“ALEXANDRU IOAN CUZA” UNIVERSITY OF IAŞI

FACULTY OF COMPUTER SCIENCE

THESIS

Using Genetic Algorithms as a Tool of Iteration of Finite State Machine Behaviors in Video Games

proposed by

Maftei Claudiu Ioan

Session: February 2020

Scientific Coordinator:

Prof. Dr. Henri Luchian

"ALEXANDRU IOAN CUZA" UNIVERSITY OF IAŞI

FACULTY OF COMPUTER SCIENCE

Using Genetic Algorithms as a Tool of Iteration of Finite State Machine Behaviors in Video Games

Maftei Claudiu Ioan

Session: February 2020

Scientific Coordinator:

Prof. Dr. Henri Luchian

Avizat,

Îndrumător Lucrare

Titlul, Numele şi prenumele: **Prof. Dr. Henri Luchian**

Data: **4 februarie 2020** Semnătura _____

DECLARAŢIE

privind originalitatea conţinutului lucrării de licenţă/disertaţie

Subsemntatul **Maftei L. Claudiu Ioan** domiciliul în **Municipiul Iași, Județul Iași** născut la data de **12.11.1997**, identificat prin CNP **1971112226808**, absolvent al Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de **Informatică** specializarea **Informatică**, promoţia **2016-2019**, declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul:

Using Genetic Algorithms as a Tool of Iteration of Finite State Machine Behaviors in Video Games

elaborată sub îndrumarea **Prof. Dr. Henri Luchian**, pe care urmează să o susţin în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă/disertaţie să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, **4 februarie 2020** Semnătură student _____

DECLARAŢIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Using Genetic Algorithms as a Tool of Iteration of Finite State Machine Behaviors in Video Games*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, **4 februarie 2020** Semnătură student _____

Table of Contents

1. Thesis Structure

The Motivation section details the minimum context necessary to be known as to argue the utility of this work, as well as the reasons it was done.

The Overview section details the research-related goals and issues, the manner in which they are addressed, as well as the domains in which the contribution of this work may be valuable.

The Related Work section mentions brief descriptions of works that have points possessing either similarity or ability to be referenced.

The Engine section offers a summary regarding the workings of the Godot 3.1 engine, as well as arguments for its choice.

The Game section offers a detailed description of the game components made for this work, in association with the design choices that drove their making (where applicable).

The Finite State Machines section offers a summary of their concept, alongside its usage in game AI.

The Comparisons with Other Forms of Behavior Selection Algorithms section compares the aforementioned approach to others, in terms of both their usage in game AI, as well as in relation to genetic algorithms.

The Finite State Machine Editor section offers a description of the tool developed to easily create Finite State Machines, as well as touching on the process of its making.

The Genetic Algorithms section summarizes the genetic algorithm concept.

The Creating a Genetic Algorithm for FSM Behaviors section details the methods and arguments regarding the implementation of FSMs evolved via GAs.

The Running the Experiment section showcases the results of using the previously detailed method.

The Conclusions section details the results of the experiment, as well as how this work may be further extended.

2. Motivation

2.1 Video Game Agents

Video games, as a medium, are interactive models of experience, differing from other forms of media in the ability of the audience to alter the content and/or content consumption; and differing from other acts of play in its rigid computational imposure of rules [1]. As such, a game provides a dynamic environment (i.e. game-state), the acting components of which are agents.

Agents are defined by their ability to perceive and change their environment [2]. The most common agent, requisite to a game, is the player-controller – the vehicle translating verbs (viz. physical interactions with the keyboard, mouse, microphone etc.) into in-game actions [3]. Video games popularized non-player agents, most commonly in the form of non-player characters (NPCs). Their control is assigned to, what is commonly referred to as, in spite of its at times limited scope, artificial intelligence (AI).

The main distinction between the designs of an agent in and out of ludic contexts lies in its goals: generally, an agent purports the maximization of success, whereas agents in games serve the maximization of player enjoyment [4] – distinction very much relevant in its translation to, what will be later coined as, fitness. An agent may have its goal scoring victory against the player in a shooter game, yet the most efficient agent is seldom desirable, due to the characteristics of skill vs. challenge (see flow [5]). The enjoyment itself is varied phenomenologically, ranging from empathy, proof of expertise, and social satisfaction, to tension, uncanny, and horror.

Regardless of this discrepancy, video game agent designs in terms of their mechanics, as opposed to their aesthetics (viz. emotional responses evoked in the player) [6] are scarcely specific. The most popular approaches are finite-state machines (hierarchical or otherwise) (FSM and HFSM respectively), behavior trees, utility systems, goal-oriented action planning (GOAP), and hierarchical task networks (HTN) [7]. In-detail comparisons are made in Comparisons with Other Forms of Behavior Decision Algorithms.

2.2 Enemy NPC Behavior Design

The manner in which enemy NPC behaviors are usually designed is by giving the AI designers tools that allow ease of creation and modification of behaviors. Those behaviors are then tested by either/both themselves and testers; the results of which lead to further iteration.

Given the manual nature of the process, there can be cases in which the behavior does not perform the task optimally (or as optimally as desired). Two approaches that may fix this are the generation of behaviors from scratch, and the optimization of already existent ones.

To be noted that there is a difference between designing intelligent agents and fun agents. In the case of agents that have gameplay options translatable to the player ones, which is to say that the AI supplies the actions for an agent that is otherwise supplied by a player, intelligence

(or at least, the appearance of it) is desired. Player vs. Player Multiplayer games employ AI that is able to mimic player behavior – in those cases, smartness and fun are tangent.

However, this work refers to agents that are meant to differ from the player one, designed to offer engaging obstacles.

2.3 Multi-Purposed Thesis

The first purpose of this work is the experimentation of using genetic algorithms as means of solving the aforementioned issue (i.e. the reaching of as-optimal-as-possible behaviors); thus offering a tool which AI designers can use in the iterative processes of designing enemy NPC behaviors.

Given that the work delves partially in the area of game design, some of the design choices behind the game made for the purpose of this work are explained.

The second purpose is the showcase of a crystallized, so to speak, translation of the knowledge and skills attained in undergraduate studies. As such, the aspects regarding the engineering of the game itself are either detailed or at least mentioned. The repository can be found at the given link.

3. Overview

3.1 Goals and Research Questions

1. *What is the manner in which genetic algorithms can be used to evolve FSM behaviors?*

The center goal is the finding of a technique which can be used to evolve FSM behaviors via genetic algorithms. The raised question is not a 'why', but a 'how', as such details regarding the theoretical reasoning as to what properties do FSM possess that allow their ability to be evolved by genetic algorithms are not discussed.

2. *What are the manners in which the above-proposed technique can be useful, if at all?*

The two ways in which the technique may prove of use are: generation from scratch of behaviors or iteration of behaviors. As the title of thesis implies, the answer is the second.

3. *What differentiates a human-designed behavior from a computer-designed one?*

This issues will arise at two particular points: how one may model a FSM phenotype into a chromosome and how their ability to fulfill their purpose can be measured.

3.2 Research Method

As to investigate the aforementioned directions, a top-down shooter game was developed; within it, enemy agents that are controlled by FSMs. For ease of development, a FSM Editor tool was developed.

In terms of the genetic algorithm features, a method of encoding and decoding, alongside various approaches of genetic operators and parameters, were developed.

Experiments showcasing different behavioral situations were run, the results of which are detailed. Those are analyzed in order to answer the research questions.

The Conclusion section contains further inquiries.

3.3 Contributions

The first domain that this work contributes to is Evolutionary Algorithms, given that it treats techniques and data regarding the usage of a genetic algorithm.

The second domain that this work contributes to is Game AI, given that the very same techniques and data offer insight into Game AI iteration.

4. Related Work

Evolving Behavior Trees: Automatic Generation of AI Opponents for Real-Time Strategy Games [10] shares a high tangency of scope. The work details the usage of genetic programming to generate enemy AI behaviors in the form of behavior trees. Despite the core differences that are: its different behavior selection algorithm, as well as its design-related problem-to-solve (as the paper refers to strategy-making), it touches upon topics that will relate, albeit in a different manner, to some of the ones this thesis touches.

Genetic Algorithm with Variable Length Chromosomes for Network Intrusion Detection [11] has a self-explanatory title. In its implementation, a chromosome represents a set of rules. It is used as a point of reference to showcase the difference between homogenous genes and the lack-there-of.

Genetic Algorithms for Drawing Bipartite Graphs [12] purports a genetic algorithm solution for the level permutation problem. Its crossover and mutation genetic operators are referenced and the differentiation between the context of a bipartite graph and a FSM behavior is detailed.

JavaGenes: Evolving Graphs with Crossover [13] developed a crossover operator for directed and undirected graphs. It suggests that its software is able to evolve pharmaceutical drug molecules and simple digital circuits. Some of the techniques portrayed in this thesis are either referred to or derived from for the purpose of this work. Its software is released under the terms and conditions of the NASA Open Source Agreement (NOSA) Version 1.1 or later.

Variable Length Genomes for Evolutionary Algorithms [14] discusses the aspects of convergence and generality of the “exG” genome developed. It is a point of reference in terms of the nature of positionality of genes.

5. The Engine

5.1 Overview

The sample game is created using the Godot 3.1 engine [8]. It is an open-source framework; as of recent gaining popularity among indie game developers.

