2016-04-28

Using genetic algorithms as a core gameplay mechanic

Bohdan Kachmar
Worcester Polytechnic Institute

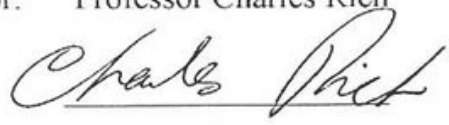
Using genetic algorithms as a core gameplay mechanic

By: Bohdan Kachmar
Oleksandr Terletskyy

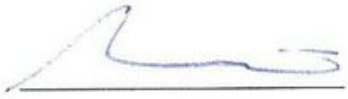
A Thesis submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirement for the Degree of Master of Science in
Interactive Media and Game Development

In this thesis we used genetic algorithms as a core gameplay mechanic for games. We created a flexible genetic algorithms framework that allowed us to iterate quickly through various designs and prototypes of games. We developed two iterations of fighting robots game and a racing game that used our framework to implement genetic algorithms. Playtesting showed that such a sophisticated game mechanic like this one can be fun and appealing to players.

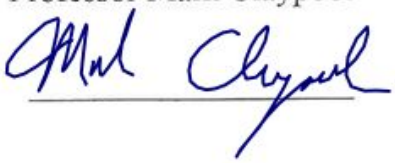
Advisor: Professor Charles Rich



Readers: Professor Brian Moriarty



Professor Mark Claypool



May 2016

Table of Contents

1. Introduction

Nowadays everyone either knows about artificial intelligence (AI) or at least has heard of it. You do not even have to be a programmer or a scientist to understand the basic idea of it. People often refer to AI as to a very complex tool that is used to solve complicated problems. Developers, on the other hand, often try to hide the process of using AI from users, by sharing only the results, which in general makes sense. We think that there is some beauty in a way that AI works to provide the appropriate results, even though it is quite hard to see from the first sight, because the process is often hidden. It is not only the result that can be interesting and appealing for the user, but also the process of getting the appropriate result.

Genetic algorithms (GA) is a topic from the world of AI that we chose to work with. The basic idea of it was taken from one of the theories of evolution that happens in real world, that is why people can use the term evolutionary algorithms to describe GA. In this paper we will use the term GA.

We think that even though GA is often considered to be a helper tool[x] that sits somewhere in the background of the development[12], there are a lot of interesting details in the process of GA itself. Moreover, it can be very interesting to watch the GA in action for the user.

We decided to built four small games that tried to utilize GA as a core gameplay mechanic in order to understand if it indeed could be interesting to players. By making GA a core gameplay mechanic, we wanted the user to play with the process of getting the results from GA.

Because of the fact that GA resembles real world evolution we selected robots as a theme for our games and prototypes. We thought that having a theme that correlates to the gameplay mechanics would help us make the gameplay more interesting and appealing to players.

Our goal is to provide the experience of playing with GA to player. By using GA as the main tool to achieve it we decided to create a space for interaction with properties of GA. Sometimes it can be tricky and even confusing, so we simplified the gameplay as much as possible while still trying to keep the desired experience of playing with GA to player.

GA, similar to real world evolution, is all about making changes to the existing content. One of our goals was to make player be always able to identify changes whenever they are made. We decided to try four different designs in each of our four games that would bring the process of GA to player in different ways.

We chose to build our demos and prototypes in Unity game engine. In order to use GA in all of our games we decided to implement our own GA framework that could be easily modified to work with each of our demos. The framework we built is extendable and is Unity friendly, but it is not dependent on Unity so it can be used with any game engine.

We used the framework we developed and Unity game engine to build four games: Fighting bots (Iteration 1), Fighting bots (Iteration 2), 3D racing bots and 2.5D racing bots. Each of the game we built was playtested to see if it was interesting for the player to play with GA and influenced the development of the next game.

2. Genetic algorithms

2.1. Background

A genetic algorithm is a search heuristic that mimics the process of natural selection. It generates solutions to optimization problems using techniques inspired by natural evolution, such as selection, mutation and crossover[13].

Genetic algorithms work with a population of individuals that evolve to increase their fitness function. Fitness function is a special measure that says how good the individual performs in its current environment. Each individual has a set of properties, which can be mutated or altered. These properties are called genes. A set of genes is called genome[14].

Evolution is an iterative process, where each population is called generation. Evolution usually starts with a randomly generated population of individuals. In each generation, every individual is evaluated using a fitness function. After that, fitter individuals are selected. This process is called selection. Figure 1.

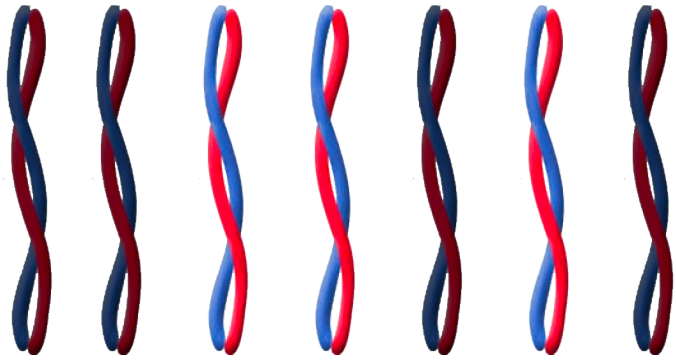


Figure 1. Selection (The least fit genomes are greyed out)

To breed a new generation, a genetic algorithm typically uses two genetic operators: crossover and mutation. To create a child, two fit parents are selected. The child inherits some genes from the first

parent and other genes from the second parent. This process is called crossover. (Figure 2) In computer language genes are represented as bits. And the inheriting genome becomes swapping bits.[13]

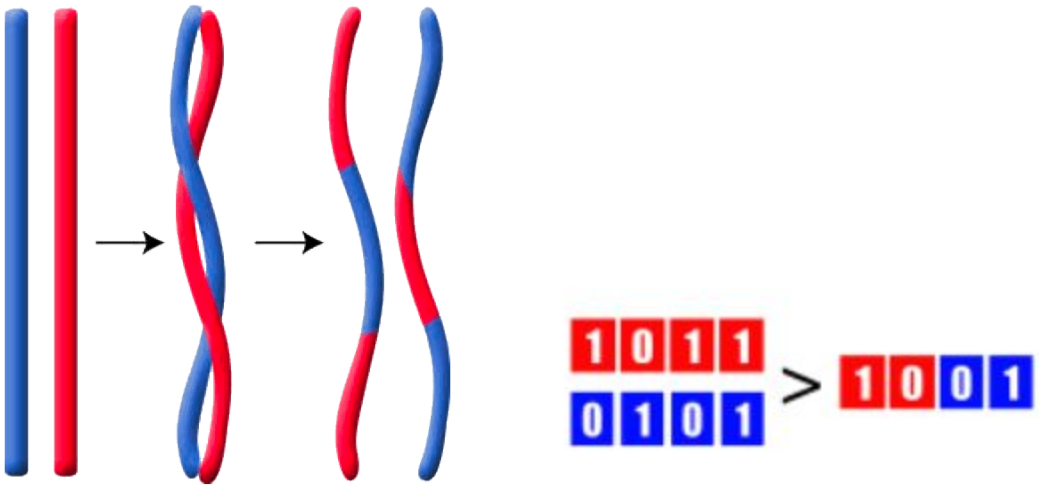


Figure 2. Crossover

The next step of genetic algorithm is mutation. In this process, some of the genes are randomly changed. Usually more suitable individuals mutate less and less suitable individuals mutate more. (Figure 3).

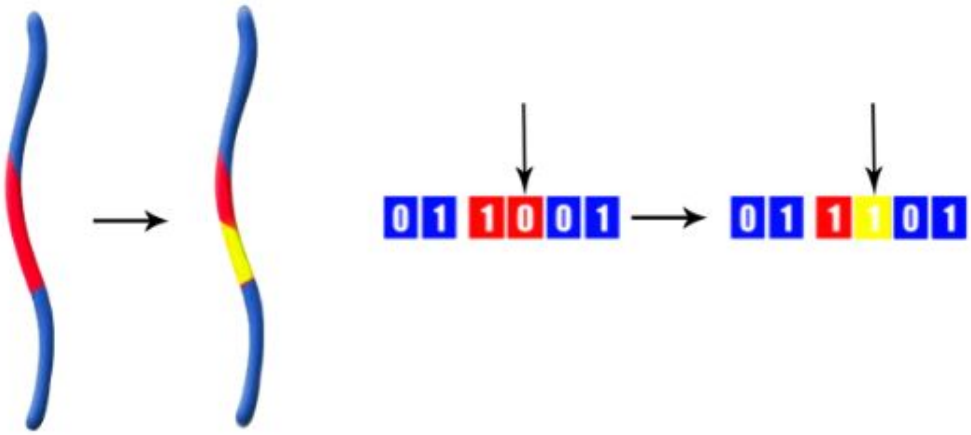


Figure 3. Mutation

2.2. Genetic algorithms framework

2.2.1 Explanation

To build four games that would use genetic algorithms as a gameplay mechanics, we needed to implement genetic algorithms. We chose Unity to be our game engine and after looking at Unity asset store[10] we did not find any existing GA tool that would allow us to incorporate GA in our games. We implemented our own genetic algorithms framework. On the figure 4 are shown the basic steps that our framework does in order to make GA work.

