DESIGN DOCUMENT

LUX AURA

Javier Villanueva Forner

53794782-J

Introduction

Lux Aura is a simplified version of Auralux[1], launched in 2013 by E. McNeill and ported to different platforms by WardRum Studios. The version we propose here is a simplification of the game in terms of game play mechanics and in graphics. This is due to the objective of the game: we want to create a controlled environment with simple but firmly defined rules to train the machine learning mechanisms we will implement later. Because of this, elements that would be included to make it more attractive to users will be ignored. We will also approach the possible implementations of the machine-learning techniques that will be programmed, as well as how they will be tweaked to adapt them to the player level.

In this document we will use the concept of “player” (or “players”). We want to clarify that, unless it is explicitly specified (case in which the term “human player” will be used), the expression will refer to all the agents with capacity to play the game. In other words, we will refer to human players and AI. We will also use the term “agent”, referring the concept of an autonomous agent with a certain behavior that interacts with the space within the game where it is.

Gameplay

Lux Aura is a real-time strategy game in which the players fight for the control of a planetary system. The objective is to be the last entity alive, that is, the last player that controls at least one planet. To achieve this, the players have to conquer and attack each other by ordering their units what to do.

These units are generated periodically in the conquered planets. Each planet will produce a certain number of units every time a certain time passes. These units will belong to the player that controls that planet. If a planet is not controlled by anyone (aka. that planet is neutral neutral) no units will be produced. The number of units a planet produces depends on the level that planet has. All planets start at level one, in which they will produce two units per second. Some planets can be upgraded by ordering units to do so. If a planet levels up, the amount of units produced per second will increase.

Each player has at least one planet under its control while it's still alive. The units that are currently in those planets can be controlled and have to be used by the player to achieve his goals. Units can be sent to enemy planets by selecting the desired number of them to conquer them. They can also be sent to a planet that belongs to the player to upgrade it or to simply wait there until the player decides to use them.

Conquering, attacking or upgrading planets has a cost. A unit can destroy another one, decrease by one the health points of a planet, increase its health points by one (if the planet belongs to the same player as the units) or increase by one the experiences points of a planet (again, if the planet belongs to the same player). Once one of this actions is performed, the unit will be destroyed. For example, let's assume that we have a planet controlled by player one with one hundred of units in it and another controlled by player two with only twenty. If player one attacks player two with his one hundred units, the units in player's two planet will be destroyed and twenty of the one hundred player's one units will be as well. That means that the eighty remaining units will decrease the same amount of health points to player's two planet and then they will be also destroyed.

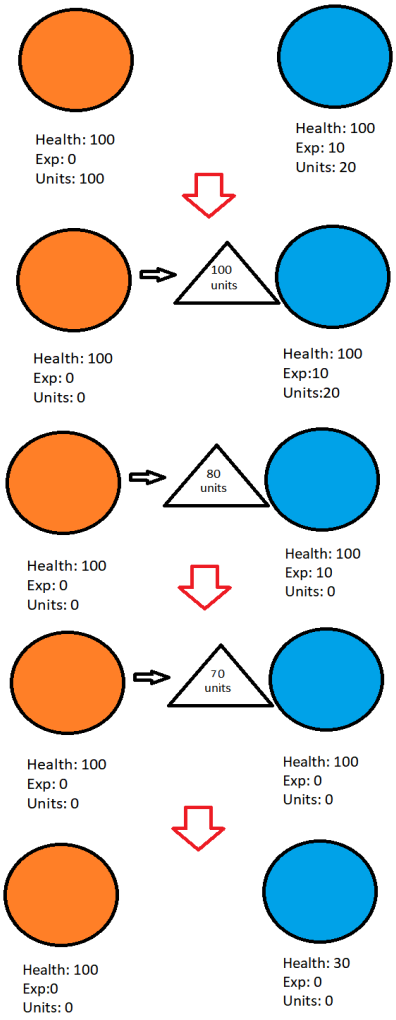

Figure 1: Example of an attack

Planets have three main attributes: health, experience and their current units. The first one is used to conquer the planet, the second one to upgrade it and the third one stores the units that are currently in that planet. Units increase periodically, but to modify the other two values, players must spend units in a planet. The health will always increase before the experience and this order will be inverted when the numbers decrease. In other words, if a planet has some experience and full health and it is attacked, the

experience will suffer damage first and, if it reaches zero and the attack continues (aka. there are still some units attacking), the health will decrease. If the player spend units to upgrade a planet, first the health will be restored and only if the health is at its maximum value the experience will increase. This can be seen at *Figure 1*.