It has its own scripting language (i.e. GDScript), which is a high-level dynamically typed (also able to be static) programming language, with syntax akin to Python. It has native support for C++, C#, Rust, Nim, and D; and community support for Python, Lua and Squirrel. Alike other competitors, it fits visual programming as well.

Its graphics engine is able to use both GLSL 3 and GLSL 2, both of which feature ability to display 2D and a 3D environments.

5.2 Reasoning

A particular advantage that favored usage is its pure 2D engine, which is not featured by competitors such as Unity and Unreal (the 2D of which is simulated in a 3D environment). This allows easier development for a 2D ¾ shooter.

Another particular advantage that favored its usage is object-oriented programming (OOP); unlike GameMaker: Studio (1.4 and 2.x), which, in spite of usage of inheritable objects, lacks a varia of common OOP functionalities. It, as of 2017, is no longer fully accessible freely, thus defeating the purpose of openness this thesis intends.

A third argument, to support the previous in terms of accessibility, is its light weight: the engine is ~51 MB.

The main argument in the usage of Godot is the speed at which a prototype can be made, due to the ability to create custom tools, live editing the game, and built-in nodes; favoring thus the reuse of already written code.

5.3 Significant Features

The game's architecture is a tree of nodes, building blocks of the game. Each node may contain a script, the code of which is run if able (be it on instantiation, on a frame update etc.). Nodes automatically run the code and inherit the properties of the hyperclasses they extend (not to be confused with node-parents). Examples of nodes used in the game include sprites, kinematic objects, viewports and collision shapes. As to differentiate, a playable character may have its sprite and collision box as children nodes, but the sprite and the collision box do not inherit the code of the character – rather, it can be intuitively regarded as a form of ownership.

A scene is a tree of nodes that is runnable. To illustrate, in Figure 1 World is the root scene, its descendants layer_bg and layer_chars, each with their own descendants. Figure 2 features the editable visual representation of the same scene.

Figure 1 – Scene Tree Architecture

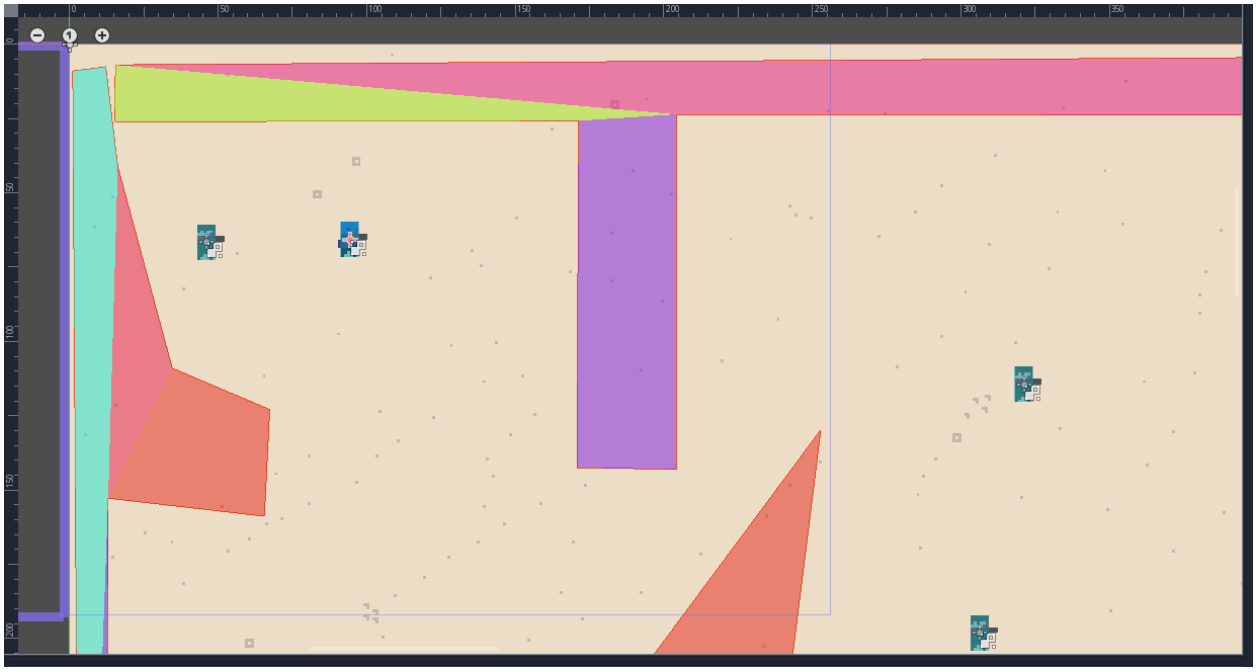

Figure 2 - Visual Representation of a Scene

Godot's editor enables live editing. The ability to modify both parameters (which can be exported/exposed in the editor, for ease of access) and code on the run facilitates testing.

A node can be converted to a tool, by the simple addition of the tool keyword in its script. Tools are run by the editor itself, not only upon the separate running of the executable. This allows the dynamic customization of what can be achieved with the editor, through the usage of the very scripting language that is otherwise used coding the game itself. This particular feature is further referenced in the Finite State Machine Editor section.

Another (albeit more obscure) feature is the ability to overwrite the set and get functions for the member variables of any given object.

The complete documentation can be found here [9].

6. The Game

6.1 Design

6.1.1 Gameplay Design

The game is a 2D ¾ perspective shooter (mechanically similar to a top-down shooter).

The player character is moveable by WASD, can reload by pressing R, aim with the cursor, and shoot by pressing the left click. The movement features acceleration and deceleration, the reload time is short, and the bullet is generated instantly; this simple-to-master control scheme does not only suffice, but it focuses the shift from the mastery of the character to the observation of enemy behavior.

There is a lack of gatherables, both in terms of ammo and replenishable health. This decision was made to emphasize the dynamic action at play; another argument is the out-of-scope nature of such a mechanic, given the purpose of this work.

The game features a single enemy type, as just to fulfill the minimum requirement for the finite state machine's behavior's proxy.

6.1.2 Polish

Particular emphasis is put on polish (i.e. subtleties that are cumulative in the nature of their impression upon the player). Short-term feedback is a form of polish, the role of which is empowering the feel of having made a choice, as well as evoking liveliness through interactions between objects. Particle effects on shooting (the muzzle flash) and on hitting elongate the purposefulness of the shot, by highlighting its start and final destination. Effects on hit (whitening) are a confirmation of damage.

Knockbacks have the general role of adding weight. Shots fired knockback the character in the opposite directions of the hit, whilst taking a shot knockbacks the character in the direction of the shot. The slight movement purports not merely a psychological effect, but a mechanical one as well, as it directly influences player decisions and the outcome of enemy behaviors.

Camera related polish effects have their effectiveness rely the on their property of being the very representation (to be read, world view) of the player. Camera shake is analogous to a literal world-shake – an emphasis of the importance of the event that happened (i.e. bullet hitting something, explosion, corpse splattering into gibs). The camera kick has a similar benefit, with the added feature of directionality – the kicking back of the camera in the opposite direction of a shot further accentuates the sense of weight and inertia of the bullet.

Freeze frames have a highly debated usage. The influence they offer is subconscious, as it allows the player to absorb more information, whilst not consciously observable. If made observable, its impact reaches the premise of the previous one, analogous to a world-freeze.

The flowers wiggle when moved by characters, bullets and explosions. In general, slight movements are used as a tool for evoking liveliness; particularly, it causes the directionality of the dynamic object to linger.

Long-term feedback is another form of polish, the role of which is facilitating the remembrance of events past. Blood splatters upon hit, and the explosions leave behind a dangerous seeming green and dark gray. The most effective form of permanence is the transformation of acting agents into corpse props upon death; the bullets apply their visual effects and knockback when hitting one, thus adding to their weight and veracity. There is a chance that, upon being hit, a corpse explodes into an array of gibs, which in turn can be moved by bullets; this feature further enhances the visceral sense of permanence of action and effect of (mostly intended) violence.

6.1.3 Graphic Design

The graphics are 2D pixel art. The art style was chosen based on both its aesthetics (evoking of retro and "gameyness") and its (relative) ease of production. The high contrast palette favors the quick recognition of characters, further favoring the observation of enemy behavior patterns.

Tweening, also called easing, more widely known as interpolation, adds a sense of natural transitioning; as such it is used in mechanical movement, and in that which appears as movement (bobbing, the wiggling of the flowers).

The weapons lag behind the character when moving, adding thus a feel of weight. The weapon is kicked back when shooting, and kicked forth when reloading, as to contour the aforementioned inertia of action.

The weapons in the possession of enemies bob up and down, mimicking breath – an explicit mark of liveliness.

Perhaps the most obvious of particularities is the surrealist weapon-wielding. Shells fly right out of the UI, and weapons are thrown when reloaded, presumably equipping a copy of the weapon with full clip. Both objects lay on the ground, purporting the before-presented permanence design pillar.

6.2 Architecture

A general direction of design (as not to call it a philosophy) is the minimization of code in the favor of generation and modifiability through the editor. As such, many variables that are often customized are exported (made editable through the editor). In regards to the scope of this work, this feature was of particular use when making the bullets and weapons, as the Inspector tab facilitated their ease of making.

6.2.1 Scenes

Scenes are runnable nodes, the children of which are objects partaking in the scene. As the documentation phrases, to run a game is to run a scene.