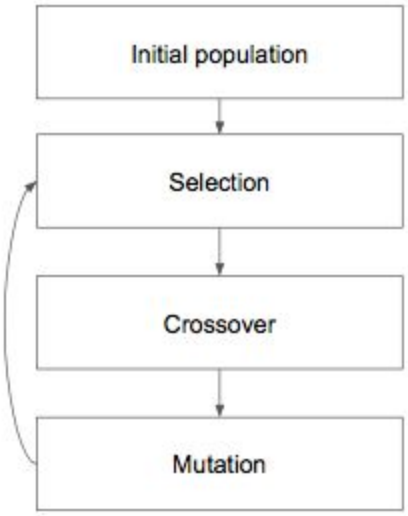


Figure 4. Steps we use in the GA framework

Our framework evolves the population of individuals that are represented with their genomes. Individual is a representation of anything that is going to be evolved using the framework. Individuals can be specified by the user of the framework by creating a class that implements an interface IIndividual about which we will talk later. On Figure 5 it is shown the example of creating different individuals.

Arrows on the figures point out four types of individuals that we made. We will describe those individuals in the design section.

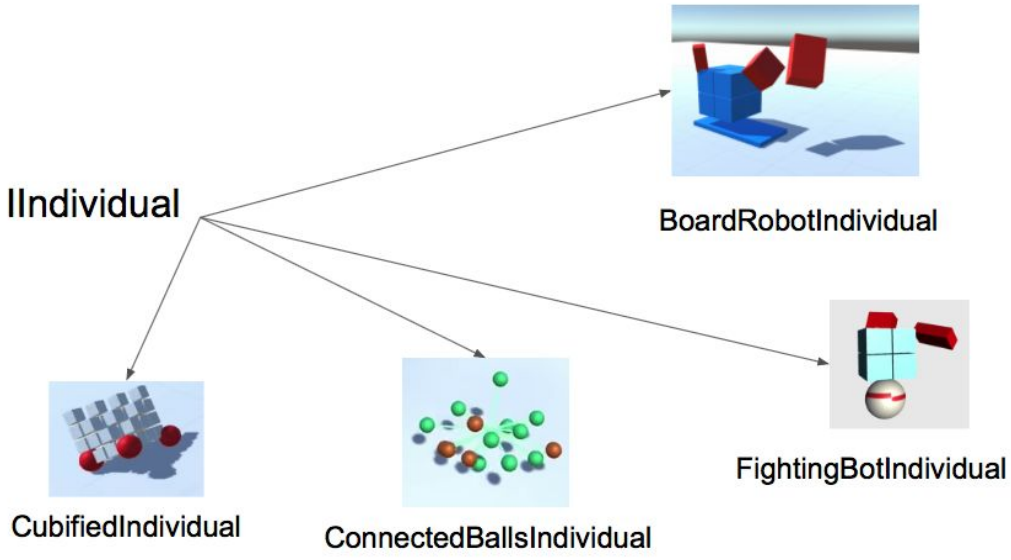


Figure 5. Different individuals

Each of the individual has its own genome and in most cases genomes of different individuals are not identical. Genome is a set of genes. Having different genome representations may require to have genes that are different from each other. That is why we made each gene being represented by a special value. Different values of the gene can be specified by implementing a specific interface IGeneValue. Figure 6 illustrates the example of creating different gene values. On the figure it is shown how we created different values for the genes that we used in the genome of the individuals. One of the examples of us creating new values for our prototypes was the following. The first iteration of our robots evolved only the size of the wheel, which can be represented as just a one floating point gene. The second generation of our robots also included the position and rotation of the arms, as well as their size. Size of the arm is a floating point gene, but the representation of the position and rotation in space should be a vector which by itself consists of three separate floating point values. A new type of gene had to be created. In one of our prototypes we added color to the representation of our robot. To implement that we

added a new type of gene which also consists of three floating point values, but has some limitations on the mutation, because we didn't want the color to mutate the same way that any vector, but change mutate the hue value only.

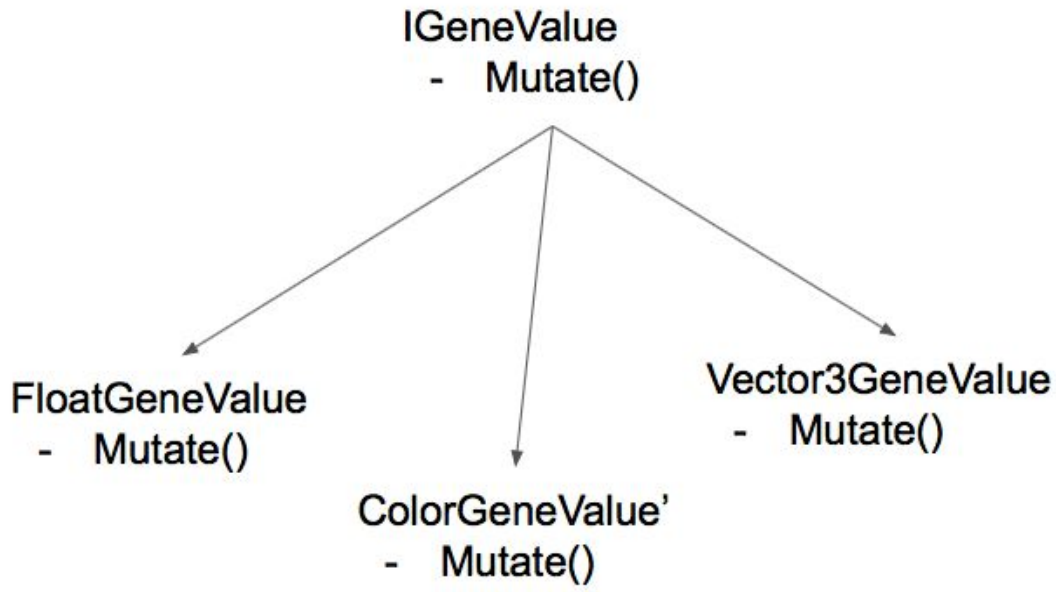


Figure 6. Different gene values

2.2.2 Technical details

Even though, our framework is supported by Unity we are not using Unity classes inside of the implementation. Therefore the framework can be easily integrated into any game engine, not necessarily Unity.

When writing custom genome representations we needed to have different types of Genes. Because of that, we decided to make the Gene functionality independent from the genetic "value" it represents. IGeneValue [Appendix A.1] is an important interface in our framework since it describes the basic behaviour of the genetic "value". By changing the implementation of "value" we can mimic having different types of genes in the individual's genome. IGeneValue has to be implemented by future custom

classes. IGeneValue ensures that all of the future values that would be used as a genetic "value" would have to know how to Mutate and Randomize themselves. IGeneValue is used as a generic type parameter in a Gene class [Appendix A.3]. Gene is able to mutate itself by mutating the value. [Appendix A.3.1].

Genome is a serializable struct [Appendix A.4] which uses IGeneValue as a generic parameter and has a collection of genes. Since genome or part of it during the evolution process is copied from one individual to another, genome is able to create its copy [Appendix A.4.1]. Genome can mutate itself [Appendix A.4.2] and make a crossover with another genes [Appendix A.4.2].

Genomes are used by individual to "individualize" themselves. Each individual implements the IIndividual interface. That interface ensures that individuals would be able to Mutate and Crossover the genome, as well as provides an option to set new genome to the individual. [Appendix A.2]. Individuals can mutate and crossover by manipulating the genome.

GeneticAlgorithm is a generic class which makes the evolution process happen. [Appendix A.5]. It operates with the array of individuals, called - population. On each step genetic algorithm is making three basic operations: Selection, Crossover, Mutation. Selection is done by sorting individuals based on their fitness function[Appendix A.5.1]. After that crossover is performed. It removes the worst half of the individuals based on their fitness score and populates new individuals.[Appendix A.5.2]. When all of the individuals are populated genetic algorithm randomly mutates the individuals. This process is done by calling Mutate function of each of the individual.[Appendix A.5.3]

During implementation of the crossover function we ran into an issue of not being able to copy genomes properly, because only references to the genomes were copied. We figured out the problem was because we needed to perform a deep copy (a technique of copying object when copy of objects are created for each reference rather than copying a reference) of the genome before applying it to another individual, or else only a reference to that genome will be copied. That is why most of the classes that we

implemented are serializable. Technique of creating a deep copy in C# requires classes to be serializable in C#.

