Performance Analysis in Implementation of a Dodgeball Agent for Video Games

Bianca-Cerasela-Zelia Blaga, Dorian Gorgan

Technical University of Cluj-Napoca
Computer Science Department
Cluj-Napoca, Romania
E-mail: zelia.blaga@cs.utcluj.ro, dorian.gorgan@cs.utcluj.ro

Abstract. Over the last few years, there has been an amazing growth in the video game industry, as great improvements have been made to the fields of graphics and engines. The development of artificial intelligence has enabled developers to create complex non-player characters (NPCs) that enhance the virtual worlds. In this paper, we present the implementation of a dodgeball agent using a recent toolkit called Unity Machine Learning Agents. We explore the capabilities of this technology and test its performance, while analyzing the ways in which it can be used to enrich a computer game.

Keywords: Artificial Intelligence, machine learning, non-player characters, video games.

1. Introduction

The video game industry has seen a shift from being exclusive to children to being part of the commercial industry. This domain has suffered changes in the game business and has brought innovations in the hardware for playing games, in interaction devices, in the available software tools, and in designing the products, as Overmars (2012) states in his book "A Brief History of Computer Games".

Gaming has a variety of applications in fields such as education, medicine, or military. Minecraft (Duncan, 2011) and Portal (Shute, 2015) are suggested by Smith (2010) as platforms for teachers to use in their classroom. Physicians can benefit from digital games by learning faster to accurately recognize trauma in patients, thus increasing their competency level, as presented by Newman (2002). Popular examples of games that are dedicated to the medical field are Prognosis (Medical Joyworks LLC, 2019), Medical School (Kongregate, 2011) and Microbe Invader (Tao, 2013). It has been a long time since the military started using games such as Janus, Simnet, or Spearhead, for training, tactics analysis, and mission preparation, work detailed in a report by Smith (2010). Moreover, video games can help

astronauts combat isolation in space, among other methods presented by Kelly & Kanas (1994).

Nowadays, artificial intelligence is added to video games in order to make them more sophisticated. Julian Togelius, an associate professor at New York's University's department of computer science and engineering who specializes in the intersection of video games and AI has summarized the recent directions in this field: "Typically when you design the game, you want to design an experience for the player. You want to know what the player will experience when he gets to that point in the game. And for that, if you're going to put an AI there, you want the AI to be predictable. Now if you had deep neural networks and evolutionary computation in there, it might come up with something you had never expected. And that is a problem for a designer." (The Verge, 2019). Such an example of subtle AI is given by Spittle (2011) and it is encountered in the game FEAR (First Encounter Assault Reconnaissance), where enemies talk about the actions they take, which are actually given by the path planning algorithms that control them. Players look at this as a realistic scenario, which proves how important the psychological factor is in perceiving the virtual world.

Shaker et al. (2010) consider that the level of excitement and enjoyability a user experiences while playing games is given by the various events encountered during gameplay, by the behaviour of characters, or by the interaction with game elements. In order to obtain a complex virtual world, researchers are studying new ways to generate content. These can range from procedural environment generation to different toolkits like DeepMind Control Suite (Tassa et al., 2018), OpenAI Gym (Brockman et al., 2016), or Unity Machine Learning Agents (Johansen et al., 2019).

In this paper, we take a look at a game engine which provides a toolkit for creating intelligent agents. We design a dodgeball non-player character (NPC) that can be integrated to give more dynamics to a digital game. In Section 2, we take a look at the characteristics of NPCs, and the tools that different game engines provide to create agents using artificial intelligence. Next, in Section 3, we present some particularities of Unity ML-Agents toolkit, which is used for the implementation of a dodgeball agent. In Section 4, we analyze the performance of the NPC, and we draw the conclusions of this work in Section 5, highlighting possible future research ideas.

2. Related work

Non-Player Characters (NPCs) are defined by Warpefelt (2016) as: "characters within a computer game that are controlled by the computer, rather than the player". They can be represented, for example, by monsters, vehicles, animals, or plants, and can play different roles. Propp (2010) gives a list of action spheres for traditional folktales prototypical characters:

- *villain* – the opponent of the hero,
- *donor* – provides the hero with a magic agent (possibly as a reward),
- *helper* – transports, rescues, or assists the hero,
- *princess* – the person the hero is trying to "acquire" through marriage or something similar,
- *the princess's father* – gatekeeper for the marriage, provides a quest,
- *dispatcher* – sends the hero on a quest,
- *hero* – can be either a seeker (goes on a quest or completes a task to fulfill the requirements of the donor and/or the father. Marries the princess.) or victim (as above, but does not go on a quest),
- *false hero* – tries to steal the glory from the hero.

