity = Vector3.zero;
                rb.angularVelocity = Vector3.zero;
            }
            child.position = Positions[i];
            child.rotation = Rotations[i];
            i++;
        }
    }

    /* var rbs = candidate.GetRigidbodyBodies();
    int rbcount = rbs.Count;
    rbs[0].transform.position = InitialLocation.position;
    for (int i = 0; i < rbcount; i++)
    {
        rbs[i].velocity = Vector3.zero;
        rbs[i].angularVelocity = Vector3.zero;
        rbs[i].position = Positions[i];
        rbs[i].rotation = Rotations[i];
    }
    */
}

```

```

        }*/
        /* candidate.stage = 0;
    }
    foreach (var candidate in Population)
    {
        var x = candidate.transform.GetComponentInChildren<Transform>(true);
        foreach (Transform child in x)
        {
            var rb = child.GetComponent<Rigidbody>();
            if (rb)
            {
                // rb.isKinematic = false;
                //rb.useGravity = true;
            }
        }
    }
}*/
}

private void IgnoreCollisionsBetween(List<Collider> allcollidersX, List<Collider>
allcollidersY)
{
    int listsize = allcollidersX.Count;
    for (int i = 0; i < listsize; i++)
    {
        for (int j = 0; j < listsize; j++)
        {
            Physics.IgnoreCollision(allcollidersX[i], allcollidersY[j]);
        }
    }
}

private void IgnoreRagdollCollisions()
{
    /*for (int i = 0; i < PopulationSize - 1; i++)
    {
        List<Collider> collidersforI = Population[i].GetAllColliders();
        for (int j = i + 1; j < PopulationSize; j++)
        {
            IgnoreCollisionsBetween(collidersforI, Population[j].GetAllColliders());
        }
    }
    */
    if(Population.Count>0)
    for (int i = 0; i < PopulationSize; i++)
    {
        List<Collider> collidersforI = Population[i].GetAllColliders();
        for (int j = 0; j < collidersforI.Count - 1; j++)
        {
            for (int k = j + 1; k < collidersforI.Count; k++)
            {
                Physics.IgnoreCollision(collidersforI[j], collidersforI[k], false);
            }
        }
    }
}

public float GetHeadPositionY()
{
    return Population[0].GetHeadY();
}

public float GetFootPositionY()
{
    return Population[0].GetFootY();
}

```

```

    }
    public List<GeneralCharacter> GetPopulation()
    {
        return Population;
    }

    public void StartMovement(int index = -1)
    {
        ready = true;
        if (index == -1)
        {
            for (int i = 0; i < PopulationSize; i++)
            {
                Population[i].gameObject.SetActive(true);
                Population[i].Act();
            }
        }
        else
        {
            Population[index].gameObject.SetActive(true);
            Population[index].Act();
        }
    }
    public List<List<List<Vector3>>> GetPopulationDNA()
    {
        return CurrentPopulationDNA;
    }
    public void SetDNA(List< List<List<Vector3>>> DnaFromNextGen)
    {
        CurrentPopulationDNA = DnaFromNextGen;
    }
}

```

Annex 3

```

using System.Collections;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Linq;
using UnityEngine;

public class EvaluationController : MonoBehaviour
{
    public List<GeneralCharacter> PopulationReference = new List<GeneralCharacter>();
    private List<float> IndividualFitnessList = new List<float>();
    private float DistanceFromStartToFinish;
    private Transform StartPos;
    private Transform Destination;
    private int bestsmithindex;
    private float headposY;
    private float footposY;
    // -- GeneticAlg
    public bool done = false;
    public List<float> GetFitnessList()
    {
        return IndividualFitnessList;
    }
    // --

```

```

void Start()
{

}

// Update is called once per frame
void Update()
{

}

public void SetPopulation(List<GeneralCharacter> population)
{
    PopulationReference = population;
    float[] fitnesslist = new float[population.Count];
    IndividualFitnessList = fitnesslist.ToList();
}

public void SetHeadPositionY(float posY)
{
    headposY = posY;
}

public void SetFootPositionY(float posY)
{
    footposY = posY;
}

public void SetTransforms(Transform start, Transform dest)
{
    StartPos = start;
    Destination = dest;
}

public void StartEvaluation()
{
    StartCoroutine("ComputeFitness");
}
IEnumerator ComputeFitness()
{
    int bestsmithindex = 0;