One of the examples of our custom genetic "values" is FloatGeneValue. This class represent floating point gene that has its range from 0.0 to 1.0.[Appendix A.6]. Floating point genetic "value" mutates itself and still stays in its minimum and maximum range.[Appendix A.6.1]. Vector3GeneValue is another example of the genetic "value". [Appendix A.7] It consists of three floating point values and has its own custom Mutation function. [Appendix A.7.1]. While implementing our demos we also created ColorGeneValue and Vector2GeneValue. In order to make our classes serializable and ensure the possibility of creating deep copy we implemented our own Vector3 representation, we called it SVector3, which is a serializable version of Unity's standard Vector3.

3. Design iterations

It is very important to choose the right theme for a game with genetic algorithms to be able to make players experience the evolution. Human experience is combination of multiple factors so if we want to succeed in making a game that uses AI as a gameplay element we need to make use of that knowledge. Many people, especially players, mention robots when refer to artificial intelligence. Robots are completely associated with AI for general public. Robots theme is also good because genetic algorithms can be applied in many ways, which gives us a lot of flexibility and doesn't limit us to a single game mechanic.

By taking a terrain with a population of robot cars and letting them to evolve for a while, cars will change dramatically and they will finally be good cars! Robots can be different, therefore we can think of another scenario of robot evolution. For instance, there can be fighting robots and they can evolve their body parts to become stronger.

3.1. Reference games

A big inspiration for our project is the TV show *Robot Wars*[9] (Figure 7). The show involves teams of usually amateur robot developers, who make their own robots to compete against each other. Those robots have all kinds of different forms, shapes and weapons and from season to season they are upgraded and improved. This process reminds as of evolution – the best robots continued to compete and new ones are created.

Figure 7. Robot Wars

Genetic Cars[1] demo uses a simple genetic algorithm to evolve random two-wheeled shapes into cars over generations. It uses Box 2D physics engine for physical simulation. Even though in *Genetic Cars* player cannot control what is happening, it is still brings a lot of fun (Figure 8). This shows that AI is very interesting to investigate and to play around with it. The user may not know anything about the theory of genetic algorithms but still can easily change some parameters and see the results.

In *Genetic Cars* they use a simple genome representation: shape (8 genes, 1 per vertex), wheel size (2 genes, 1 per wheel), wheel position (2 genes, one per wheel), wheel density (2 genes, 1 per wheel) and chassis density (1 gene). Fitness function is a maximum distance that cars can travel. Between each two races, selection, crossover and mutation are applied.

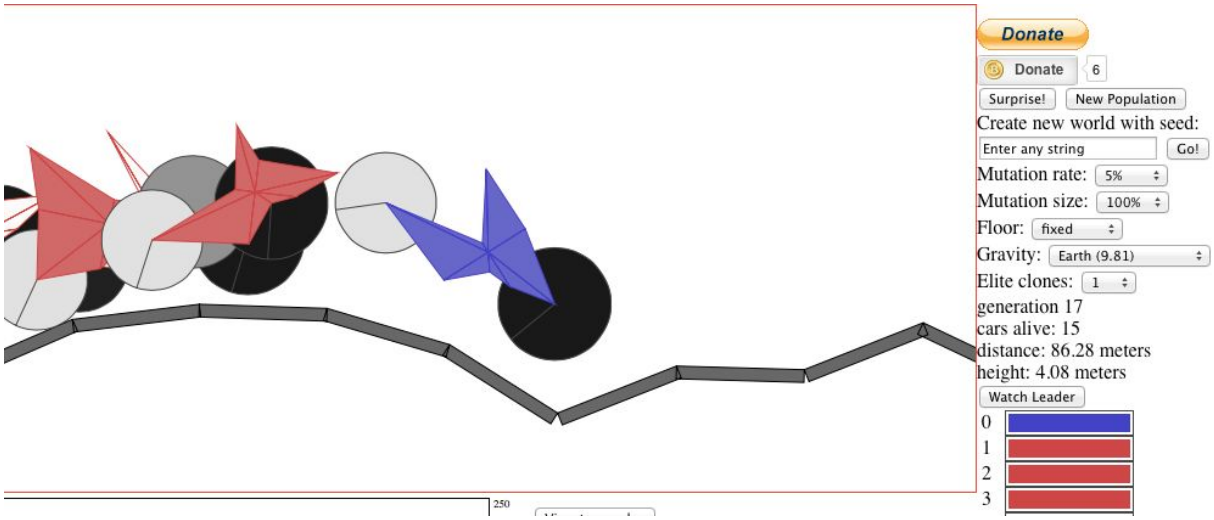


Figure 8. Genetic Cars.

Egg worm generator[2] (Figure 9 a) is a game that evolves creatures that learn how to move. Egg worms should go only to the right side of the screen and fast as possible. Genetic algorithm is a key gameplay element as well.

Figure 9. a) Egg Worm Generator b) Cambrian Explosion

Cambrian Explosion[3] (Figure 9 b) is also a worm simulator. Genetic algorithms are used to evolve worm movement. Each worm consists of given amount of segments. Player can control their physical representation as well as control the process of evolution by changing the genetic algorithms parameters.

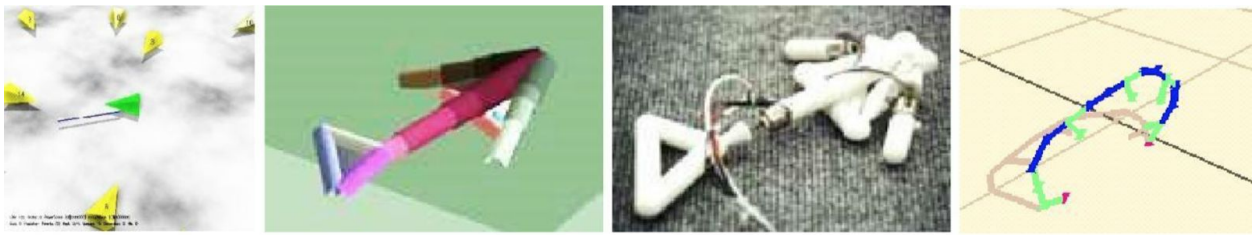


Figure 10. a) Combat Bots b) The Golem Project c) Sophia – Walking Star

Combat Bots[11] (Figure 10 a) is a game where player tries to hit bots with a bat, which are evolving using neural networks and genetic algorithms. The goal for bots is to deal as much damage as possible to the player while avoiding being hit. This is a good example of evolving interesting group behaviors using genetic algorithms. For instance, bots will sacrifice themselves for the group to deal more damage to player. Genome representation is weights and nodes of the neural network. Neural network has inputs based on player position according to bot (left, right, etc.) and outputs what action to make (move left, right, forward).

The Golem Project[5] is Hod Lopson's and Jordan Pollock's work (Figure 10 b) at Brandeis University aimed to evolve a moving electro-mechanical system from scratch. They generate a population of candidate robots composed of some repertoire of building blocks, where fitness function is their ability to move. After simulating the evolution they build real prototypes.

Hierarchically Regular Locomoting Robots (Genobots)[6] (Figure 10 c) created by Dynamical & Evolutionary Machine Organization evolve both the morphology and the controllers for different robots. They use genetic algorithms to create different creatures and later build some prototypes in real life. It is not a game but is beautiful to watch.

Agar IO[8] (Figure 11) is a very popular browser multiplayer game. Main game mechanic is to grow bigger in relatively fast game sessions. In this game even if you are very big you can lose all your power very quickly.

Figure 11. Agar IO

Clash of Clans[7] is a freemium mobile MMO strategy game. In this game player develops his village, upgrades buildings and trains different kinds of troops to compete with other villages. In our prototypes we also used a mechanic where player evolves his character during multiple game sessions and competes with enemies whenever he thinks he is ready.

3.2. Fighting bots iteration 1

After trying out different kinds of robots with different genomes and multiple fighting scenarios we chose the one that inspired us the most. Robot body contains of a main cube that drives on top of a sphere and has three parallelepiped hands. Genome of the robot consists of one float variable that determines the size of a wheel and three sets of nine float variables that determine size (3 floats), position (three floats) and rotation (three floats) of hands. Size of a wheel determines speed and torque power of

robot. Hands can be used for both defence and offence, depending on size and placement (Figure 12). When hands of other robots hit main cube - robot dies: all his hands randomly fall apart and the body with a wheel freezes in the air on top of the fighting place with a nice “heaven” sound effect.