Additionally, NPCs can also play different roles, like providing services, guarding places, getting killed for loot, supplying background information (history, lore), or just making the place look busy as Bartle (2004) states. These typologies are present in many games, such as Mario Series (Togelius et al., 2009), Bioshock Infinite (Lizardi, 2014), Skyrim (Puente & Tosca, 2013), or Dota 2 (Drachen et al., 2014).

Sagredo-Olivenza et al. (2017) consider that machine learning represents a method to avoid conditional logic for NPCs. It enables the creation of complex worlds that mimic the real environment for a more immersive gameplay. Algorithms are created that model behavior based on predictions that are made from past events to create actions in the future. Typically, this process consists of the following steps: gathering a dataset that is representative for the problem that must be learned, converting the dataset in numerical form, creating a model (neural network, logistic regression, random forests, etc.) and training it in order to minimize the loss function, inserting new data and training again until the best possible results are achieved, and collecting the predictions and inserting them in the game.

Unity provides a toolkit called ML-Agents (Johansen et al., 2019) that contains a framework for creating intelligent agents for computer games. One

example is represented by Marathon Environments (Booth, 2019), which is a field of research dedicated to video game researchers who are interested in applying bleeding edge robotics into the domain of locomotion and AI for digital gaming. It is based on previous work done by DeepMind Control Suite (Tassa et al., 2018) and OpenAI Gym (Brockman et al., 2016), and re-implements the classic set of Continuous Control benchmarks typically seen in Deep Reinforcement Learning literature. Marathon Environments was released alongside Unity ML-Agents v0.5 and includes four continuous control environments: Walker, Hopper, Humanoid, and Ant. All environments have a single ML-Agent brain, with continuous observations and actions.

Holden et al. (2017) present a neural network that generates new frames of animation in order to provide more realistic movements to avatars that look similar to humans. These learn to walk, climb, or even jump over obstacles, while maintaining their balance. Stere & Trăușan-Matu (2017) used artificial intelligence techniques to generate musical accompaniment, while Toma et al. (2017) created a game for vocabulary acquisition, proving the large variety of topics agents can cover.

A Recurrent Neural Network has been trained to play Mario Kart (Lei et al., 2019), by learning to predict what controller inputs a player would use in any given situation, rather than having the goal to win. This is a method that works well for games that can be represented by a time series, such as a racing, and for agents that get as input visual information. In Unreal Engine, Belle et al. (2019) explain that artificial intelligence can be created by using behavior trees. These are systems that determine which behavior an agent should perform given the scenario. For example, whether it should fight or run away depending on its health amount. Manzoor et al. (2018) present a digital game called Hysteria, which is a third person shooter game.

In the next sections, we will focus on Unity as the primary game engine, describe the tools that it provides to developers and implement our own agent. We have chosen this framework because it is new and there is little research in the literature showcasing various scenarios in which the agents can be used. Compared to other game engines, Unity requires less resources than Unreal Engine 4, as explained in a study by Nilsson (2019) that analyzes the battery consumption of a mobile phone when running video games developed with these tools.

3. Implementation of a dodgeball agent

The machine learning methodology that we use is reinforcement learning. It has the following particularities: the agent learns an optimal policy (also called behavior) by trial and error, it needs feedback on its action, and its actions affect the future state it receives. The tools used are: game engine Unity 2018.2.1, Unity ML-Agents 0.6.0, Anaconda 5.1.0, Python 3.6, and Tensorflow 1.4.0.

We have divided the project into the following elements:

- *world* – 2D canvas limited in four directions (up, down, left, right) by a border,
- *agent* – duck that senses its environment (gets feedback through sensors),
- *enemies* – volleyballs that have different movement rules,
- *goal* – the duck has to avoid collision with the borders and the balls for as long as possible,
- *reward* – for each time step, the score increases,
- *punishments* – if collision happens, the score decreases and the training is reset.

These components can be see in Figure 1, where the duck agent is placed in the middle of the game space, between the 4 borders of the playing area, and the enemies are spawned from 3 different locations on the right border. The ratio between the game area, the duck and the volleyball sizes is 650:4:1.

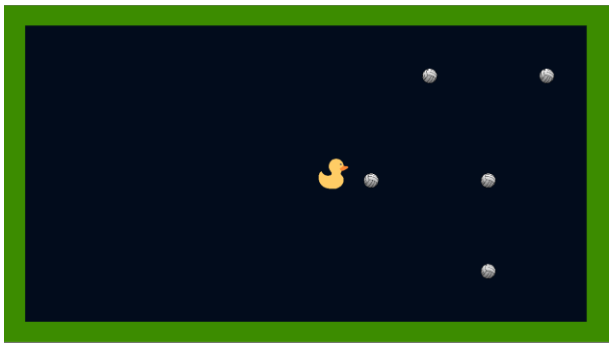


Figure 1. Simple game setup – the duck agent starts at the center of the screen, while the balls spawn at 3 locations on the right border.