Planets can belong to a player or be neutral. Neutral planets won't produce units and will have health points that players will need to reduce (sending units) to conquer them. Planets that belong to a player will produce units and are harder to domain, because before reducing their health points, the units and experience points have to be eliminated first.

It is important to notice that if a player attacks a planet but does not conquer it, the other player will have to "overpower" their attack before trying to conquer it. This translates to what can be sen in *Figure 2*. The health points that other players reduced have to be restored before conquering the planet.

To make the game more friendly for the player, each player will have its own representative color. Planets and spaceships will be of this color, to identify them easily. When a planet is neutral, its color will be grey until it is conquered (as seen in *Figure 2*).

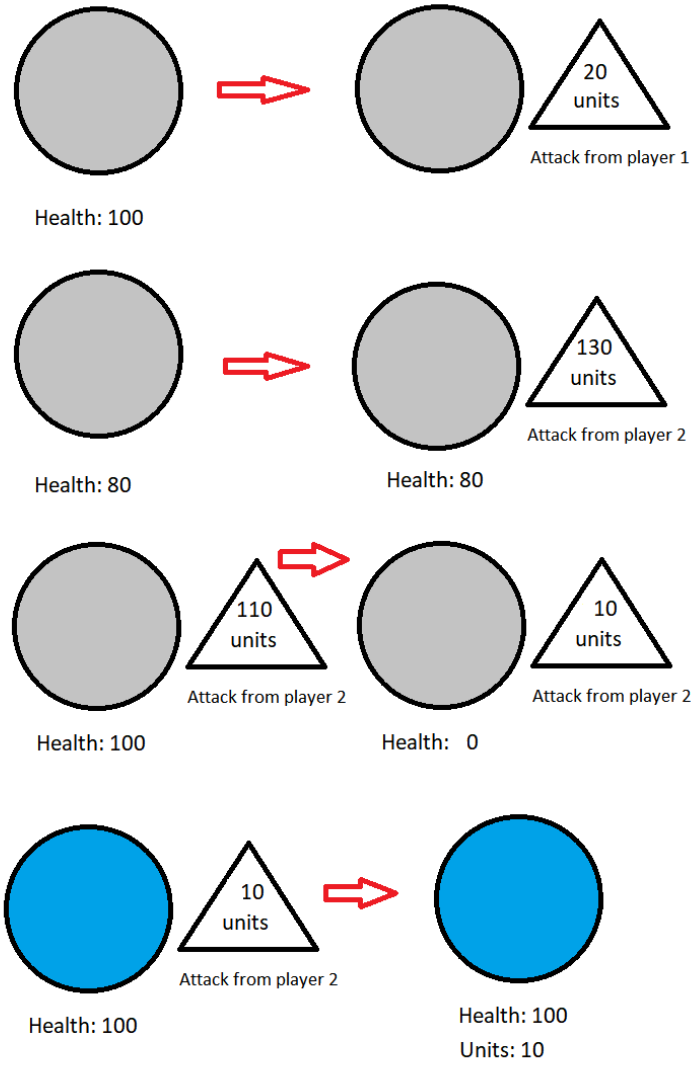

Figure 2: Attack to a neutral planet step by step

Mechanics

The mechanics of the game define what can happen or be done while playing it. In our case, these are centered in the units. The main mechanism players have to interact and change the state of the world are their units. Thus, the main mechanics are:

- Unit Generation: Periodically, units will be generated in every planet that belongs to a player. The number of units that planet has will increase depending on that planet level. No player interaction is required for this to happen.
- Upgrading Planets: If a planet can be upgraded, once enough units have been sent to it and the experience reaches the required amount, the planet will level up. This will imply a minor visual change (the planet will grow in size) and will increase its units production ratio. This ratio, as well the experience needed to level up the planet, will evolve as follows:
  - Level 0: 2 units per second (initial level).
  - Level 1: 4 units per second (100 experience points required).
  - Level 2: 6 units per second (150 experience points required).
- Unit Selection: Each player will be able to select the number of units that needs from the planets that controls. The number of units that can be selected goes from zero to all the units available in the planet, and this can be extended to every planet the player has under his domain.
- Unit movement: Once some units have been selected, they can be sent to any planet in the system.
  - If that planet belongs to the player, the units will heal it, increase its experience points if the health it's at its maximum value or just be stored there if the health and experience can't be increased.
  - If the planet belongs to another player or it is neutral, the sent units will destroy the units that are stored in it, decease its experience points and lastly, decrease its health points, again, as seen in *Figure 1.#Imagen1|graphic*

  These attacks will manifest in a spaceship that will move from the attacker to the objective. They will become effective once the spaceship reaches the objective. It is important to notice that, while in the original game units can collide with each other and are shown independently, in *Lux Aura* the units move in huge space crafts and do not collide with each other.
- Conquer planets: If there are enough units sent to a planet that belongs to another player that the health points of it reach zero (*Figure 1*), the planet then will become neutral and, again, if its health points reach zero (*Figure 2*), it will be conquered by the player that sent the units. When a planet is conquered, it will turn the color of the player that conquered it.
- Camera movement: To be able to see the entire system, the camera moves in a plane parallel to the plane that contains the planets and can be zoomed in and out.

