Self-optimizing evaluation function for first person shooter combats in 3-dimensional environments

Jonathan J. Allarassem
Department of Computer Science
Grove City College
Grove City, PA
AllarassemJJ20@gcc.edu

Brian Dellinger
Department of Computer Science
Grove City College
Grove City, PA
DellingerBJ@gcc.edu

ABSTRACT
Given a 3D discrete randomly generated environment E and a sniper S capable of shooting a target, it is not obvious which strategy should S follow to be successful at one versus one combat. It is not clear what a good strategy even looks like for the sniper agent S in the environment E. The approach of this paper to solve this problem is to use Particle Swarm Optimization to learn good strategies. Two types of fights were accounted for: static fights (no motion of the agent) and dynamic fights (the agent can move around). Despite the stochastic nature of the approach, results suggest the existence of a unique best strategy specially in the case of static fights.

KEYWORDS
Particle Swarm Optimization, Sniper, First Person Shooter learning algorithm. Evaluation function, Learning the evaluation function, Self- optimizing evaluation function, videogame.

1 Introduction

For a sniper agent living in a 3-dimensional reality and facing an enemy, finding a set of rules to win as many fights as possible, independently of the environment is a hard problem. It is not clear what approach the sniper should follow to win. This paper presents a way for a sniper agent in a 3-dimensional discrete environment, to learn the best strategy to win fights. A strategy, in our case, is simply a way to prioritize certain features of the environment over others. The strategy is implicitly defined by the evaluation function that shows how much an agent values each bloc in the 3D playground. The goal for the sniper is simply to learn a winning evaluation function for the given environment.

In this study, we consider two types of fights. The first one is a static sniper fight. The two agents fighting are not allowed to move. The second type of fight considers two agents and allow them to run over the map according to the gradient of their evaluation function periodically shooting at each other. For these two types of fight, we want to find the parameters that will maximize the number of wins.

2 Different approaches to solve the problem

2.1 Evaluation function optimized through a genetic algorithm

Another approach that was considered, is to model a sniper as a set of binary parameters encoded in such a way that the number of bits for one parameter describes how important that parameter (the number of bits works like a weight for each parameter). After the encoding is done, the approach is to use those binary weights to generate an evaluation function. At every generation, breed the snipers with the more successful evaluation functions discarding the unsuccessful ones. This method comes from the work of William Tunstall-Pedoe for chess algorithms [1]. Despite the forwardness of this approach, it did not quite fit our problem for multiple reasons.

The first reason is that to compare different evaluation functions, one needs an example of what the correct evaluation should be like. If two Ais are playing chess for example, and one move a pawn to A4 the other AI moves its pawn to G6, we need a way to assess the different moves so that the AI's evaluation functions can be corrected or rewarded. In his paper, Dr. Pedoe had access to a database of 400 Grandmaster's replays and was ranking the evaluation functions based on how many times they were reproducing world Grandmaster's moves. In our case, we do not know what a good move is for the sniper. We do not have access to tons of replays of people playing sniper games hence the approach of this paper would not work.

2.2 Self optimizing evaluation function

As mentioned in the previous section, learning the right evaluation function implies that the agent has some ways to know beyond his own assessment of the environment, what is a good move and what is not. Because of that, the idea of a self-optimizing evaluation function seems like a plausible solution. By self-optimizing, it is understood that the evaluation function will update itself based on some heuristic to maximize or minimize some values. This is assuming that

the heuristic is generic enough that it will work for all the cases.

In the paper "Self-optimizing evaluation function for Chinese-chess"[2], the author uses an algorithm called Particle Swarm Optimization to update the candidate solution to the problem. Because the approach does not require any pre knowledge of what a good solution is and update the candidate solution until some convergence is reached, it was a good fit for the problem of maximizing a sniper's strategy.

3 Current approach using Particle Swarm Optimization

3.1 Presentation of the idea and more background on our approach

In this paper, we solve the problem of learning the right evaluation function for the sniper by using a modified version of the particle swarm optimization algorithm. The pseudocode was modified to account for things specific to the context of a sniper in a 3D environment.

The particle Swarm optimization algorithm (PSO) inspires itself from natural phenomenon like birds searching for food in a cornfield [3].