    //yield return StartCoroutine(LastFrameDistanceFromDestinationFitness());
    yield return StartCoroutine(MediumDistanceToDestinationFitness());
    //yield return StartCoroutine(EncourageWalking());

    Debug.Log("Smith no " + bestsmithindex + " is the best");
    NotifyFinish();
    //PopulationReference[bestsmithindex].gameObject.GetComponent<Renderer>().material.color = Color.red;
}
IEnumerator LastFrameDistanceFromDestinationFitness()
{
    for (int i = 0; i < 24; i++)
    {
        bestsmithindex = 0;
        for (int j = 0; j < IndividualFitnessList.Count; j++)
        {
            //Debug.DrawLine(PopulationReference[j].GetRigidbody()[0].position,
            Destination.position + new Vector3(0, j / 10 * 30, 0), Color.blue, 0.5f);

```



```

        //var DistanceToFinish = Vector3.Distance(PopulationReference[j].GetRigidBodies()[0].position, Destination.position + new Vector3(0,j/10*30,0));
        var DistanceToFinish = Vector3.Distance(PopulationReference[j].GetRigidBodies()[0].position, Destination.position);
        var NormalizedDistanceToFinish = GetNormalizedValue(DistanceToFinish, 0f, DistanceFromStartToFinish);

        IndividualFitnessList[j] = (1 - NormalizedDistanceToFinish);
        if (IndividualFitnessList[j] > IndividualFitnessList[bestsmithindex])
        {
            bestsmithindex = j;
        }
        //Log("Measured" + (i+1) + " - smith" + j + " value " + Vector3.Distance(PopulationReference[j].GetRigidBodies()[0].position, Destination.position));
    }
    yield return new WaitForSeconds(0.5f);
}

IEnumerator EncourageWalking()
{
    for (int i = 0; i < 24; i++)
    {
        bestsmithindex = 0;
        for (int j = 0; j < IndividualFitnessList.Count; j++)
        {
            var DistanceToFinish = Vector3.Distance(PopulationReference[j].GetRigidBodies()[0].position, Destination.position);
            var NormalizedDistanceToFinish = GetNormalizedValue(DistanceToFinish, 0f, DistanceFromStartToFinish);

            var DistanceBetweenYs1= Mathf.Abs(PopulationReference[j].GetHeadY() - headposY);
            var NormalizedHeadDistanceFromOptimal1 = GetNormalizedValue(DistanceBetweenYs1, 0f, headposY);

            var DistanceBetweenYs2 = Mathf.Abs(PopulationReference[j].GetFootY() - headposY);
            var Normalized_YDistanceBetweenHeadAndFoot = GetNormalizedValue(DistanceBetweenYs2, 0f, headposY);
            if(PopulationReference[j].GetFootY() > PopulationReference[j].GetHeadY())
            {
                Normalized_YDistanceBetweenHeadAndFoot = 0;
                NormalizedHeadDistanceFromOptimal1 = 1;
            }
            IndividualFitnessList[j] += (1f - NormalizedDistanceToFinish) * 0f + Normalized_YDistanceBetweenHeadAndFoot * 0f + (1 - NormalizedHeadDistanceFromOptimal1) * 1f ;
            // IndividualFitnessList[j] += (1f - NormalizedDistanceToFinish) * 0f + (1f - NormalizedHeadDistanceFromOptimal1) * 0.5f + (1f - NormalizedHeadDistanceFromOptimal2) * 0.5f;

            if (IndividualFitnessList[j] > IndividualFitnessList[bestsmithindex])
            {
                bestsmithindex = j;
            }
            //Log("Measured" + (i+1) + " - smith" + j + " value " + Vector3.Distance(PopulationReference[j].GetRigidBodies()[0].position, Destination.position));
        }
        yield return new WaitForSeconds(0.5f);
    }
}

```

```

    }
    for (int j = 0; j < IndividualFitnessList.Count; j++)
    {
        IndividualFitnessList[j] /= 24f;
    }
}

