ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μελέτη και υλοποίηση δέντρων συμπεριφοράς σε περιβάλλον Unity3D

Ουρανία Γ. Σπαντίδη

Επιβλέποντες **Ιζαμπώ Καράλη,** Επίκουρη Καθηγήτρια
Βασίλης Αναστασίου, Υποψήφιος Διδάκτωρ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2016

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μελέτη και υλοποίηση δέντρων συμπεριφοράς σε περιβάλλον Unity3D

Ουρανία Γ. Σπαντίδη
Α.Μ.: 1115201000100

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Ιζαμπώ Καράλη,** Επίκουρη Καθηγήτρια
Βασίλης Αναστασίου, Υποψήφιος Διδάκτωρ

ΠΕΡΙΛΗΨΗ

Η παρούσα πτυχιακή εργασία αποσκοπεί στην υλοποίηση δέντρων συμπεριφοράς (behavior trees) στη μηχανή βιντεοπαιχνιδιών Unity3D, με σκοπό τη δημιουργία μίας εύκολης και γενικής χρήσης βιβλιοθήκης. Θα μελετηθούν και αναλυθούν οι διαφορές των δέντρων συμπεριφοράς με άλλες τεχνικές που χρησιμοποιούνται στην ανάπτυξη τεχνητής νοημοσύνης στη βιομηχανία των παιχνιδιών. Ένας ακόμη στόχος αυτής της πτυχιακής είναι η διευκόλυνση στη μελέτη και τη δημιουργία δέντρων συμπεριφοράς, με γνώμονα την υλοποιημένη βιβλιοθήκη.

Στο πρώτο κομμάτι της παρούσας πτυχιακής εργασίας θα παρουσιαστεί μια επισκόπηση της ανάπτυξης της Τεχνητής Νοημοσύνης στη βιομηχανία των βιντεοπαιχνιδιών αλλά και της σταδιακής ένταξης του συγκεκριμένου πεδίου στην ακαδημαϊκή κοινότητα. Θα γίνει επίσης σύγκριση με υπάρχουσες τεχνικές ανάπτυξης Τεχνητής Νοημοσύνης σε βιντεοπαιχνίδια, ανάλυση των πλεονεκτημάτων και των μειονεκτημάτων κάθε τεχνικής και ανασκόπηση στις δυνατότητες επεκτασιμότητας της ήδη υπάρχουσας μορφής των δέντρων συμπεριφοράς.

Στο δεύτερο κομμάτι της εργασίας, θα περιγραφεί η μηχανή παιχνιδιών Unity3D καθώς και η αρχιτεκτονική των δέντρων συμπεριφοράς, καθώς και η εφαρμογή τους σε ένα παιχνίδι Αρπαγής Σημαίας (capture the flag). Θα αναλυθεί η χρήση της εφαρμοζόμενης βιβλιοθήκης στο παιχνίδι, η αρχιτεκτονική του δέντρου συμπεριφοράς που υλοποιήθηκε, καθώς και οι δυνατότητες επέκτασης της βιβλιοθήκης αλλά και η εφαρμογή της σε διαφορετικού τύπου βιντεοπαιχνίδι.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνητή Νοημοσύνη

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: δέντρα συμπεριφοράς, Unity3D, C#, Capture the Flag

ABSTRACT

On this thesis, I have implemented a behavior tree structure on the Unity3D game engine, in order to create an easy and practical library. Differences among behavior trees and other artificial intelligence techniques used in the videogames industry will be presented and analyzed. An additional goal of this thesis is to provide a facilitation in studying and creating behavior trees, driven by the implemented library.

The first part of this thesis will provide an overview of the general growth of artificial intelligence in the videogames industry, but also the gradual integration of this field in the academic community. There will be a presentation of the different techniques used in artificial intelligence in videogames and an analysis on their advantages and disadvantages compared to behavior trees. There will also be an overview of the potential scalability of the current form of behavior trees.

The second part of this thesis will present the Unity3D game engine and the general architecture of behavior trees. Analysis of the implemented library and elaboration on its application on a Capture the Flag game will follow. Finally, possible extensions will be discussed, as well as suggested implementation patterns for different types of videogames.

SUBJECT AREA: Artificial Intelligence

KEYWORDS: behavior trees, Unity3D, C#, Capture the Flag

Στη μητέρα μου Εργίνα, στον πατέρα μου Γιώργο και στον αδερφό μου Δημήτρη, που μου έμαθαν να αγαπώ, να ονειρεύομαι και να ρισκάρω.

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω τους επιβλέποντες, επίκουρη καθηγήτρια Ιζαμπώ Καράλη και υποψήφιο διδάκτορα Βασίλη Αναστασίου για τη συνεργασία και την πολύτιμη συμβολή τους στην ολοκλήρωση της.

Περιεχόμενα

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

1. ΕΙΣΑΓΩΓΗ

1.1 Τεχνητή Νοημοσύνη και βιντεοπαιχνίδια

Η τεχνητή νοημοσύνη στα βιντεοπαιχνίδια αποσκοπεί στο να δώσει ζωή σε εικονικούς κόσμους, καθορίζοντας την αλληλεπίδραση του χρήστη με ένα παιχνίδι. Συχνά σκεφτόμαστε την τεχνητή νοημοσύνη σε ένα βιντεοπαιχνίδι ως κάτι τετριμμένο, όπως για παράδειγμα ένα ρομπότ (bot) που εκτελεί βασικές λειτουργίες όπως αποφυγή πυρών. Η δυνατότητα όμως που δίνουν οι τεχνικές τεχνητής νοημοσύνης πάει πολύ πιο μακριά από τη δημιουργία έξυπνων πρακτόρων ή πιο πειστικών ρομπότ. Όσο η τεχνητή νοημοσύνη των βιντεοπαιχνιδιών συγκλίνει με γενικές τεχνικές και έννοιες τεχνητής νοημοσύνης, μπορεί να αλλάξει ο τρόπος με τον οποίο παίζοντα τα παιχνίδια, ακόμα και τον τρόπο με τον οποίο αυτά αναπτύσσονται. [14]

1.1.1 Προκλήσεις στη σύγχρονη ΤΝ

Για τους προγραμματιστές, οι προκλήσεις πάνω στην τεχνητή νοημοσύνη είναι άρρηκτα συνδεδεμένες με την εμπειρία που έχει ο παίκτης. Ένας έξυπνος πράκτορας σε ένα βιντεοπαιχνίδι έχει να επιτύχει στόχους καθαρά ψυχαγωγικούς, αλλά και να είναι χρηστικός στους προγραμματιστές.

Στην πλειοψηφία, η τεχνητή νοημοσύνη στα παιχνίδια είναι σχεδιασμένη έτσι ώστε να είναι όσο ευφυής χρειάζεται προκειμένου να ανταποκρίνεται στις προσδοκίες του παίκτη. Για αρκετά παιχνίδια αυτό σημαίνει πως ο πράκτορας πρέπει να συμπεριφέρεται με έναν προβλέψιμο και μη ευφυή τρόπο.

Στη σύγχρονη τεχνητή νοημοσύνη γίνεται η απόπειρα να δημιουργηθούν συστήματα που αποσκοπούν στη μίμηση αληθινής βιολογικής ευφυίας. Σε αυτόν τον τομέα γίνεται εστίαση στη μάθηση (learning), μέσω της οποίας οι υπολογιστές μαθαίνουν να εκτελούν και να βελτιστοποιούν εργασίες χωρίς να έχουν προγραμματιστεί ρητά να τις εκτελέσουν.