1.Algorithm (w, φ_p, φ_g)
2.for each particle $i = 1, ..., S$ do
3.Initialize the particle's position with a uniform distribution random vector: $x_i \sim U(b_{lo}, b_{up})$
4.Initialize the particle's best-known position to its initial position: $p_i \leftarrow x_i$
5.if $f(p_i) < f(g)$ then
6. update the swarm's best-known position: $g \leftarrow p_i$
7.Initialize the particle's velocity: $v_i \sim U(-|b_{up}-b_{lo}|, |b_{up}-b_{lo}|)$
8.while a termination criterion is not met do:
9. For each particle $i = 1, ..., S$ do
10. For each dimension $d = 1, ..., n$ do
11. Pick random numbers: $r_p, r_g \sim U(0,1)$
12. Update the velocity: $v_{i,d} \leftarrow w\, v_{i,d} + \varphi_p\, r_p\, (p_{i,d}-x_{i,d}) + \varphi_g\, r_g\, (g_d-x_{i,d})$
13. Update the particle's position: $x_i \leftarrow x_i + v_i$
14. if $f(x_i) < f(p_i)$ then
15. Update the particle's best-known position: $p_i \leftarrow x_i$
16. if $f(p_i) < f(g)$ then
17. Update the swarm's best-known position: $g \leftarrow p_i$

The algorithm takes in 3 parameters. There is w that represents the inertia of the swarm (how much we care about having all particles close together during the search), φ_p which describes how much the particles should care about the local maximum and φ_g which describes how much the particles care about the global maximum during the search (our implementation has w=1, $\varphi_p = \varphi_g = 1.5$).The algorithm starts by initializing all the particles of the swarm and their best-known maximums to themselves. It then goes through the list of particles and sets the particle with the maximum altitude as the best-known particle for the swarm.

It finally initializes the velocities of each particle (which describes the way in which they will move through that high-dimensional space during the search). If the best-known position is not converging yet, the velocity vector is updated according to the heuristic in the pseudocode as well as the particle's values (line .11, .12).

After the update, the algorithm checks if any of the local maximums needs to change or if the global maximum known needs to change. The algorithm extrapolates truths about those biological searches and applies them in the domain of function optimization (still a search problem).

In this study, PSO provided a way to describe an algorithm that finds the best strategy for a sniper to win more games in a randomly generated 3D environment. In addition to the particles in the swarm, there are groups of static agents that are not allowed to evolve and are used to measure the overall improvement of the swarm after each iteration of the PSO. Snipers of the swarm are not fighting each other because it could have been hard to measure progress since they would all be evolving at the same time according to the red queen hypothesis [4]. The agent gets assigned an altitude by fighting all the static snipers and gets modified afterwards accordingly to the algorithm's internal heuristic (line .12, .13).

Beyond the obvious, our modified version allows a set of vectors in higher dimensions that represent snipers, to be iteratively modified in such a way that they will maximize some n-dimensional scalar field f. This scalar field f maps snipers to their chances of winning a fight. It essentially searches for a global maximum in a stochastic manner (line .11) without actively computing a gradient. The diagram below summarizes well the approach of this paper. Let us designate as "squad" a group of different snipers, also assume snipers and fighters are inter-changeable.

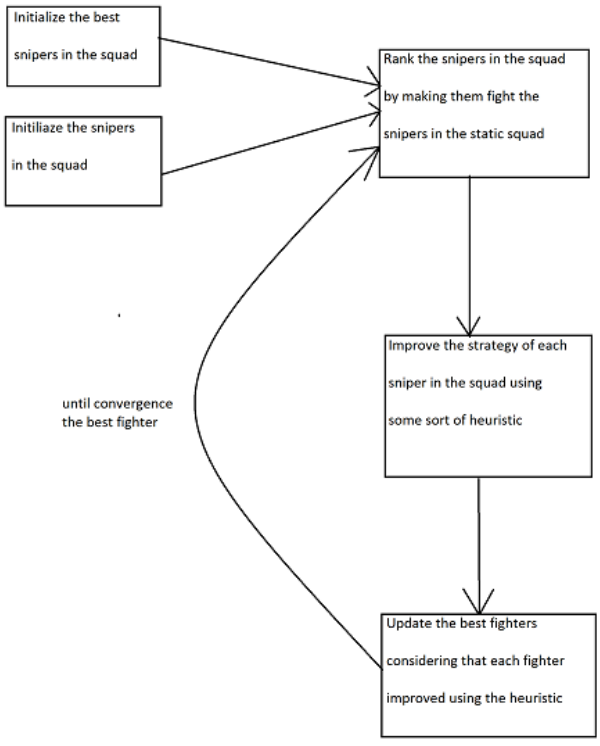

Figure .1

More formally, given a static environment, and a set of snipers that are represented by 4-dimentional vectors, we want to maximize the function f:

$$f: \mathbb{R}^4 \rightarrow \mathbb{R} \quad (1)$$

$$f: < u_0, u_1, u_2, u_3 > \mapsto v \quad (2)$$

With *v* being the altitude or score given to the agent,

$$v = \frac{1}{|S|} \sum_{i=0}^{|S|} \chi(U, i)$$

Where $\chi(U, i)$ is defined below:

$$\chi: \mathbb{R}^5 \to \{0,1\}$$

$$\chi(U, i) = \begin{cases} 1, & \text{U beats static sniper } S_i \\ 0, & \text{otherwise} \end{cases}$$

U $=< u_i >_{i=0,1,2,3}$ being the set of parameters identifying agents and over which f will be optimized and S the set of all static snipers. Each sniper has an evaluation function, that describes the current strategy (encoded through the parameters).