IEnumerator MediumDistanceToDestinationFitness()
{
    var DistanceFromStartToFinish = Vector3.Distance(StartPos.position, Destination.position);
    for (int i = 0; i < 24; i++)
    {
        bestsmithindex = 0;
        for (int j = 0; j < IndividualFitnessList.Count; j++)
        {
            var DistanceToFinish = Vector3.Distance(PopulationReference[j].GetRigidbody()[0].position, Destination.position);
            var DistanceFromStart = Vector3.Distance(PopulationReference[j].GetRigidbody()[0].position, StartPos.position);

            var NormalizedDistanceToFinish = GetNormalizedValue(DistanceToFinish, 0f, DistanceFromStartToFinish);
            var NormalizedDistanceFromStart = GetNormalizedValue(DistanceFromStart, 0f, DistanceFromStartToFinish);
            IndividualFitnessList[j] += ((1 - NormalizedDistanceToFinish) * 1f); //+ (NormalizedDistanceFromStart * 0f);
            if (IndividualFitnessList[j] > IndividualFitnessList[bestsmithindex])
            {
                bestsmithindex = j;
            }
            //Log("Measured" + (i+1) + " - smith" + j + " value " + Vector3.Distance(PopulationReference[j].GetRigidbody()[0].position, Destination.position));
        }

        yield return new WaitForSeconds(0.5f);
    }
    for (int j = 0; j < IndividualFitnessList.Count; j++)
    {
        IndividualFitnessList[j] /= 24f;
    }
}

public void SetDistanceFromStartToEnd(float dist)
{
    DistanceFromStartToFinish = dist;
}

private void NotifyFinish()
{
    done = true;
}

public void ResetState()
{
    done = false;
}

private float GetNormalizedValue(float value, float minValue, float maxValue)
{

```

```

        //if (minValue == maxValue)
        //    return 0.5f;
        if (value > maxValue)
            return 1f;
        return (value - minValue) / (maxValue - minValue);
    }
}

```

Annex 4

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using Random = UnityEngine.Random;

public enum SelectionMethod
{
    RouletteWheel,
    Rank,
    Tournament
}

public class GeneticOperationsController : MonoBehaviour
{
    // Start is called before the first frame update
    public bool done;
    public SelectionMethod SelectMethod = SelectionMethod.RouletteWheel;
    public bool Elitism = true;
    public int EliteCount = 2;
    public float MutationRate = 0.02f;

    private int PhysicsSteps = 500;
    private int RigidBodyPerIndividual;
    private List<List<List<Vector3>>> PopulationDNA;

    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
    }

    public List<int> Select(List<float> FitnessList)
    {
        List<int> indexes = new List<int>();
        switch(SelectMethod)
        {
            case SelectionMethod.RouletteWheel:
                float fitness_sum = FitnessList.Sum();
                int popsize = FitnessList.Count;
                List<float> selection_probabilities = new List<float>(popsize);
                foreach (var fitness in FitnessList)

```

```

        {
            selection_probabilities.Add(fitness/fitness_sum);
        }
        List<float> selection_probabilities_cumulated = new List<float>(new
float[popsize+1]);
        //roulette
        selection_probabilities_cumulated[0] = 0f;
        for (var j = 0; j < popsize; j++)
        {
            selection_probabilities_cumulated[j+1] = selection_probabilities_cumulated[j] + selection_probabilities[j];
        }
        for (var i = 0; i < popsize; i++)
        {
            var rand = Random.Range(0.0000001f, 1);
            indexes.Add(BinarySearch(selection_probabilities_cumulated, rand));
        }
        break;
    case SelectionMethod.Rank:
        break;
    case SelectionMethod.Tournament:
        break;
    }
    Debug.Log(String.Join(", ", indexes));
    return indexes;
}
public void SetPopulationDNA(List< List<List<Vector3>>> dna)
{
    PopulationDNA = dna;
}
public void ComputeNextGeneration(List<float> FitnessList, out List<List<List<Vector3>>> NewDNA )
{
    //Debug.Log("Fitnesslist has " + FitnessList.Count + " values.");
    //Debug.Log("Fitness: " + String.Join(", ", FitnessList));

    PhysicsSteps = PopulationDNA[0].Count;
    RigidBodyPerIndividual = PopulationDNA[0][0].Count;
    List<List<List<Vector3>>> NewDNA = new List<List<List<Vector3>>>();
    var halfpopsize = FitnessList.Count / 2;

    List<int> survivors = Select(FitnessList);
    /*foreach (var x in PopulationDNA)
    {
        NewDNA.Add(x);
    }*/

    for (int i= 0; i< halfpopsize ;i++)
    {
        var parent1index = survivors[Random.Range(0, survivors.Count)];
        var parent2index = survivors[Random.Range(0, survivors.Count)];
        NewDNA.AddRange(CrossOver(parent1index, parent2index));
        // Debug.Log("Crossed over: " + parent1index + ", " + parent2index);
    }
    for(int j = 0; j< NewDNA.Count;j++)
    {
        //Mutation_RigidBody(j, NewDNA);
        Mutation_WholePhysicsStep(j, ref NewDNA);
    }
}