Οι προκλήσεις σε αυτήν την πλευρά της τεχνητής νοημοσύνης περιλαμβάνουν τη δημιουργία ενός πράκτορα που μπορεί να μάθει και να προσαρμόζεται σε νέα περιβάλλοντα, δημιουργώντας έτσι εύρωστα συστήματα που καταλαβαίνουν ομιλία και ήχο, μπορούν να εξάγουν μοτίβα αλληλεπίδρασης και επίσης η ανάπτυξη πράκτορα που μπορεί να εξελιχθεί και να αναλαμβάνει μη τετριμμένες ενέργειες προς διεκπεραίωση. [1]

1.1.2 Βιομηχανία και Ακαδημαϊκή Κοινότητα

Από τη Διάσκεψη Προγραμματιστών Παιχνιδιών (Game Developers Conference - GDC) το 2002 και την ακαδημαϊκή σύνοδο της Διεθνούς Ένωσης Προγραμματιστών Παιχνιδιών (International Game Developer Association - IGDA academic summit), η IGDA έχει οργανώσει πολλαπλές συζητήσεις με ακαδημαϊκούς και ηγέτες της βιομηχανίας, ώστε να συζητήσουν πως μπορεί να γεφυρωθεί το χάσμα που τους χωρίζει. Έκτοτε, το εν λόγω “χάσμα” αποτέλεσε ένα επανεμφανιζόμενο θέμα σε πολλά

[1] Graft, K. (2015, September 22). Gamasutra: When artificial intelligence in video games becomes...artificially intelligent.

συνέδρια σχετιζόμενα με παιχνίδια, ενώ μάλιστα έχει παρουσιαστεί σε κάθε GDC έκτοτε.

Αν και υπάρχει πλήθος εμποδίων προς μια συνεργασία, αυτό δεν αφήνει να εννοηθεί πως δεν μπορεί να ακμάσει. Υπάρχουν πολυάριθμα ιδρύματα που έχουν κατοχυρώσει επιτυχημένες συμπράξεις βιομηχανίας-ακαδημαϊκής κοινότητας, το Carnegie Mellon University (Πίτσμπουργκ, Ηνωμένες Πολιτείες), το University of California (Καλιφόρνια, Ηνωμένες Πολιτείες), Massachusetts Institute of Technology (Κέμπριτζ, Ηνωμένες Πολιτείες) είναι μόνο λίγα από τα ιδρύματα που έχουν προχωρήσει σε τέτοιου είδους συνεργασίες. Κάθε ένα από αυτά έχει προσελκύσει πολλούς υψηλού κύρους συνεργάτες της βιομηχανίας με ποικίλα ερευνητικά πεδία στο ενεργητικό τους και σημαντική χρηματοδότηση.

Το 2002 παρατηρήθηκε μείωση των αιτήσεων για πανεπιστημιακά τμήματα επιστήμης υπολογιστών, και αύξηση των εσόδων της βιομηχανίας βιντεοπαιχνιδιών κατά πολλά δισεκατομμύρια. Το άμεσο αποτέλεσμα ήταν η μεγάλη ζήτηση για εργασία στον τομέα της ανάπτυξης παιχνιδιών, που είχε όμως ως προϋπόθεση εργαζόμενους με κατάρτιση στην επιστήμη των υπολογιστών. Αυτά τα δύο γεγονότα ενωμένα μαζί οδήγησαν σε μία κατάσταση όπου η βιομηχανία των βιντεοπαιχνιδιών παρουσίαζε κρίση στις προσλήψεις, αφού έψαχνε συγκεκριμένα εκπαιδευμένους εργαζόμενους που εκείνη την περίοδο χαρακτηρίζονταν ως σπάνια ομάδα ανθρώπων. Επομένως η χρυσή τομή που συνδύαζε τις ανάγκες της βιομηχανίας και της ακαδημαϊκής κοινότητας, ήταν η ίδρυση τμημάτων που θα έδιναν τη δυνατότητα απόκτησης πτυχίου που θα έφερε τον τίτλο Προγραμματισμού Παιχνιδιών.

Το University of Abertay and Dundee (Νταντί, Σκωτία) ήταν το πρώτο ίδρυμα που προσέφερε τη δυνατότητα απόκτησης τέτοιου είδους πτυχίου, γεγονός που πυροδοτήθηκε, ως άμεση συνέπεια, από τις αιτήσεις της βιομηχανίας για καλύτερα εκπαιδευμένους επαγγελματίες στην επιστήμη των υπολογιστών. [3]

Τόσο η ακαδημία όσο και η βιομηχανία οφείλουν να προβούν σε αμοιβαία κατανόηση και σεβασμό των σκοπών τους. Για παράδειγμα, μέσω κοινών ερευνητικών δραστηριοτήτων, μπορούν να προκύψουν οφέλη και για τις δύο πλευρές.

2. ΟΡΓΑΝΩΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΣΥΜΠΕΡΙΦΟΡΩΝ ΠΡΑΚΤΟΡΩΝ

Υπάρχουν ποικίλες τεχνικές υλοποίησης συμπεριφορών πρακτόρων. Η πιο ισχυρή προσέγγιση είναι αυτή της σκληρά κωδικοποιημένης λογικής (hard-coded logic), η οποία στηρίζει έναν μηχανισμό που επιτρέπει τη δημιουργία μιας εύλογης εμφάνισης ευφυίας σε έναν πράκτορα, μέσω ενός πλήθους If/Else συνθηκών. Είναι αρκετά απλή στη δομή και τη δημιουργία ως λογική, όμως φέρει και αρκετά μειονεκτήματα. Μία αλλαγή σε ένα κομμάτι κώδικα επιφέρει την αναγκαιότητα αναπροσαρμογής και όλου του υπόλοιπου. Αυτό καθιστά επίσης δύσκολη τη δομοστοιχείωση (modularity) και την αναπαραγοντοποίηση (refactoring), γεγονός που ισοπεδώνει οποιαδήποτε πιθανότητα ευελιξίας του συστήματος. Συνεπώς αν και χαρακτηρίζεται από ταχύτητα στην υλοποίηση, καταλήγει να είναι αργή σε δοκιμές (testing) αφού απαιτεί κάθε φορά επανάληψη της μεταγλώττισής της.

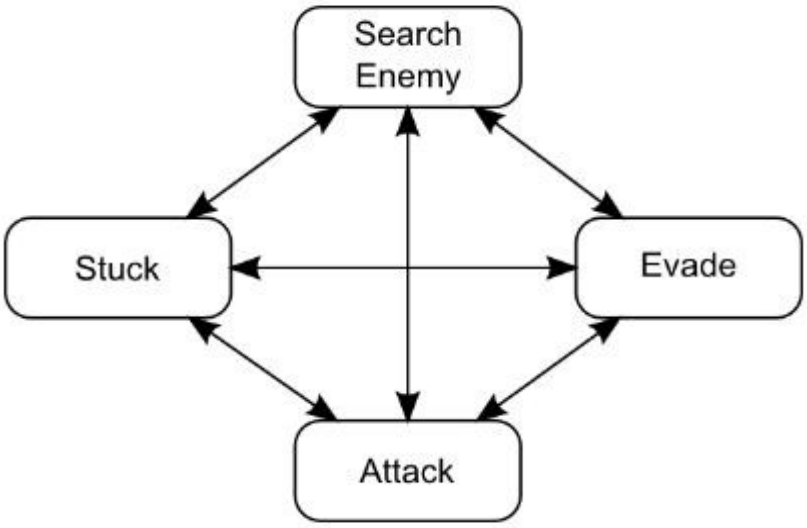
Οι πράκτορες μπορούν επίσης να υπάγονται σε μια υλοποίηση συμπεριφοράς δεσμών ενεργειών (scripts), η οποία έχει παραπλήσια συμπεριφορά με την hard-coded logic, αλλά χαρακτηρίζεται σε αντίθεση με αυτήν από γρηγορότερο testing λόγω της δυναμικής εφαρμογής των δεσμών.

Σε αυτές τις τεχνικές προστίθενται και οι μηχανές πεπερασμένων καταστάσεων (finite state machines) που αποτελούν μία γρήγορη και ευέλικτη τεχνική προγραμματισμού τεχνητής νοημοσύνης με βασικό μειονέκτημα τη μη δυνατότητα επαναχρησιμοποίησης δομών της, και επίσης τα δέντρα συμπεριφοράς (behavior trees), τα οποία εξαλείφουν τα πιθανά μειονεκτήματα των προαναφερθέντων τεχνικών ενώ ταυτόχρονα προσφέρουν επιπρόσθετες δυνατότητες.