Let g be the evaluation function for the agents above. Let $U=< u_i >_{i=0,1,2,3}$ be a sniper and $W=< w_i >_{i=0,1,2,3}$ a bloc of the environment described by a 4-dimensional vector.

$$g: \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R} \quad (3)$$

$$g : (U, W) \mapsto \text{y} = UW^T \quad (5)$$

With $U = < u_0, u_1, u_2, u_3 >$ and $W = < w_0, w_1, w_2, w_3 >$, we get the expression below

$$y = \sum_{i=0}^{n} u_i w_i \ (6)$$

y describes how much the agent *U* values bloc *W*. Since the parameters *U* varies from agents to agents this measure is subjective and probably not an accurate evaluation given the goal of winning more fights.

3.2 Vectorization of the agents and blocs

3.2.1 Mapping each bloc to a 4-dimensional vector

For the current task, it was necessary to have a way to access different attributes of a bloc. One could have used the Cartesians coordinates of the blocs for example and do the optimization on top of it. However, considering how specific our task was, and how much was known about what snipers care about, it would have been bad not to take advantage of sniper-specific features relatively easy to compute. Instead of using the coordinates of the blocs, we used 4 features that are extracted from each bloc in the 3D environment.

a. Area Cover

This is a measure of how much of the visible world the agent can see from his spot. Let σ_c be the area cover. If n is the number of blocs that are in the agent's line of sight and N, the number of cubes that one agent can walk on, we get σ_c the following way:

$$\sigma_c = \frac{n}{N}$$

The number of walkable cubes is read from a variable and the number of cubes in line of sight is found by doing a circular ray cast and counting the number of hits. The bloc highlighted in orange below has a decent area cover for example.

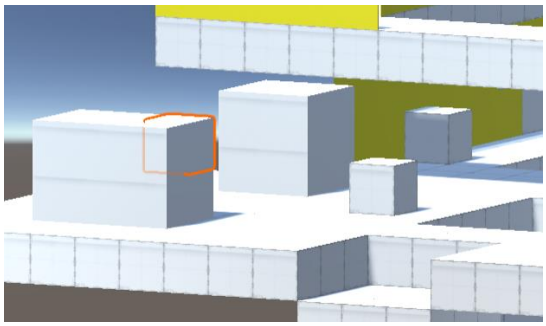

Figure .2

b. Average height above line of sight

Let $h_\mu$the average height above line of sight. It is a measure of how high you are compared to the blocs that you can see. For a better formulation, let us define the scalar field h_{z_0}, as following:

$$h_{z_0}: \mathbb{R}^3 \to \mathbb{R} \ (7)$$

$$h_{z_0} :< x, y, z > \mapsto z - z_0 \ (8)$$

with z_0 the height of reference for the given environment.
Let us also define h_{max} which is the maximum height among the blocs that you can see. We can find h_μ the following way:

$$h_\mu(p_i) = \frac{1}{h_{max} \bar{h}_{z_0}} h_{z_0}(p_i) \ (9)$$

with $\bar{h}_{z_0}$ the average value of the function h_{z_0} (which is technically the average height among the blocs you can see) and $p_i = < x_i, y_i, z_i >$ describing the sniper's position. The average value $\bar{h}_{z_0}$ is computed as following:

$$\bar{h}_{z_0} = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} h_{z_0}(\omega) \ (10)$$

With Ω the set of blocs you can see and $h_{z_0}(\omega)$ the height of bloc ω (ω representing a bloc's coordinate). A good average height above the line of sight tends to also mean a good area covered. It would make sense because the higher the sniper is, the more blocs of the terrain he would be able to see. The bloc highlighted in orange below has a very good average height above the line of sight for example.

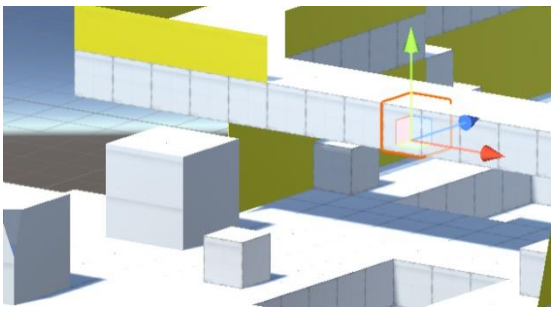

Figure .3

This sniper also displays a very good area covered as it can see a lot of the walkable cubes.

c. Average Cover

It is a measure of how much of the enemies you can see, cannot see you. Let μ_c be the average cover. Assuming the average visibility of the cubes from the line of sight μ_v has been calculated and normalized, the average cover is found the following way:

$$\mu_c = 1 - \mu_v \ (11)$$

With the normalized average visibility computed in the following way:

$$\mu_v = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} v(\omega) \ (12)$$

With Ω the set of all blocs in line of sight. And the scalar field v is defined as following:

$$v: \mathbb{R}^3 \rightarrow \mathbb{R} \ (13)$$

$$v :< x, y, z > \mapsto p \ (14)$$

With p the percentage of visibility from that bloc. It is a metric about defense compared to the two mentioned earlier. The orange block below is a good example displaying a good average cover.