Controls

The human player will control the game using mouse and keyboard. This will allow for displacement and unit selection and control.

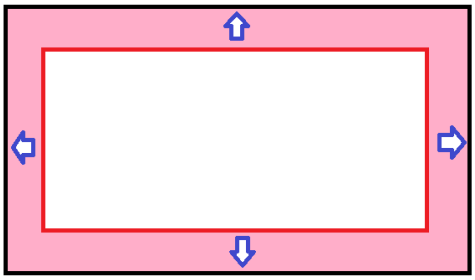

Figure 3: Scheme of mouse control. When the cursos enters the red area, the camera will move in the direction of the arrows. In the corners, it will move in the direction of the two closest arrows.

Moving the mouse to the edges of the screen will make that camera move in that direction. This will allow the player to move freely around the map. If the left control button is pressed, the camera won't move.

If the player scrolls the mouse wheel up and down or presses the "w" or "s" keys, the camera will zoom in and out respectively. This will allow the player to control the amount of the map visible at any moment.

If the player right-clicks in a planet that belongs to him, 1/5 of that planet units will be selected Those unit will be subtracted from the planet and added to a text that will be placed near the cursor, so the amount of selected units is visible at any moment. If he or she clicks with the left button, all the units of the planet will be selected.

If there are units selected and the player clicks in an empty area, they will be deselected and returned to their planets. If a planet that belongs to another player is clicked, each planet will send the units that were selected to that planet. To do the same with a planet that already belongs to the player (to upgrade or heal it), it has to be right-clicked while holding the left control key.

Figure 4: Control scheme

Levels

The levels that can be found in the game are quite similar. Set in space, they represent systems of planets and stars. The planets that can be found there are the only intractable elements in the game. This are the agents that will store the units and that will be conquered and leveled up.

Their composition is simple: a group of planets that are distributed through the map. Some of these planets will belong from start to a player while other will start being neutral. The distribution itself is quite meaningless. Stars, background and other elements besides from planets that can be found are only there for decoration.

Figure 5 shows examples of levels in the original game and in our own game:

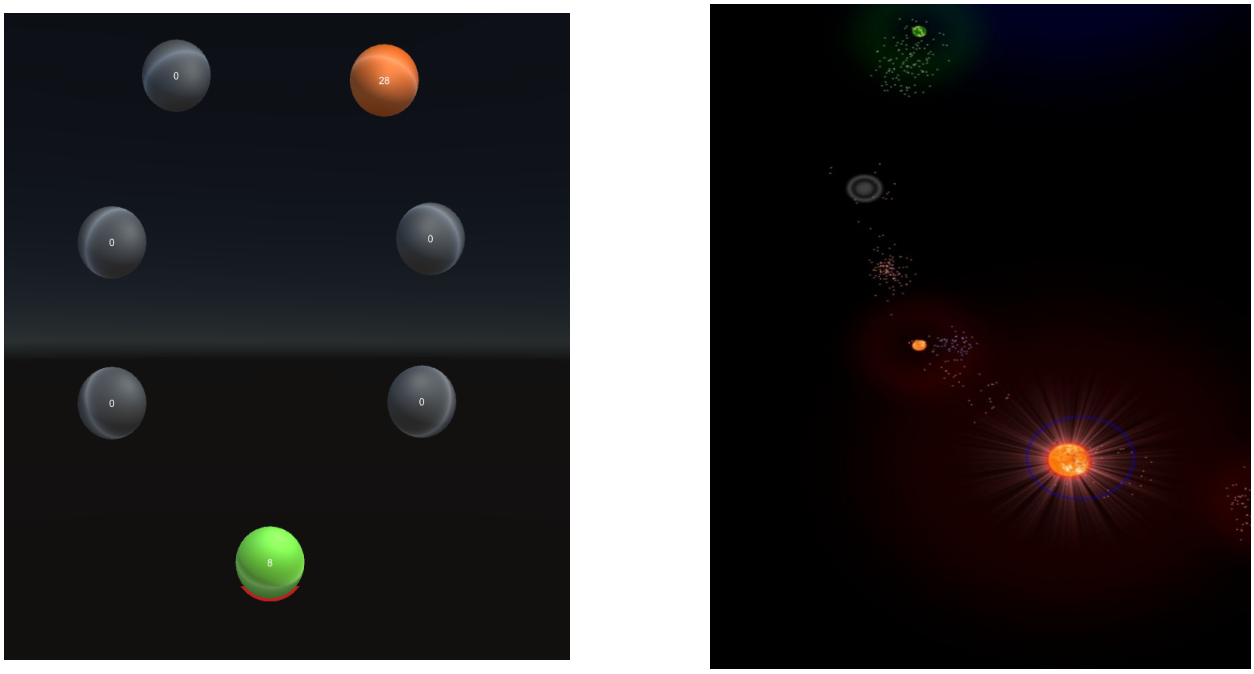

Figure 5: Example of AuraLux level (left) and a recreation (right)

Flowchart

In *Figure 6* the main flowchart of the game can be seen. The game will start in the main menu. There, the human player will be able to choose to play the game or to exit. If he or she wants to play, the level selection screen will be shown. This screen will display the different levels of the game, including the training one. Once one is chosen, the level will load and he game will start. The player will be able to exit the game and return to the main menu at any moment if he pauses the game and chooses to do it. If the game is won or lost he will also be redirected to the main menu.

