“ALEXANDRU IOAN CUZA” UNIVERSITY OF IAȘI

FACULTY OF COMPUTER SCIENCE

BACHELOR THESIS

Application of K-means Clustering in Video Games

Proposed by

Andrei Ghiran

Session: 06.2020

Scientific coordinator

Prof.lect.dr. Moruz Alex

“ALEXANDRU IOAN CUZA” UNIVERSITY OF IAȘI

FACULTY OF COMPUTER SCIENCE

Application of K-means Clustering in Video Games

Andrei Ghiran

Session: 06.2020

Scientific coordinator

Prof.lect.dr. Moruz Alex

SUMMARY

INTRODUCTION

From the day I started my studies at the faculty of computer science I knew I wanted to pursue a career in game developing. To be one step closer to my career of choice I decided that for my bachelor thesis I would combine my passion for game developing and the limitless applications of machine learning. Using the Unity engine I developed a simple game that showcases the K-means algorithm by using it to train an Artificial Intelligence adversary for the player.

The game has a simple premise, the player controls a blue square that can only move on a grid in the 4 cardinal directions and must navigate a simple maze and reach the goal, the AI adversary, represented by a red square, with the same movement constraints as the player and only moving at the same time as him, must intercept and bump into them to end the game. The adversary doesn’t know the position of the player at all times, he has a vision radius in which he can detect the player, acquire his position and save it in a file for later use. The K-means algorithm uses the saved player locations to form clusters and determine an advantageous position from where the adversary can intercept the player, the using the A-star algorithm the adversary will find a path to this more advantageous position and follow it.

1. SIMILAR APPLICATIONS

Information on the use of machine learning techniques in video games is mostly known known publicly through research projects as most gaming companies don't publish specific information about their intellectual property. The most common use of clustering algorithms in the domain of video games is the classification of player behaviour in massive multiplayer online games (MMOs). These behaviours range from completing missions to exploring the game world and even spending time with other payers. The classification is used by the developers to understand what most players enjoy doing the game and this knowledge influences the further development of the game. K-means clustering is also commonly used for procedural content generation.

The most popular application of machine learning in games is the use of deep learning agents that compete with professional human players. Another popular type of agent is the computer vision-based AI player. The most significant application of machine learning has been done on games such as Dota 2, the StarCraft series, Pong and Doom. Games that did not originate as video games such as Chess and Go have also been subject to machine learning research.

1.1. Deep Learning Agents

Deep learning focuses heavily on the use of artificial neural networks that learn to solve complex tasks. Deep learning uses multiple layers of artificial neural networks and other techniques to progressively extract information from an input. Due to this complex layered approach, deep learning models often require powerful machines to train and run on.

Chess is considered a difficult AI problem due to the computational complexity of its board space, its state space being 10^120 possible board states. Similar strategy games are often solved with some form of a Minimax Tree Search. These AI agents have been able to beat professional human players, such as the historic 1997 Deep Blue versus Garry Kasparov match. Since then, machine learning agents have only shown even greater success.

Go poses an even more difficult AI problem then chess. The state space of Go is approximately 10^170 possible board states, much greater then the state space of chess.

Google's 2015 AlphaGo was the first AI agent to beat a professional Go player. It used a deep learning model to train the weights of a Monte Carlo tree search. The deep learning model consisted of 2 Artificial Neural Networks, one network to predict the probability of potential moves made by the opponent and the other network to predict the win chance of a given state. The combination of the deep learning model and the Monte Carlo tree search allowed the agent to explore the state tree much more efficiently than if it was using just the Monte Carlo tree search.

AlphaGo was initially trained on games against human players and later on games against itself. Later, in 2017, another implementation of AlphaGo, AlphaGoZero, was made public. This implementation was able to entirely train by playing against itself and it would train much faster then its predecessor. In the final stages of its training AlphaGoZero would develop new advanced strategies that, until then, were never used by a Go player.