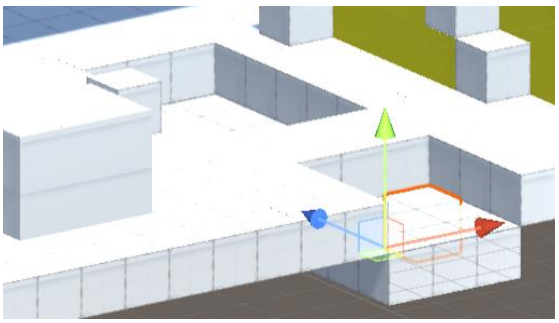

Figure .4

d. Area of cubes that cannot see the sniper

This is a metric of how much enemies cannot see you. Let $\sigma_{\bar{c}}$ be that feature. It is derived easily from the area cover:

$$\sigma_{\bar{c}} = 1 - \sigma_c \ (15)$$

This is another defensive feature. The spot in orange below is an example of cube that most other cubes in the environment cannot see.

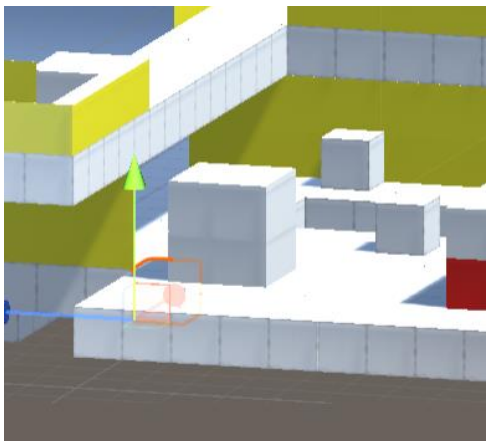

Figure .5

e. Summarization of the vectorization of a cube

Every cube in the universe is mapped to the following vector $<\sigma_c, h_\mu, \mu_c, \sigma_{\bar{c}}>$ where each component refers to one of the feature mentioned earlier.

3.2.2 Mapping each sniper to a 4-dimensional vector

Each sniper is identified by a 4-dimensional vector that encodes its strategy. The 4 weight values encode how much importance the sniper gives to the different parameters we

used to vectorize the cube (first the average cover, second the area cover etc.) Once this is encoded, the evaluation function can map each bloc of the environment to a score describing how much the sniper will like a specific spot by using (6).

4. Results and visualizations

4.1 Assumptions and games

Simulations were run in Unity using a 3D randomly generated environment. This environment is recycled for the rest of the simulations. Since the simulations were run using Unity, the scripts and all the code is written in C#.

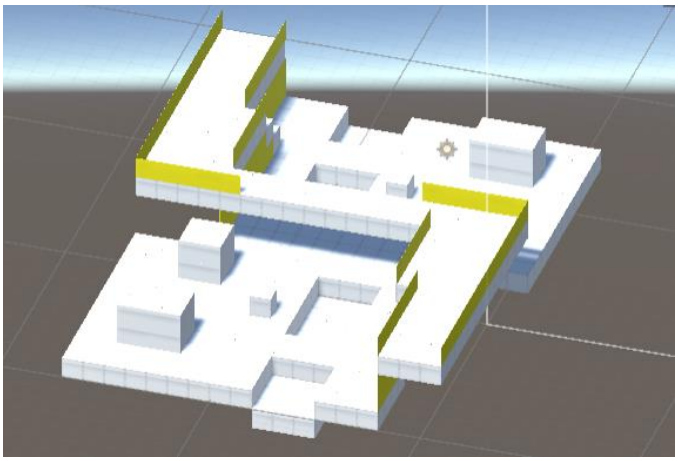

Figure .6

To run those simulations successfully and in a relatively small amount of time, the algorithm made assumptions about the snipers and the fights. The first assumption is that snipers do not have an orientation (when checking for a shot, the sniper shoots around instead of just in front of him). The second assumption is that after one hit, the sniper dies. The game does not allow for recovery or number of lives. After one hit, the sniper dies and the fight ends. On top of these two assumptions, depending on the format of the fights we had additional specifications.