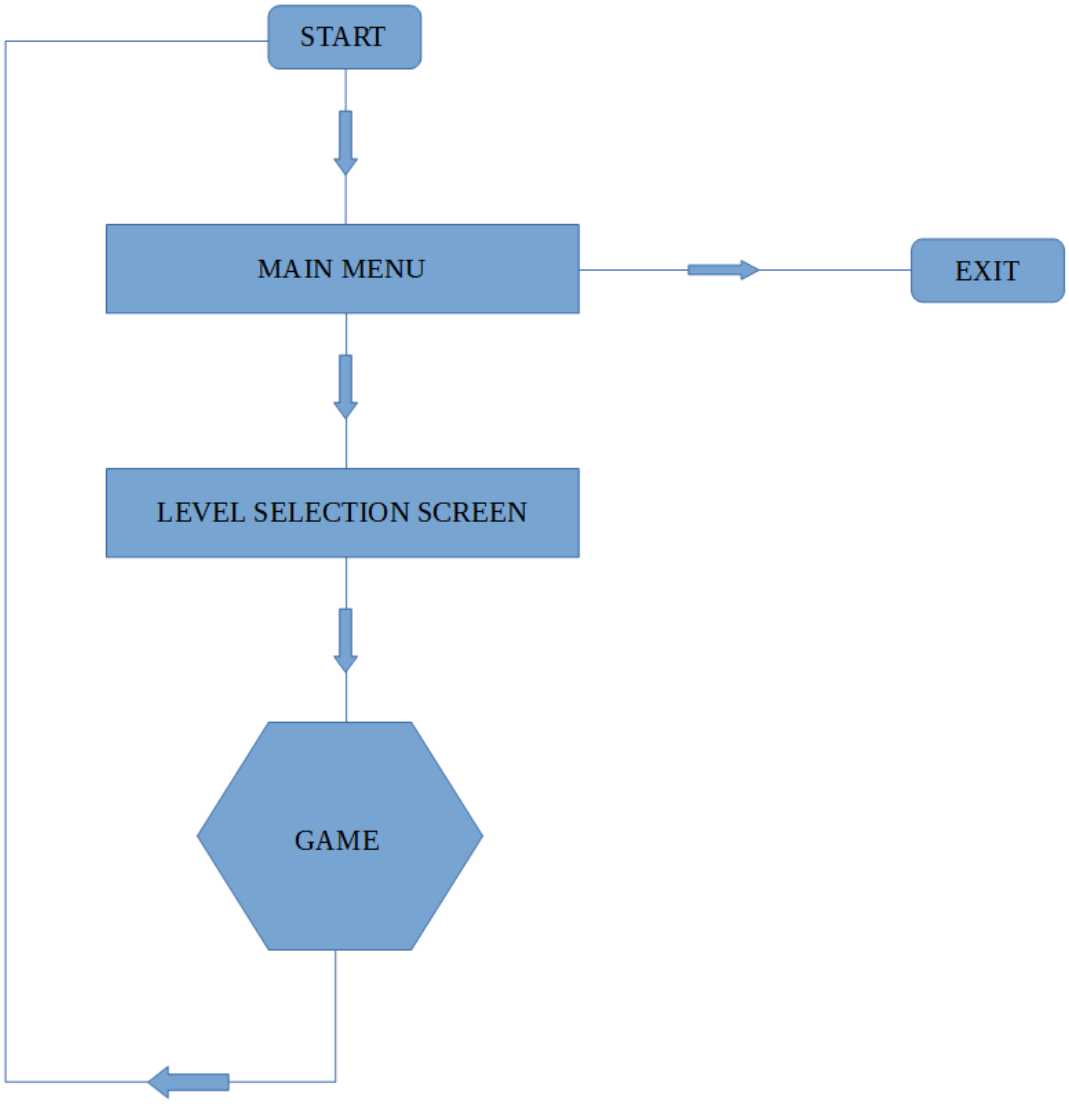

Figure 6: Flow of the game

Camera and graphics

In this section we will discuss the graphics of the game and how the player will see the world. It is necessary to remember the reader that, since our game is a copy of another one and its main purpose it's not to entertain, but rather to serve as a support for our machine learning techniques to develop, the graphical part of the game will be simplified.

This means that the colorful and beautiful scenarios of Auralux will be substituted with simple spheres and numbers. This will allow for a better demonstration of the decision making level that the machine has been able to achieve, because the total number of units will be easily readable at all times. This will be harder if the original approach (each unit is a point that orbits the planet to which belongs) would have been followed.

Human player will be able to see the world through a camera that will cover the whole level. It will be situated in front of the level and will point at it perpendicularly. As it has been mentioned before, It can be moved by moving the pointer close enough to the limits of the screen. The camera will have limitations to avoid moving too far away from the planets.

The units will never be visible. They will be shown as numbers in the planets (a number that will display the total units a planet has at the moment) and as a spaceship with another number when they move (the number showing how many units are being transported in the ship).

Health and experience of the planets will be shown using a circle that will wrap it. This circle will fill as the health and experience increase or decrease. There will be two: one for the health (white) and another one for the experience (red). These indicators will only be displayed when the health of a planet is not full or when the experience of a planet is not zero.

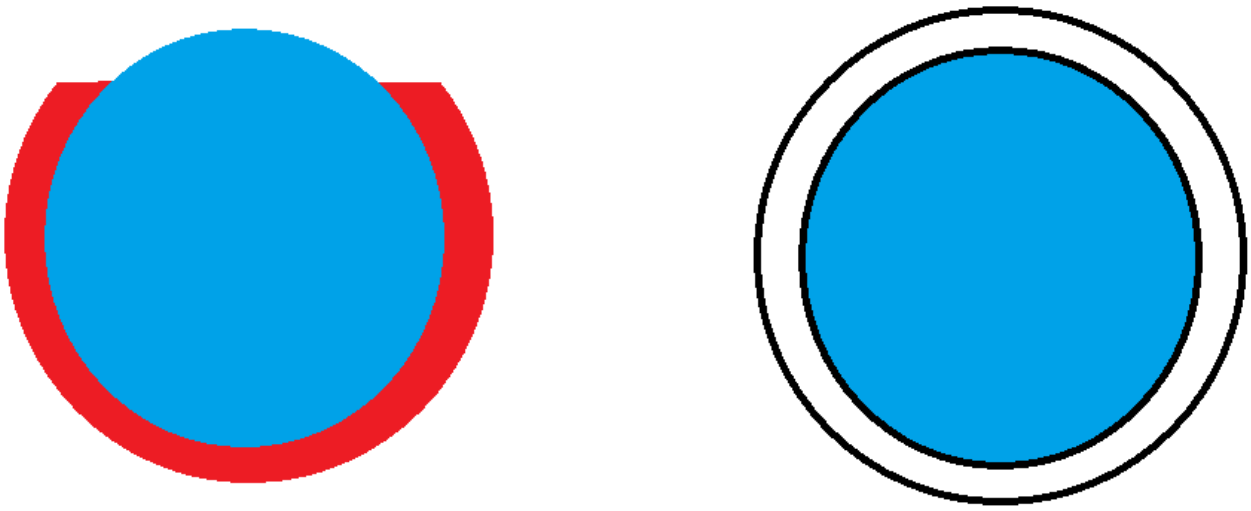

Figure 7: Experience and health indicators wraping the planet in blue. The experience one is not full like the health (right) one.

Threads and events

This section covers some of the requisites that will have to be satisfied in order to allow the machine learning techniques to be implemented correctly and without affecting the game's performance.

The game has to have an event system dependent of the system's clock. This is necessary for a couple of reasons:

1. By doing so, the events that occur periodically will happen independently of the game's performance.
2. Using the clock provides with a more precise way of controlling for how long a task should perform.