The StarCraft series is a series of real-time strategy video games that have become very popular environments for AI research. Blizzard, the game's developer, and DeepMind have collaborated to release a public StarCraft 2 environment for AI research.

Alphastar was the first AI agent to beat a professional StarCraft 2 player without any in-game advantages. Initially for its deep learning network the agent used a simplified zoomed out version of the gamestate as imput, but later it was updated to play using a camera like other human players. Up to the date of writing this document the code or architecture of the model have not been publicly released by the developers of Alphastar, but they have listed several machine learning techniques that they used, such as relational deep reinforcement learning and long short-term memory.

Initially Alphastar was trained with supervised learning, it watched replays of many human games in order to learn basic strategies, later it trained against different versions of itself an was improved trough reinforcement learning. The final version was successful but it was only trained to play on a specific map and matchup.

Dota 2 is a multiplayer online battle arena (MOBA) game. Because of its complexity traditional AI agents have not been able to play at the same level as professional human players. The only widely published information on AI agents attempted on Dota 2 is OpenAI's deep learning Five agent.

OpenAI Five utilized separate Long Shor-Term Memory networks to learn each hero in the game. It trained using a reinforcement learning technique known as Proximal Policy Learning. It trained against itself on a system containing 256 GPUs and 128,000 CPU cores accumulating 180 years of game experience each day. OpenAI Five's first public appearance occurred in 2017, it won a one-on-one game against a professional Dota 2 player known as Dendi. The following year the agent advanced so much that it could performe as a team and in a 2019 series of games beat the 2018 Dota 2 champion team.

1.2. Computer vision-based agents

Computer vision focuses on training agents to gain a high-level understanding on digital images and videos. Many computer vison techniques incorporate forms of machine learning and have been applied to various video games. This technique focuses on interpreting game events using visual data, in some cases AI agents, using model-free techniques, have learned to play games without any direct connection to internal game logic, solely using video data as input.

Pong is a table tennis sports game featuring simple two-dimensional graphics, manufactured by Atari and originally released in 1972. Andrej Karpathy, Tesla's director of artificial intelligence, demonstrated that a neural network with just one hidden layer is capable of being trained to play Pong based solely on screen data.

Doom (1993) is a first person shooter (FPS) game. A group of student researchers from Carnegie Mellon University used computer vision techniques to create an agent that could play the game using only image pixel input from the game. The students used convolutional neural network layers to interpret image data and output valid information to a recurrent neural network which was responsible for outputting game moves.

Other uses of machine learning in video games focus more on generating procedural content rather than AI agents that interact with the player. Procedural content generation is the process of generating data algorithmically rather than manually. Examples of procedurally generated content can be found in the weapons of Borderlands 2, the map layouts of Minecraft and the entire universes in No Man’s Sky. Common approaches to procedural generation include techniques that involve grammars, search-based algorithms, and logic programing. These approaches require a human to manually the range of content possible, but machine learning is theoretically capable of learning the features of possible content when given examples to train on, thus reducing the time spent on defining the range of desired content. Machine learning techniques used for content generation include Long Short-Term Memory, Recurrent Neural Networks, Generative Adversarial Networks and K-means clustering.

Galactic Arms Race is a space shooter video game that uses neuro-evolution powered procedural content generation to generate unique weapons for the player, based on his personal preferences. This game was a finalist in the 2010 Indie Game Challenge and its related research paper won the Best Paper Award at the 2009 IEEE Conference on Computational Intelligence and Games. The content was generated using a form of neuro-evolution called cgNEAT.

Machine learning is not a new addition to the domain of video games, but that does not mean there is no room for new and innovative application of its techniques. Over the years these techniques have continued to evolve and will continue to do so with the development of new video games and technologies.

2. IMPLEMENTATION

The project is developed with the Unity game engine, which is a cross-platform platform game engine developed by Unity Technologies. The engine can be used to create three-dimensional, two-dimensional, virtual reality and augmented reality games as well as simulations, animations and other experiences. The programing language associated with the unity engine is C#, all the scripts comprising the project are written in this programing language.