For static fights, we do not allow the agents to move on the map. For this type of game, the agent must pick a spot and remain there for the entirety of the fight. If there are no hit, we call it a draw otherwise we give one point to the winning sniper.

For dynamic fights, the sniper can move on the map following the best bloc adjacent to the bloc where the sniper stands. There are no loop check, however if after 50 iterations nobody died, we stop the game and call it a draw. The dynamic fights do not account for the other players' s position. The sniper simply optimizes its evaluation function.

4.2 Static fights

4.2.1 Observations

The PSO implementation for static fights takes in average 1 to 2 minutes to run. For the static fights, after running the PSO, the algorithm sorts the walkable blocs in decreasing order based on how much the sniper values them and then outputs the coordinates of the best bloc in the terrain according for the winner sniper. For all these runs we consider 100 snipers in the swarm and around 10 in the static sniper pool.

Looking at blocs instead of weights is a simple and useful reduction because snipers are only characterized by their weights vector or in other words, their evaluation function, and for the static fights, the only thing that matters is where they will be standing on the 3D playground.
The algorithm for static games has the number of iterations of PSO as a parameter. It allows us to look at the winning agent (truly the winning bloc) at different point in the search process. Below are some data points for winning positions after 20, 500 iterations that were coming back very frequently.

4.2.1.1 PSO up to n=20 iterations

Bloc (-7, -1, -8)

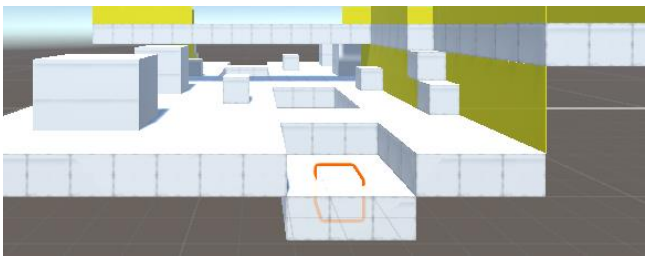

Figure 6.

Bloc (-6, -1, -7)

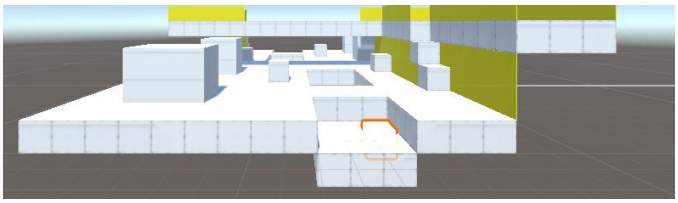

Figure 7.

Bloc (-11, -1, -12)

Figure 8.

4.2.1.2 PSO up to n=500 iterations

Below are some data points for winning positions after 500 iterations

Bloc (-11, -1, 12)

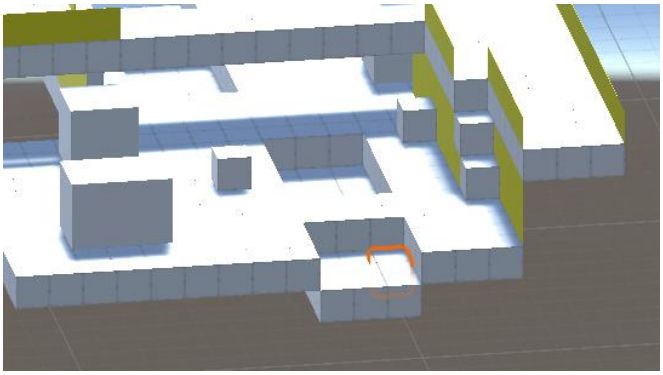

Figure 9.

Bloc (-6, -1, -7)

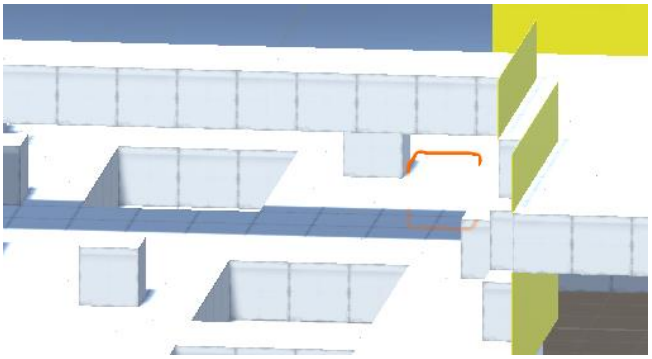

Figure 10.