3.1. Unity ML-Agents toolkit

Unity Machine Learning Agents (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. The learning environment, also seen in Figure 2, is composed of an Academy which is coordinated by a Python script written in Tensorflow – a framework for training deep learning models, including agents.

An academy is composed of one or more brains. The brain encapsulates the decision-making process, and controls the agents, which play two roles in this model. They are used both for observation of the environment, and they perform actions that bring the agent closer to the desired goal.

Every agent must be assigned a brain, but you can use the same brain with more than one agent. There are four types of brains: external – decisions are made using the Python API, internal – decisions are made using an embedded pre-trained model, player – decisions are made using real input, and heuristic – decisions are made using hard-coded behavior.

There also exist multiple training modes: single-agent – single brain with single agent, simultaneous single-agent – single brain and multiple agents, adversarial self-play – two agents with a single brain and opposite rewards, and cooperative multi-agent – single/multiple brain and multiple agents and same rewards.

Some examples of training modes are reinforcement learning, curriculum

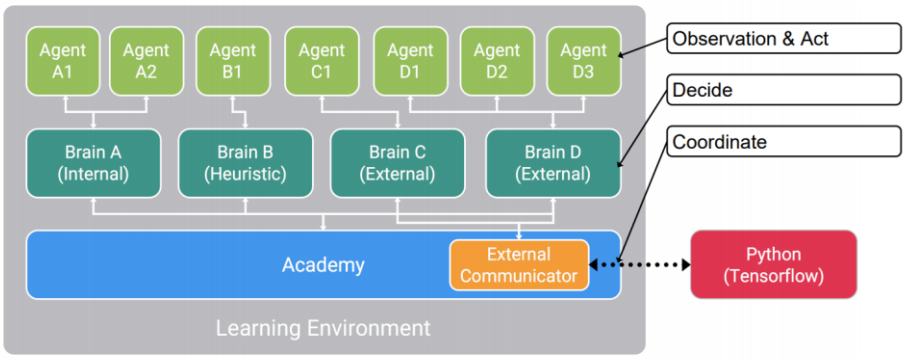


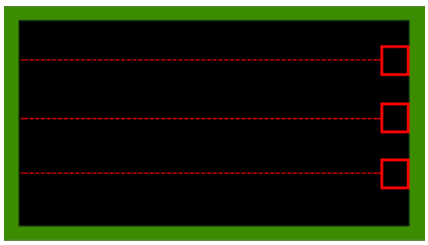
Figure 2. Unity ML-Agents framework components (Juliani, 2019)

learning, and imitation learning. Sutton & Barto (2018) consider that Reinforcement learning (RL) is a type of machine learning where agents learn how to behave in an environment to maximize their performance at a task. Thus, an agent interacts at each time step by receiving a state or observation of its environment and acting on it. It also gets feedback on its action by a reward. This type of learning can usually be modeled as a Markov Decision Process (MDP) or Partially Observable MDP (POMDP). They help define the relationships between the agent's observations, actions and reward or punishment functions. Furthermore, they can update the reward function and indicate when the agent has reached a terminal state. Observations are gathered from the environment through various sensors that compute, for example, the distance to objects, the intersection with a desired point, etc.

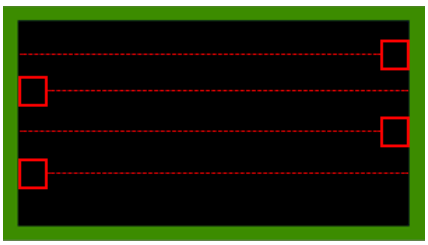
3.2. Project Creation and Specifications

After installing the necessary tools with their respective versions, we create a new Unity project. In `Project Settings`, go to `Edit – Project Settings – Player`, and in the `Inspector panel – Resolution and Presentation`, check the box `for Run in Background`. Additionally, in the `Inspector Panel – Other Settings`, add the tag `ENABLE_TENSORFLOW` in the `Scripting Define Symbols` area and set `the Scripting Runtime Version` to `.NET 4.6`. These are very important steps to run the training in the learning environment.

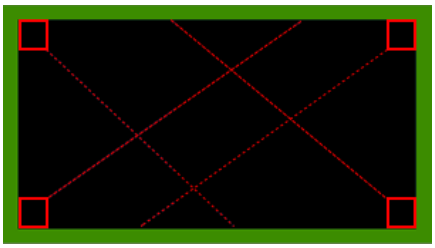
The next step is to download the ML-Agents and import the package in the Unity project. Inside it, make a new folder called `Dodgeball` and create a `2D Project Scene`, in which we create a `BallSpawner`, that has attached a script to it for randomly spawning volleyballs. Another important aspect is the spawn logic. There are two possibilities to generate the enemies. The first one is to have static points on the borders from where the balls are generated linearly. The second one would be to have random starting points on the borders, with collision logic added to the balls when hitting a planar surface. Thus, we would only spawn a limited number of such enemies, which would ignore collision with each other, as this would only complicate matters and it is not our objective. In total, we have 6 game scenarios, as shown in Figure 3.