```

```

// Elitism - preserving the best x individuals from the current generation.
if (Elitism == true)
{
    int[] bestindexes = GetBestX(FitnessList);
    for (int i = 0; i < bestindexes.Length; i++)
    {
        NewDNA[i] = (DeepClone(PopulationDNA[bestindexes[i]]));
    }
    Debug.Log("Elitism: " + String.Join(", ", bestindexes));
    float[] vals = new float[bestindexes.Length];
    for (int i=0;i<vals.Length;i++)
    {
        vals[i] = FitnessList[bestindexes[i]];
    }
    Debug.Log("Elitism values: " + String.Join(", ", vals));
}
//PopulationDNA = NewDNA;
NewDna = NewDNA;
// printV(PopulationDNA[0].Item1, PopulationDNA[1].Item1);
}
private int[] GetBestX(List<float> FitnessList)
{
    int[] result = new int[EliteCount];
    float[] maxvalues = new float[EliteCount];
    var maxlen = maxvalues.Length;
    for (int i =0; i< FitnessList.Count; i++)
    {
        for(int j = 0; j < maxlen; j++)
        {
            if(FitnessList[i] > maxvalues[j])
            {
                for(int k = maxlen - 2; k >= j; k--)
                {
                    maxvalues[k+1] = maxvalues[k];
                    result[k + 1] = result[k];
                }
                maxvalues[j] = FitnessList[i];
                result[j] = i;
                break;
            }
        }
    }
    return result;
}

private void printV(List<List<Vector3>> a, List<List<Vector3>> b)
{
    for(var i =0; i < a.Count; i++)
    {
        string txta = "A: ";
        string txtb = "B: ";
        for(int j= 0; j< a[i].Count;j++)
        {
            txta += a[i][j].ToString();
            txta += ", ";
            txtb += b[i][j].ToString();
            txtb += ", ";
        }
        Debug.Log(txta);
        Debug.Log(txtb);
    }
}

```

```

    }
}
private List<List<Vector3>> DeepClone(List<List<Vector3>> x)
{
    List<List<Vector3>> result = new List<List<Vector3>>();
    foreach (var i in x)
    {
        var c = new List<Vector3>();
        foreach (var j in i)
        {
            c.Add(new Vector3(j.x,j.y,j.z));
        }
        result.Add(c);
    }
    return result;
}
private List<List<float>> DeepClone(List<List<float>> x)
{
    List<List<float>> result = new List<List<float>>();
    foreach (var i in x)
    {
        var c = new List<float>();
        foreach (var j in i)
        {
            c.Add(j);
        }
        result.Add(c);
    }
    return result;
}
public List<List<List<Vector3>>> CrossOver(int charindex1, int charindex2)
{
    List<List<Vector3>> torque_result_X = new List<List<Vector3>>();
    List<List<Vector3>> torque_result_Y = new List<List<Vector3>>();
    List<List<Vector3>> torque_X = DeepClone(PopulationDNA[charindex1]);
    List<List<Vector3>> torque_Y = DeepClone(PopulationDNA[charindex2]);

    var cutcount = Random.Range(1, PhysicsSteps);
    SortedSet<int> cutindexes = new SortedSet<int>();

    for (int k=0; k< cutcount;k++)
    {
        cutindexes.Add(Random.Range(1, PhysicsSteps));
    }

    int startI = 0;
    int endI = 0;
    bool swap = false;
    foreach(var idx in cutindexes)
    {
        endI = idx;
        if (swap == false)
        {
            torque_result_X.AddRange(torque_X.GetRange(startI, endI - startI));
            torque_result_Y.AddRange(torque_Y.GetRange(startI, endI - startI));
        }
        else
        {
            torque_result_X.AddRange(torque_Y.GetRange(startI, endI - startI));
            torque_result_Y.AddRange(torque_X.GetRange(startI, endI - startI));
        }
    }
}