The main scene of the game is test.tscn, containing (in the order of draw) the background layer, tilemaps for the floors and the benches, the character layer (further containing the player character, a number of enemies, and the props; the draw order of which is Y-sorted), the bullets layer (which initially contains no instance; it becomes populated with bullets and visual effects (VFX)), the camera, a tilemap for the walls, the static body itself for the walls, the UI layer containing the ammo_display, and the navigation map (indicating which space is moveable through for the enemy pathfinding).

The 2D Godot editor is a scene editor, scene-relatedly indicating the node hierarchy in a tab, and specific node properties in another.

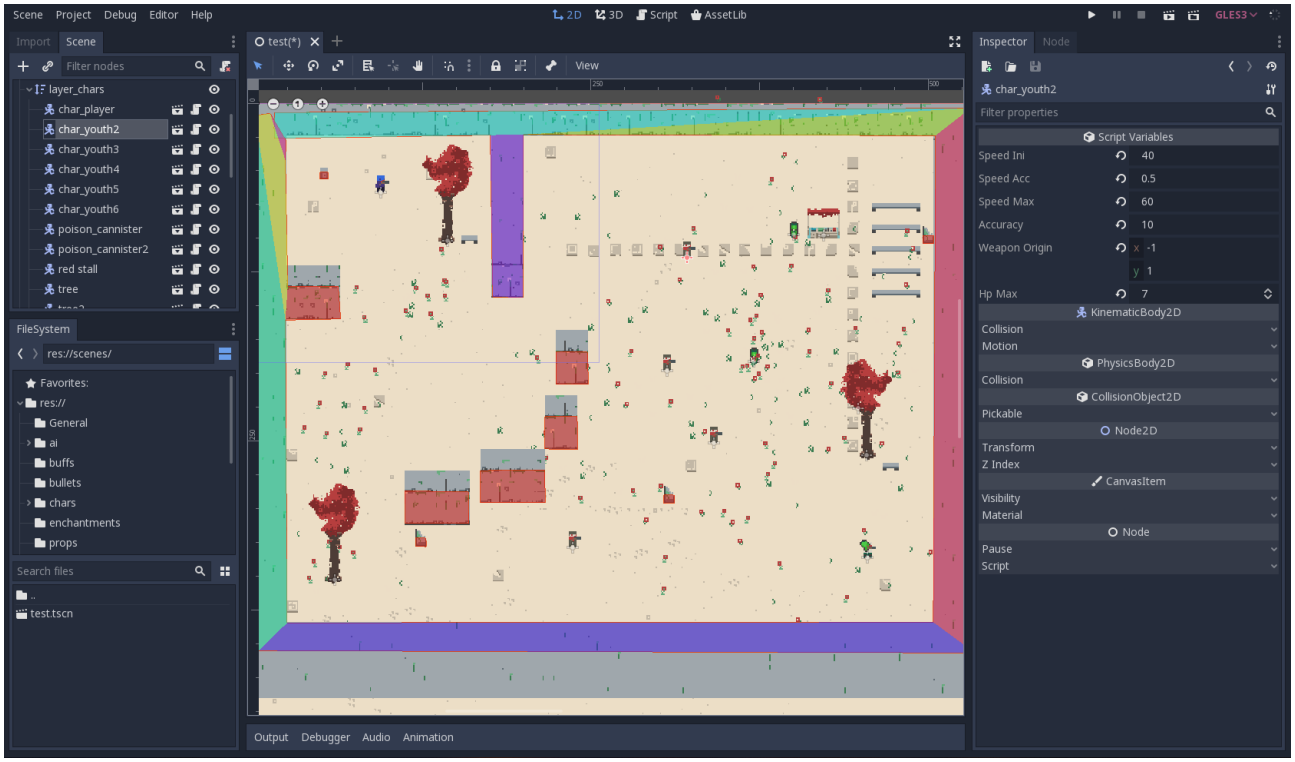

Figure 3 - Scene Editor

A feature of particular usefulness is the ability to instantiate scenes into another scene. The characters, props, and certain scene objects (as can be seen in the demo application) are scenes, which can be instantiated. In the previous figure, the layer_chars Y-sort node contains a series of characters and props, which do not have editable children nodes. In the following figure, the player character scene, instantiated above, contains the sprite of the character, the sprite of the weapon, the collision shape of the character, the area of the feet (used to wiggle the flowers) and a raycast inherited (see Characters) from the parent Char scene, which is used by the enemies for line of sight detection, but not by the player.

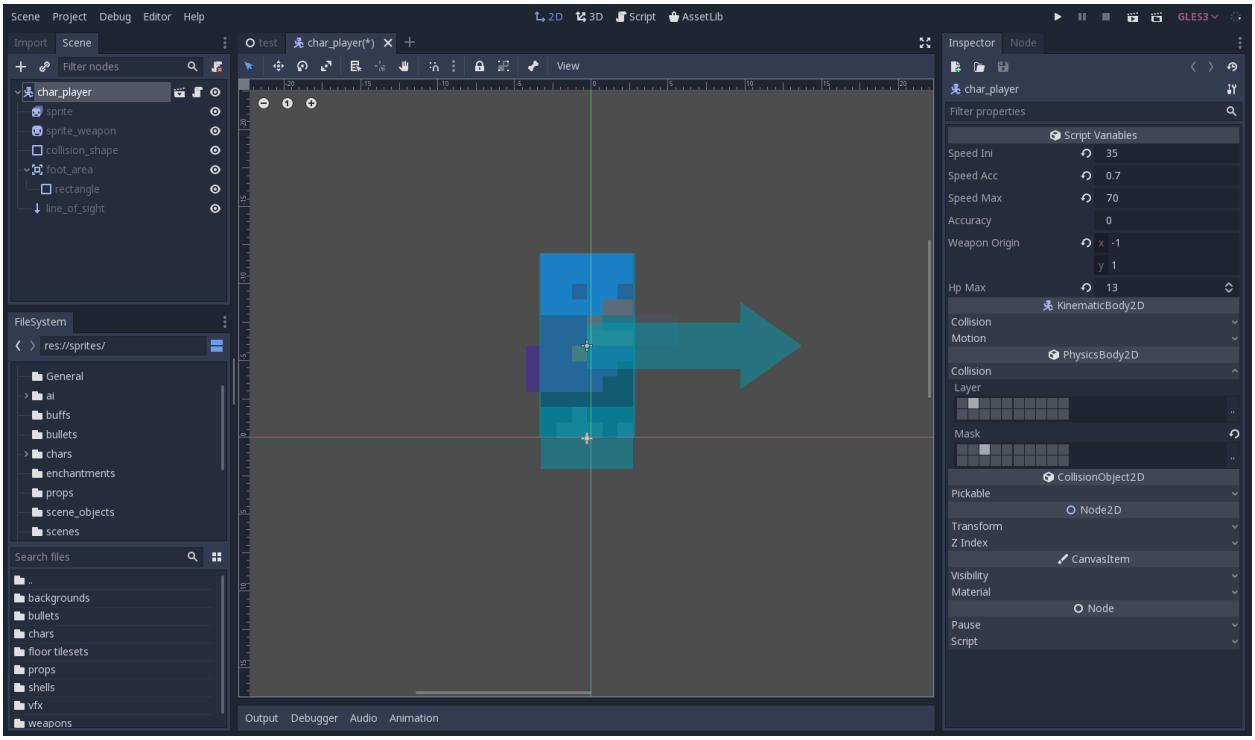

Figure 4 - Player Character Scene

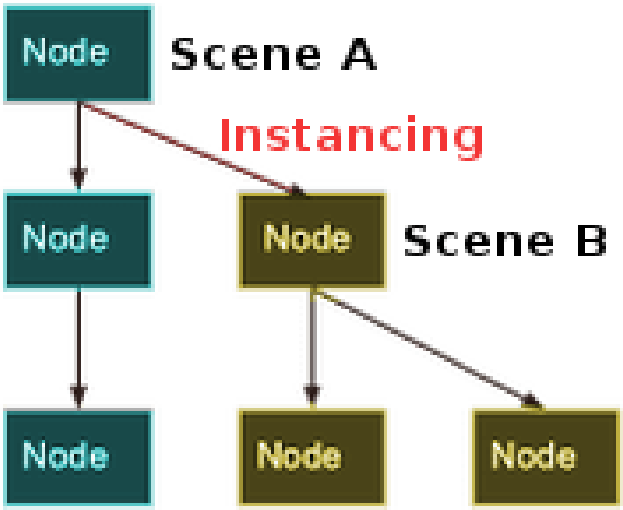

Figure 5- Instancing, as shown in the documentation [16]

Scenes can be instantiated not only in the editor, but in the code as well. The bullet scene is instantiated whenever a bullet is meant to be fired, the modified according to the weapon type, the direction of firing, and whatever other effects (such as buffs and enchantments; see Buffs; see Enchantments) may apply.

6.2.2 Inheritance

Both scenes and scripts possess inheritance. In the case of scenes, it allows the inheritance of the nodes, whilst in the case of scripts, it allows the inheritance of variables, data structures and functions.

6.2.3 Singletons

The singleton design pattern purports the identity of a class and its instance. In the demo, the Constants class contains general utilities (in terms of functions), and the widely-used reference to the collision layer enumerator.

6.2.4 Characters

Characters are kinematic bodies (vid. collision bodies meant to be controlled; they are not affected by physics). They are controllable by both players and agents.