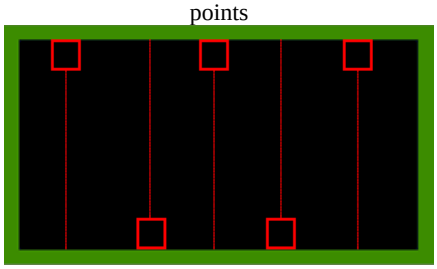
a) Scenario 1 – 3 right spawning points



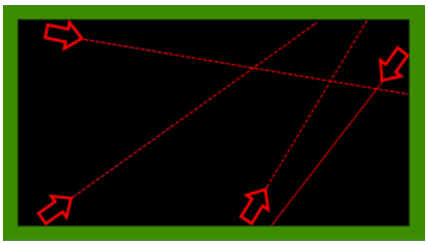
b) Scenario 2 – 2 left and 2 right spawning points



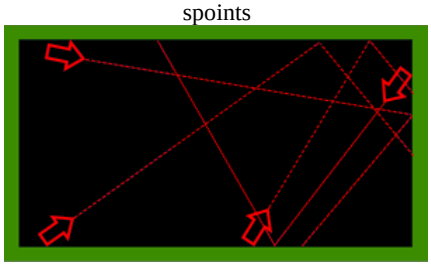
c) Scenario 3 – 4 corner spawning points



d) Scenario 4 – 3 top and 2 bottom spawning spoints



e) Scenario 5 – 4 random spawning points on edges, with no collision logic

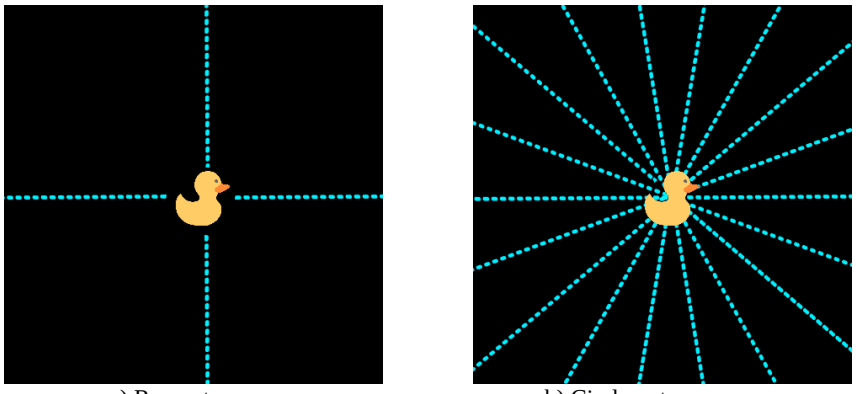


f) Scenario 6 – 4 random spawning points on edges, with collision logic

Figure 3. Spawning logic for the enemies (volleyballs)

For the academy part, which is attached to the duck object, we add logic in the `Brain` C# script. As inputs, we have the four borders, the force applied to our agent to get it to move, the agent's position, and a `Boolean` variable for the crashing logic. The state that we collect in the agent's brain is related to the distances to the objects from the scene, be it borders or balls. In the method `AgentStep`, we control the agent's movement in the scene. It takes as input the actions that come from the brain and performs according to them.

The bounce collision logic is modelled by attaching the correct materials to the borders and balls, and using a `RaycastHit2D` physics component. This takes into account the surface normal of the hit and reflects the ball.



a) Raycast sensor b) Circlecast sensor rays

Figure 4. The two types of sensors used in our game implementation

There are two ways for the agent to sense its environment (Figure 4):

Raycast – which is 4 directional (up, down, left and right),

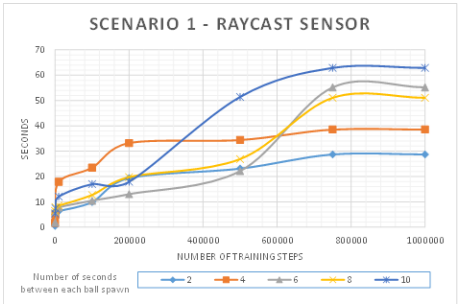
Circlecast – which is represented by rays that are at 20 degrees apart on a circle (18 rays in total).