```

```

        }
        startI = endI;
        swap = !swap;
    }
    endI = PhysicsSteps;
    if (swap == false)
    {
        torque_result_X.AddRange(torque_X.GetRange(startI, endI - startI));
        torque_result_Y.AddRange(torque_Y.GetRange(startI, endI - startI));
    }
    else
    {
        torque_result_X.AddRange(torque_Y.GetRange(startI, endI - startI));
        torque_result_Y.AddRange(torque_X.GetRange(startI, endI - startI));
    }
    // last index from list -----> last physicstep (last frame).

    //printV(rep11, horizontal_X);

    List<List<Vector3>> replacement_X = torque_result_X;

    List<List<Vector3>> replacement_Y = torque_result_Y;

    List<List<List<Vector3>>> result = new List<List<List<Vector3>>>();
    result.Add(replacement_X);
    result.Add(replacement_Y);
    return result;
}

public void Mutation_RigidBody(int charindex, ref List< List<List<Vector3>>> NewDNA)
{
    var chance = 0f;
    for (int affected_step = 0; affected_step < PhysicsSteps; affected_step++)
    {
        var torque_list = NewDNA[charindex][affected_step];
        var len = torque_list.Count;
        for (int i = 0; i < len; i++)
        {
            chance = Random.Range(0f, 1f);
            if (chance < MutationRate)
            {
                // Debug.Log("Smith" + charindex + " mutated gene " + affected_step);
                torque_list[i] = (new Vector3(Random.Range(-1f, 1f), Random.Range(-
1f, 1f), Random.Range(-1f, 1f)));
                /*var a = torque_list.SequenceEqual(NewDNA[charindex][af-
fected_step]);

                if (a==false)
                {
                    Debug.LogError("not the same");
                }
            }
        }
    }
}

public void Mutation_WholePhysicsStep(int charindex, ref List<List<List<Vector3>>>
NewDNA)
{

```

```

var chance = 0f;
for (int affected_step = 0; affected_step < PhysicsSteps; affected_step++)
{
    chance = Random.Range(0f, 1f);
    if (chance < MutationRate)
    {
        var torque_list = NewDNA[charindex][affected_step];
        var len = torque_list.Count;
        for (int i = 0; i < len; i++)
        {
            torque_list[i] = (new Vector3(Random.Range(-1f, 1f), Random.Range(-
1f, 1f), Random.Range(-1f, 1f)));
        }
        //Debug.Log("Smith" + charindex + " mutated gene " + affected_step);
    }
}

private int BinarySearch(List<float> a, float item)
{
    int result_index = 0;
    int start_index = 0;
    int end_index = a.Count - 1;
    while (start_index <= end_index)
    {
        result_index = start_index + (end_index - start_index) / 2;
        if (item > a[result_index])
            start_index = result_index + 1;
        else
            end_index = result_index - 1;
        if (a[result_index] == item)
        {
            return result_index - 1;
            //return the index before the found number.
        }
    }
    return a[result_index] < item ? result_index : result_index - 1;
    //return the index closest to the number, on the left side.
}
}

```

Annex 5

```

using System;
using System.Collections.Generic;
using System.IO;
using TMPro;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.WSA;

public class MenuController : MonoBehaviour
{
    // Start is called before the first frame update
    private static MenuController instance = null;
    public Button LoadButton, ResumeButton, LeftArrow, RightArrow, BackToFolderView;
    public Slider SpeedSlider;
    public GameObject SavePanel;
    public GameObject ItemsWindow;
    public ReplayController ReplayComponent;
}

```



```

public static bool IsReplayRun = false;
public static float speed = 1;
public List<GameObject> FileItems;
public List<GameObject> FolderItems;
public GameObject FolderIcon;
public GameObject FileIcon;
private string CurrentFolderPath;
private void Awake()
{

}

}
void Start()
{
    Init();
    ResetMenus();
    if (IsReplayRun)
    {
        TriggerReplayMenu();
    }
    else
    {
        TriggerGeneralMenu();
        if(!ReplUtils.IsCreated_Folder())
        {
            CreateCurrentFolder();
        }
    }
}
private void CreateCurrentFolder()
{
    string uniqueness = Guid.NewGuid().ToString();
    string path = Path.Combine(UnityEngine.Application.persistentDataPath, uniqueness);
    Directory.CreateDirectory(path);
    ReplUtils.AssignCurrentFolder(path);
}
// Update is called once per frame
void Update()
{
}
private void GoBackToSim()
{
    SavePanel.SetActive(false);
    ResumeButton.gameObject.SetActive(false);
    LoadButton.gameObject.SetActive(true);
    Time.timeScale = speed;
}

private void HideFolderIcons()
{
    for (int i = 0; i < FolderItems.Count; i++)
    {
        FolderItems[i].SetActive(false);
    }
}
private void RemoveFileIcons()
{
}