The project can be divided in three integral components: the AI adversary, the player characters and the environment.

2.1. AI adversary

In the editor the AI agent game object is comprised of a red square sprite, aRigidbody2D, two BoxCollider2D components and the PlayerDetector and Navigator scripts. One of the colliders together with the rigid body component are responsible for the correct interaction between the AI and the walls, they assure that the collision actually happens. The second collider has the is Trigger property checked and acts as the vision field of the AI.

The core components of the AI are the K-means clustering algorithm, found in the PlayerDetector.cs script, for determining an advantageous position and the A-star algorithm, found in the Navigator.cs script, for calculating a path to that position.

2.1.1. The K-means clustering algorithm

Clustering is used for partitioning a set of objects into groups, called clusters. Objects that are similar, according to a certain similarity measure, are places in the same cluster, while dissimilar objects are placed in different groups.

The K-means algorithm aims to partition n objects $(x_1, x_2, \dots, x_n)$, in k $(\leq n)$ clusters $\boldsymbol{S} = \{S_1, S_2, \dots, S_k\}$, in which every object belongs to the nearest cluster center. The most common algorithm uses an iterative refinement technique.

Given an initial set of k means $m_1^{(1)}, m_2^{(1)}, \dots, m_k^{(1)}$, represented by centroids corresponding to each clusters, the algorithm alternates between two steps:

Assignment: The algorithm assigns each object to the cluster with the nearest mean, that with the minimum Euclidean distance between the object and the centroid corresponding to that cluster. If an object has the same minimum Euclidean distance to multiple centroids, it will be randomly assigned to one of the cluster that correspond to those centroids.

$S_i^{(t)} = \left\{x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \forall j, 1 \leq j \leq k \right\}$, where t represents the current iteration.

Update: Recalculate the means (the position of the centroids) for the new clusters.

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

The algorithm stops when, after 2 consecutive iterations the assignments no longer change. The algorithm does not guarantee to find the global optimum partitioning, the result may depend on the initial clusters. As the algorithm is usually fast, it is possible to run it multiple times and obtain different clusters.

The algorithm is most commonly used to assign objects to the nearest cluster by Euclidean distance. Using a different distance function may result in the algorithm never stopping.

Methods commonly used for initializing the clusters are **Forgy** and **Random Partition**. The Forgy method randomly choses k objects from the data set and uses these as the initial means. The Random Partition method first randomly partitions all the objects to one of the clusters and then proceeds to the update step, computing the initial means to be the centroids of the clusters randomly assigned points.

Illustrated example of the K-means algorithm:

Given a set of objects, represented by the squares in Figure 2.2.1.1**,** the k-means algorithm is used to partition them in k = 3 clusters.

Step 1: k initial means, represented by the colored circles are randomly placed within the data domain. (*Figure 2.2.1.1*);

Step 2: k clusters are created by assigning each object to the closest mean. (*Figure 2.1.1.2*);

Step 3: the cluster centroids become the new means (*Figure 2.2.1.3*)

Step 4: steps 2 and 3 are repeated until the cluster assignments don't change anymore (*Figure 2.1.1.4*);

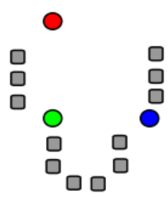
Figure 2.1.1.1

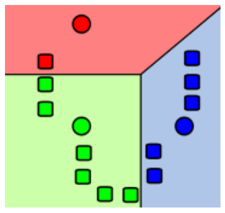
Figure 2.1.1.2

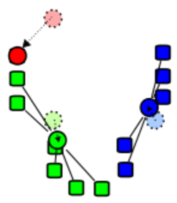
Figure 2.1.1.3

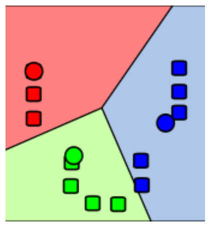
Figure 2.1.1.4

Implementation of K-means clustering Algorithm

In the project the k-means algorithm is used to partition the position where the AI adversary has encountered the player in previous sessions and calculate an advantageous position to were the AI can move to intercept the player. The code implementation is found in the PlayerDetector.cs script.