After setting the `Brain Type` as `External`, we train the agent with different numbers of steps for each type of scenario and sensor. We first want to test how well can the agent dodge the balls when these are generated linearly, using first the Raycast sensor, then the Circlecast sensor. We expect the latter to perform better, given its larger number of sensing rays. Then, we want to see how many seconds can the agent survive in the environment, depending on how many balls are spawned. These scenarios are deterministic, as the generation logic does not change. Therefore, the last two scenarios aim to test the performance of the duck agent when the balls display a more complex logic, being spawned at random locations and even with collision properties.

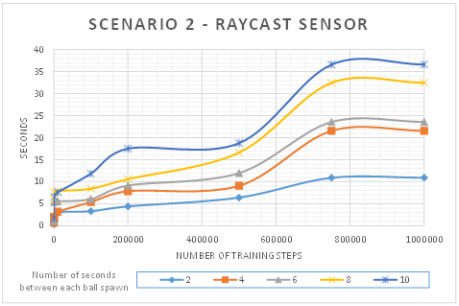
4. Results

In order to evaluate the performance of the agent, we have altered several parameters of the game components. We tried having different numbers of volleyballs in the environment, as well as various spawning points and collision logic. We also changed the number of training steps for the duck agent, to observe when an optimal performance is achieved.

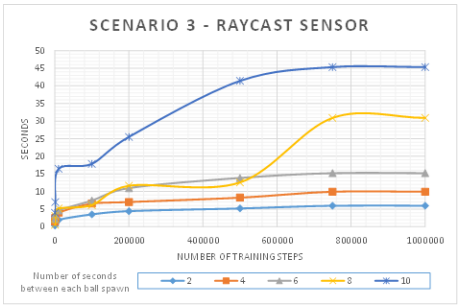
4.1. Results analysis for scenarios 1 – 5 with Raycast sensor



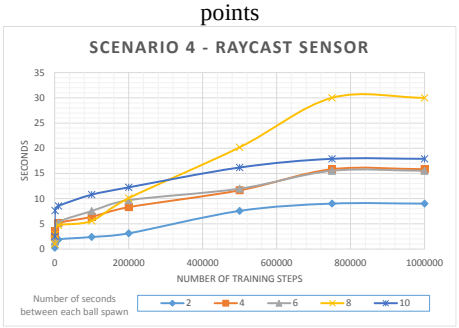
a) Scenario 1 – 3 right spawning points



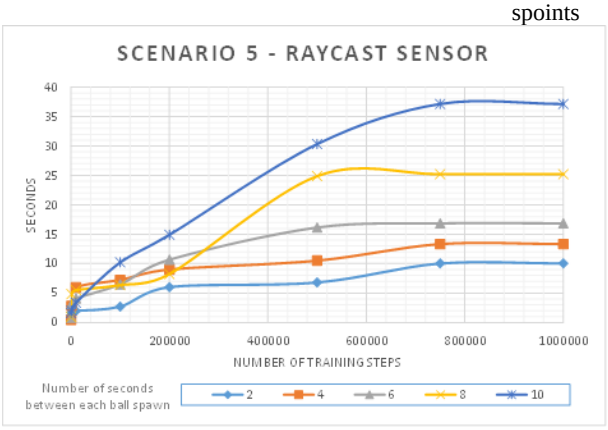
b) Scenario 2 – 2 left and 2 right spawning points



c) Scenario 3 – 4 corner spawning points



d) Scenario 4 – 3 top and 2 bottom spawning spoints



e) Scenario 5 – 4 random spawning points on edges, with no collision logic

Figure 5. Charts showing the runtimes in seconds for scenarios from 1 to 5, using Raycast sensor, with varying numbers of seconds between ball spawn (2 to 10)

From the charts presented in Figure 5, we can observe that in 4 out of 5 scenarios, the best performance is encountered when the number of seconds between the spawning of the balls is 10. Surprisingly, for Scenario 4, the agent performs better when the spawning time is 8 seconds, as it is able to better generalize a strategy in this case. When the balls are spawned almost continuously, the duck agent performs the worst, having a low survivability time.

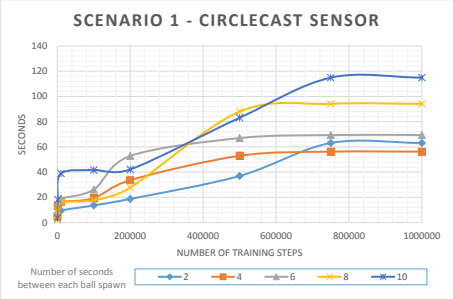
For scenarios 1, 3 and 5, we notice that the training stabilizes after 500 000 steps, while for the other two, this happens after 750 000 steps. Therefore we can say that it is sufficient to train the agent for half a million steps, as increasing this number will not bring much improvement in the surviving time in the game environment.