Their movement gains acceleration in time; then it resets upon stopping. A knockback effect is applied at the end of a movement, as to simulate deceleration. The function itself is called by the knockback effect, so that the logic behind collision and flower-wiggling is kept in the same location. It is also called by the pathfinding specific to enemy characters

Characters have stats, which are Resources (vid. data containers that load only once; includes Textures, Weapons, Stats, Sound etc.). Stats (i.e. health, initial movement speed, cooldown between attacks, accuracy etc.) indicate definitory and differential character-related properties that are augmentable by an outside source – notably in the demo, the hectic enchantment.

The weapon the characters own is, as well, a resource. It contains the data for the bullet and shooting pattern. The generated bullets accounts for both the direction of the shot and the cardinal positioning based on the location of the gun as wielded in that frame (given the bobbing, weapon kick, rotation etc.).

Upon being hit by either an unfriendly bullet or an explosion, characters are damaged. The damage function, after the implicit reduction of health, generates a lingering blood sprite in the background layer, buffs the character with a white effect, and adds an aggro token if the character is an enemy.

In the case of damage causing health lower or equal to zero, the character dies, creating a corpse prop in its stead, causing fear to nearby enemies by adding a negative brave token.

An array of factions the character is part of and factions the character is enemy of necessary for the behaviors to be later described.

6.2.5 Buffs

Buffs are outer effects attachable to characters and props. Their design, inspired by the decorator design pattern, allows for widely needed and unspecific functionalities. In the demo, the knockback buff moves props and characters, the white buff draws a whitened copy of their texture (as to indicate taken damage), and the stat buff is used to increase the attack cooldown of the weapons in enemy possession, through the hectic enchantment.

6.2.6 Bullet

The bullet architecture features two distinct objects: a BulletResource and the bullet itself. The resource is a data container for the type of bullet that is shot; as such, it is highly editable through Godot's Inspector tab, as to ease the process of creating new bullets or altering already existent ones.

The bullet itself is a moveable Area2D (vid. area defined by a collision shape, able to detect collisions without having a body). Upon instantiation the bullet data is loaded based on the fired weapon; the shooter is as well added to the list of already hit objects, as to prevent further collision. A sprite child node is created to fashion an effect similar to a glow.

Whenever a body is detected within its collision boundaries, if the body was not already hit, and it is not of the same faction as the initial character, an effect is caused, the results of which are varied based on the type of object hit. In most cases a hit VFX is created, the object may be damaged, knocked back, and/or the bullet may lose one of its pierces (which is to say, numbers of objects it can still hit until destruction).

6.3.7 Enchantments

Enchantments are nodes attached to a weapon that describe additional behavior. The de facto enchantment in the demo application is the hectic enchantment which adds additional cooldown to the character shooting the weapon.

6.3.8 Props

Props are interactive objects that are not characters. In the demo, this includes corpses (knockbackable, damageable), flowers (wiggleable), gibs (knockbackable, damageable), poison canisters (collidable, damageable).

6.3.9 Background

Viewports are subviews of the scene, usable both as render targets for the draw events of the objects, as well as dynamic textures. The background sprite object generate a viewport child which attaches automatically to a separate texture rectangle (a board) used as the render target. Children added to the viewport are drawn on the board, using the settings native to the local viewport, as opposed to the innate game-specific one.

Its blend_texture function is called by other objects as to imprint a permanent texture upon the game world.

6.3.10 Camera

The camera is the interface between the game window and the viewport. Given that the game scene contains a single camera, it is automatically attributed to the root viewport. Its position is a linearly interpolated point between the player character and the cursor, as to allow a degree of freedom of view, evoking similitude with the real-life ability and choice of sight directionality. The choice of directionality offers the intrinsic benefit and disfavor of seeing ahead vs. seeing nearby.

The slight tweening between the original position of the camera and the new one has the sole purpose of adding a sense of cinematic kinesis.

6.3.11 Shaders

Godot's shading language is similar to the OpenGL Shading Language. The sole shader used in the demo is a canvas item shader that converts all RGB values to 1, thus turning a texture white whilst maintaining its original alpha values.

The shader is attached to a material, which is attached to the sprite itself.

6.3.12 VFX

The VFX objects are mostly void of mechanical interaction with the characters and the props, except for the explosion, that is an Area2D, similar in functionality to a static ephemeral bullet.

The painter is the only VFX-categorized node that is not a visually-purposed scene in and of itself. It is a moving generator of sprites, used to spawn trails of blood and green or black goo through calling the background's blend_texture function.

6.3.13 Weapons

Weapons, alike BulletResources are (as the mentioned similarity may imply) resources. They contain the data regarding the shooting patterns of the gun, as well as the effects the shots have in terms of self-knockback, camera shake and kick, VFX spawning and, nevertheless to mention, the bullet(s).

7. Finite State Machines

Finite state machines are automata defined by a list of states, the initial state, as well as transitions between the states. Within the context of game AIs, each state is (usually) associated to a behavior the agent does. The conditions that trigger a transition to fire, which is to say to transfer the quality of "activeness" to another state, compound the behavior selection feature of the algorithm.

As mentioned in Artificial Intelligence for Games [15], FSMs *make up the vast majority of decision making systems used in current games*.

7.1 States

Each state asserts a form of control over the FSM's agent, given either by its inherited (and inheritable) attributes, or by state-specific functions. A state asserts its control only if it is active.

It uses event-based programming, in the form of signals, to trigger its specific behavior.

The Aim State, upon entering causes the agent to modify its current shooting angle to face a certain chosen target, modified by an added inaccuracy. It triggers its done-signal immediately, which is to say, the very same frame the state was entered in.

The Approach State, upon entering, computes a navigation path to its chosen target. On a randomly chosen frequency within a determined interval, the path is re-computed. While active, the body's speed is calculated and the body's coordinates updated. Upon reaching the target, or upon the inexistence of the target, the done-signal is fired.

The Shoot State, upon entering, triggers the body's shoot function. The state is done immediately.

The Stand State, upon entering, resets the body's speed.

7.2 Transitions

Transitions transfer the "active" quality from one state to another.

Each transition contains a reference to an entry point, which is a state within the same FSM that the active variable is turned to false, and an exit point which is the state within the same FSM the active variable of which is turned to true. To note that its implementation allows for a distribution of exit points, however, for the sake best fitting the scope of this thesis, it is unused.

Given that the transitions are not multithreaded, their concurrency is limited by their order in the scene tree (objects that are upper in the tree execute first – which is to say, transitions that are upper in the tree trigger first).

The On-Done Transition is fired when the entry state signals a certain amount of times that it is done. Upon firing, the amount of times required is randomized (given the interval).

The On-CD (Cooldown) Transition is fired when a certain time has passed since the entry state has become active. Upon firing, the required amount of time is randomized (given the interval).

8. Comparisons with Other Forms of Behavior Decision Algorithms

8.1 Hierarchical Finite State Machines

A Hierarchical Finite State Machines (HFSM) adds hierarchically nested states, orthogonal regions and an extension of transition types. A behavior managed by HFSMs can be converted to FSMs, albeit in a more convoluted manner; as such, it can be regarded that solving the issue of using genetic algorithms on FSMs can also be extended to the same issue for HFSMs.

A direct (and more elegant) solution is approachable (see Conclusions).

8.2 Decision Trees

A decision tree is a tree subclass, the inner nodes of which are decision points, and the leaves of which are behaviors. A random decision tree (RDT) is a decision tree that contains at least one non-deterministic decision point.

8.2.1 Lack of Memory/History

FSMs, by design, allow for a direct control over the order of state-activation, equivalent thus to direct control over the order of behaviors. RDTs are history-agnostic, the previous states being unable to affect the current one outside of the means offered by the behavior itself.

As such, for the RDT to take into account previous states, the behaviors are to modify the inputs of the RDT through additional variables.

8.2.2 Ease of Design

Their highly modular nature benefits simple designs, but, present a burden when designing more complex and conditional behavioral patterns. Conjunctions and disjunctions of decisions require the repetition of leaves that fall in the false result of the statement.

Naïve implementations risk being convoluted [7].

8.2.3 Genetic Programming

Given that genetic programming is a technique used for evolving programs modeled as trees, one might argue that it is more approachable to use the very same technique for evolving behavior selection algorithms in the form of decision trees. This particular approach has been done (see Related Work).

The emphasis of differentiation is put not on the term of raw, as one might colloquially refer to as, efficiency, but in terms of scope: FSMs stand as another form of behavior selection algorithms widely used, benefitting of different approaches to enemy AI design.

Behavior trees can be modeled as FSMs and not vice-versa, as such this issue cannot be considered a subclass of the aforementioned.

8.3 Goal-Oriented Action Planners & Hierarchical Task Networks

As stated in Game AI Pro 2 – Collected Wisdom of Game AI Professionals [7], Goal-Oriented Action Planning (GOAP) is a *technique pioneered in 2005 by the developers of F.E.A.R*. It allows the AI to solve problems through chaining a list of actions, each with their own prerequisites and effects, as to find the transition from one world-state to another (that is intended by the goal).

GOAP work by backwards chaining search, whereas Hierarchical Task Networks work by forwards chaining search.

8.3.1 Reactivity vs Proactivity

FSMs are reactive in nature: given a certain world-state, a FSM chooses whether or not to fire a transition which chances the current active state (to be read: current selected behavior).