2.1 Δέντρα Συμπεριφοράς και Μηχανές Πεπερασμένων Καταστάσεων

Οι Μηχανές πεπερασμένων καταστάσεων (Finite State Machines) θεωρούνται κλασικό εργαλείο υλοποίησης Τεχνητής Νοημοσύνης σε βιντεοπαιχνίδια. Κάθε χαρακτήρας - έξυπνος πράκτορας (intelligent agent) χαρακτηρίζεται από την ύπαρξή του σε καταστάσεις (states) και εναλλαγές αυτών. Σε αυτό το σημείο ορίζεται και η μετάβαση από κατάσταση σε κατάσταση (transition). Η αυξανόμενη πολυπλοκότητα των μεταβάσεων είναι εφικτό να διαχειριστεί από Ιεραρχικές Μηχανές Πεπερασμένων Καταστάσεων (Hierarchical Finite State Machines). [20]

Μόνο μία κατάσταση μπορεί να εκτελείται ανά πάσα στιγμή, η οποία καλείται τρέχουσα κατάσταση (current state). Μία κατάσταση παύει να χαρακτηρίζεται ως τρέχουσα, όταν μεταβιβάσει την εκτέλεση σε μία άλλη κατάσταση. Οι μεταβάσεις αυτές λαμβάνουν χώρα μόνο μέσα στην τρέχουσα κατάσταση, για παράδειγμα η παρούσα κατάσταση είναι αυτή η οποία θα αποφασίσει αν θα εμφανιστεί ή όχι μία μετάβαση.



Εικόνα 1: Γραφικό μοντέλο μίας FSM

Πηγή: http://guineashots.com/

Μία FSM είναι μία αφηρημένη μηχανή, που μπορεί να συλληφθεί και ως γραφικό μοντέλο που επιτρέπει τον εύκολο μοντελισμό των συμπεριφορών του πράκτορα, παρέχοντας τοιουτοτρόπως μία εύκολη κατανόηση του μοντέλου. Επίσης είναι υπολογιστικά αποδοτική, αφού όλες οι συνθήκες αλλαγής βρίσκονται μέσα στην τρέχουσα κατάσταση και δεν εκτελείται παράλληλα καμία άλλη κατάσταση.

Το κυρίως πρόβλημα που εμφανίζεται στις FSM, έγκειται στο πώς γίνεται επαναχρησιμοποίηση ορισμένων states σε διαφορετικό γενικό πλαίσιο (context), γεγονός που οφείλεται στο ότι η φιλοσοφία των FSM είναι χαμηλού επιπέδου με αποτέλεσμα πολλές φορές την αναγκαία υλοποίηση διπλότυπων συμπεριφοράς (behavior). Κατ' επέκταση δημιουργείται ένα ακόμη πρόβλημα, η ανικανότητα επαναξιοποίησης των FSM σε διαφορετικά βιντεοπαιχνίδια ή ακόμα και σε διαφορετικά μέρη του ίδιου βιντεοπαιχνιδιού. Απαιτείται συνεπώς προσαρμογή σε μία FSM ώστε να ανταποκρίνεται στις ειδικές περιπτώσεις ενός προβλήματος.

Σε αντίθεση με τις FSM, τα Δέντρα Συμπεριφοράς (Behavior Trees) χαρακτηρίζονται τόσο από επαναχρησιμοποίηση όσο και από βιωσιμότητα, αφού είναι δυνατή η ύπαρξη υπό - δένδρων (subtrees) που ενεργούν πρακτικά σαν μία οντότητα BT που επιστρέφει το αποτέλεσμα εκτέλεσης των child nodes της πίσω στο BT από το οποίο κλήθηκε. Υπάρχει η δυνατότητα ύπαρξης ενθυλακωμένης λογικής σε μορφή υπό - ρουτίνας (subroutine), γεγονός που επιτρέπει εν τέλει την ύπαρξη επαναχρησιμοποίησης λογικής καταστάσεων. [2]

2.1.1 Γιατί έφτασε το τέλος των FSM

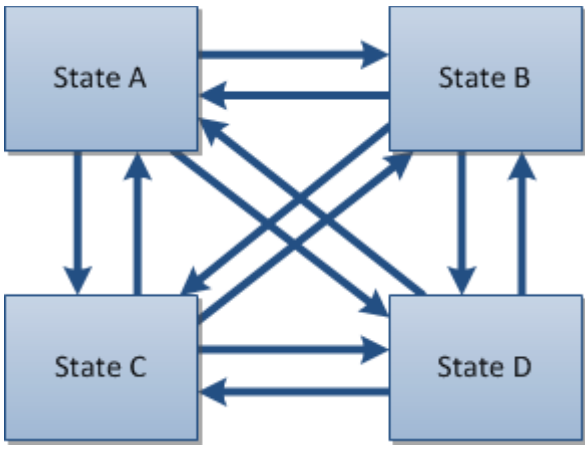
Οι FSM ήταν εξαιρετικά δημοφιλείς την περασμένη δεκαετία και οδήγησαν στην ανάπτυξη αρκετά ψυχαγωγικών παιχνιδιών. Πλέον όμως η πρώτη θέση σε προτίμηση

[2] R. D., Pereira. (2014, August 10). GuineaShots: An Introduction to Behavior Trees – Part 1.

τεχνικών ανάπτυξης τεχνητής νοημοσύνης πρέπει να παραδοθεί σε τεχνικές διαφορετικές από τις FSM, αφού αυτές χαρακτηρίζονται από αρκετά προβλήματα και μειονεκτήματα.

- Είναι ανορθόδοξες : απαιτούν τη σύνδεση μίας κατάστασης (state) με την επόμενη μέσω μίας ρητής μετάβασης.
- Είναι χαμηλού επιπέδου (low-level) : η μόνη δυνατότητα που δίνεται στον προγραμματιστή είναι να επεξεργαστεί τις μεταβάσεις από μία κατάσταση σε μία άλλη. Δεν υπάρχει κάποιος τρόπος να συγκρατήσει μοτίβα που επαναλαμβάνονται.
- Η λογική τους είναι περιορισμένη : όπως είναι επίσημα ορισμένες, είναι υπολογιστικά περιορισμένες, δηλαδή δε μπορούν να χρησιμοποιηθούν ως γενικευμένος φορμαλισμός υπολογισμών, καθώς δεν παρέχουν δομές αναδρομής.
- Απαιτούν επεκτάσεις δημιουργημένες από χρήστη : οι προγραμματιστές συνήθως χρησιμοποιούν επεκτάσεις ώστε να κάνουν τις FSM χρήσιμες στην πράξη. Οι FSM είναι θεωρητικά περιορισμένες, οπότε χρειάζονται εισαγωγή εξωτερικής λειτουργικότητας για να υλοποιήσουν συγκεκριμένα χαρακτηριστικά. Χρειάζεται επομένως μόχλευση (leverage) της υποβόσκουσας γλώσσας προγραμματισμού για την υλοποίηση οντοτήτων όπως για παράδειγμα μετρητών, χρονομέτρων, ή οποιασδήποτε μορφής μνήμη.
- Είναι δύσκολο να τυποποιηθούν : είναι πολύ δύσκολο να επαναχρησιμοποιηθούν σε πολλά παιχνίδια ή σε διαφορετικά μέρη της μηχανής (engine).
- Δεν είναι προσχεδιασμένες : οι FSM δρουν σε μία αντιδραστική λειτουργία (reactive mode) και ασχολούνται μόνο με συμβάντα και καταστάσεις. Για να αντιμετωπίσουν όλα τα πιθανά σενάρια, πρέπει να επεξεργαστούν όλες τους οι μεταβάσεις χειροκίνητα.
- Έχουν προβλήματα συγχρονισμού : όταν εκτελούνται πολλαπλές καταστάσεις σε παραλληλία, το πιθανό αποτέλεσμα είναι αδιέξοδα (deadlocks), αλλιώς πρέπει να επεξεργαστούν όλες με τρόπο τέτοιο ώστε να είναι συμβατές.
- Κλιμακώνονται πτωχά : δεν κλιμακώνονται αποδοτικά, οπότε καταλήγουν να επεξεργάζονται και να αντιμετωπίζονται ως μία μεγάλη οντότητα λογικής.
- Απαιτούν μεγάλη ένταση εργασίας : απαιτείται προσεκτική χειρωνακτική δουλειά για τη διασύνδεση καταστάσεων. Με την εμπλοκή περισσότερων καταστάσεων (states), χρειάζονται όλο και περισσότερες διασυνδέσεις καταστάσεων (transitions). [3]

[3] Champandard, A. J. (2007, December 28). AiGameDev: 10 Reasons the Age of Finite State Machines is Over.