The coordinates of locations were the AI detects the player are stored in a text file named "points.txt" located in the project directory. To write and read from the file I used StreamWriter and StreamReader objects from the System.IO namespace. The coordinates are used to create Vector2 objects from the UnityEngine namespace, those store two-dimensional coordinates. To display points and cluster centers I used GameObjects also from the UnityEngine namespace.

For handling collections of coordinates and GameObjects I used the List object from System.Collections.Generic. Clusters being represented by a List object containing Lists of game objects.

To display a step by step execution of k-means I used the InvokeRepeating(string methodName, float time, float repeatRate) method from the UnityEngine. MonoBehaviour class. When called the method methodName will be invoked after time seconds and then repeatedly every repeatRate seconds. Calling CancelInvoke() will cancel all invoke calls.

The code has been structured in different methods that are called in order to do certain tasks. Next I will detail the methods and how they interact in order to implement the K-means algorithm.

- **OnTriggerEnter2D(Collider2D coll)** is called when the collider coll enters and stays inside the trigger collider attached to the AI, that acts like the vision field. The method checks if coll belongs to the player character, if that is the case the coordinates of the player character are added to the positions list;
- **initialize_Cluster_Centers_and_Points()** method iterates trough the ols_points list creates GameObjects with the coordinates of the points and finds the minimum and maximum values of the x and y. Then creates cl_number cluster centers, assigns them a color, adds them to the cluster_centers list and places them at random coordinates between the minimum and maximum x and y.
- **clusterize_Points()** initializes the clusters list then iterates trough the points leaded from points.txt, computes the Euclidean distance between the point and the first cluster center in the cluster_centers list then iterates trough the rest of the list, computes the Euclidean distances and compares them. The point is added to the corresponding cluster in the clusters List and is assigned that clusters color.
- **move_Cluster_Centers()** iterates trough the clusters list, calculates the new coordinates for the cluster centers and counts the number of centers that have been moved. The new cluster center coordinates are computed by calculating the arithmetic means of the x an y coordinates for each cluster, the means become the new z and y coordinates of the cluster center. If the new center coordinates are different from the old ones the variable moved gets incremented by one;
- **SetNavGoalCoords()** calculates the centroid of the cluster centers, rounds the coordinates and adds or subtract 0.5 from x or y to make sure the coordinates land on the movement grid, and then moves the NavigationGoal game object at those coordonates;
- **K_means()** calls **clusterize_Points()**, **move_Cluster_Centers()** in a loop and count the number of iterations. The loop ends when the number of moved centroids is 0. After the loop a message is constructed containing the number of

iterations, this message is displayed on the screen, if the application is in Developer Mode. Lastly **SetNavGoalCoords()** is called;

- **Save()** writes the coordinates stored in the positions list in points.txt;
- **Load()** reads the player coordinates from points.txt and place them in the old_points list;
- **make_Colors()** generates 25 color objects by using 3 for loops nested one into another, with counters going from 1 to 0 in steps of 0.5, it creates a Color objects with those counters as RGB values and adds those objects to the colors list;
- **toggle_display_Clusters()** iterates the points and cluster centers lists and makes the corresponding game objects visible;
- **Start()** is called once at the start of the application, the method initializes the list objects used in the script, calls **Load()**, **make_Colors()**, **initialize_Cluster_Centers_and_Points()**, initializes the moved variable with the number of clusters, so that the k-means algorithm execute at leas one iteration, and, if the application is in Play Mode, calls **K_means()**;
- **Update()** is called at the end of every frame and it is used for handling keyboard inputs and checking to see if the cluster centers have not been moved and call the CancelInvoke() method;
- **OnTriggerStay2D(Collider2D coll)** similar to **OnTriggerEnter2D(Collider2D coll)** is called when another collider **coll** enters the trigger collider of the AI, but it is also called as long as coll is inside the AI trigger collider. The method checks if coll belongs to the player and moves the navigation goal game object to the players coordonates.
-
- **OnApplicationQuit()** is called when the application is closed, inside this method there is only a call of **Save()**.