GOAPs and HTNs are proactive in nature: given a certain world-state, they find the list of actions to be performed as to react a certain desired world-state.

This is a high-level design problem which heavily affects the agent behavior design direction/philosophy.

8.3.2 Emergence

Given its mechanisms, the designer's influence over the agent's behavior is lessened, or, one may argue, diverted. The agent is able to act in unforeseeable manners through an unchecked for chaining of actions.

This can result in emergent situations which can work both in the favor and the disfavor of the designer's intended experience.

8.3.3 Rule-based

Genetic Algorithm with Variable Length Chromosomes for Network Intrusion Detection (see Related Works) presents a technique useable for the evolution of phenotypes able to be modeled as a series of rules, as is the case with GOAPs and HTNs. As such, using this form of behavior selection algorithm as the chosen form for a genetic algorithm is approachable.

Alike the nature of the differentiation mentioned in the Decision Tree subchapter, the nature of differentiation lies in the subtleties benefitting of different approaches to enemy AI design. In consideration of the disparity of the emergence-quality and the lack-there-of, a rule-based design also shifts the ontological state equaling behaviors to a more phenomenological sequence of actions equaling apparent behaviors.

9. Finite State Machine Editor

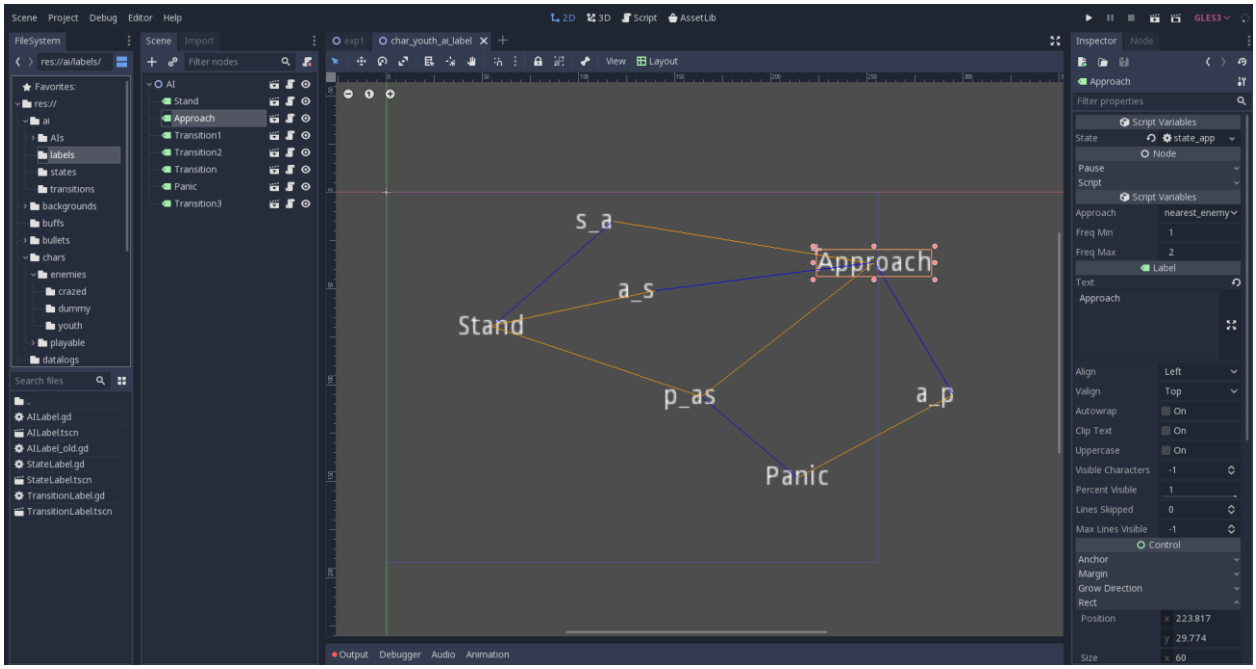

Figure 6 - Finite State Machine Editor

As to easily create FSMs for the purpose of debugging as well as having what to use genetic algorithms to iterate on, a visual editor was created. Godot's existence of the "tool" keyword which causes a scene to be runnable both in the editor and in the game itself, favors the devise of custom-tailored editors, made through means identical to any other in-game script or scene.

The AI Label scene contains States and Transitions, altogether encompassing a representation of the FSM used by the enemy characters. The initial_states array determines which states are considered initial. Note that this feature extends the FSM concept, as it allows for multiple activated states at once.

Through the file system, places and transitions can be instantiated, named, and moved around freely. Places and transitions are scenes, as such their functionality is coded separately, as to encapsulate the specificity of each component, as well as benefit from lack of clutter. Alike any other node, the instantiation and movement is done by drag and drop.

The blue lines of the transition indicate the input states, whilst the orange ones indicate output states.

As not to create a different State and Transition scene for each type, dynamic properties are used. A property is a member variable of an object. Given that each graphically-represented scene is either of a StateLabel type, or TransitionLabel type, and that the type of state/transition is referred to as an external script, their properties are not accessible.

In order to make those properties editable, which is to say visible in the editor, the functions that set and get the properties of the Label are overwritten to refer to a dictionary which copies the initial properties, but adds the properties of the desired type script as well. It is to be noted that this process was more convoluted and time-demanding than the actual creation of a Label for each type.

The editor scene, when run, generates an FSM based on the created structure and saves it as a packed scene (which can be later unpacked and encoded into a chromosome).

10. Genetic Algorithms

Genetic algorithms (GAs) are metaheuristics, subclass of evolutionary algorithms.

GAs are often used for solving optimization problems. The solution (or phenotype) is encoded as a set of variables called genes, the entireness of which is called a chromosome/genotype.

The first step is the generation of a population, that is to say a series of possible solutions in the search space.

The second is selecting which individual solutions will breed new individuals of the next generation; the means of selection vary, as well as the means of determining the worth of each solution (called a fitness function).

The third is the application of genetic operators, as to create a new generation. The two vanilla genetic operators are crossover (an exchange of the genes between two chromosomes) and mutation (modifications of individual genes).

The process is repeated until a chosen number of generations passed, until a singularity is met, a solution satisfying the desired criteria is met, or any other desired termination condition.

Metaheuristics can be applied as to improve the performance of the GA.

11. Creating a Genetic Algorithm for FSM Behaviors

11.1 Encoding/Decoding

The chromosome is an array of dictionaries, each containing the name of the node, whether or not it is initial (when referring to states), its type (to be read: script) and an array of other parameters, if any (transitions having their entry and exit points included here); thus, each gene contains all the data necessary for the creation of its corresponding state or transition.

Note: in the implementation, to know which parameters are relevant, their names are mentioned in a separate array “exportable_vars”.

This modeling benefits from cvasi-homogeny - all genes are dictionaries, and all have the same properties. This has been done as to maintain representation simplicity, as opposed to having a separation between states and transitions, a separate (heterogenous) gene containing the name(s) of the initial state(s) etc. It is a form of cvasi-homogeny as the apparent homogeny is applicable to only a single genetic operator.

The benefit of the array representing a graph, as opposed to an actual graph, is the ease of processing each individual gene.

11.2 The Continuity Problem

Given that transitions contain dependencies, meaning they require the existence of at least one particular state to be referred to within the same FSM, the genes cannot be considered homogenous.

For this reason, the vanilla point crossover, as presented in Genetic Algorithm with Variable Length Chromosomes for Network Intrusion Detection [11] is far from any use.

Variable Length Genomes for Evolutionary Algorithms [14] offers a position-related feature that depends on homogeneity, as such it also cannot be used. On the presumption that they are homogenous, the dependencies cannot be treated without converting the float index described in the paper to a graph one – as such, that approach is of no use, despite its tangency to the notion of positionality/topology.

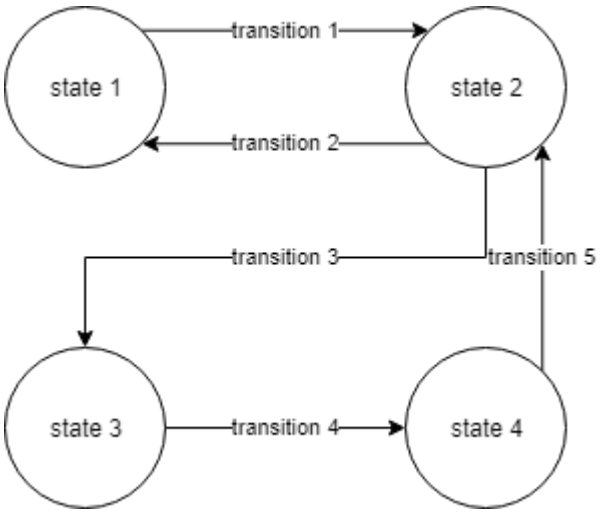

Figure 7 - An FSM Example

11.3 Crossover

Two of the crossover variants considered are disqualified given the arguments above. Genetic Algorithms for Drawing Bipartite Graphs [12] offers a crossover method that can fit the model, albeit after particular acrobatics, so to speak.