Εικόνα 2: Μεταβάσεις μεταξύ καταστάσεων σε FSM

Πηγή: http://intrinsicalgorithm.com/

Η βιομηχανία εξελίσσεται και με την πάροδο των χρόνων όλο και λιγότεροι προγραμματιστές επιλέγουν τη χρήση FSM και στρέφονται σε άλλες εναλλακτικές ανάπτυξης τεχνητής νοημοσύνης, όπως για παράδειγμα τα BT.

2.1.2 Δέντρα Συμπεριφοράς

2.1.2.1 Δέντρα Συμπεριφοράς ως Γλώσσα Μοντελισμού

Στην ακαδημαϊκή κοινότητα, τα BT ορίζονται ως επίσημη, γραφική γλώσσα μοντελισμού (modelling language), που χρησιμοποιείται πρωτίστως σε συστήματα και τεχνολογίες λογισμικού (software engineering). Αυτά τα δέντρα χρησιμοποιούν μία καλά ορισμένη σημειογραφία (notation) για την ξεκάθαρη εκπροσώπηση των εκατοντάδων - ή ακόμα και χιλιάδων - απαιτήσεων φυσικών γλωσσών (natural language), που χρησιμοποιούνται τυπικά για την έκφραση των αναγκών πληροφοριακών συστημάτων μεγάλης κλίμακας (large-scale software-integrated system).

Στην ανάπτυξη βιντεοπαιχνιδιών, τα BT αποτελούν δομές για ανάπτυξη τεχνητής νοημοσύνης και υλοποίηση συμπεριφορών πρακτόρων. Παρακάτω γίνεται περιγραφή και ανάλυση της δομής και αρχιτεκτονικής τους.

2.1.2.2 Ισχύς των BT έναντι των FSM

Η υπεροχή των BT έναντι των FSM διαφαίνεται από πολλούς παράγοντες. Η κυριότερη είναι πως ένας agent του οποίου οι ενέργειες έχουν υλοποιηθεί σε FSM, γνωρίζει ποια ενέργεια εκτελεί εκείνη τη στιγμή και ποια ενέργεια μπορεί να εκτελέσει μετέπειτα. Η λογική πίσω από τις FSM έρχεται σε αντιπαράθεση με αυτήν που εισάγουν τα BT, όπου ο agent έχει επιπλέον βαθύτερη γνώση για το πώς να παρθεί η απόφαση της επόμενης ενέργειας.

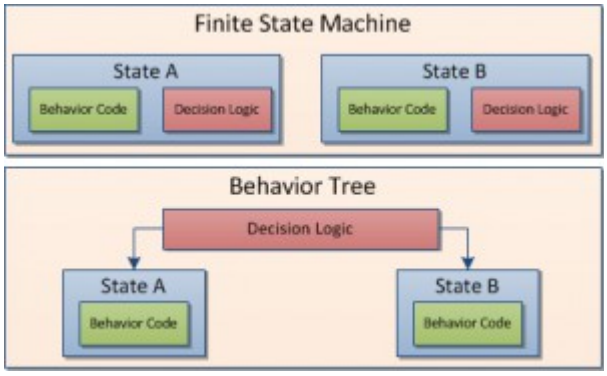
Ένα BT αποτελείται από κόμβους σε ιεραρχία, δηλαδή κόμβους-γονείς (parent nodes) συνδεδεμένους με κόμβους-παιδιά (child nodes). Μέσω των parent nodes δημιουργείται η αλληλουχία εκτέλεσης των παιδιών του. Όταν ένας κόμβος αποφανθεί το αποτέλεσμα

της εκτέλεσής του, το στέλνει στον γονέα. Αυτό το αποτέλεσμα μπορεί να είναι ένα από τα ακόλουθα: επιτυχία (success), αποτυχία (failure) και σε εκτέλεση (running).

Με βάση την περιγραφή της φιλοσοφίας τους, τα πλεονεκτήματα των BT μπορούν να απαριθμιστούν ως εξής:

- **Συντηρησιμότητα**: οι κόμβοι μπορούν να σχεδιαστούν ως ανεξάρτητοι μεταξύ τους, επομένως όταν αφαιρείται ή προστίθεται ένας κόμβος από ένα δένδρο, δεν απαιτείται κάποια αλλαγή στην υπόλοιπη δομή του.
- **Επεκτασιμότητα**: ένα δένδρο με πολλούς κόμβους μπορεί να θεωρηθεί ως σύνολο από subtrees και επομένως να μπορεί να αναπαρασταθεί γραφικά με σαφήνεια και ευκολία.
- **Επαναχρησιμοποίηση**: όπως προαναφέρθηκε, οι κόμβοι χαρακτηρίζονται από ανεξαρτησία, κατά συνέπεια και τα subtrees είναι ανεξάρτητα, οπότε και μπορούν να χρησιμοποιηθούν ξανά σε άλλα δένδρα ή και σε τελείως άλλα projects.
- **Προσανατολισμός στον στόχο**: εξαιτίας της δενδρικής δομής, ανεξάρτητοι κόμβοι μπορούν να συσχετίζονται και να συνδυάζονται, δίνοντας έτσι τη δυνατότητα σχεδιασμού subtrees για συγκεκριμένο στόχο χωρίς να χάνεται η ευελιξία του μοντέλου.
- **Παραλληλισμός**: τα BT έχουν στη διάθεσή τους κόμβους παραλληλισμού (parallel nodes) που μπορούν να εκτελούν όλους τους τους child nodes ταυτόχρονα χωρίς να χάνεται ο έλεγχος της εκτέλεσης του μοντέλου. [4]

Η δύναμη και υπεροχή των BT έγκειται στην ικανότητά τους να διαχωρίζουν καταστάσεις από συμπεριφορές. Ο κώδικας που καθορίζει τη συμπεριφορά του πράκτορα δεν υφίσταται σε επανάληψη μέσα σε κάθε κατάσταση που έχει μία FSM, αλλά η λογική είναι διαχωρισμένη σε μία ανεξάρτητη αρχιτεκτονική. Έτσι μπορεί να εκτελεστεί μόνη της, είτε συνεχόμενα είτε όταν χρειαστεί.

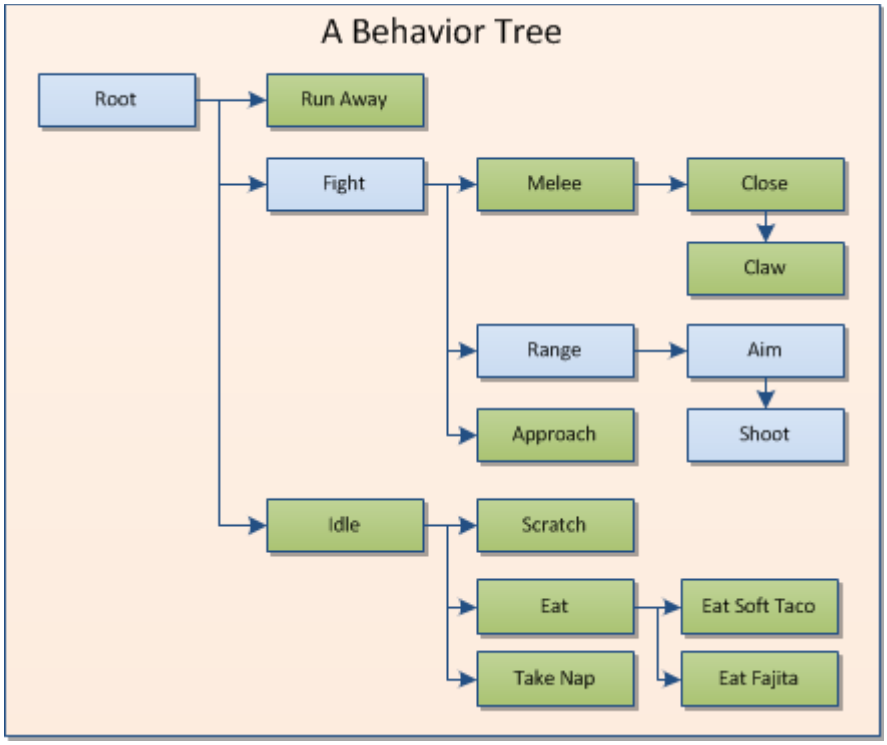


Εικόνα 3: Σύγκριση μίας FSM με ένα BT

Πηγή: http://intrinsicalgorithm.com/

[4] R. D., Pereira. (2014, August 10). GuineaShots: An Introduction to Behavior Trees – Part 2.