4.2. Results analysis for scenarios 1 – 5 with Circlecast sensor

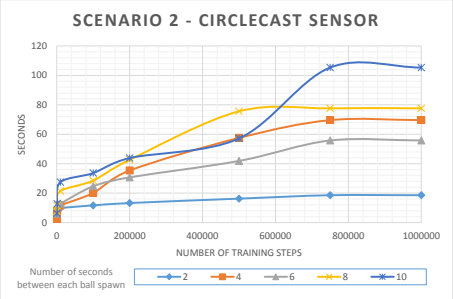
When the Circlecast sensor is used, we can notice from the graphs presented in Figure 6 that the agent is performing the best when the time between the generation of the enemies increases gradually. The best results are obtained when there is a 10 seconds delay, while the worst for 2 seconds delay. As the scenarios increase in difficulty, the agent can not improve its performance after a certain number of training epochs. Therefore, we notice that the performance stabilizes after 500 000 steps in the last 3 scenarios, while, for the first two, more training steps are necessary.

4.3. Results analysis for scenario 6 with Raycast and Circlecast sensors

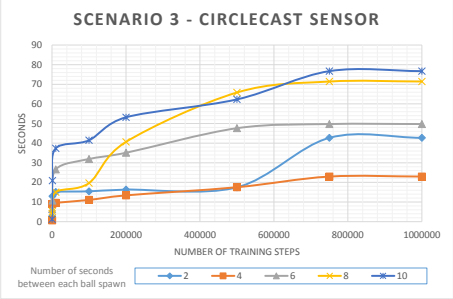
For the last use case, we analyze the performance of the duck agent for scenario 6, where the balls have collision properties and are spawned randomly. Comparing the charts from Figure 7 and Figure 8, we notice that the agent is able to survive more, regardless of the time it takes for balls to be spawned. On average, the survival time is tripled when using the Circlecast sensor as compared to the Raycast one. We can also see that in the first case, increasing the number of training steps brings only small improvements in the performance, while in the second case, training for more epochs shows some increase in the agent's performance.



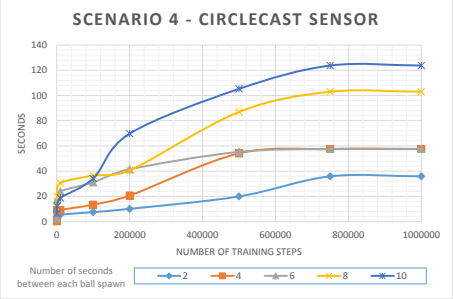
a) Scenario 1 – 3 right spawning points



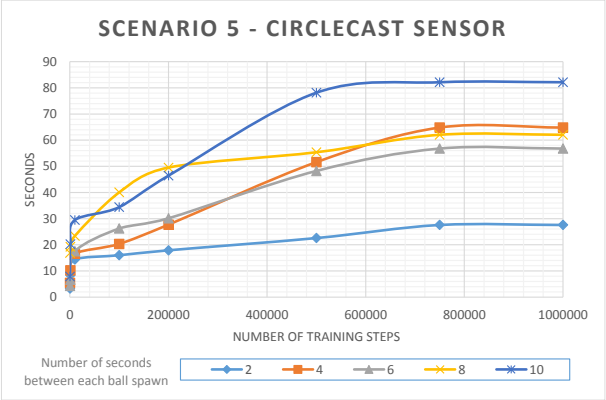
b) Scenario 2 – 2 left and 2 right spawning points



c) Scenario 3 – 4 corner spawning points



d) Scenario 4 – 3 top and 2 bottom spawning spoints



e) Scenario 5 – 4 random spawning points on edges, with no collision logic

Figure 6. Charts showing the runtimes in seconds for scenarios from 1 to 5, using Circlecast sensor, with varying numbers of seconds between ball spawn (2 to 10)

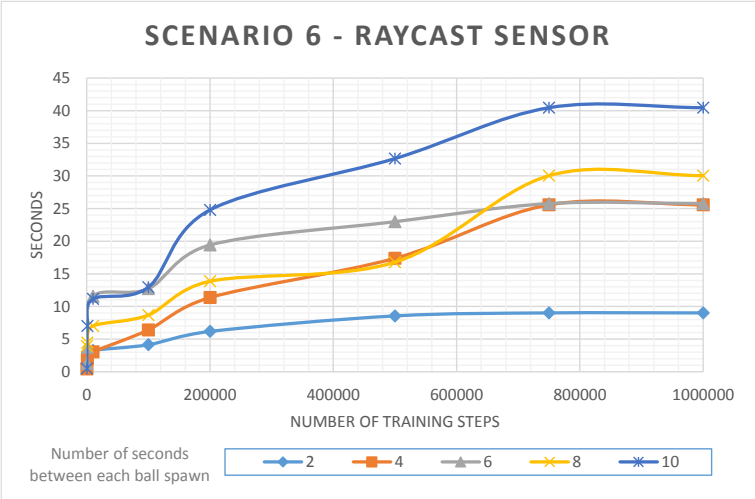


Figure 7. Chart showing the runtimes in seconds for scenario 6, using Raycast sensor, with varying numbers of seconds between ball spawn (2 to 10)