```

```

        for (int i = 0; i < FileItems.Count; i++)
        {
            Destroy(FileItems[i]);
        }
        FileItems.Clear();
    }
    private void PopulateFileList(string folderpath)
    {
        List<string> filenames = ReplUtils.GetSaveFileNames(folderpath);
        if (folderpath != CurrentFolderPath)
        {
            RemoveFileIcons();
            for (int i = 0; i < filenames.Count; i++)
            {
                GameObject file = Instantiate(FileIcon, ItemsWindow.transform);
                var textcomponent = file.GetComponentInChildren<TMP_Text>();
                textcomponent.text += filenames[i];
                file.GetComponentInChildren<Button>().onClick.AddListener(delegate {
                    IsReplayRun = true;
                    Time.timeScale = speed;
                    ReplayComponent.LoadReplay(Path.Combine(folderpath, textcompo-
nent.text));
                });
                FileItems.Add(file);
            }
            CurrentFolderPath = folderpath;
        }
        else
        {
            ShowFileIcons();
        }
    }
    private void ShowFileIcons()
    {
        foreach (var fileicon in FileItems)
        {
            fileicon.SetActive(true);
        }
    }
    private void HideFileIcons()
    {
        foreach (var fileicon in FileItems)
        {
            fileicon.SetActive(false);
        }
    }

    private void PopulateFolderList(string[] paths)
    {
        for (int i = 0; i < paths.Length; i++)
        {
            GameObject folder = Instantiate(FolderIcon, ItemsWindow.transform);
            string formattedtext = Path.GetDirectoryName(paths[i]).Substring(0, 4) +
"...";
            folder.GetComponentInChildren<TMP_Text>().text = formattedtext;
            var idx = i;
            folder.GetComponentInChildren<Button>().onClick.AddListener(delegate
            {
                HideFolderIcons();
                PopulateFileList(paths[idx]);
            });
        }
    }

```

```

        BackToFolderView.interactable = true;
    });
    FolderItems.Add(folder);
}
}
private void ShowFolderIcons()
{
    foreach (GameObject foldericon in FolderItems)
    {
        foldericon.SetActive(true);
    }
}
private void OpenLoadTab()
{
    Time.timeScale = 0f;
    if (FolderItems.Count == 0)
    {
        string[] folderpaths = ReplUtils.GetFolderNames();
        PopulateFolderList(folderpaths);
    }
    else
    {
        ShowFolderIcons();
    }
    SavePanel.SetActive(true);
    ResumeButton.gameObject.SetActive(true);
    LoadButton.gameObject.SetActive(false);
}
private void Init()
{
    BackToFolderView.interactable = false;
    BackToFolderView.onClick.AddListener(delegate
    {
        HideFileIcons();
        ShowFolderIcons();
        BackToFolderView.interactable = false;
    });
    LoadButton.onClick.AddListener(OpenLoadTab);
    ResumeButton.onClick.AddListener(GoBackToSim);

    if (ReplayComponent.GetIndex() == ReplayComponent.PopulationComponent.PopulationSize - 1)
    {
        RightArrow.interactable = false;
    }
    if (ReplayComponent.GetIndex() == 0)
    {
        LeftArrow.interactable = false;
    }
    LeftArrow.onClick.AddListener(delegate
    {
        ReplayComponent.ChangeCandidate(-1);
    });
    RightArrow.onClick.AddListener(delegate
    {
        ReplayComponent.ChangeCandidate(1);
    });
}

```

```
        SpeedSlider.value = speed;
        SpeedSlider.onValueChanged.AddListener(delegate {
            speed = SpeedSlider.value;
            Time.timeScale = speed; });
    }
    private void ResetMenus()
    {
        SavePanel.SetActive(false);
        ResumeButton.gameObject.SetActive(false);
        LoadButton.gameObject.SetActive(false);

        LeftArrow.gameObject.SetActive(false);
        RightArrow.gameObject.SetActive(false);
        //SpeedSlider.gameObject.SetActive(false);
    }
    private void TriggerReplayMenu()
    {
        LeftArrow.gameObject.SetActive(true);
        RightArrow.gameObject.SetActive(true);
        LoadButton.gameObject.SetActive(true);

        //SpeedSlider.gameObject.SetActive(true);
    }
    private void TriggerGeneralMenu()
    {
        LoadButton.gameObject.SetActive(true);
    }
}
```