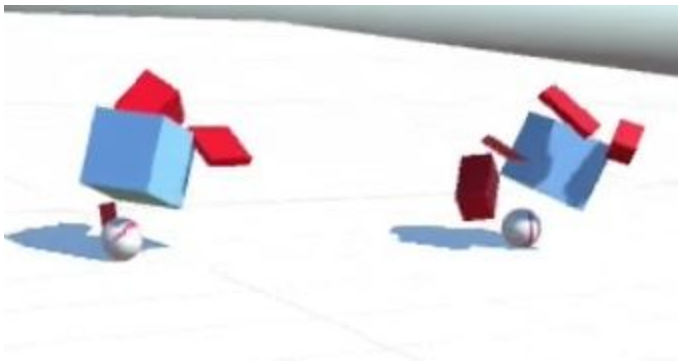


Figure 12. Fighting bots from iteration 1

Ten robots of this type are fighting in a deathmatch scenario. In game mode camera is set to top down view, but for research purposes we can easily control it in editor mode. The strategy of robots is simple: each robot randomly chooses single opponent and tries to crash him. In this setup some robots might evolve as fast attacking robots, and other could be slower and use defencive hand placement. After each round three best robots are saved and displayed in head-up display for the next round. For this prototype evolution is not implemented, so every time random robots are fighting (Figure 13). Video preview of this prototype is available here: *https://goo.gl/fCBw24*.

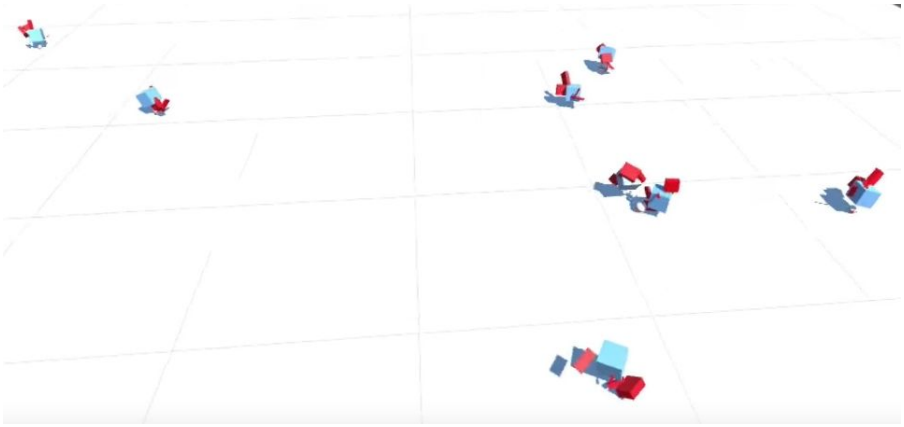


Figure 13. Deathmatch scenario from iteration 1

In this iteration player is just a spectator, he cannot interact with robots. After playtesting we saw that users enjoy seeing all the different kinds of random robots fighting in a simple setup. So we decided to continue working on this kind of robots with some slight changes in physics model and other parameters and make it into a playable game prototype, to just a visual demo.

3.3. Fighting bots iteration 2

After playtesting and analyzing first prototype we saw a lot of flaws and based on feedbacks decided to physics model of robot (Figure 14). We adjusted sizes and durabilities of all the parts and instead of one main cube new robot got four smaller cubes and to die robot had to lose all four of them. In this case robots became more durable and this made one on one combats a little longer and more interesting to watch. What is more, this opened new evolution strategies for hand placements. Video review of the iteration is available here: https://www.youtube.com/watch?v=O1Vm06YSkdQ.

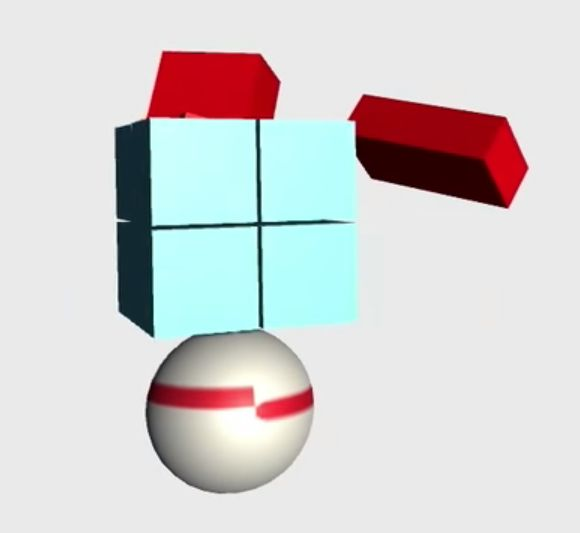


Figure 14. Fighting bot from iteration 2

3.3.1. Main menu

The goal of the game is to evolve your robot against different enemies and finally beat all of them in the deathmatch mode. In main menu player can see blacklist of enemy robots (with possibility to preview them) and current challenge (set of three robots to fight against in death match). Together with actual robot physics model - schematic Figure of DNA is used to represent his structure. In this example we use colors (red, green and blue) to represent coordinates, rotations and sizes (x, y and z). Both physics modes and DNA model are rotating for user to be able to see it from different sides.

User's robot, his evolutions and progress is saved between game sessions in XML files.

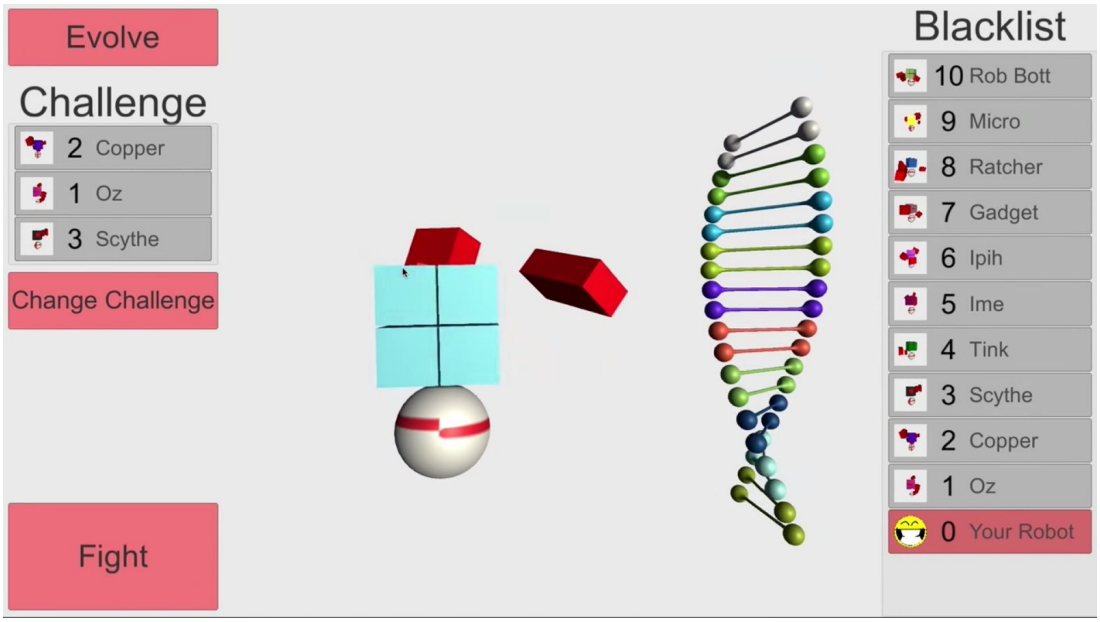


Figure 15. Main menu

3.3.2. Fighting mode

The goal of the game is to go to the top of the blacklist (Figure 16). To do this player has to defeat all the enemies in a fighting mode.