Το κύριο πλεονέκτημα αυτής της διαφοράς που εντοπίζεται ανάμεσα σε BT και FSM, είναι πως όλη η λογική αποφάσεων βρίσκεται σε ένα και μόνο μέρος. Μπορεί να αναπτυχθεί μία πολύπλοκη συμπεριφορά πράκτορα, χωρίς να υπάρξει πρόβλημα σχετικά με το πώς θα επιτευχθεί συγχρονισμός ανάμεσα στις διαφορετικές καταστάσεις (states). Αν γίνει προσθήκη μίας νέας συμπεριφοράς, γίνεται προσθήκη του κώδικα που θα καλέσει τη συμπεριφορά σε ένα μέρος, αντί να γίνει προσθήκη σε κάθε κατάσταση που ήδη υπάρχει σε μία FSM. Αντίστοιχα, αν πρέπει να γίνει κάποια επεξεργασία ή αλλαγή σε μία συμπεριφορά, μπορεί να γίνει η επεξεργασία της σε ένα σημείο και όχι από την αρχή σε όλες τις καταστάσεις.



Εικόνα 4: Παράδειγμα BT

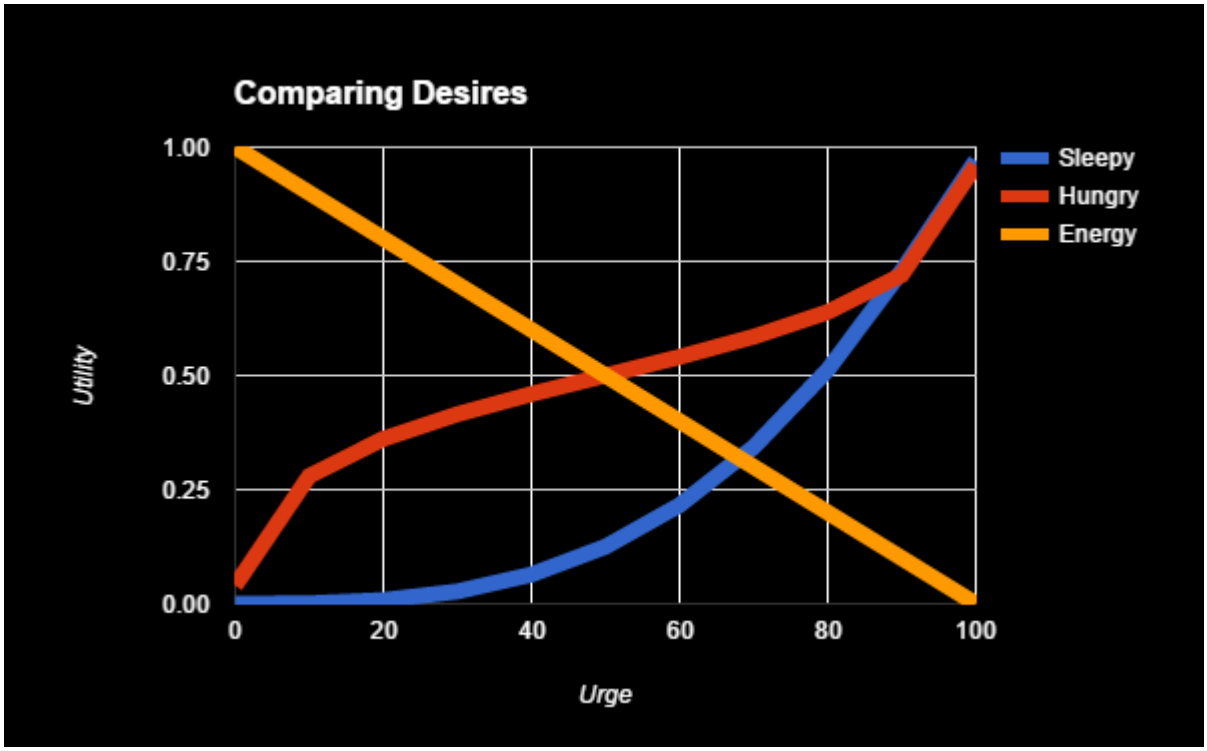
Πηγή: http://intrinsicalgorithm.com/

Στο παράδειγμα της εικόνας 4, ο πράκτορας έχει αποφασίσει να επιτεθεί (Fight), με απομακρυσμένη επίθεση (Range), η οποία απαιτεί πρώτα στόχευση (Aim) και έπειτα εκτόξευση βλήματος (Shoot). Φαίνεται λοιπόν πως άλλο ένα πλεονέκτημα των BT είναι πως υπάρχει μια πιο επίσημη μέθοδος προς την υλοποίηση συμπεριφορών. Μέσα από μία συλλογή εργαλείων, μοτίβων και κατασκευών, μπορούν να υλοποιηθούν αρκετά εκφραστικές συμπεριφορές, ακόμα και σειριακές συμπεριφορές που προορίζονται να εκτελούνται μαζί, όπως στο παράδειγμα. [10]

2.1.3 Utility AI: ένα νέο πρότυπο ΤΝ

Το Utility AI λειτουργεί ως εξής: αναγνωρίζει τις επιλογές διαθέσιμες στον πράκτορα και επιλέγει την καλύτερη βαθμολογώντας τις ανάλογα με την περίσταση. Έχει φανεί ως μία εξαιρετικά ορθά λειτουργική μέθοδος για αρκετούς λόγους.

Οι καμπύλες επιτρέπουν στο Utility να λάβει αποφάσεις μέσω ενός μεγάλου φάσματος επιλογών. Ένα παράδειγμα της αξιοποίησης των καμπυλών φαίνεται στην εικόνα 5, όπου χρησιμοποιούνται για να δώσουν προτεραιότητα σε επιθυμίες. Όλες οι επιθυμίες έχουν μία συνάρτηση εισαγωγής που είναι κανονικοποιημένη στο διάστημα 0 - 100. Η δράση που σχετίζεται με την καμπύλη με την μεγαλύτερη χρησιμότητα (κανονικοποιημένη στις τιμές 0 και 1) εκτελείται. Η χρησιμότητα του ύπνου (καμπύλη sleep) αυξάνεται εκθετικά, η χρησιμότητα της αξιοποίησης ενέργειας μειώνεται γραμμικά (καμπύλη energy) και η χρησιμότητα της πείνας (καμπύλη hungry) είναι παρούσα τις περισσότερες στιγμές, εκτός κι αν έχει ήδη προηγηθεί κάποιο γεύμα. Η ορμή μπορεί να είναι διαφορετική για κάθε επιθυμία. Η επιθυμία που θα έχει ο πράκτορας για να κοιμηθεί μπορεί να εξαρτάται από την ώρα που έχει περάσει από τον τελευταίο ύπνο. Η ορμή σχετιζόμενη με τη σπατάλη της ενέργειας μπορεί να σχετίζεται με την εκάστοτε ώρα μέσα στην ημέρα και αντίστοιχα η ορμή για φαγητό μπορεί να σχετίζεται με την ώρα που έχει περάσει από το τελευταίο γεύμα.