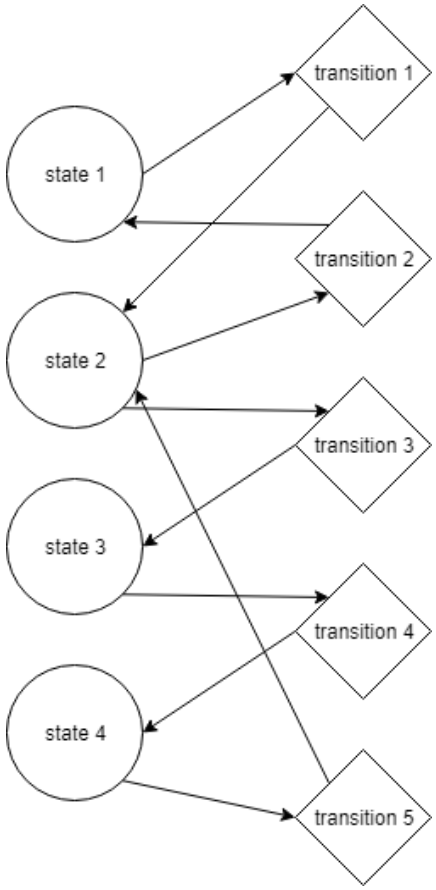

Figure 8 - The Same FSM Behavior in a Bipartite Representation

The transitions can be considered nodes as well, as such the FSM is represented as a bipartite graph. Given this setup, there are two problems that occur. The first is that the original

algorithm described in the paper is for a fixed sized chromosome. The second is that, even if adapted for variable length, it is agnostic to the topology of the graph, required for the behavioral continuity.

This approach did, however, inspire that the connectivity of a state holds importance – as it is an approximation of how "influential" that state is. As such, a connectivity score can be calculated based on the number of transitions that a certain state has; as for transitions, their "influence" depends on how "influential" their affiliated states are.

$$score(state) = \left|\{t \mid t.entry_{point} == state \text{ or } t.exit_{point} == state\}\right|$$

$$score(transition) = score\left(transition.entry_{point}\right) + score(transition.exit_{point})$$

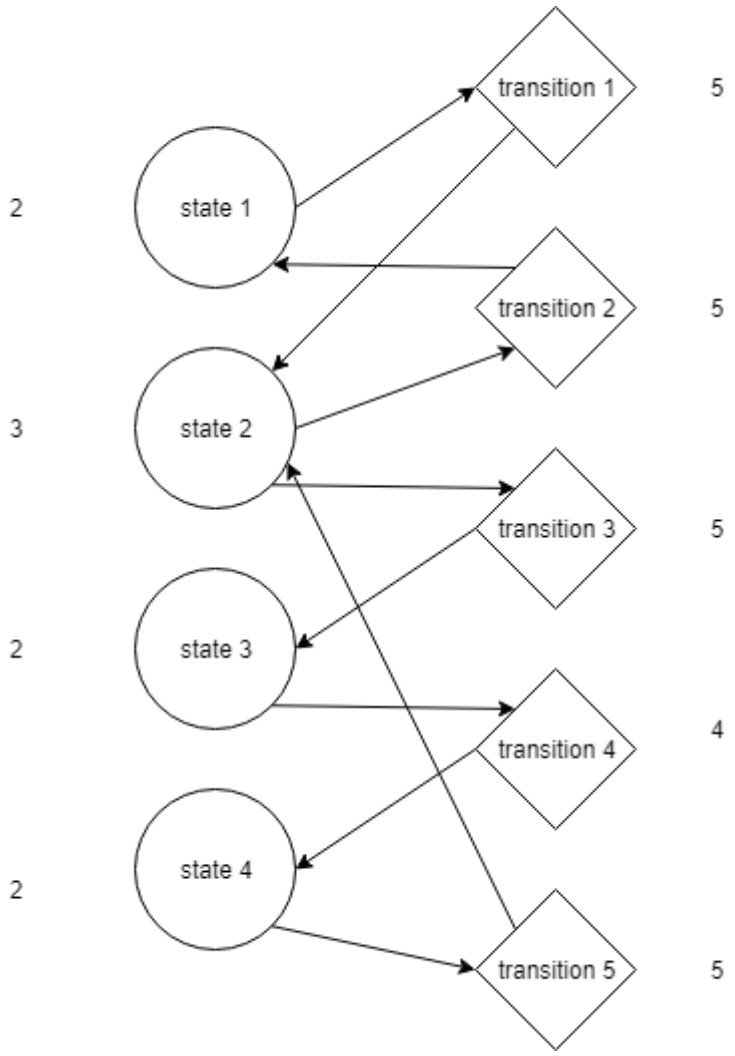

Figure 9 - Connectivity Score Example

The connectivity score does not offer a method of crossover, but rather it hints towards which parts are to be exchanged.

A variation similar to the one showcased in JavaGenes: Evolving Graphs with Crossover [13] was developed as the method of exchange between the genes.

```
states_to_swap= [ ][ ] #a two-dimensional array as there are two chromosomes, 0 and 1

transition_to_swap = [ ][ ]

elected_transition = select(transition) #based on either connectivity score or a random distribution

transitions_to_swap.append(elected_transition)

states_to_swap.append(elected_transitions.entry_point ∪ elected_transitions.exit_point)

transitions_to_swap.append(transitions_between(states_to_swap)) #all other transitions inbetween

while(extension_chance happens): #selecting a chance-based amount of transitions/states to be swapped

    t0, t1 = get_adjacent_transition(transitions_to_swap)

    if t0 is null and t1 is null: break

    if t0 is not null:

        transitions_to_swap.append(0, t0)

        states_to_swap.append(t0.entry_point ∪ t0.exit_point)

    if t1 is not null:

        transitions_to_swap.append(1, t1)

        states_to_swap.append(t1.entry_point ∪ t1.exit_point)

swap_transitions()

swap_states()

add_any_remaining_states()

add_any_remaining_transitions()
```

The extension chance is a float value between 0 and 1 that is attributed to a chance which, while it occurs, adds another set of transitions and states to the list that are meant to be swapped. The graphical explanation of the algorithm is recommended.