2.1.2 The A-star Algorithm

The A-star(A*) algorithm is a graph traversal and path finding algorithm. It was published by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute in 1968. It can be seen as an extension of Edsger Dijkstra's 1959. A* guides its search using heuristics to achieve a better performance that its predecessor.

A* is applied on weighted graph, starting at a specific starting node, its goal is to find a path to a given destination node, having the smallest cost, this cost can represent the distance traveled, the time traveled, etc. To find this path the algorithm creates a tree of paths with the start node as its root and extends those paths one at a time until it reaches thew destination node. A* selects the path that minimizes $f(n) = g(n) + h(n)$ where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n and $h(n)$ is an additional heuristic that guides the algorithm by estimating the cost of the cheapest path from n to the destination. As long as the heuristic function never overestimates the actual cost to get to the goal A* is guaranteed to find the cheapest path from start to destination. A good heuristic is the straight-line distance between n and the destination.

The algorithm most commonly uses a priority queue in which the nodes with the lowest $f(n)$ have priority. At each step of A* the top node is popped from the queue, the f and g values of its neighbors are updated and they are added to the queue. The algorithm until the destination node is popped from the queue or the queue is empty. To get the sequence of nodes that lead to the destination each node in the path keeps track of its predecessor, when A* stops the destination node will point to the previous node, and so one, until a node's predecessor will be the start node.

Implementation of A-star Algorithm

In the project the A* algorithm is used to find the shortest path to the advantageous position calculated by K-means. The heuristic ($h(n)$) used is the Euclidean distance between a node and the destination. The code implementation is found in the Navigator.cs script.

To represent the priority queue elements I used KeyValuePair<Vector3, float> objects from the System.Collections.Generic namespace, where the Vector3 objects represents the node and the float value is $f(n)$. The priority queue is a List of KeyValuePair objects which are ordered by the float value, at the end of each step.

To record the visited nodes so that A* does not expand them again I used HashSet<Vector3> in which the Vector3 objects represents visited nodes.

To record the predecessors of each node I use a Dictionary<Vector3, Vector3> object, in which the first Vector3 object represents a node and the second one its predecessor. A Dictionary <Vector3, int> object is also used to record the $h(n)$ values.

Next I will detail the methods fount in Navigator.cs and how they interact in order to implement the K-means algorithm. I will not go in depth on the behaviour of methods already detailed in the K-means section such as Start() and Update().

- **getDistance(Vector3 start, Vector3 end)** returns the Euclidean distance between start and end using the Math methods Sqrt and Pow;
- **isWall(Vector3 place)** return true if the place coordinates correspond to a wall gameObject and false otherwise;
- **onGoal(Vector3 coords)** returns true if the coords coordinates correspond to the navigation goal and false otherwise;
- **A_star_path_finder(Vector3 start)** implements the A* algorithm and computes the shortest path from start to the position of the Navigation Goal game object. First all the data structures used in the algorithm are instantiated, such as the visited list, A_star_stack list of KeyValuePair objects, from and cost dictionaries for recording predecessors and $g(n)$ and a proto_path list. The start node is added to the stack, its predecessor is recorded as itself and its cost to 0. In a while loop the firs element in the stack is popped, marked as visited and added to the proto_path list. Next