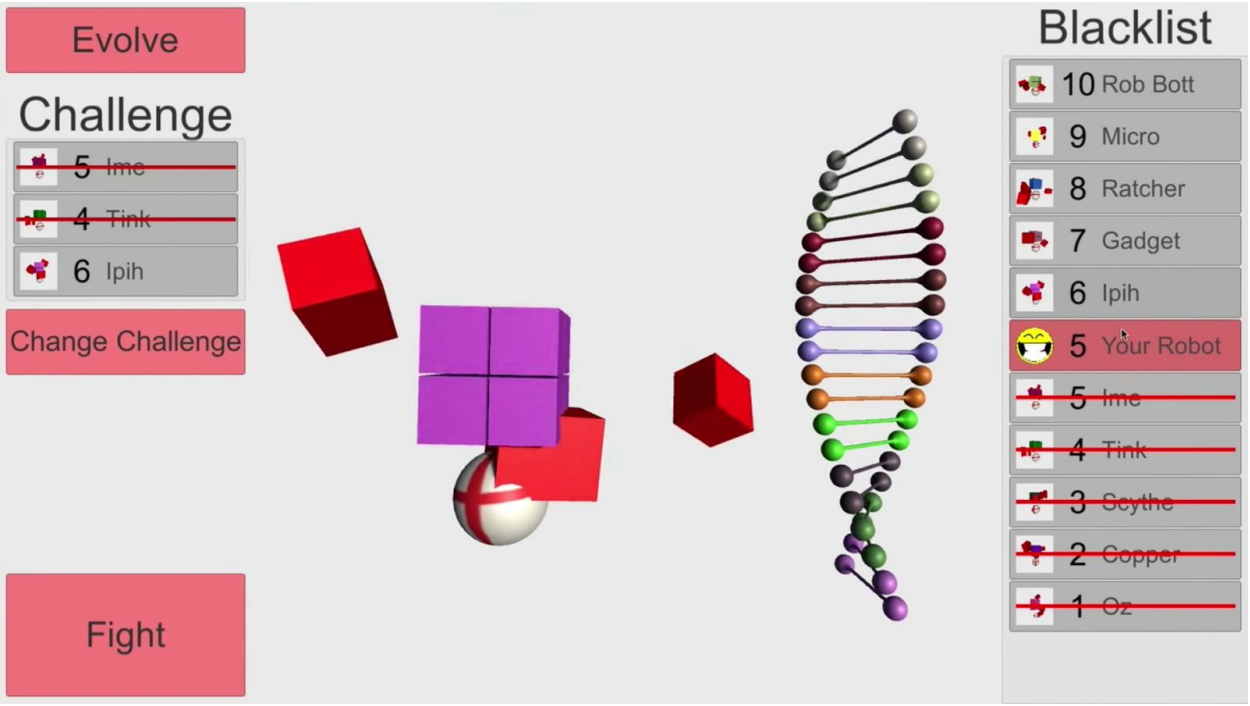


Figure 16. Main menu after a few rounds of deathmatch

The fighting mode is a death match with four randomly placed robots on the plain map. Camera is set to third person mode to be able to easily track your robot (Figure 17). Game continues until either player's robot dies or all the enemies die. This mode is very similar to the mode in first iteration, but has some adjustments to make it more interesting to play.

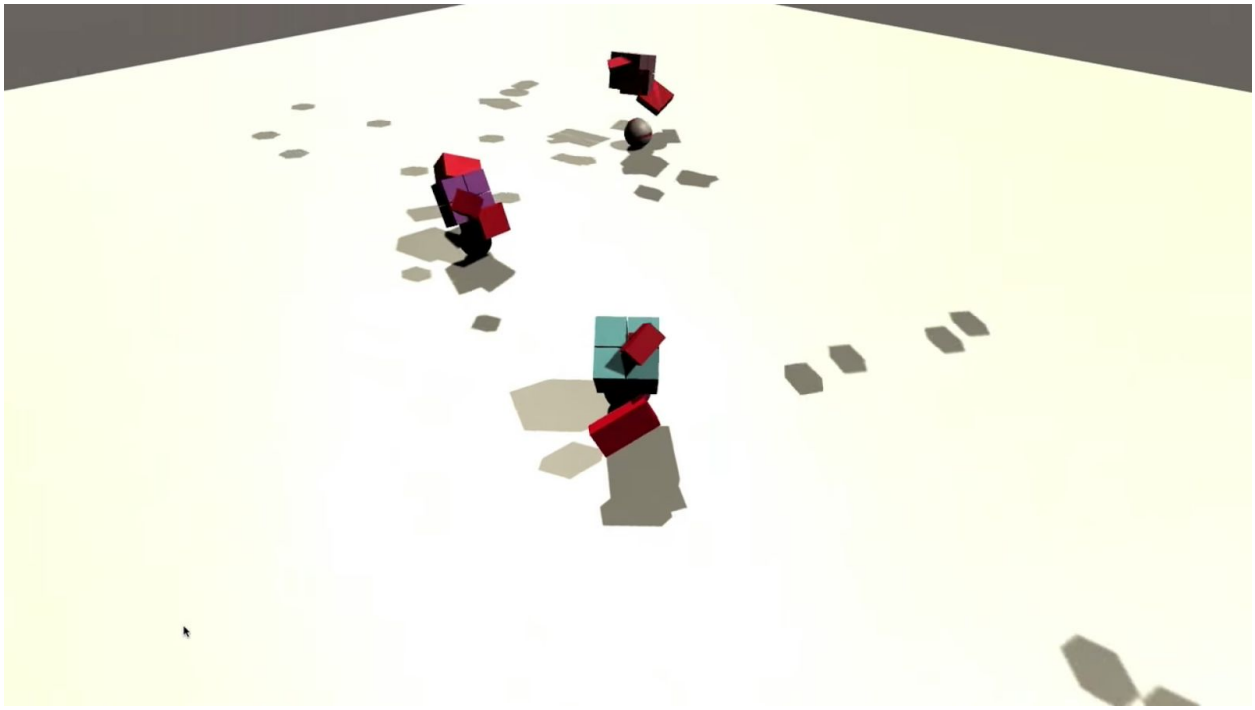


Figure 17. Fighting mode

Fighting mode had the same physics issues as evolution mode, so we applied the same mechanic changes and result was quite satisfying.

3.3.3. Evolution mode

At the beginning of the game player has his default random robot, so to change him, player has to evolve him. Also when in one of challenges player finds a bot that is always winning, player can evolve his own robot to become better against that specific enemy. That is why there is a possibility to choose evolution enemy in main menu. Initial setup is the following (Figure 18): five same enemy robots are on the right side of the screen and five slightly mutated versions of player's current robot are on the left side of the screen. One fittest (current) robot is always visible in head-up display. Each robot fights corresponding robot one on one until either one of them or both die. In case one robot stays alive - he returns to his home position. After all the robots die or return to home position Genetic algorithm

(selection, crossover and mutation) is applied to player's robots and new population of five robots appears. This allows player to train his robot against tough enemy to be able to beat him and move up the blacklist.

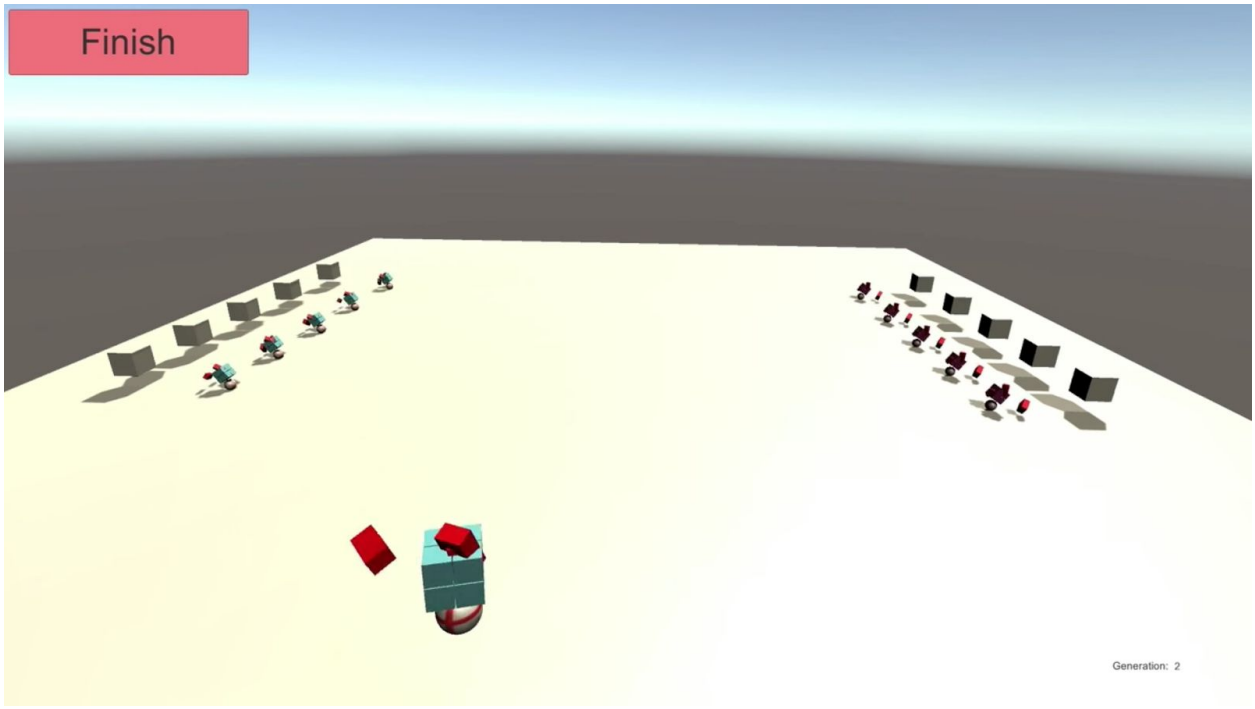


Figure 18. Initial setup of evolution mode.

After testing different setups on this scene we encountered physics problems with these robots. When two robots are close to each other and try to go towards each other - physics engine cannot handle it properly. This is why with the same setup of robots results can be different. And this means that fitness of robot depends not only on his structure, but also on the randomness of physics engine.