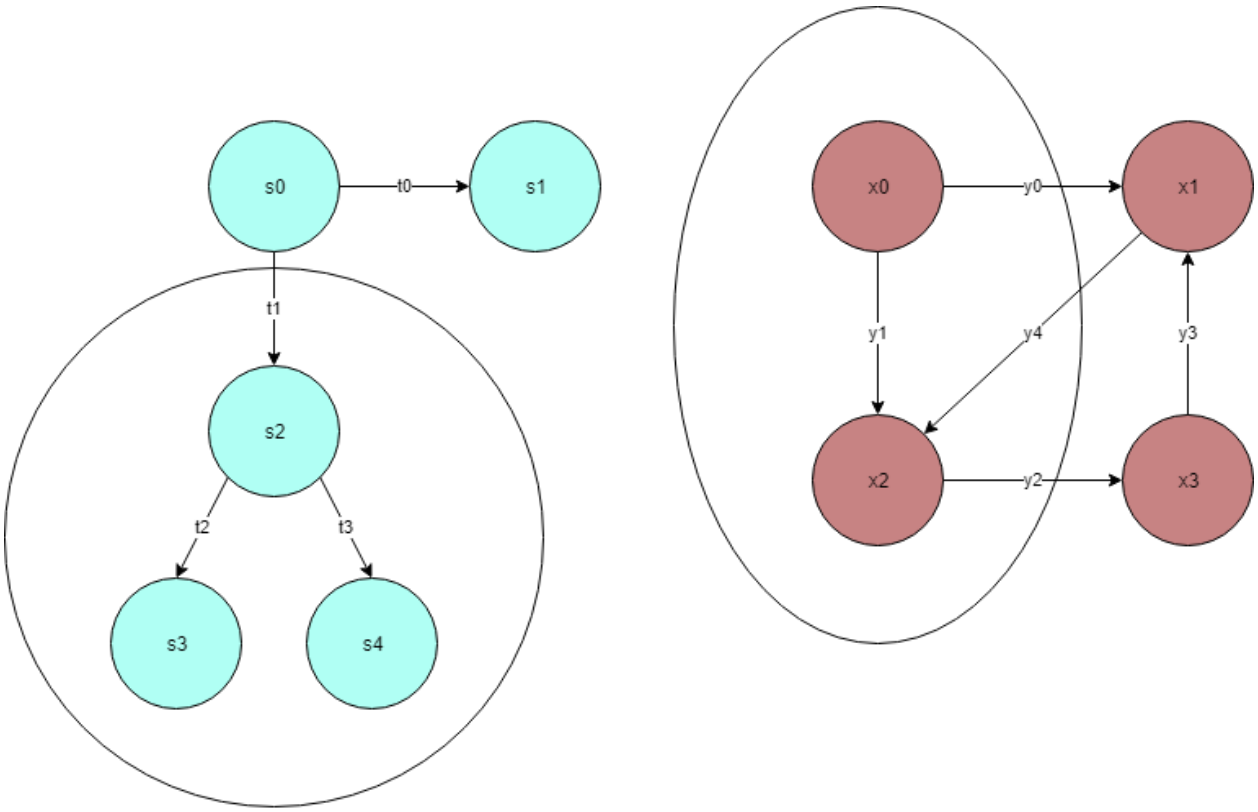

Figure 10 - Example of Parents

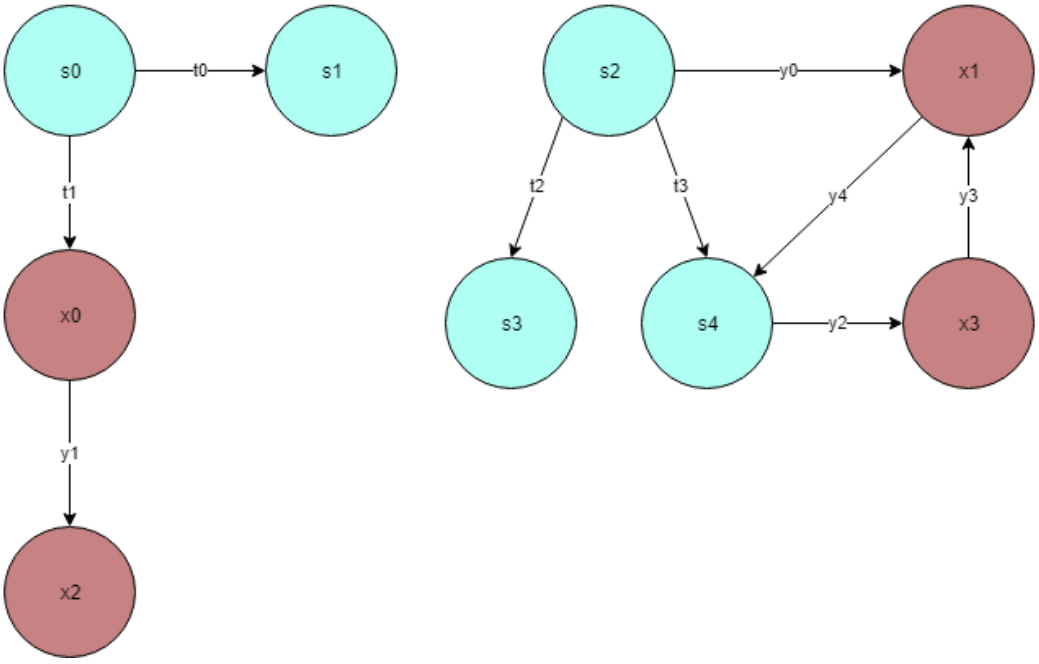

Figure 11 - Example of Crossovered Children

The main distinguishing point between the scope of the algorithm presented in the JavaGenes paper and this one is the consideration that the transitions are, in and of themselves, genes meant to be crossovered.

11.4 Hard Mutation

The distinction (between soft and hard) is necessary as FSM-related mutations and node-related mutations have a different scope.

The first type of hard mutation is the addition of a new random state or transition.

The second type of hard mutation is the removal of a random gene.

The third type of hard mutation is the changing of type of a gene, from one state type to another or one transition type to another.

The fourth type, concerning only transitions, is the change of their entry and/or exit point. Given that this mutation effect concerns only transitions, it causes the hard mutation operator to also take into account the genotype's heterogeny.

11.5 Soft Mutation

Given that each gene may contain parameters defining that particular gene's behavior, they should be mutable as to allow the full exploration of the search space. As such, a soft mutation operator is introduced which has the ability to modify float values by a desired percentile range, integers by +/- 1 and Strings by selecting another of their available options (if any).

It is the only genetic operator that is heterogeny-blind, which is to say, it is unaffected by it.

11.6 Junk Code

Non-coding DNA refers to sequences of DNA that do not encode protein sequences [16]. Alike, what used to be referred to as, "junk DNA", there are genes that do not translate into behavior. They may be transitions without states that can become active, states disconnected from the rest of the FSM, or even entire subgraphs that are disjoint.

Those non-coding or "junk" sequences of genes are not removed. They have the ability to both resurge, which is to say be crossovered or mutated in a manner that allow manifestation of their corresponding behavior, or corrupt working-coding. Given that they contain the history of the chromosome throughout generations, they minimize the risk of gene deletions causing loss of desirable traits.

12. Running the Experiment

As to run the experiment, cell-like rooms were created, which contain all the nodes necessary for running a single individual's behavior and assessing its performance. These are multiplied to obtain a population, and each is reset to obtain a new generation. The population is run concurrently.

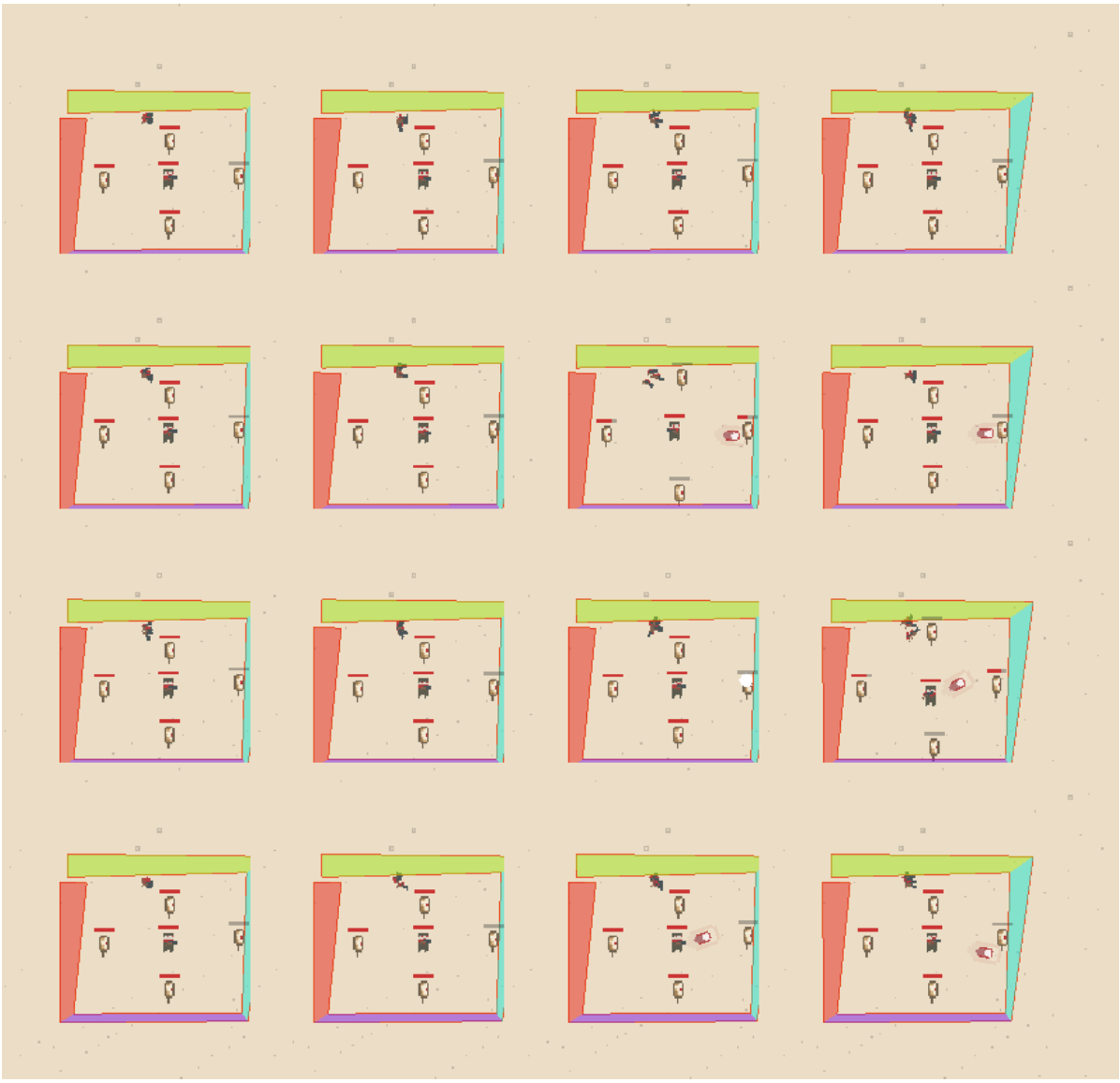

Figure 12 - Running the Experiment

The experiment is meant to find a FSM that deals as much damage as possible, in the ideal case, killing all nearby enemies, in a set time. There are an infinite amount of FSMs that can accomplish this feat; and if simplified to only their core behavior, more than one. Any FSM that evolves to cycle between aiming and shooting obtains the maximum damage dealt. In the case of a tighter time constraint, evolution of approaching the target can also be noted.

The fitness score is calculated as damage dealt to the targets. The bullets deal 3 damage, each target can take up to 12 damage, as such the maximum fitness score is 48. To note: wherever a fitness-dependent chance is 0, it is turned to 1 automatically (e.g. roulette wheel selection).

The initial population is formed of heavily mutated individuals that all contain the required states to achieve maximum fitness, but not the transitions. The pre-mutation phenomenological behavior is simply shooting, as to not rely on hard mutation to discover that shooting leads to fitness.