Εικόνα 5: Παράδειγμα Utility AI

Πηγή: http://gamasutra.com

Είναι απλή στο σχεδιασμό και μπορεί να χαρακτηριστεί από δομές σε φυσική γλώσσα, κάνοντας έτσι πιο εύκολη τη συνεννόηση μεταξύ προγραμματιστή - σχεδιαστή βιντεοπαιχνιδιών. Δεν υπάρχει λόγος αναφοράς όρων όπως σύνθετοι κόμβοι, κόμβοι παραλληλίας κλπ. Αποτελεί θέμα στο οποίο γίνονται αποδεκτά ασαφείς όροι (fuzzy terms). Επίσης είναι εύκολα επεκτάσιμη τεχνική. Οι κανόνες βαθμολόγησης μπορούν να προστεθούν σε ήδη υπάρχουσα τεχνητή νοημοσύνη, αυξάνοντας έτσι την πιστότητα και τη λειτουργικότητά του. Επιπροσθέτως, η απλότητα στο σχεδιασμό και η ευκολία με την οποία ένας πράκτορας μπορεί να επεκταθεί, μειώνει δραστικά την ύπαρξη σφαλμάτων

(bugs) και αυξάνει την παραγωγικότητα. Με αυτόν τον τρόπο είναι εφικτή η αύξηση της ποιότητας της τεχνητής νοημοσύνης που αναπτύσσεται, αφού μένει περισσότερος χρόνος για ανάπτυξη πιο πολύπλοκης τεχνητής νοημοσύνης με καλύτερη συμπεριφορά. [5]

2.1.4 Σύγκριση Utility AI με BT

Τα BT χαρακτηρίζονται από μία λογική ροή που μπορεί να ακολουθηθεί, ενώ ο πράκτορας θα πρέπει να είναι αρκετά ανιχνεύσιμος (traceable) ως προς τις κινήσεις και τις πράξεις του. Είναι επίσης εφικτό να σχεδιαστούν πολλά διαφορετικά δέντρα τα οποία μπορούν να ενωθούν σε ένα τελικό δέντρο. Γεγονός που φυσικά επιφέρει το μειονέκτημα της μεγαλύτερης απαίτησης σε χρόνο και σχεδίαση ώστε να στηθεί το τελικό μεγάλο δέντρο.

Το Utility δρα πιο έξυπνα και μπορεί να φέρει εις πέρας τις ίδιες εργασίες με πιο απλές μεθόδους τεχνητής νοημοσύνης. Ο τρόπος που δρα μπορεί επίσης να αλλάξει απλά με την τροποποίηση των τύπων που χρησιμοποιεί για να προσδιορίσει τη χρησιμότητά του. Το να βρεθούν όμως οι σωστοί τύποι είναι το πιο δύσκολο έργο στο Utility και μπορεί να αποδειχτεί πολύ δύσκολο να προσκομιστεί μία συμπεριφορά που προσδοκεί ή θέλει ο προγραμματιστής.

Ενδείκνυται συνεπώς η χρήση BT για δημιουργία απλών αυτόνομων πρακτόρων με πολύ προβλεπόμενες ενέργειες. Για πράκτορες πιο ρεαλιστικούς που μπορούν να χαρακτηριστούν και ως σκεπτόμενοι, το Utility αποτελεί μία καλύτερη επιλογή. [6]

2.2 Σύγκριση με Δέντρα Αποφάσεων

2.2.1 Δέντρα Αποφάσεων (Decision Trees)

Τα DT αποτελούν μία από τις πιο βασικές τεχνικές ΤΝ. Η βασική διαφορά τους με τα BT έγκειται (όπως υποδεικνύεται και από τις ονομασίες τους) στην κεντρική φιλοσοφία της κάθε δομής: τα DT παίρνουν αποφάσεις ενώ τα BT ελέγχουν συμπεριφορές. Κάθε απόφαση παρμένη από ένα DT οδηγεί σε κάποια αντίστοιχη συνέπεια.

Τα BT και τα DT είναι δύο διαφορετικές δομές. Τα BT είναι προσανατολισμένα στο στόχο και αντιδραστικά, επομένως αρμόζουν πιο πολύ σε σχεδιασμό πρακτόρων και αποφάσεων σε ένα περιβάλλον παιχνιδιού. Τα DT είναι ένα εργαλείο καλύτερο για τον ορισμό των προδιαγραφών αποφάσεων που στηρίζονται στη χρησιμότητα μιας ενέργειας για μία δεδομένη κατάσταση.

Λόγω της απλότητας της δομής και της συνοχής τους, τα DT αποτελούν μία τεχνική εύκολη σε κατανόηση και υλοποίηση. Είναι εφικτό να εκτιμηθούν κάθε φορά οι χειρότερες, καλύτερες και αναμενόμενες τιμές για διαφορετικά σενάρια. Τα DT επίσης είναι ευέλικτα ως προς τον συνδυασμό τους με άλλες διαφορετικές τεχνικές απόφασης και διαθέτουν την ευχέρεια διαχείρισης τόσο κατηγορηματικών, όσο και αριθμητικών δεδομένων.

---

[5] Rasmussen, J. (2016, March 27). Gamasutra: Are Behavior Trees A Thing of The Past?

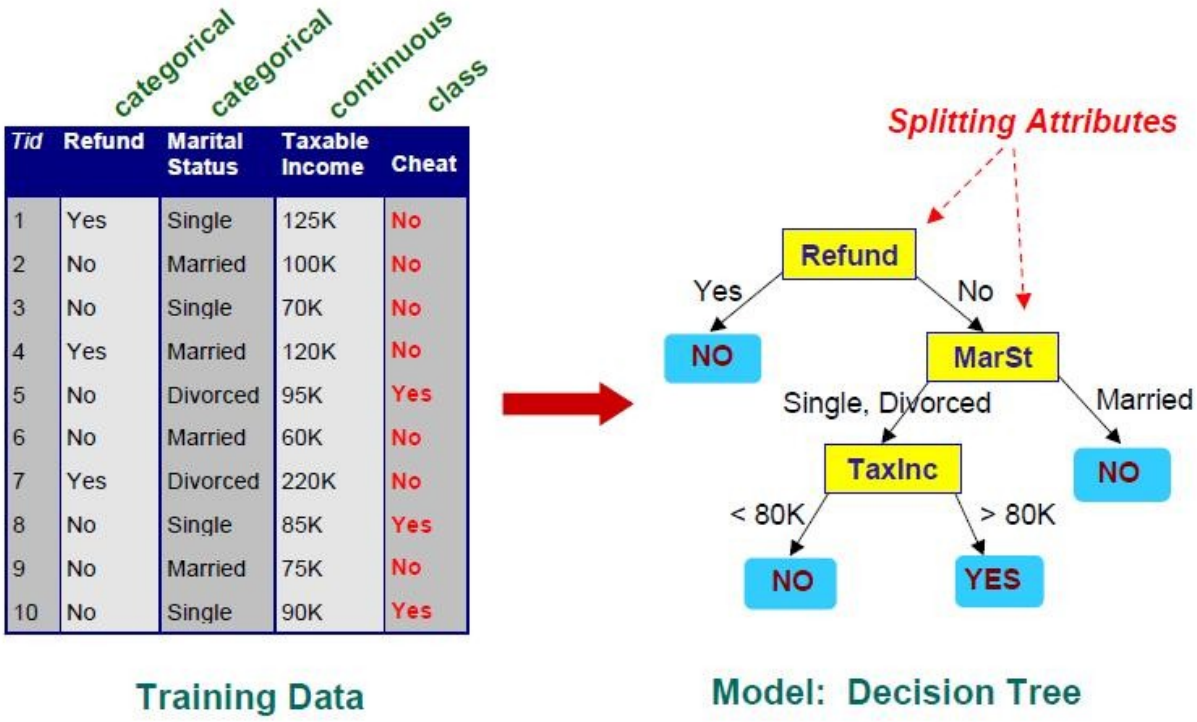
[6] Smith, D. (2014, May 31). Deven Smith Blod: Thoughts on AI: Utility and Behavior Trees.

Σε αντίθεση με τα DT, τα BT δεν παίρνουν μόνο αποφάσεις, αλλά τοποθετούν και σε αλληλουχία τις ενέργειες του πράκτορα. Είναι πιο ευέλικτα, ολοκληρωμένα, αφοσιωμένα σε έναν σκοπό, διαισθητικά και αυτόνομα. [7]

2.2.2 Αδυναμίες των DT

Τα DT αξιολογούνται από τη ρίζα έως το φύλλο κάθε φορά. Για να λειτουργήσει ένα DT ορθά, οι κόμβοι παιδιά κάθε κόμβου γονέα πρέπει να αναπαριστούν όλες τις πιθανές αποφάσεις για τον εκάστοτε κόμβο. Αν ένας κόμβος μπορεί να λάβει απάντηση όπως “ναι”, “όχι”, “ίσως”, πρέπει να υπάρχουν τρεις κόμβοι παιδιά, όπου ο κάθε κόμβος παιδί αποτελεί αναπαράσταση της κάθε πιθανής απάντησης.