Annex 6

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class ReplayController : MonoBehaviour
{
    public CameraFollow SpecialCam;
    private static List<float> ScoreList;
    private static int charindex = 0;
    public PopulationControlller PopulationComponent;

    // Start is called before the first frame update
    void Awake()
    {
        if(MenuController.IsReplayRun == false)
        {
            gameObject.SetActive(false);
        }
    }
    private void Start()
    {
        if(MenuController.IsReplayRun == true)
        {
            SpecialCam.enabled = true;
        }
    }
}
```

```

        SpecialCam.target = PopulationComponent.GetPopulation()[charindex].GetSpine();
        PopulationComponent.StartMovement(charindex);
    }

}

public int GetIndex()
{
    return charindex;
}

// Update is called once per frame
void Update()
{
}

public void ChangeCandidate(int direction)
{
    charindex += direction;
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    /*SetupCharacter();
    ResetCharacter();
    character.SetActive(true);
    individual.Act();*/
}

public void LoadReplay(string name)
{
    List<List<List<Vector3>>> data;
    ReplUtils.LoadSimulation(name, out data, out ScoreList);
    PopulationComponent.SetDNA(data);
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
}

```

Annex 7

```

using UnityEngine;

public class CameraFollow : MonoBehaviour
{
    public Transform target;
    public float smoothTime = 0.3f;
    public float smoothSpeed = 0.125f;
    public Vector3 offset;
    private Vector3 velocity = Vector3.zero;
    private void FixedUpdate()
    {
        Vector3 desiredPosition = target.position + offset;
        Vector3 smoothedPostion = Vector3.SmoothDamp(transform.position, desiredPosition, ref velocity, smoothTime * Time.deltaTime);
        transform.position = smoothedPostion;

        transform.LookAt(target);
    }
}

```

Annex 8

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics.Tracing;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GeneticAlg : MonoBehaviour
{
    private static int simcounter = 0;
    public PopulationControlller PopulationComponent;
    public EvaluationController EvalComponent;
    public Transform InitialLocation;
    public Transform DesiredLocation;
    public GeneticOperationsController GeneticOperationsComponent;
    public int NumberOfIterations = 20;

    private void Awake()
    {
        if(MenuController.IsReplayRun == true)
        {
            gameObject.SetActive(false);
        }
    }
    private void SetRefs()
    {
        EvalComponent.SetPopulation(PopulationComponent.GetPopulation());
        EvalComponent.SetTransforms(InitialLocation,DesiredLocation);
        EvalComponent.SetDistanceFromStartToEnd(Vector3.Distance(InitialLocation.position, DesiredLocation.position));
        GeneticOperationsComponent.SetPopulationDNA(PopulationComponent.GetPopulationDNA());
        PopulationComponent.SetEliteCount(GeneticOperationsComponent.EliteCount);
        PopulationComponent.SetSimCounter(simcounter);
    }
    // Start is called before the first frame update
    void Start()
    {
        SetRefs();
        if (simcounter < NumberOfIterations)
        {
            Debug.Log("started step " + simcounter);
            StartCoroutine("Coordinate");
        }
    }

    // Update is called once per frame
    void Update()
    {
    }

    private void StartSimulation()
    {
        PopulationComponent.StartMovement();
        EvalComponent.StartEvaluation();
    }
}