and if checks for the stopping condition, if the popped node is the destination, and the loop is stopped if so. Next 4 Vector3 objects representing the current nodes neighbors, in practice those are the 4 direction the AI can move in (up, down, left, right). As you can see the graph is generated as the algorithm searches, because the A* algorithm does not need the entire graph at the beginning of the search generating it dynamically is more memory efficient. The 4 directions are added to a newly initialized direction list, a series of if-else statements determine the order in which the directions are placed in the list by comparing the current node's coordinates with the coordinates of the destination (if the destination is above the node then the neighbor "up" is placed first, then "left", "right", and so on). The different order of the neighbors is both an optimization and a solution for a bug where, for certain destinations, the path would encircle the destination before arriving in it. For each neighbour that is not a wall and hasn't been visited the cost and predecessor are updated accordingly then if the node exists in A_star_stack its $f(n)$ value is updated based on the Euclidean distance and the cost function, if it doesn't exist in the stack it is added. A_star_stack is sorted by comparing the values of the KeyValuePair objects inside it. After the while loop end, signifying the stop on A*, the last node in the proto_path list is the destination node, using from Dictionary to obtain the predecessor of the destination node and all other nodes until the start node is reached, a path from the destination node to the start node is obtained and reversed it gives the desired path.

- **CalculateMoves()** iterates trough the path list, converts the coordinates in the list to moves represented by integers from 0 to 3 (0 – left, 1 – right, 2 – down, 3 – up) and add those moves to the moves list;
- **Move()** is called when the player moves so that the AI agent moves at the sae time. If the AI is not at the navigation goal it uses a counter move_number to pick the next move from the moves list. If the it is already at the goal position it will randomly chose one of the 4 moves. After making a random move instead of recalculating a path with A* the method **after_random_move(int choice)** is called which reinitializes the moves list and depending on the value of choice adds a move in the moves list that will move the AI back to the goal location;
- **OnCollisionStay2D(Collision2D coll)** is used to end the game and display a game over message on the screen if the AI collides with the player and handle the collisions with walls;
- **make_path()** instantiates game objects corresponding to the coordinates stored in the path list and stores them in a list named path_objs. Those game objects are used for displaying the path.
- **toggle_display_Path()** toggles the visibility of the navigation goal and path_objs game objects.

- **Update()** is used for checking if the navigation goal is moved and if so then the **A_star_path_finder** method is called with the AI current position as the start node and after that **CalculateMoves** is called. This method also handles keyboard inputs for displaying the path and navigation goal;
- **Start()** initializes the endPosition and rb variables, used for the movement system which will be detailed next, saves the navigation goal position, used to check when the goal moves and initializes move_number with he value 0.

2.1.3 The Movement System

The Navigator.cs script is also responsible for the movement system of the AI. The movement system uses 2 important variables: rb, in which the Rigidbody2D component of the AI is stored, and a Vector3 object named endPosition.

In the **Start** method rb is initialized with the Rigidbody2D of the AI and endPotition with rb’s coordinates. When the player moves the method **Move** is called and the coordinates of endPosition are modified. In the **FixedUpdate** method called at the beginning of the physics cycle, which can happen more than once per frame, if the rb coordinates and endPosition do not match using the Verctor3 method **MoveTowards(Vector3 current, Vector3 target, float maxDistanceDelta)** which returns a Vector3 position between the points specified by current and target, moving no farther than the distance specified by maxDistanceDelta. Then the AI is moved to the new position by using the Rigidbody2D method MovePosition(Vector2 position) to move its Rigidbody2D component to position coordinates.

To handle collisions with walls in the **OnCollisionStay2D** method the endPosition coordinates are changed back to the old rb coordinates.

2.2. Player Character

In the editor the Player Character game object is comprised of a blue square sprite, a Rigidbody2D and a BoxCollider2D component and the PlayerControler.cs script. The Rigidbody2D and BoxCollider2D components serve the same purpose as here as in the AI game object.

The PlayerControler.cs script is responsible for the movement of the player character and is similar to the movement system of the AI agent, although it has 2 major differences. In the **Start** method an object adv_move of type Navigator, as in the Navigator script, is instantiated with the Navigator script component of the AI game object. The **Update** method handles arow keys inputs and when the player makes a move it uses the adv_move object to call the Move method from the AI’s Navigator script, making the AI move at the same time with the player. The Update method also uses a Boolean variable called move to check if the player character is currently moving and to only register another input when the player character has stopped, thew purpose of this check is to prevent the player from moving diagonally by inputting 2 moves in quick succession, for example right and then up.