The events that have to be implemented are the following:

- Unit creation: This event will fire periodically and will affect every planet in the level. If the planet belongs to a player, the number of units it possess will increase depending on its current level.
- AI-Tick: This event refers not only to AI, but to machine learning techniques too. It simply indicates when the enemies (the non – human players) have to take a decision. This event will start those process.
- Montecarlo Stop: Only for the Montecarlo Tree Travrsal. This event will stop the simulations and the construction of the tree and force the process to return a value for the best action. More information in its corresponding section (Montecarlo Tree Traversal)

These ticks will all start a thread. This is because the tasks that have to be done can be very expensive in terms of resources and CPU time, so, in order to avoid the game for stop responding and to have to coordinate the tasks execution with the main game process, they will execute separately. Note that we are talking about threads, not *corrutines*. Because threads are more flexible and do not depend on the Unity Engine, they will be used instead of the engine's solution for multitasking. This, however, has some implications: no functions or attributes used in the engine will be accessible.

Although it might seem like a huge obstacle, it is not. For the first event, unit creation, the number of units in each planet increases if it has been conquered. No Unit-specific properties are needed. The second one, AI-tick, uses the data stored in the planets, so ,again, there is no need for using the unavailable aspects of the engine. For the last one, it is simply a timer to tell a process to stop.

Training Game

The “normal” version of the game will have a graphical component and will execute at a velocity slow enough for the human user to be able to play. However, we need to not only train our neural model, but we also need an environment where games execute as fast as possible to test the efficiency of our machine learning techniques. This is why we need a version of the game independent of the graphical components and, overall, as simple as possible.

Here is when the training version will come in handy. This version of the game has only the core mechanics of the game, those that include only numbers, and has no graphical aspect. In other words, it is composed only by a simplified version of the main classes (planets, players...) that only retain the minimum attributes and methods to be able to execute the game. That means that the planets, for example, will store the current units they have, the health, experience, etc; but there won’t be any graphical aspects or other elements of interaction. It is simply a version of the game that will be able to execute itself and display the results on the console.

In *Figure 8* the execution flow of this version can be seen. The entry point is a class that depends on Unity execution order. When the scene starts, this class reads the provided number of games that will have to be played and reads the level information that can be found in the scene. Then, it recreates the level using the simplified version we talked about and starts playing until the necessary amount of games have been executed.

In this version there are not events. Instead, the game is played by turns. A turn corresponds to one *Unit Creation* event. Every 20 turns, an AI Tick is fired. This causes the different AI types to start their own thread to decide which action to take. When the Montecarlo Tree has enough nodes or certain time has passed, this phase will end (note that the montecarlo method is the one that needs more time and thus, the last one to finish). After that, the decisions will be executed (the players will do nothing or they will attack each other) and the cycle will repeat itself.

The attacks here will also work differently. Once an attack is issued, it will be stored in a list. Based in the speed in which they move in the “normal” version of the game and the frequency at which the Create Unit events happen, they will have a number of turns to reach their objective. At the beginning of each turn, before the corresponding units are added to the planets, the list will be traversed and the number of turns remaining for the attack to reach its objective will be reduced accordingly. Once this number reaches 0, the attack will trigger. The idea is to recreate the “normal” version as best as possible.

After the attacks have been updated, a check for victory will be done. If some player won, the overall results of the session will be updated and the game will restart (players and planets will restore its initial condition). If there are no games to be played left, the session will end and a results screen will be displayed.