Bloc (-7 -1,-7)

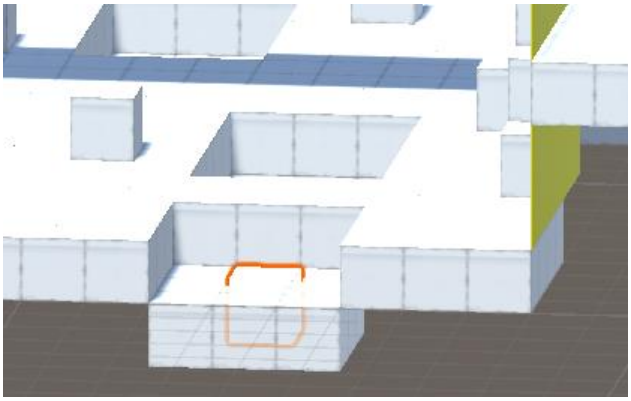

Figure 11.

Bloc (-10, -1, -12)

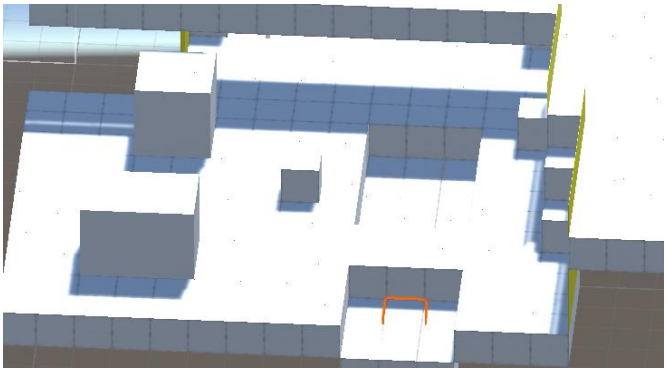

Figure 12.

Bloc (-13, 3, 2)

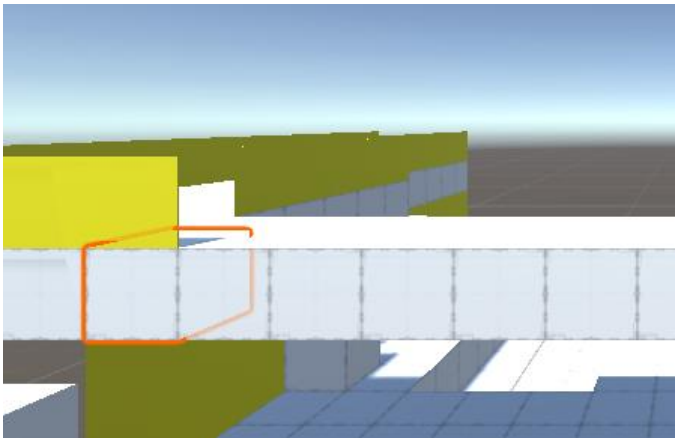

Figure 13.

4.2.2 PSO up to n=1000 iterations

Bloc (-2,3,-9)

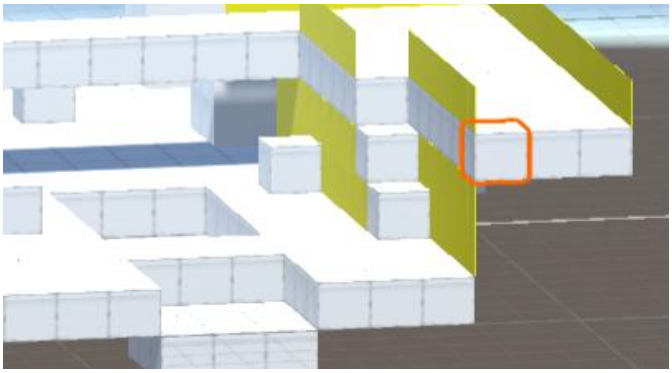

Figure 14.

Bloc (-11,-1,12)

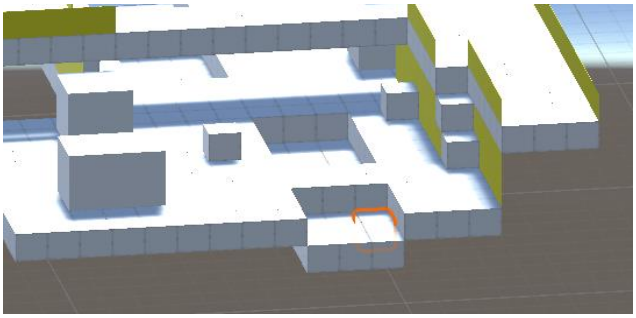

Figure 15.

4.2.3 Interpretations

After running the game multiple times and recording the winning blocs, a pattern becomes more and more obvious. Most of the blocs that were picked have a lower height than the blocs in their surrounding and have some cover from bottom to half of the body of the sniper at least.
Whether it is when n=20, 500 or 1000 iterations, we most if not all the winning blocs were all half buried on one or the other corner of the map. Below are a couple of examples:

Bloc (-6, -1, -7) (n=20 iterations)

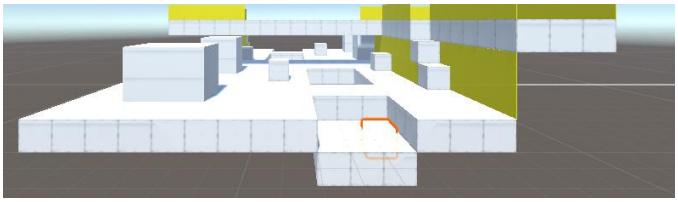

Figure 16.