To overcome these issues we changed the wheel of the robot to a moving board (Figure 19). Hard connection of body and board turned out a good solution to lots of physics issues. After this we developed another strategy for robots: after gaining speed robots slide into each other (in this case force isn't applied to robots during the collision). In case both robots survive the collision they go away from each other and then gain speed to collide again. This strategy improvement made one on one combats a lot more interesting to watch and during the playtesting all users liked that.

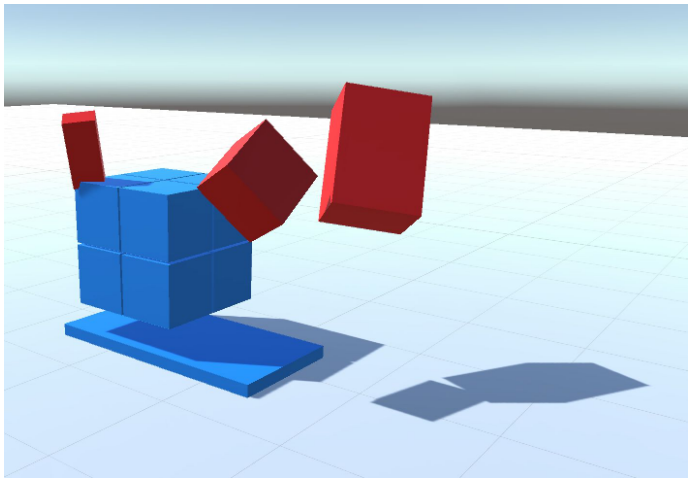


Figure 19. Changed physics model for second iteration robots

With these features we dramatically reduced randomness of combat, but still it was not enough. We know that in evolution process randomness is always present, so why did we want to get rid of it? Because randomness makes the fitness function not stable. And the less stable fitness function is - the more fit individuals will be discarded by selection and the longer time it will take actually evolve in a good way. And due to the fact that the game is supposed to be played by user (not evolving overnight), we don't have that time.

For example if somehow a very strong type of robot evolves, other individuals will want to breed with him, but due to the randomness he can lose a fight against an easy enemy and his genome will be discarded by genetic algorithm.

3.4. 3D cars

After playing around with physics model changes and robot behaviour changes we decided that it will not be feasible to completely resolve these issues, so we decided to change the topic and return to our initial inspiration of cars driving in 2D space. Both racing and fighting are competitive mechanics, but in racing games car to car interactions are not main element, so evolution works in a different way. This

means that even if collisions of two physical objects are not calculated in the same way every time - this doesn't noticeably affect the evolution process.

To test different prototypes we used the same template based genetic algorithm framework that we developed earlier. And with the new models and new environment we were able to simulate exactly the same outcomes from the same starting conditions.

We rapidly iterated through different types of racing robot representations and first successful prototype is called "Cubification" (Figure 20). These cars are defined by placement of 8 vertices of polyhedron. The polyhedron is filled with small cubes from the inside and wheels are attached to four of the vertices. The size of polyhedron defines the number of cubes, size, center of mass and weight of the robot. What is more - those models were driving on a 3D surface and showed very good and nice-looking results.

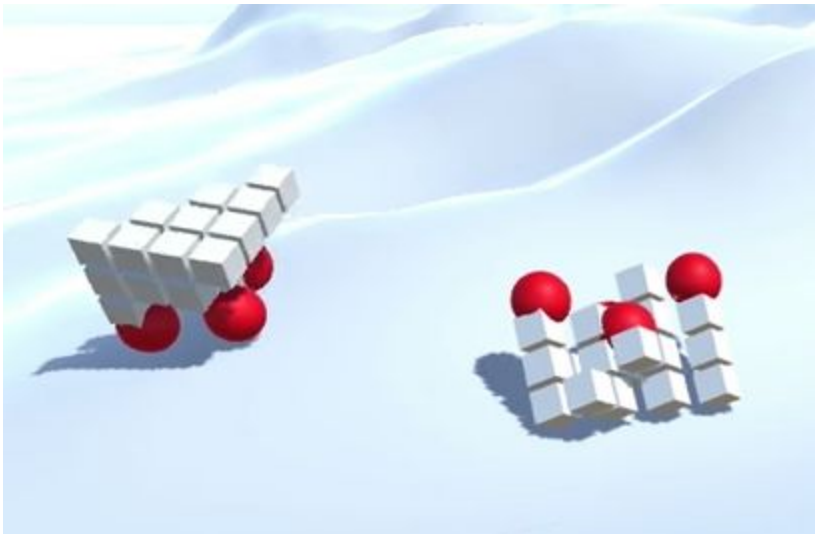


Figure 20. "Cubification" model

Second successful prototype of racing robots is called "Connected balls" (Figure 21). These robots are defined by positions of the wheels and type of their connection (wheels are connected to the

center of mass, wheels are connected to each other one by one and all the wheels are connected to the main wheel).

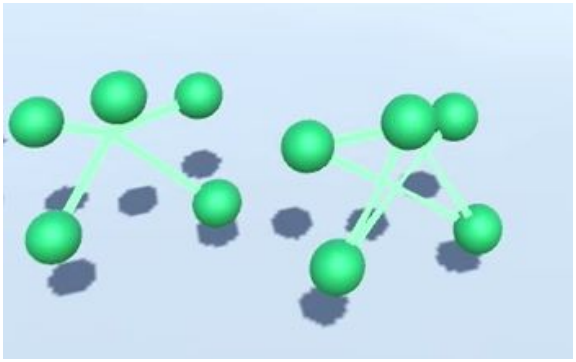


Figure 21. "Connected balls" model

These robots showed very nice results in obstacle overcoming, since they could turn upside down and still drive on other wheels. So to make it harder we increased the number of wheels to 15 and made 5 of them inactive. This iteration of racing robots showed the biggest potential, so we created a round map with hills and aim in the middle and fully applied genetic algorithms to this setup (Figure 22).

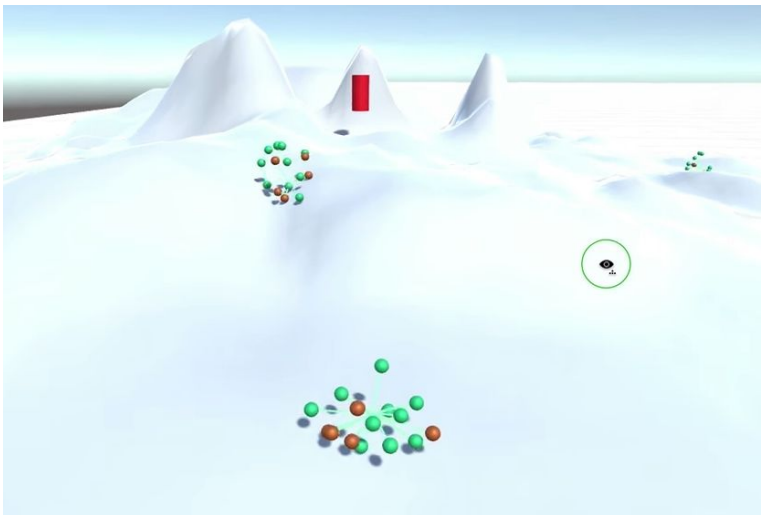


Figure 22. Setup for second iteration of "Connected balls"

During the evolution these robots sometimes learned how to hide disabled wheels inside, so only active wheels were outside. What is more, some robots turned out to get more rounded shape to be able to

equally drive on all sides and other became more flat to have more active wheels touching the ground at the same time.

During the playtesting we saw that the biggest problem in this setup is that it's too hard to recognize your car. Since all the wheels are very similar and cars drive on different sides - the only way to memorise them is to look very closely for some time, but constant evolution made this very hard.

3.5. 2.5D cars

After implementing 3D cars we decided to make our next iteration. We made an assumption that simplifying the 3D space to 2D would make the evolution process become more controllable from algorithmic point of view and easier to see for the player. In order to make the game more interesting to watch we created 2.5D space. By that we mean that some parts are done in 3D and some parts in 2D. Video demo of this iteration is available here: https://www.youtube.com/watch?v=FYppYEVib6M.

We generated a 3D terrain by using fractals algorithm "squares and diamonds".