2.3. Environment

The environment is comprised of a simple maze, the green goal area and a minimal user interface. The maze is constructed using a Grind game object which contains a Tilemap object which allow the placement of sprites on the aforementioned grid, adding a Tilemap Collider 2D to the tilemap object will attack a Collider2D component to all the sprites on the grid. The Composite Collider 2D component attached to the Grid object merges all the individual sprite colliders into a singular collider, this is done to prevent the characters getting stuck on phantom vertexes which appear when 2 colliders are aligned next to each other. The green goal area is composed of a green rectangle sprite, a BoxCollider2D trigger component and a script called GoalControler.cs. When the Player enters the goal area's collider the OnTriggerEnter2D method from GoalControler is called, the method stops the game and displays a game over message on the screen.

The user interface consists of a pair of buttons for starting the app in different modes and text messages that appear on the screen when the game is over. When pressed 2 buttons call their respective methods from the GameMaster.cs script. The GameMaster.cs script is attached to the GameMaster game object and contains the 2 methods used to activate the environment: **setup_play_mode()** and **setup_dev_mode()**. On application start the GameMaster deactivates all the components in the scene except the main camera and the user interface, when one of the buttons is clicked it's respective method is called and the environment is activated and made visible, depending on which button was clicked a game object named PlayMode is either activated or not, this game object is used by the other scripts to check if the application is in play mode or developer mode. This script is also responsible with closing the application when the Escape key is pressed, the check is made in the Update method.

3. USER MANUAL

After starting the application you must chose one of two modes, Play Mode or Developer Mode, by pressing one of the corresponding buttons.

Play Mode is intended to exhibit how the game would behave in a normal play session. All the computing is done in the background. The player controls the blue square and the goal is to reach the green area by navigating the grey maze and avoiding the AI adversary, represented by the red square.

Controls:

- Use the arrow keys to move the player character (blue square). You can only move in a single direction at a time.
- Use the “Escape” key to close the application.

Developer Mode is intended to exhibit the internal mechanics of the game. When the session starts in developer mode the AI does not run the k-means algorithm right away and it will not do so until a specific key is pressed. In this mode there is a list of important keyboard inputs.

Controls:

- Use the arrow keys to move the player character (blue square). You can only move in a single direction at a time.
- Press “D” to display a graphical representation of useful information.
  - The white points represent all the positions recorded and used by the k-means algorithm;
  - The colored triangles represent the means of the clusters;
  - The dark red diamond represents the AI’s navigation goal;
  - The small red squares represent the path calculated by the A-star algorithm;
  - The grey, transparent square represents the AI vision field;
- Press “I” to execute one iteration of the K-means algorithm. The points, if displayed on the screen, will change color according to the cluster they have been assigned to and the means will move to the centroids of the corresponding clusters.
- Press “K” to start running the K-means algorithm, one iteration will be executed every half a second and when the algorithm stops a text with information regarding the number of iterations will appear at the bottom of the screen.
- Use the “Escape” key to close the application.

When the player enters the goal area or the AI touches the player character the game will stop and a message will appear on the screen displaying the result of the game.

CONCLUSION

The range of machine learning applications in the domain of video games is already broad and with the passing of time it is only going to get even broader. From what has been detailed in chapter 1 we can certainly find a purpose for machine leaning powered AI in video games aside from being the subject of research projects.

The complex AI agents that learn to play multiplayer games and can even beat professional players like OpenAI Five, AlphaStar and even AlphaGoZero can serve as training partners for the same players they defeated or they can provide entertainment by being places in AI only tournaments where different research teams can showcase new machine learning techniques. The AIs that learn how o play single player games can be adapted into play testers that can work much faster than their human counterparts reducing development time and labour costs. The difficulty of a game that uses AI enemies implemented with machine learning can be modified just by using training data from different stages in the AI's training.