Bloc (-11, -1, -12) (n=20 iterations, 500 iterations, 1000 iterations)

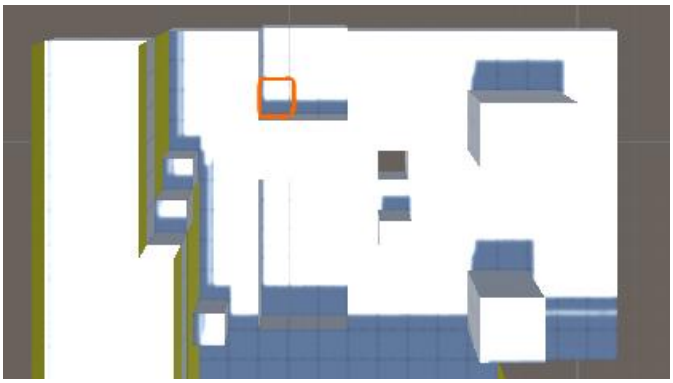

Figure 17.

Bloc (-7, -1, -8) (n=50)

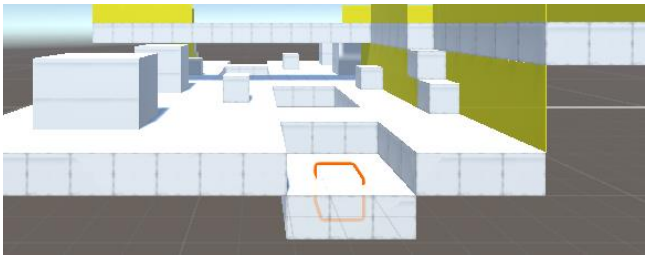

Figure 18.

It is obvious, looking at these pictures that the winning snipers have the same preferences. The physical position of the bloc does not necessarily matter. Earlier we have discussed how we could map every 3-dimensional bloc to a 4-dimensional bloc that we then use for the optimization. The claim here is that even though the position might differ, the 4-dimentional representation of the winning blocs is converging to one specific vector. The similarities between all the winning blocs is leading us to make this conjecture.

Most of those blocs are trenches, that meaning, they are usually buried 1-unit into ground which provides a good average cover. They are at the corners of the map most of the time and open on the entire map, which provides a great area cover and increase the area of blocs that cannot see you.

The surprise here is that the average height above the line of sight is not very big. The human intuition is to think of snipers as doing better in higher places (which we do have 2 winning blocs that turn out to be very high (see Figure 13., Figure 14.)) however it is clear that this is not the trend on this map. The best strategy here is to go for trenches, not for highly elevated portions of the map.

4.3 Dynamic fights

4.3.1 Observations

Dynamic fights are very similar to static fights expect that we allow the agent to move around following the direction of maximum increase in the evaluation function when looking at all the adjacent blocs.
Instead of outputting a bloc like in static fight, we show a heatmap of the evaluation function of the winning sniper.
For the heatmaps below, the number of iterations for PSO is 150, the number of static agents is 50 and 10 swarm members.
Below are a couple of heatmaps with their winning vector sniper following the legend about the coloring of the evaluation function scoring.

Figure 19. Legend

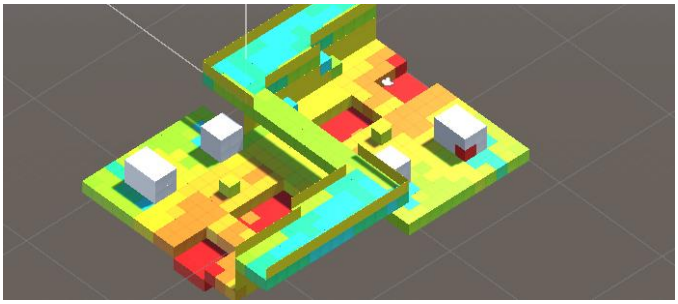
Figure 20.

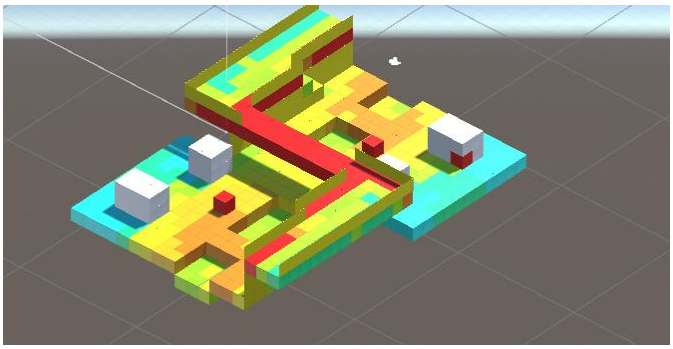
Figure 21.