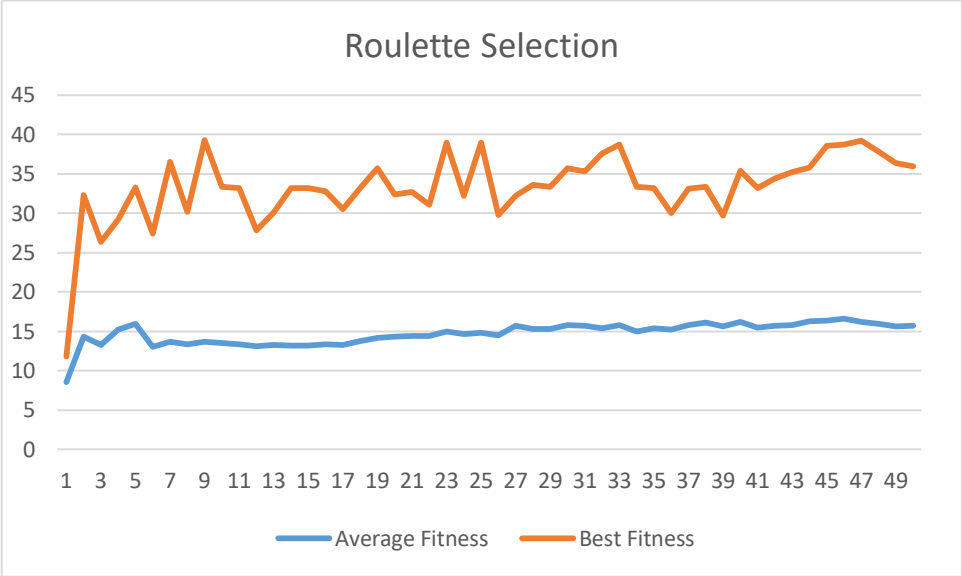

Figure 13 - Experiment 1: Roulette Selection

From the graph above one might deduce that a more elitist approach can lead to an overall better performance, as the next graph implies.

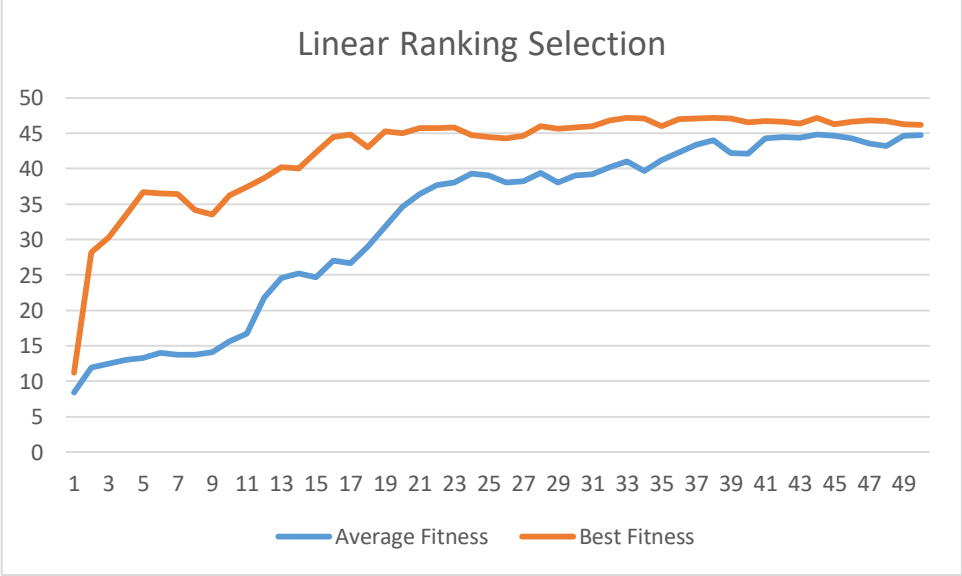

Figure 14 - Experiment 1: Linear Ranking Selection

Note that this method led to several of its iterations to reach the maximum fitness score.

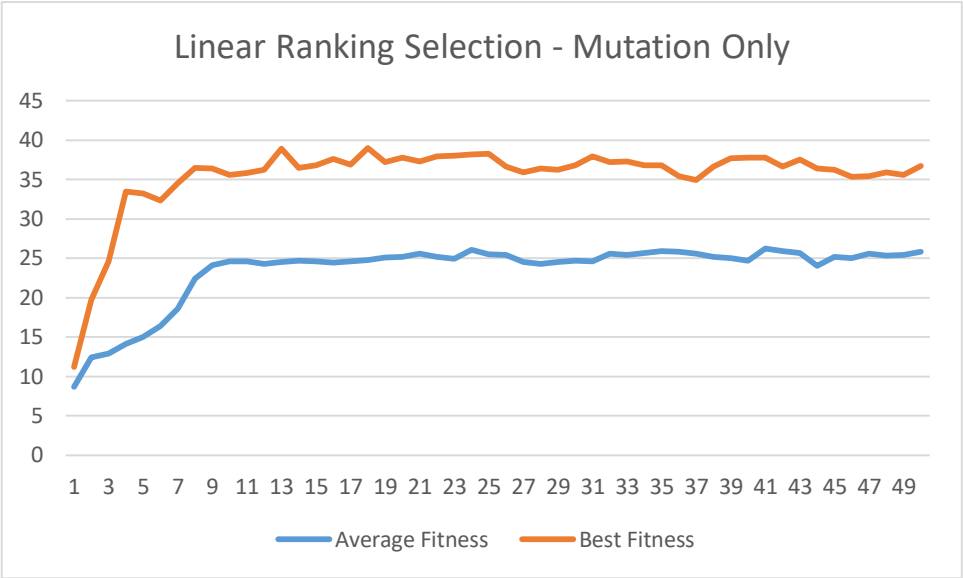

Figure 15 - Linear Ranking Selection - Mutation Only

In order to ascertain the utility of the crossover function, and as such to address this problem within the method of a genetic algorithm and not a randomized heuristic, the graph above hints towards two aspects: the first is that mutation alone does not better the fitness of the overall population; for the very same reason, the chance of occurrence of the precise required mutation is small, as such the maximum fitness does not cross the 40 line.

13. Conclusions

13.1 Results

1. What is the manner in which genetic algorithms can be used to evolve FSM behaviors?

The manner was explained in the Creating a Genetic Algorithm for FSM Behaviors and Running the Experiment sections. It is not the only manner available, but the goal of finding at least one was attained.

Given the results from the experiment, it can be concluded that the chosen parameters and genetic operators do purport results as expected of a problem solvable by GAs. Based on the fact that the crossover method chosen increases fitness over generations: the method detailed is not a mere convoluted randomized heuristic; and that the model possesses continuity translatable to behavioral sequences.

2. What are the manner in which the above-proposed technique useful, if at all?

The generation of FSM behaviors from scratch is incidence-dependent, which is to say that it requires a high amount of memory and processing time (as to allow the necessary behavior to randomly emerge).

The technique can, however, be used for the iterative design process of finding behaviors that can fulfill a certain desired role.

3. What differentiates a human-designed behavior from a computer-designed one?

In regards to non-coding (or junk) genes, they are inexistent in human-designed behaviors. A human designed behavior is also less redundant, which is to say that the number of states and transitions required for the desired behavior is (usually) minimized.

A computer-designed one can reach behavior sequences otherwise not thought about by the game AI designer.

Given the benefits of both, it is clear that computer-designed behavior selection algorithms (in the scope of this thesis) fill the niche of exploitation of the search space, as opposed to its exploration; thus, their utility as a tool of iteration of design, as opposed to direct design.

13.2 Future Work

Given that this thesis treats using GAs a tool of iterating FSM designs, a direct extension is expanding upon the nature of behavior selection algorithms that it iterates, such as HFSMs or petri nets.

The crossover operator exchanges the region of the chromosomes approximated to be of importance by its connectivity score; while the mutation operators are agnostic to the importance of a gene. A method of detecting what sequence of states and transitions leading to

the behavior that causes the increase in fitness can prove of use as a more efficient tool to manipulate the behavioral topology.

Experiments with different methods of selection, such as tournaments, 1 on 1 duels between agents, and manual selection can be employed. They are, however, outside of the scope of this work, as agent vs. agent behavior becomes heavily specialized.

References

[1] Ian Bogost (year). Introduction: Media Microecology in How to Do Things with Video Games

[2] Stuart J. Russel, Peter Norvig (1995) 2.1 Intelligent Agents - Introduction in Artificial Intelligence – A Modern Approach

[3] What Games Are (2012). The Difference between Actions and Verbs. https://www.whatgamesare.com/2012/02/the-difference-between-actions-and-verbs-game-design.html

[4] CRC Press (2014). Chapter 1: What is Game AI? in Game AI Pro 2 in Collected Wisdom of Game AI Professionals

[5] Jenova Chen. Flow in Games

[6] Robin Hunicke, Marc LeBlanc, Robert Zubek (2004). MDA: A Formal Approach to Game Design and Game Research

[7] CRC Press (2014).. Chapter 4: Behavior Selection Algorithms: An Overview in Collected Wisdom of Game AI Professionals

[8] Godot 3.1. https://godotengine.org

[9] Godot Documentation. https://docs.godotengine.org/en/3.1/index.html

[10] Hallvard Jore Christensen, Jonatan Wilhelm Hoff (2016). Evolving Behavior Trees: Automatic Generation of AI Opponents for Real-Time Strategy Games

[11] Suni Nilkanth Pawar, Rajankumar Sadashivrao Bichkar (2015). Genetic Algorithm with Variable Length Chromosomes for Network Intrusion Detection

[12] Erkki Mäkinen, Mika Sieranta (1994) : Genetic Algorithms for Drawing Bipartite Graphs

[13] NASA Technical Reports Server et al., Al Globulus, Sean Atsatt, John Lawton, Todd Wipke (2018): JavaGenes: Evolving Graphs with Crossover

[14] C.-Y Lee, E.K. Antonsson (2000): Variable Length Genomes for Evolutionary Algorithms

[15] Ian Millington, John Funge (2009). 5.3. State Machines – Artificial Intelligence for Games

[16] https://en.wikipedia.org/wiki/Non-coding_DNA