Οι ταξινομητές (classifiers) των DT οργανώνουν μια σειρά από δοκιμαστικές ερωτήσεις και συνθήκες μέσα σε μία δομή δέντρου. Στην εικόνα 6 φαίνεται ένα παράδειγμα ενός DT που προβλέπει αν ένα άτομο κλέβει. Μέσα στο DT, η ρίζα και οι εσωτερικοί κόμβοι περιλαμβάνουν δοκιμαστικές συνθήκες/ κριτήρια για να ξεχωρίσουν καταχωρήσεις που έχουν διαφορετικά χαρακτηριστικά. Όλοι οι τελικοί κόμβοι έχουν μία ετικέτα “ναι” ή “όχι”.



Εικόνα 6: Παράδειγμα ταξινομητή DT

Πηγή: http://mines.humanoriented.com/

Αν και τα DT είναι εύκολα αναγνώσιμα από τον άνθρωπο και παρουσιάζουν απλότητα σε κατανόηση και υλοποίηση, χαρακτηρίζονται και από αρκετές αδυναμίες. Οι υπολογισμοί μπορούν να γίνουν πολύ πολύπλοκοι, ειδικά αν αρκετές τιμές είναι αβέβαιες ή αν κάποια αποτελέσματα είναι συνδεδεμένα μεταξύ τους. Στην περίπτωση

[7] https://prezi.com/ez06l7qeja6s/copy-of-ai-decision-tree-behaviour-tree/

που κάποιο δέντρο είναι σε μεγάλο βαθμό σύνθετο, μπορεί να μην καταφέρει να βρεθεί στη θέση μίας ορθής γενίκευσης από τα δεδομένα εκπαίδευσης. [8]

Το σενάριο της χείριστης απόδοσης ενός δέντρου συμπεριφοράς, εξαρτάται από το βάθος του δέντρου. Σε κάθε εκτέλεση κάποιου σεναρίου γίνεται επαναξιολόγηση της διαδρομής από τη ρίζα του δέντρου μέχρι το τελικό φύλλο που περιέχει τη συμπεριφορά, έτσι ώστε να φτάσει η εκτέλεση στο τελικό αποτέλεσμα. Αυτό φέρνει σαν αποτέλεσμα δυσκολίες στην υλοποίηση κάποιων πιο σύνθετων επιλογών συμπεριφοράς, που μέσα στη γενική απλότητα σχεδιασμού των DT αποφέρει προβλήματα. [21]

2.2.3 Υπεροχή των BT στο σχεδιασμό βιντεοπαιχνιδιών

Όταν έχουμε να κάνουμε με DT, ολόκληρο το δέντρο αξιολογείται όπως προαναφέρθηκε από τον κόμβο της ρίζας έως τον κόμβο του φύλλου ώστε να υπάρχει αποτέλεσμα, επομένως η εκτέλεση δεν είναι βασισμένη σε μία δεδομένη κατάσταση. Αντιθέτως, όσον αφορά τα BT, η εκτέλεση είναι άρρηκτα συνδεδεμένη με την κατάσταση στην οποία βρίσκεται ο πράκτορας αναφορικά με το δέντρο.

Το γεγονός πως η συμπεριφορά σε ένα BT δεν αξιολογείται από τη ρίζα σε κάθε ανανέωση, είναι η κύρια διαφοροποίησή του από ένα DT. Έτσι ένα BT δεν χαρακτηρίζεται από φτωχή υποστήριξη για εκτέλεση και παρακολούθηση συμπεριφορών κατά την πάροδο του χρόνου. Για παράδειγμα, αν γίνει επαναφορά (reset) ενός BT, τότε τυπικά οι κόμβοι ακολουθιών θα επαναφέρονται κάθε φορά.

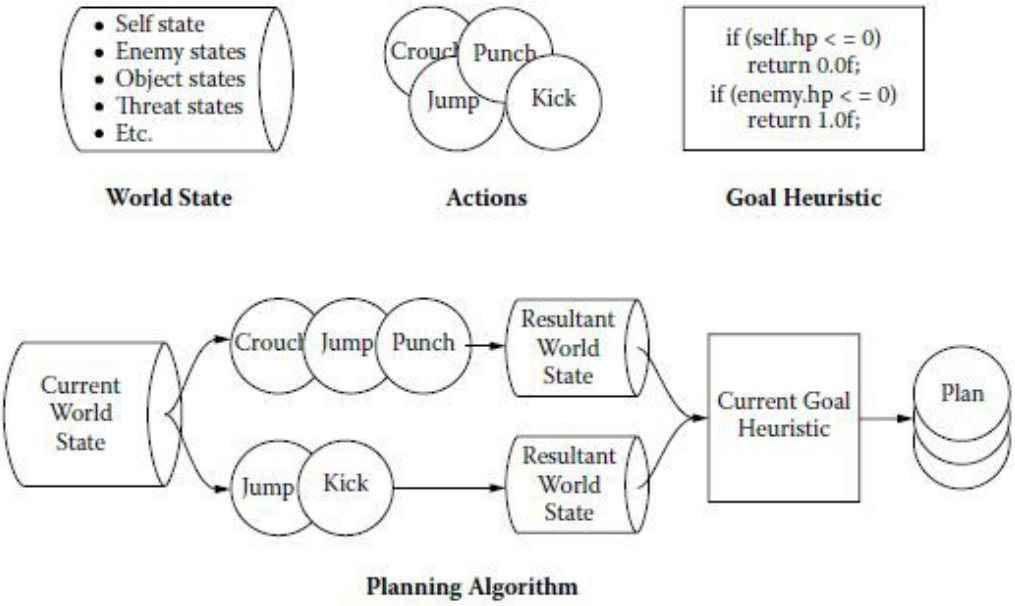
2.3 Συστήματα Σχεδιασμού (Planners)

Ένα σύστημα σχεδιασμού στην τεχνητή νοημοσύνη για βιντεοπαιχνίδια, χρησιμοποιείται για να δημιουργήσει μία ακολουθία από στοιχειώδεις ενέργειες για να ολοκληρωθεί κάποιος στόχος, σε σχέση με τη δοθείσα κατάσταση του τρέχοντος κόσμου (world). Αυτή η ακολουθία ενεργειών ονομάζεται σχέδιο (plan). Το σύστημα σχεδιασμού διατηρεί ένα μοντέλο της κατάστασης του κόσμου (world model), μία συλλογή από όλες τις στοιχειώδεις ενέργειες διαθέσιμες σε έναν πράκτορα, και έναν ευρετικό μηχανισμό για στόχο (goal heuristic). Το μοντέλο της κατάστασης του κόσμου περιέχει ο,τι πληροφορία χρειάζεται ο ευρετικός μηχανισμός σχετικά με τον κόσμο. Για παράδειγμα, μία κατάσταση του κόσμου μπορεί να περιλαμβάνει μία λίστα με όλους τους διαθέσιμους εχθρούς (πράκτορες ή μη) και τα επίπεδα της ζωής τους. Το σύστημα σχεδιασμού καταλαβαίνει πώς κάθε ενέργεια δρα πάνω στην τρέχουσα κατάσταση του κόσμου και πώς κάθε σχέδιο δρα πάνω της. Για παράδειγμα, μία δράση “κλωτσιάς” (kick) κάνει ζημιά στον αντίπαλο αν είναι σε κοντινή απόσταση, ενώ μία δράση “άλματος” (jump) φέρνει τον πράκτορα πιο κοντά στον εχθρό.