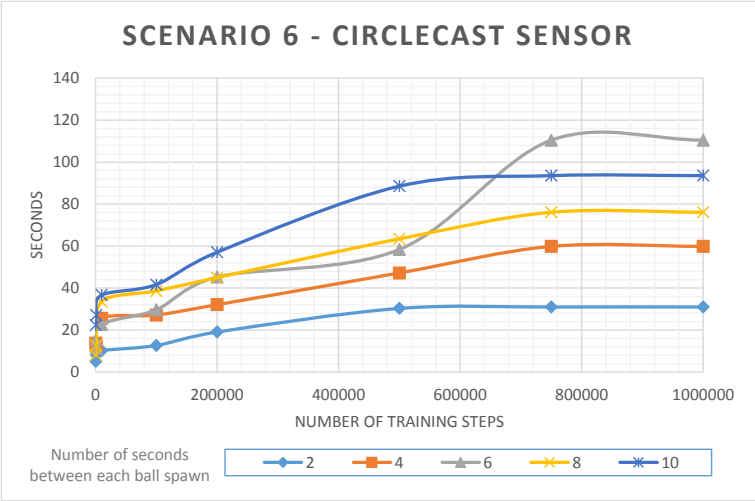


Figure 8. Chart showing the runtimes in seconds for scenario 6, using Circlecast sensor, with varying numbers of seconds between ball spawn (2 to 10)

4.4. Discussion

Table 1. Best survival times of the duck agent based on game scenario, sensor type and number of enemy volleyballs (S = seconds, R = Raycast, C = Circlecast)

S	Scenario 1		Scenario 2		Scenario 3		Scenario 4		Scenario 5		Scenario 6	
	R	C	R	C	R	C	R	C	R	C	R	C
2	28,72	63,05	10,89	18,56	5,96	42,71	9,01	35,89	10,03	27,57	9,01	30,99
4	38,54	56,21	21,55	69,60	9,95	22,98	15,83	57,56	13,33	***64,82***	25,57	59,86
6	***55,18***	69,37	23,57	55,76	15,22	49,74	15,52	57,63	16,85	56,79	25,77	**110,44**
8	51,07	***94,12***	***32,46***	***77,63***	***30,91***	***71,41***	**30,00**	***102,98***	***25,21***	62,05	***30,04***	76,10
10	**62,89**	**114,87**	**36,59**	**105,32**	**45,32**	**76,71**	***17,89***	**123,75**	**37,15**	**82,14**	**40,46**	***93,55***

In Table 1, we have summarized the best survival times of the duck agent in the 6 game scenarios, for various spawning times between the enemy volleyballs (from 2 to 10 seconds). We have highlighted the first and second best results, and we notice these are encountered for cases when $S = 10$ or $S = 8$ seconds, meaning that the agent can better adapt to a less aggressive type of environment. Exceptions occur for scenarios 1, 5 and 6, when sometimes this interval is not that important.

In conclusion, we appreciate that the duck agent obtained a satisfactory performance. We also notice that Unity ML-Agents is a very helpful tool for using artificial intelligence in video games, as it allows to easily tweak training parameters like the game configuration or number of training epochs. To the authors' knowledge, the work presented in this paper is not found in any previously published articles and therefore can serve as foundation for future research in the domain of artificial intelligence.

5. Conclusions

In this paper, we experimented with the toolkit Unity ML-Agents to implement a dodgeball agent represented by a duck that has to survive in an environment by avoiding volleyballs. We surveyed the literature to understand the recent advances in the domain, and we analyzed the results of our own experiment. The work we carried aimed to test the limits of artificial intelligence when applied to non-player characters, and we found out that the framework we used provides the necessary tools for any researcher looking to develop a large variety of dynamic game components. In the future, it would be interesting to add another dimension to the agent and make it 3D. Moreover, experimenting with other game engines would give us insight in other types of learning methods for a better cross-domain comparison.

References

Bartle, R.A., Designing virtual worlds. *New Riders*, 2004.

Belle, S., Gittens, C. and Graham, T.N., Programming with Affect: How Behaviour Trees and a Lightweight Cognitive Architecture Enable the Development of Non-Player Characters with Emotions. In 2019 *IEEE Games, Entertainment, Media Conference (GEM)* (pp. 1-8). 2019.

Booth, J. and Booth, J., Marathon environments: Multi-agent continuous control benchmarks in a modern video game engine. *arXiv preprint arXiv*:1902.09097. 2019.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., Openai gym. *arXiv preprint arXiv*:1606.01540. 2016.