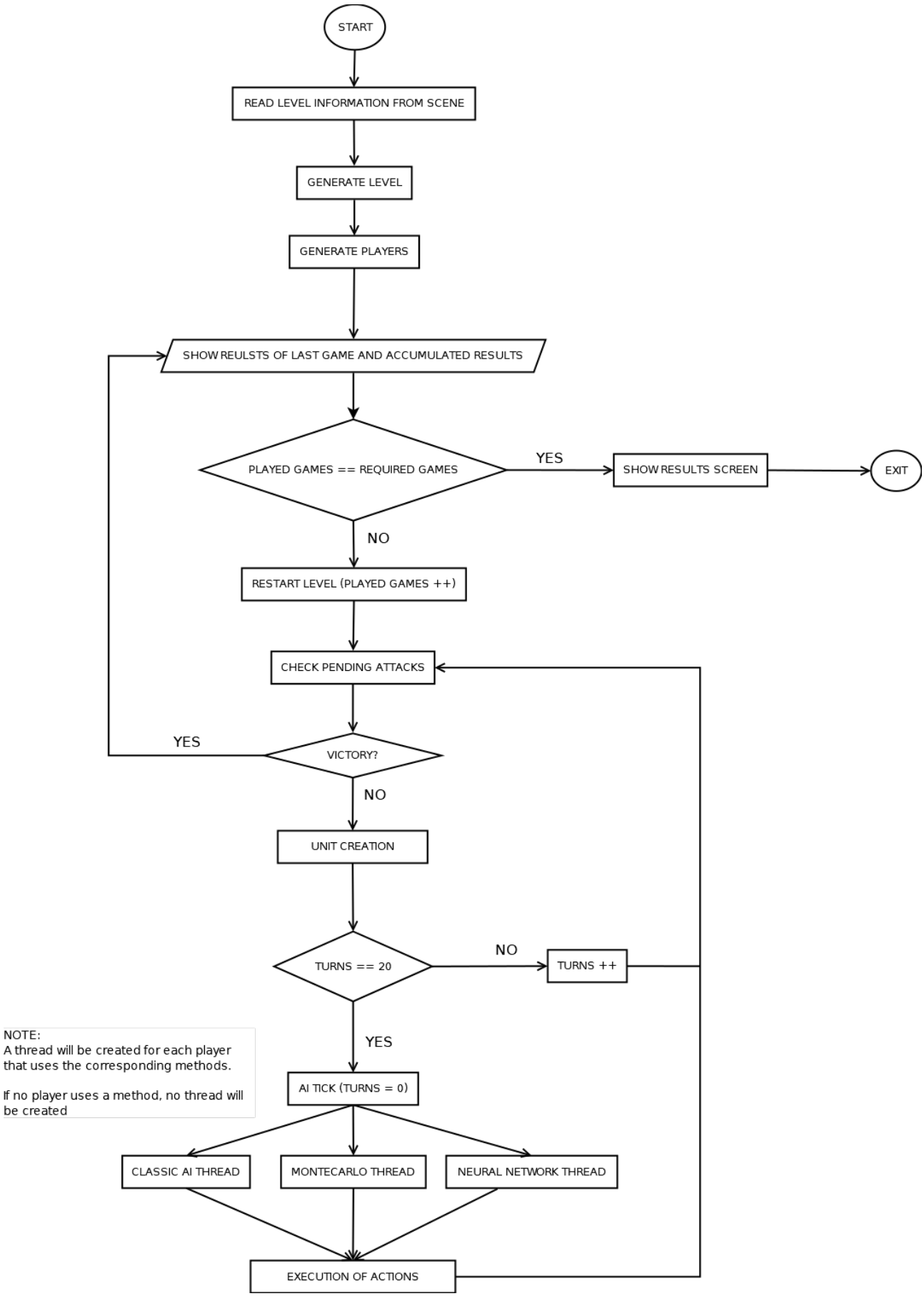

Figure 8: Training session execution flow

Classic AI

The game will also include a “classical” version of an AI. This version will use decision trees at its core and will play by the same rules as its machine-learning counterparts. The purpose for its inclusion its mainly comparative. It will be used to play against the machine-learning techniques to train them and to check its progression.

It will also be used as a test for these techniques adaptation capabilities. Once both are finished and fully trained in the case of the neural networks, this AI will be used as a reference. Hundreds of games will be played facing up the classic AI and its self-learning companions to check that the wins and looses of the classic AI reach the desired point in which the wins are slightly bigger than the looses. This will check the capacity of the Montecarlo and the neural network techniques to adapt to a player and to adjust its difficulty.

Montecarlo Tree Traversal

Montecarlo tree traversal is a method that retrieves the best action that can be done given the state of the current game. It starts with a copy of the current state of the game being played and basically explores every action that can be done. It chooses a random action, executes it and then simulates the game taking random decisions until it is finished. If the player that executes the method wins, a positive score is added to that action. If it loses, a negative score will be added.

This process is repeated creating a tree. We start with the current state of the game. A random action is selected and the state after that action is taken is stored as a new node. After that, the game is simulated from that state until the end, without creating new nodes. This process is repeated exploring possibilities until the process is interrupted. Once this happens, the action with a better score is chosen.

Scores start in the node from which the simulation started and scale up the tree until they reach the root node. While this happens, the score modifies the previous scores that were stored in those nodes first, until modifying the top scores. This top scores are the ones that will be used in order to decide which action is the best one among all possibles.

There is an important thing that needs to be clarified regarding how the implementation will be done. First of all, there are to approaches regarding possible actions that can be taken:

1. There is an “attack” movement for each planet. That means that, if there are 5 planets in a level, there will be 6 possible moves (“attack” for each of the planets and wait), if the number of planets is 8, then there are 9 actions and so on.
2. There are only three options: attack an enemy, use units in the player’s planets and wait.

The first one is more precise and straight forward, but it makes the tree grow faster and, in consequence, the necessary time to obtain good results grows too. In comparison, the first option will make the tree grow at an n^x rate and the second one at an 3^x, being “n” the number of planets and “x” the tree depth.