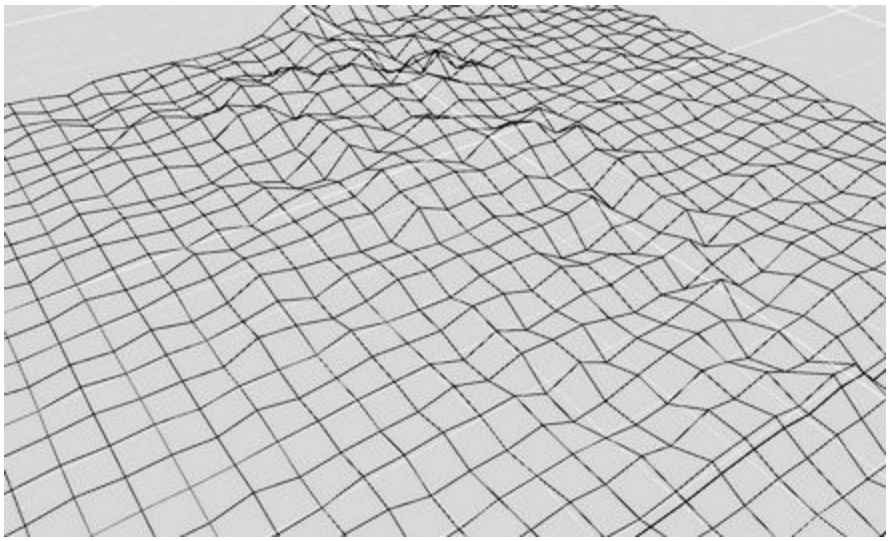


Figure 23. Squares and Diamonds terrain generation

Generating it programmatically allowed us to modify the difficulty of the terrain by tweaking just a few parameters of the algorithm. The goal of this iteration was to evolve our cars in 2D. So we decided

to use a slice of the terrain as the road for the 2D car. When steering the car would steer in 3D space, therefore changing the appearance of 2D road for user. Finally, it looked like the following Figure.

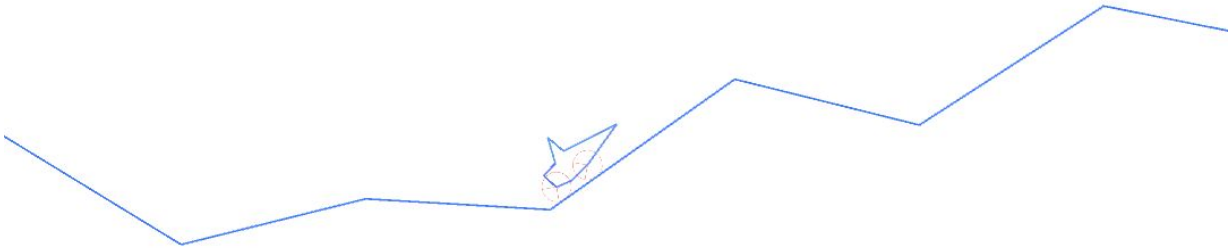


Figure 24. 2D Car on a "slice" from 3D world

In order, to make the game feel more than just a 2D we added a possibility to look at the same 2D scene from a 3D perspective. Video preview of this mode is available here: https://goo.gl/a0OsCt.The next figure shows our final result..

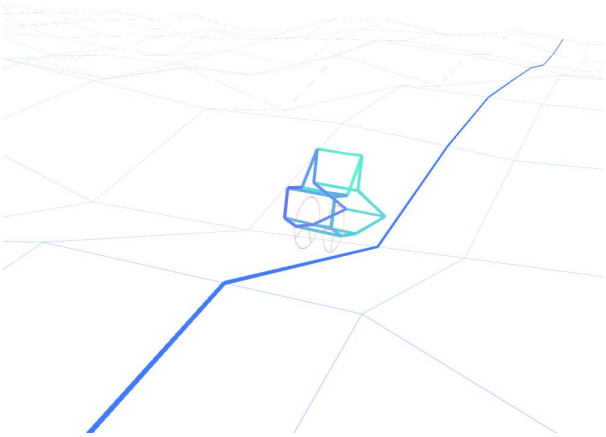


Figure 25. 3D view on a 2D car

During evolution the car would evolve in 2D space. The genome includes positions of the car body vertices and wheel sizes. In evolve mode player can adjust the time scale, so this allows him to both look at particular cars how they look like, how they overcome obstacle in slow speed and enjoy quick evolutionary results on high speed. The video demo is available here: https://goo.gl/ssBF94.

The whole idea in this demo was two make the car change in the real-time during the race. Car takes the evolution information from the evolve mode and morphs its shape from one evolution to another during the race. The race appeared to be very dynamic since both the car and the terrain were changing at the same time.

When playing user is able to try three different camera modes, experiencing the game in both 2D and “3D” modes. This gives him impression that it is actually a 3D world with a 3D car, but actually all the physics is calculated in 2D. So by adding some graphics like a dune buggy instead of plain lines, desert instead of lines in terrain and actual road instead of just one highlighted line - user will feel the 3D world with just some 2.5D limitations. And player’s skill will be in correct timed turns to avoid rough parts of the 3D terrain better than his opponents.

4. Conclusion

4.1. Playtesting

Playtesting was important part of our development process. We didn't do any official playtesting protocols, but we used to show all the results to our friends and IMGD students. During the development process we had 5 people who playtested each iteration of our prototypes and 14 people who played and gave feedback on four main prototypes. Occasionally we also gave our intermediate results to random students in WPI and sometimes their feedback was even more crucial than feedback of regular playtesters.

At the very beginning of our work we made a small prototype with two cars that were pushing each other out of the block. Those cars had simple body structures and their wheel sizes mutated randomly. After playing around with it all our friends really enjoyed it. What is more important, we saw a great potential in this topic and decided to choose it as our thesis project.

When we finished making first iteration of our fighting bots we tried to show it to as many people as possible. After initial playtests users advised us to add sound effects to the demo and this made it a lot better. At the end of last year we brought this prototype to IMGD game showcase and got a lot of positive reviews and very important feedback.

After that we decided that we want to make a full game to be able to playtest it completely, not just some parts. So we added menu with blacklist, evolution and fighting mode. After some playtesting sessions users told us that they don't like the fact that they are fighting against random unknown opponents every time, so we decided to add challenges to our game. In general users liked it, so we decided to keep it. We knew that it would be very nice to be able to preview opponents in menu, but only after users told us the same we decided to actually implement this feature.

While making our iterations we changed the appearance of robots and the game to make the evolution become easy to notice. But the biggest problem that we realised after playtesting sessions was the fact that actually users didn't care about genetic representation of robots, or about genetic algorithms. So that is why we decided to add a model of DNA that is rotating together with robot and paint it in colors, that represent robot's genome. After this feature was added players started to notice differences in DNA of different robots and they told us that it made them actually think about genetics while playing the game.

When we switched to racing robots and designed "Connected balls" model we gave it to playtesters and they told us that all those robots look too similar. As developers we were able to see all the details of each of them, but for actual users they turned out the same. That is the main reason why we decided to continue exploring other robotic racing models.

When we gave first prototypes of 2.5D cars to users - they liked them a lot. The biggest amount of good feedback we received after we implemented possibility to switch between 2D and 3D space with the same car.

Overall playtesters gave us a lot of useful advices on how to adjust our prototypes to be more interesting and fun to play.

4.2. Post Mortem

While designing a game with a core game mechanic like genetic algorithms it is important to keep in mind that it may be hard to create the role for the player in game, even though the mechanic looks very interesting and appealing to player. On the other hand, the game can be very fun even with less interaction.

By finding out what is exactly fun about the mechanics we were using, we managed to emphasize on specific parts in our small demos. For instance, showing the evolution progress in 2D allowed us to give the player much clearer picture of what is changing in between the evolutions.

What went right?

We developed a flexible genetic algorithm framework. It really helped us a lot in iterating through different genetic representations of the same robots and through whole different types of individuals. Before choosing robotic theme we implemented this framework even to a tower defence style prototype.

We developed four prototypes, one of which was a full game prototype with winning condition.

We playtested all our results with both regular playtesters and with random players and this gave us a lot of feedback and inspiration.

We researched a new field of gameplay mechanics based of the beauty of genetic algorithms.

What went wrong?

We had a lot of unexpected physics problems while detecting collisions between fighting bots. With the same initial setups combat results turned out very random. Even though we changed and tweaked our physics model and their behaviours, we weren't able to remove the randomness.

This made our fitness function not stables, which means that we had to run a lot more generations of evolution to get good outcomes, but in case of tablet game we didn't have all that time.