```

```

private void StopSimulation()
{
}
private void ResetState()
{
    //PopulationComponent.ResetState();
    EvalComponent.ResetState();
    List<Tuple<List<List<Vector3>>, List<List<float>>, List<List<Vector3>>>> CurrentPopulationDNA;
    List<float> scorelist = new List<float>();
    //ReplUtils.LoadSimulation("save0", out CurrentPopulationDNA, out scorelist);
    //PopulationComponent.SetDNA(CurrentPopulationDNA);

    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
bool ReadyToGo()
{
    return PopulationComponent.ready;
}

IEnumerator Coordinate()
{
    GeneticOperationsComponent.SetPopulationDNA(PopulationComponent.GetPopulationDNA());
    List<List<List<Vector3>>> DnaFromNextGen = new List<List<List<Vector3>>>();
    while(!ReadyToGo())
    {
        yield return null;
    }
    StartSimulation();
    EvalComponent.SetHeadPositionY(PopulationComponent.GetHeadPositionY());
    EvalComponent.SetFootPositionY(PopulationComponent.GetFootPositionY());

    while (! EvalComponent.done)
    {
        yield return new WaitForSeconds(0.5f);
    }
    ReplUtils.SaveSimulation(PopulationComponent.GetPopulationDNA(), EvalComponent.GetFitnessList(),simcounter);

    GeneticOperationsComponent.ComputeNextGeneration(EvalComponent.GetFitnessList(),out DnaFromNextGen);
    PopulationComponent.SetDNA(DnaFromNextGen);
    ResetState();
    simcounter++;
}
}

```

Annex 9

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using UnityEngine;

```

```

public static class ReplUtils
{
    public static int populationsize = 100;
    public static int steps = 500;
    public static int filecount = 0;
    public static string assignedfolderpath = "";

    public static bool IsCreated_Folder()
    {
        if (assignedfolderpath != "")
        {
            return true;
        }
        else return false;
    }
    public static void AssignCurrentFolder(string path)
    {
        assignedfolderpath = path;
    }
    public static void SaveSimulation(List<List<List<Vector3>>> CurrentPopulationDNA,
List<float> scorelist, int simcount)
    {
        var values = ConvertToSerializable(CurrentPopulationDNA);
        //GetSaveFiles();
        SaveData data = new SaveData(values, scorelist);
        BinaryFormatter bf = new BinaryFormatter();
        Debug.Log("Saved in " + "save" + simcount);
        Debug.Log("path2 = " + assignedfolderpath);
        FileStream file = File.Open(Path.Combine(Application.persistentDataPath, assigned-
folderpath, "" + simcount), FileMode.Create);
        bf.Serialize(file, data);
        file.Close();
    }

    public static string[] GetFolderNames()
    {
        var folders = Directory.GetDirectories(Application.persistentDataPath);
        return folders;
    }
    public static List<string> GetSaveFileNames(string folderpath)
    {
        var saveslist = Directory.GetFiles(folderpath);
        List<string> filenames = new List<string>();
        foreach (var x in saveslist)
        {
            filenames.Add(Path.GetFileName(x));
        }
        //filenames = filenames.FindAll(s => s.Contains("save"));
        return filenames;
    }
    public static List<float> ConvertToSerializable(List<List<List<Vector3>>> CurrentPop-
ulationDNA)
    {
        List<float> values = new List<float>(populationsize * 500 * 51);
        foreach (var list in CurrentPopulationDNA)
        {
            foreach (var x in list)
            {
                foreach (var y in x)
                {

```



```

        values.Add(y.x);
        values.Add(y.y);
        values.Add(y.z);
    }
}
}
return values;
}

public static void LoadSimulation(string path, out List<List<List<Vector3>>> CurrentPopulationDNA, out List<float> scorelist)
{
    CurrentPopulationDNA = new List<List<List<Vector3>>>();
    scorelist = new List<float>();
    if (File.Exists(path))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream fs = new FileStream(path, FileMode.Open);
        SaveData data = formatter.Deserialize(fs) as SaveData;
        fs.Close();
        ConvertToGameData(data, out CurrentPopulationDNA, out scorelist);
    }
    else
    {
        Debug.Log("Save file does not exist.");
    }
}

public static void ConvertToGameData(SaveData data, out List<List<List<Vector3>>> CurrentPopulationDNA, out List<float> scorelist)
{
    CurrentPopulationDNA = new List<List<List<Vector3>>>();
    scorelist = new List<float>();
    if (data != null)
    {
        List<float> dna = data.dna;
        List<float> score = data.score;

        int idx = 0;
        for (int i = 0; i < populationsize; i++)
        {
            List<List<Vector3>> torque = new List<List<Vector3>>();
            for (int p = 0; p < steps; p++)
            {
                List<Vector3> tlist = new List<Vector3>();
                for (int o = 0; o < 17; o++)
                {
                    tlist.Add(new Vector3(dna[idx++], dna[idx++], dna[idx++]));
                }
                torque.Add(tlist);
            }
            CurrentPopulationDNA.Add(torque);
        }
        //Debug.LogError(CurrentPopulationDNA.Count);
    }
}
}

```