Drachen, A., Yancey, M., Maguire, J., Chu, D., Wang, I.Y., Mahlmann, T., Schubert, M. and Klabajan, D., Skill-based differences in spatio-temporal team behaviour in defence of the ancients 2 (dota 2). In 2014 *IEEE Games Media Entertainment*, 2014.

Duncan, S.C., Minecraft, beyond construction and survival. *Well Played: a journal on video games, value and meaning*. 2011.

Holden, D., Komura, T. and Saito, J., Phase-functioned neural networks for character control. *ACM Transactions on Graphics*. 2017.

Johansen, M., Pichlmair, M. and Risi, S., Video Game Description Language Environment for Unity Machine Learning Agents. In 2019 *IEEE Conference on Games (CoG)* (pp. 1-8). 2019.

Juliani, A., *Unity ML-Agents: A flexible platform for Deep RL research*, available online: https://www.microsoft.com/en-us/research/uploads/prod/2018/03/FinalAJulianiMSR_2018.pdf, visited: 01.06.2019.

Kelly, A.D. and Kanas, N., Leisure time activities in space: a survey of astronauts and cosmonauts. *Acta Astronautica, 32(6)*, pp.451-457. 1994.

Kongregate, *Medical School*, 2011.

Lei, J., Chen, S. and Zheng, M., *Using Machine Learning to Play the Game Super Mario Kart*. 2019.

Lizardi, R., BioShock: Complex and alternate histories. *Game Studies*, 2014.

Manzoor, M.I., Kashif, M., Saeed, M.Y. and Campus, P., Applied Artificial Intelligence in 3D-game (HYSTERIA) using UNREAL ENGINE 4. *Applied Artificial Intelligence*. 2018.

Medical Joyworks LLC, *Prognosis: Your Diagnosis*, Google Play. 2019.

Newman, J., In search of the videogame player: the lives of Mario. *New media & society*. 2002.

Nilsson, R., *Battery Performance Comparison Of Unreal Engine 4 And Unity Applications Running On Android*. 2019.

Overmars, M., *A Brief Histry of Computer Games*. Department of Information and Computing Sciences, Faculty of Science, Utrecht University. 2012.

Propp, V., *Morphology of the Folktale* (Vol. 9). University of Texas Press. 2010.

Puente, H. and Tosca, S., The Social Dimension of Collective Storytelling in Skyrim. In *DiGRA Conference*. 2013.

Sagredo-Olivenza, I., Gómez-Martín, P.P., Gómez-Martín, M.A. and González-Calero, P.A. Combining neural networks for controlling non-player characters in games. In *International Work-Conference on Artificial Neural Networks* (pp. 694-705). Springer, Cham. 2017.

Shaker, N., Yannakakis, G. and Togelius, J., Towards automatic personalized content generation for platform games. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2010.

Shute, V.J., Ventura, M. and Ke, F., The power of play: The effects of Portal 2 and Lumosity on cognitive and noncognitive skills. *Computers & education*. 2015.

Smith, R., The long history of gaming in military training. *Simulation & Gaming*. 2010.

Spittle, S. Did This Game Scare You? Because it Sure as Hell Scared Me! FEAR, the Abject and the Uncanny. *Games and Culture, 6(4)*, pp.312-326. 2011.

Stere, C.C. and Trăuşan-Matu, Ş.. Generation of musical accompaniment for a poem, using artificial intelligence techniques. *Romanian Journal of Human-Computer Interaction*, 10(3), pp.250-270. 2017.

Sutton, R.S. and Barto, A.G., Reinforcement learning: An introduction. MIT press. 2018.

Tao, L., *Microbe Invader*. 2013.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D.D.L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A. and Lillicrap, T., Deepmind control suite. *arXiv preprint arXiv*:1801.00690. 2018.

The Verge, How artificial intelligence will revolutionize the way video games are developed and played, https://www.theverge.com/2019/3/6/18222203/video-game-ai-future-procedural-generation-deep-learning, visited: 01.06.2019.

Togelius, J., Karakovskiy, S., Koutník, J. and Schmidhuber, J., Super mario evolution. In 2009 *ieee symposium on computational intelligence and games* (pp. 156-161). 2009.

Toma, I., Alexandru, C.E., Dascalu, M., Dessus, P. and Trausan-Matu, S., 2017. Semantic taboo–a serious game for vocabulary acquisition. *Romanian Journal of Human-Computer Interaction*, 10(2), pp.147-162. 2017.

Unity Machine Learning Agents Toolkit, https://github.com/Unity-Technologies/ml-agents, visited: 01.06.2019.

Warpefelt, H., *The Non-Player Character: Exploring the believability of NPC presentation and behavior*. Doctoral dissertation, Department of Computer and Systems Sciences, Stockholm University. 2016.