Another aspect to take into account is the gap between states. The first approach might be to think that every turn in the training session and every tick in the normal game should be considered a state. But, since the AIs only take decisions when their ticks are triggered, the states will be the situation of the game when this events happen. This reduces the tree’s depth without affecting precision, because the AI is not able to react to what happens between AI events.

Finally, how this technique will adapt to the player levels has to be discussed. There are two main ways of avoid obtaining always the best move possible, giving the player an easier challenge. These are:

1. Introducing a chance of choosing a random action despite the results obtained (or to choose the second best one)
2. Reducing the time the algorithm has to run or the maximum number of nodes that can be generated.

By using one (or both) of these approaches, we can adjust dynamically the difficulty of the AI.

Neural Networks

Neural networks are composed of a series of nodes linked togheter. We can differ three parts: input layer, hidden layer(s) and output layers. In the input layer the values that are needed to take a decision are used. This values are transformed in the hidden layers and then the desired values are appear in the output layer. The values are transformed using numbers that are obtained though training sessions using genetic algorithms. These algorithms will start by generating random numbers and selecting the combinations that obtain the best results.

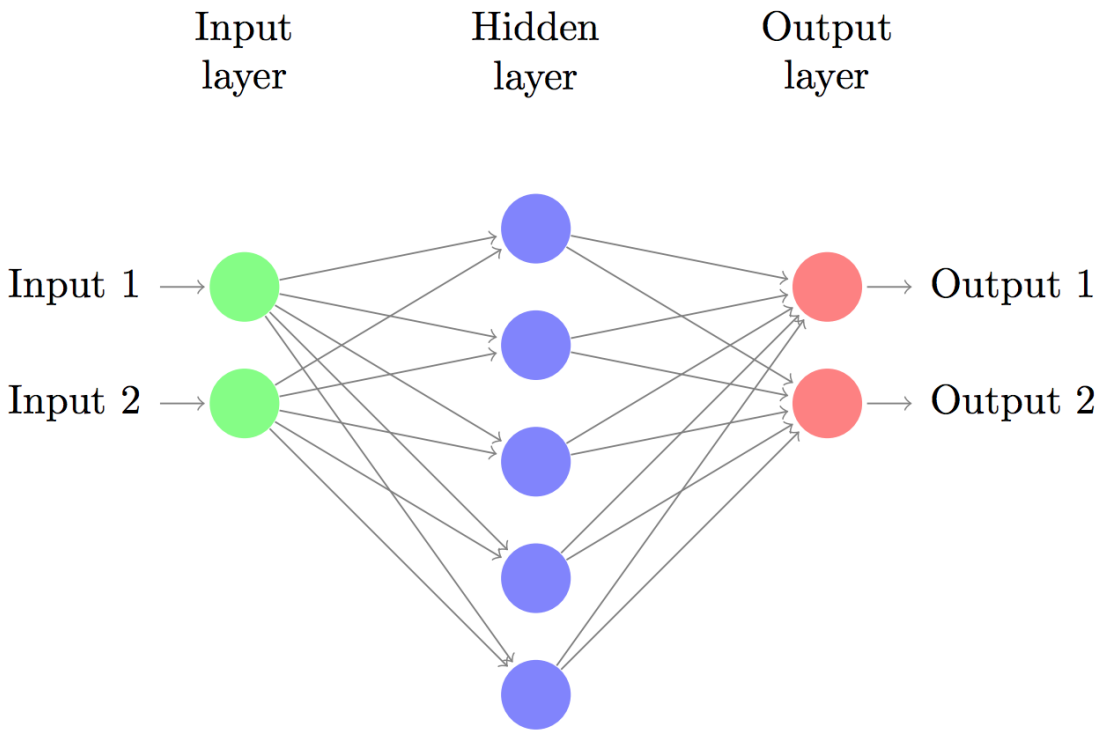

Figure 9: Example of a neural network

Something that has to be taken into account about this technique is that, unlike Montecarlo Tree Traversal, neural networks need to be trained, as it has already been said. That means that the proposed solution for the number of actions that can be taken has to be reduced to three: “attack”, “improve” and “wait”. Moreover, the entry values cannot be related to the planets or the other players directly, because then they will depend on the level and a network won’t be valid for more than one specific level. Instead, general values that include all the planets or enemies have to be used, because they are independent of the actual number of planets or levels. For instance, we cannot use variables like “number of planets of player 1”, because it depends on a singular player that might or might not be there in another level. “Enemy unit generation ratio“ will be a good alternative, because it is independent of the number of players and planets.

To adjust this technique to the player’s skill level, again, two different approaches can be taken:

1. Like int the Montecarlo case, a randome chance of choosing the non-optimal movement can be introduced.
2. The sequence of numbers that determines the best action can be altered by a percentage that depends on the player performance.