References

1. “Genetic Cars”. Web. http://rednuht.org/genetic_cars_2
2. “Egg worm generator”. Web. http://goo.gl/soIU1s
3. “Cambrian Explosion”. Web. http://www.cambrianexplosion.com/index.html
4. Jacob Schrum, Risto Miikkulainen. Evolving Agent Behavior In Multiobjective Domains UsingFitness-Based Shaping, Portland, Oregon, Genetic and Evolutionary Computation Conference,July 2010
5. “The Golem Project”. Web. http://www.demo.cs.brandeis.edu/golem/design.html
6. “Hierarchically Regular Locomoting Robots”. Web. http://www.demo.cs.brandeis.edu
7. “Clash of Clans”. Web. http://supercell.com/en/games/clashofclans/
8. “Agar.io”. Web. http://agar.io/
9. “Robot Wars” TV show. Web. http://www.robotwars.tv/
10. Unity asset store. Web. https://www.assetstore.unity3d.com/en/
11. Jacob Schrum, Risto Miikkulainen. “Constructing Complex NPC Behavior via Multi-Objective Neuroevolution”. Web. http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-018.pdf
12. “Good examples of Genetic algorithms solutions” discussion. Web. http://stackoverflow.com/questions/1538235/what-are-good-examples-of-genetic-algorithms-genetic-programming-solutions
13. “Creating a generic algorithm for beginners” article. Web. http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3
14. [14“Learning: Genetic Algorithms” lecture from MIT OpenCourseWare. Web. https://www.youtube.com/watch?v=kHyNqSnzP8Y

Appendix A: Genetic algorithms framework

A.1. IGeneValue

```
1. public interface IGeneValue
2. {
3.     void Mutate (float mutationRange = 1.0f);
4.     void Randomize();
5.     System.Object Value();
6.     void SetValue(System.Object value);
7. }
```

A.2. IIndividual

```
1. public interface IIndividual<T> where T : IGeneValue
2. {
3.     void Mutate(float mutationRange = 1.0f);
4.     float Fitness();
5.     Genome<T> GetGenome();
6.     void SetGenome(Genome<T> genome);
7.     void Randomize();
8.     void IncreaseGeneration();
9. }
```

A.3. Gene

```
1. [Serializable]
2. public struct Gene<T> where T : IGeneValue
3. {
4.     private T value;
5.
6.     public T Value
7.     {
8.         get
9.         {
10.            return this.value;
11.        }
12.        set
13.        {
14.            this.value = value;
```

```
        }
    }

```

A.3.1. Gene mutation

```
    /// <summary>
    /// Mutates the gene
    /// </summary>
    /// <param name="mutationRate"> Mutation range [0.0f, 1.0f] </param>
    public void Mutate(float mutationRange = 1.0f)
    {
        value.Mutate(mutationRange);
    }

```

A.4. Genome

```
[Serializable]
public struct Genome<T> where T : IGeneValue
{
    private Gene<T>[] genes;

    public Gene<T>[] Genes
    {
        get
        {
            return genes;
        }
        set
        {
            genes = value;
        }
    }
```

A.4.1. Genome. Creating deep copy

```
    public Genome<T> CreateDeepCopy(Genome<T> inputcls)
    {
        MemoryStream m = new MemoryStream();
        BinaryFormatter b = new BinaryFormatter();
        b.Serialize(m, inputcls);
        m.Position = 0;
        return (Genome<T>)b.Deserialize(m);
    }
```

A.4.2. Genome. Mutation

```
25.     public void Mutate(float mutationRange = 1.0f)
26.     {
27.         for (int i = 0; i < this.genes.Length; i++)
28.         {
29.             genes[i].Mutate(mutationRange);
30.         }
31.     }
```

A.4.3. Genome. Crossover

```
32.     public static Genome<T> Crossover(Genome<T> genome1, Genome<T> genome2)
33.     {
34.         if (genome1.genes.Length != genome2.genes.Length)
35.         {
36.               Debug.LogError("Cannot crossover genomes with different amounts of
   genes genome1: " + genome1.genes.Length + " genome2: " + genome2.genes.Length);
37.         }
38.
39.         int crossoverPoint = UnityEngine.Random.Range(0, genome1.genes.Length);
40.
41.         Genome<T> result = new Genome<T>();
42.         result.genes = new Gene<T>[genome1.genes.Length];
43.
44.         Genome<T> genome1Copy = genome1.CreateDeepCopy(genome1);
45.         Genome<T> genome2Copy = genome2.CreateDeepCopy(genome2);
46.
47.         for (int i = 0; i < crossoverPoint; i++)
48.         {
49.             result.genes[i] = new Gene<T>(genome1Copy.genes[i].Value);
50.         }
51.
52.         for (int i = crossoverPoint; i < genome1.genes.Length; i++)
53.         {
54.             result.genes[i] = new Gene<T>(genome2Copy.genes[i].Value);
55.         }
56.
57.         return result;
58.     }
59. }
```

A.5. GeneticAlgorithm

```
1. public class GeneticAlgorithm<IndividualT, T> where IndividualT : IIndividual<T> where T : IGeneValue
2. {
3.     private IndividualT[] population;
4. 
5.     public IndividualT[] Population
6.     {
7.         get
8.         {
9.             return population;
10.         }
11.         set
12.         {
13.             population = value;
14.         }
15.     }
```

A.5.1. GeneticAlgorithm. Selection

```
16.     private void Sort()
17.     {
18.         for (int i = 0; i < population.Length - 1; i++)
19.         {
20.             for (int j = i + 1; j < population.Length; j++)
21.             {
22.                 if (population[i].Fitness() < population[j].Fitness())
23.                 {
24.                     IndividualT temp = population[i];
25.                     population[i] = population[j];
26.                     population[j] = temp;
27.                 }
28.             }
29.         }
30.     }
31.     /// <summary>
32.     /// Sorts in the way that the fittest element is first
33.     /// </summary>
34.     private void Selection()
35.     {
36.         Sort();
37.     }
```

A.5.2. GeneticAlgorithm. Crossover

```
38.     /// Remakes second half of individuals based on first (fitter) half
39.     /// </summary>
40.     private void Crossover()
41.     {
42.         for (int i = 0; i < population.Length / 2; i++)
43.         {
44.             Genome<T> gene1;
45.             Genome<T> gene2;
46.                             population[population.Length  /  2  +  i].SetGenome(
    Genome<T>.Crossover(
47.                     population[i].GetGenome(), population[i + 1].GetGenome()));
48.         }
49.     }
```

A.5.3. GeneticAlgorithm. Mutation

```
50.     private void Mutation()
51.     {
52.         // Range (1.0f, 10.0f)
53.         float mutationDecreaser = 1.0f;
54.
55.         for (int i = 1; i < population.Length; i++)
56.         {
57.             population[i].Mutate(0.3f);
58.         }
59.     }
```

A.5.4. GeneticAlgorithm. Step

```
60.     public void Step()
61.     {
62.         for (int i = 0; i < population.Length; i++)
63.         {
64.             population[i].IncreaseGeneration();
65.         }
66.         Selection();
67.         Crossover();
68.         Mutation();
69.     }
```

A.6. FloatGeneValue

```
1. public struct FloatGeneValue : IGeneValue
2. {
3.     private float value;
4.
5.     public float MinValue;
6.     public float MaxValue;
```

A.6.1. FloatGeneValue. Mutation

```
7.     public void Mutate(float mutationRange = 1.0f)
8.     {
9.         float delta = mutationRange / 2.0f;
10.
11.        value += UnityEngine.Random.Range(-delta, delta);
12.
13.        ClampValueToMinMax();
14.    }
```

A.7. SVector3

```
1. [Serializable]
2. public struct SVector3
3. {
4.     public float x;
5.     public float y;
6.     public float z;
7.
8.     public SVector3(float _x, float _y, float _z)
9.     {
10.        x = _x;
11.        y = _y;
12.        z = _z;
13.    }
14.
15.    public Vector3 ToVector3()
16.    {
17.        return new Vector3(x, y, z);
18.    }
19.
20.    public static SVector3 zero = new SVector3(0.0f, 0.0f, 0.0f);
21.    public static SVector3 one = new SVector3(1.0f, 1.0f, 1.0f);
22. }
```

A.8. Vector3GeneValue

```
1.  [Serializable]
2.  [DataContract(Name = "Vector3GeneValue", Namespace =
    "http://www.geneticwars.com")]
3.  [KnownType(typeof(IGeneValue))]
4.  /// <summary>
5.  /// All gene values in project are [0; 1]
6.  /// </summary>
7.  public struct Vector3GeneValue : IGeneValue
8.  {
9.      [DataMember(Name = "value")]
10.     private SVector3 value;
11.
12.     [NonSerializedAttribute]
13.     public SVector3 MinValue;
14.     [NonSerializedAttribute]
15.     public SVector3 MaxValue;
```

A.8.1 Vector3GeneValue - Mutation

```
16.     public void Mutate(float mutationRange = 1.0f)
17.     {
18.         float delta = mutationRange / 2.0f;
19.
20.         float valueX = value.x + UnityEngine.Random.Range (-delta, delta);
21.         float valueY = value.y + UnityEngine.Random.Range (-delta, delta);
22.         float valueZ = value.z + UnityEngine.Random.Range (-delta, delta);
23.
24.         value = new SVector3 (valueX, valueY, valueZ);
25.
26.         ClampValueToMinMax ();
27.     }
```