Future developments for this project include:

- Extending the AI behaviour so multiple agents can work together and divide multiple advantageous positions among themselves;
- Writing a function that can determine a good (maybe even optimal) number of clusters by identifying bottle neck areas in the layout of the level;

BIOGRAPHY

Anders Drachen, Rafet Sifa, Christian Bauckhage, Christian Thurau, 2012. “*Guns, Swords and Data: Clustering of Player Behavior in Computer Games in the Wild”.* 2012 IEEE Conference on Computational Intelligence and Games.

Justesen, Niels; Bontrager, Philip; Togelius, Julian; Risi, Sebastian 2019. “*Deep Learning for Video Game Playing”*. IEEE Transactions on Games. 12: pages 1–20.

Silver, David; Huang, Aja; Maddison, Chris J.; Guez, Arthur; Sifre, Laurent; van den Driessche, George; Schrittwieser, Julian; Antonoglou, Ioannis; Panneershelvam, Veda 2016. *"Mastering the game of Go with deep neural networks and tree search"*. Nature. Volume 529, Issue 7587: pages 484–489.

The AlphaStar team, 2019. *“AlphaStar: Mastering the Real-Time Strategy Game StarCraft II”*. https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

Jakub Pachocki, Michael Petrov, Henrique Pondé, Przemysław Dębiak, Brooke Chan, Jonathan Raiman, Szymon Sidor, David Farhi, Jie Tang, Greg Brockman, Christy Dennison, Susan Zhang, Filip Wolski, 2018. *“OpenAI Five”* https://openai.com/blog/openai-five/

Andrej Karpathy, May 31, 2016. *“Deep Reinforcement Learning: Pong from Pixels”* http://karpathy.github.io/2016/05/31/rl/

Guillaume Lample, Devendra Singh Chaplot, February 2017. *“Playing FPS games with deep reinforcement learning”* AAAI'17: Proceedings of the Thirty-First AAAI Conference on Artificial IntelligenceFebruary 2017 pages 2140–2146 https://dl.acm.org/doi/10.5555/3298483.3298548

Wesley Yin-Poole, July 2012. *“How many weapons are in Borderlands 2?”* https://www.eurogamer.net/articles/2012-07-16-how-many-weapons-are-in-borderlands-2

"*Terrain generation, Part 1*" March 2011. The Word of Notch. https://notch.tumblr.com/post/3746989361/terrain-generation-part-1

Simon Parkin, July 2014 *“No Man’s Sky: A Vast Game Crafted by Algorithms”* https://www.technologyreview.com/2014/07/22/12940/no-mans-sky-a-vast-game-crafted-by-algorithms/

Summerville Adam, Snodgrass Sam, Guzdial Matthew, Holmgard Christoffer, Hoover Amy K., Isaksen Aaron Nealen Andy, Togelius Julian, September 2018. *"Procedural Content Generation via Machine Learning (PCGML)"*. IEEE Transactions on Games. Volume 10 Issue 3 pages 257–270.

Erin J. Hastings, Ratan K. Guha, Kenneth O. Stanley, September 2009. *“Evolving Content in the Galactic Arms Race Video Game”.* Proceedings of the IEE Symposium on Computational Intelligence and Games (CIG09). Piscataway, NJ:IEEEWinner of the Best Paper award at CIG-2009.

Hamerly Greg, Elkan Charles, 2002. *"Alternatives to the k-means algorithm that find better clusterings"*. Proceedings of the eleventh international conference on Information and knowledge management (CIKM).

Hartigan J. A., Wong M. A. 1979. *"Algorithm AS 136: A k-Means Clustering Algorithm"*. Journal of the Royal Statistical Society, Series C. Volume 28 Issue 1 pages 100-108.

Wikipedia the free encyclopedia, "*k-means clustering"* Demonstration of the standard algorithm images 1-4.

Russell Stuart J. 2018. Artificial intelligence a modern approach. Norvig, Peter (4th ed.). Boston: Pearson.

Unity Technologies 2020. *"Unity Documentation"* https://docs.unity3d.com/ScriptReference/index.html