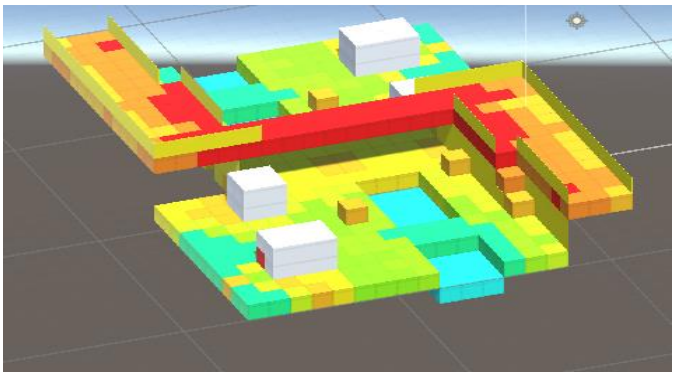
Figure 22.

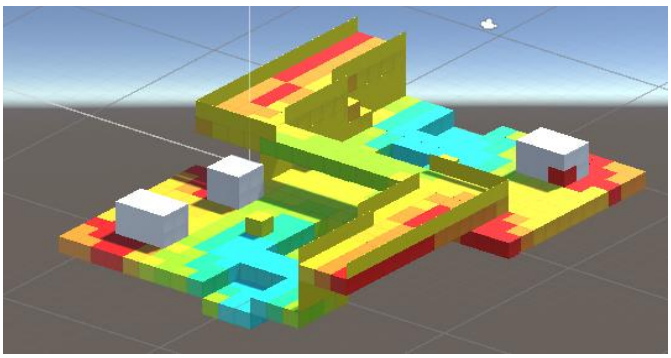
Figure 23.

4.3.2 Interpretations

It was clear, after running the algorithm many times, that there was no (or at least very little) pattern to observe and almost nothing consistent in the heatmaps. Different winning snipers had radically different strategies which was way different from what we were observing in the earlier game (static games).

This might be because we randomly start the sniper somewhere on the map. In fact, the sniper does not choose its best spot just like earlier, here the sniper is randomly assigned a bloc in the universe of walkable blocs and start moving from there. In addition to that, it has been conjectured earlier that, the place where the sniper starts matters in his success in fights. Consequently, it is possible that very good snipers got "killed" earlier in the process simply because they had bad luck.

The heatmaps that we observe might be the ones of the best snipers known for that run, but it is also equally likely that they represent snipers that were lucky enough to be spawned at good spots and made it to the end.

4.4 Conclusion and Future work

The future will focus mostly on improving the simulation of dynamic fights. The first problem to solve will be to make sure that good snipers do not die at the very start of the process. Because of the randomness of the dynamic version of our algorithm, it was hypothesized earlier that the snipers that we observe may not be the best ones because the very good ones may have died because they spawned at very bad spots. It was conjectured based on the work on the static snipers that the spots where snipers stand matter in their success.
The second problem to solve is the absence of ways to account for the motion of the other player. The displacement of the agents, as of now, is solely based on moving to the highest ranked adjacent bloc. Even if his enemy was behind him, the sniper would have blindly followed the gradient of the evaluation function to a new bloc and would have taken a shot even if the chances of getting a hit were higher from the original bloc where he was standing,

References

[1] William Tunstall-Pedoe. 1991. Genetic algorithms optimizing evaluation functions: Semantic scholar. (January 1991). Retrieved December 14, 2022, from https://www.semanticscholar.org/paper/Genetic-Algorithms-Optimizing-Evaluation-Functions-Tunstall-Pedoe/a03c457be586d0b1dd3eeb8b87a26b02dff086e7
J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] Xiang-Ran Du, Min Zahang, and Xizhao Wang. 1970. [PDF] self-optimizing evaluation function for Chinese-chess: Semantic scholar. (January 1970). Retrieved December 14, 2022, from https://www.semanticscholar.org/paper/Self-Optimizing-Evaluation-Function-for-Du Zahang/b7dab7636024e4004ad3dae66c07c4cd4d6c8279

[3] K J. Kennedy and R. Eberhart. Particle swarm optimization. Proceedings of ICNN'95 - International Conference on Neural Networks. DOI:http://dx.doi.org/10.1109/icnn.1995.488968.

[4] Anon. Red queen hypothesis. SpringerReference. DOI:http://dx.doi.org/10.1007/springerreference_142753