Ο ευρετικός μηχανισμός του στόχου αξιολογεί ένα δοθέν σχέδιο από το πόσο πολύ συμβάλλει στην επίτευξη του στόχου του. Στο ίδιο παράδειγμα, ένας ευρετικός μηχανισμός διαμάχης (combat) θα έδινε υψηλό βαθμό αξιολόγησης σε ένα σχέδιο που θα είχε ως αποτέλεσμα την πρόκληση ζημιάς σε εχθρούς. Επομένως, αν ο πράκτορας είναι κοντά σε έναν εχθρό, ένα σχέδιο υψηλής αξιολόγησης μπορεί να περιλαμβάνει

[8] http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/lguo/decisionTree.html

απλά και μόνο μία δράση κλωτσιάς. Αν όμως ο πράκτορας είναι αρκετά μακριά από τον εχθρό και η δράση κλωτσιάς δεν μπορεί να προκαλέσει ζημιά, το πλάνο θα λάβει χαμηλή αξιολόγηση. Αλλά αν γίνει προσθήκη μίας δράσης άλματος πριν τη δράση της κλωτσιάς, ο πράκτορας θα μπορεί να κινηθεί επιθετικά και να προκαλέσει ζημιά στον εχθρό, ένα σχέδιο που θα είχε υψηλή αξιολόγηση. Με όλα αυτά τα κομμάτια διαθέσιμα, το σύστημα σχεδιασμού μπορεί να δημιουργήσει σχέδια που μπορούν να επιτύχουν υψηλού επιπέδου στόχους δυναμικά, ανεξάρτητα από την τρέχουσα κατάσταση του κόσμου.



Εικόνα 7: Παράδειγμα συστήματος σχεδιασμού

Πηγή: http://www.gameaipro.com/

Ενώ η ευελιξία των συστημάτων σχεδιασμού είναι ένα μεγάλο πλεονέκτημα, μπορεί να είναι επίσης και ένα εξίσου μεγάλο μειονέκτημα. Συχνά, οι σχεδιαστές θέλουν να έχουν περισσότερο έλεγχο πάνω στην ακολουθία των ενεργειών που ένας πράκτορας μπορεί να εκτελέσει. Ενώ είναι θεμιτό ο πράκτορας να μπορεί να κάνει ένα άλμα ακολουθούμενο από μία κλωτσιά από μόνος του, θα μπορούσε όμως και να δημιουργήσει μια ακολουθία από πολλαπλά συναπτά άλματα. Αυτό σπάει την ψευδαίσθηση της νοημοσύνης που ο πράκτορας θα έπρεπε να παράγει. Υπάρχουν τεχνικές για να αποφευχθούν τέτοια ανεπιθύμητα σχέδια, αλλά είναι δύσκολο να γίνει πρόβλεψη όλων των περιπτώσεων όπου το σύστημα σχεδιασμού δε θα αντιδράσει ορθά ή σε επαρκώς μικρό χρόνο. Κάτι τέτοιο προκαλεί αποστροφή προς τα συστήματα σχεδιασμού εκ μέρους των σχεδιαστών και προγραμματιστών, εφόσον οι ίδιοι επιθυμούν περισσότερο έλεγχο πάνω στη συμπεριφορά των χαρακτήρων που αναπτύσσουν.

Αυτός είναι ο κλασικός συμβιβασμός που οι σχεδιαστές και προγραμματιστές τεχνητής νοημοσύνης αντιμετωπίζουν επί μονίμου βάσεως: η επιλογή ανάμεσα σε μία πλήρως

σχεδιασμένη (εύθραυστη) τεχνητή νοημοσύνη που παρέχουν τα BT και την πλήρως αυτόνομη (απρόβλεπτη) τεχνητή νοημοσύνη που παρέχουν τα συστήματα σχεδιασμού.

Για να δοθεί μία λύση σε αυτό το φαινομενικά δυικής υπόστασης ζήτημα, έχουν αναπτυχθεί υλοποιήσεις BT και συστημάτων σχεδιασμού, όπως το υβριδικό σύστημα BT και Planners. [22]

3. ΑΡΧΙΤΕΚΤΟΝΙΚΗ BEHAVIOR TREES

3.1 Γενική Περιγραφή

Τα BT είναι ένας τρόπος ελέγχου της ροής εκτελέσεων σε έναν πράκτορα, ιδιαίτερα όταν έχουμε να κάνουμε με αποφάσεις. Μία τυπική υλοποίηση ενός BT συμπεριλαμβάνει μία ρίζα (root) που είναι το σημείο εισαγωγής για την ροή εκτέλεσης. Ύστερα η ρίζα οδηγείται στα κλαδιά του δέντρου που είναι σύνθετοι κόμβοι (composite nodes) είτε κόμβοι διακοσμητές (decorator nodes). Τέλος, τα αποτελέσματα από κάθε συνθήκη που περιέχουν αυτοί οι κόμβοι, οδηγούν σε εργασίες (tasks) που αργότερα θα αναλυθούν ως κόμβοι φύλλα (leaf nodes). Η αναδρομική δομή του BT επιτρέπει την αναδρομική προσομοίωση σύνθετων συμπεριφορών.

Επίσης σημαντικοί για την εκτέλεση και εξέλιξη ενός BT είναι και οι μαυροπίνακες (blackboards) που επιτρέπουν στο BT να έχει πρόσβαση σε διαμοιραζόμενες μεταβλητές και δεδομένα με ευκολία.

3.2 Event loop

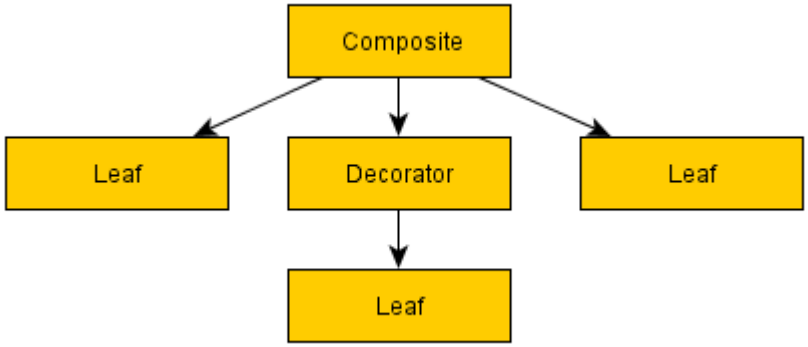
Ένα event loop (βρόχος συμβάντων), αποτελεί μία κατασκευή μέσα σε ένα πρόγραμμα που εκτελεί λειτουργία ελέγχου και αποστολής συμβάντων που ακολουθούν μια συνθήκη εκκίνησης. Ορίζονται συναρτήσεις επανάκλησης (callback functions) που καλούνται ανά ορισμένα διαστήματα ή κατά την ικανοποίηση μιας συνθήκης.

Τα BT μπορούν να διαθέσουν ένα ξεχωριστό React Tree (δέντρο αντίδρασης) που θα εκτελείται όποτε γίνεται λήψη ενός event. Το RT διαθέτει λογική που αποστέλλει το event επιλέγοντας την κατάλληλη αντίδραση. Σημειώνεται πως αυτό το υπόδενδρο εκτελείται παράλληλα με την κύρια συμπεριφορά και απαιτεί δεσμευτές πόρων (resource allocators) ώστε να εξασφαλιστεί η μη ύπαρξη συγκρούσεων. [21] Αυτοί οι δεσμευτές πόρων βοηθούν στην αποφυγή εμφάνισης μη ρεαλιστικών συμπεριφορών από την παράλληλη εκτέλεση κόμβων ενός BT. Η πιο απλή μορφή ενός τέτοιου δεσμευτή μπορεί να είναι ένα σύστημα εξυπηρέτησης με σειρά προτεραιότητας (first-come-first-served system). [9]

3.3 Χαρακτηριστικά των BT κόμβων

Οι κόμβοι ενός BT μπορούν να κατηγοριοποιηθούν σύμφωνα με τρία πρότυπα: σύνθετος (composite), διακοσμητής (decorator) και φύλλο (leaf). Κάθε μορφή κόμβου υπόκειται σε ένα από τα τρία αυτά πρότυπα. Στην εικόνα 8 φαίνεται μία γενική μορφή ενός υπόδενδρου BT.

[9] Champandard, J. (2007, July 11). AiGameDev: Using Resource Allocators to Synchronize Behaviors.



Εικόνα 8: Παράδειγμα γενικής δομής υπόδενδρου ενός BT

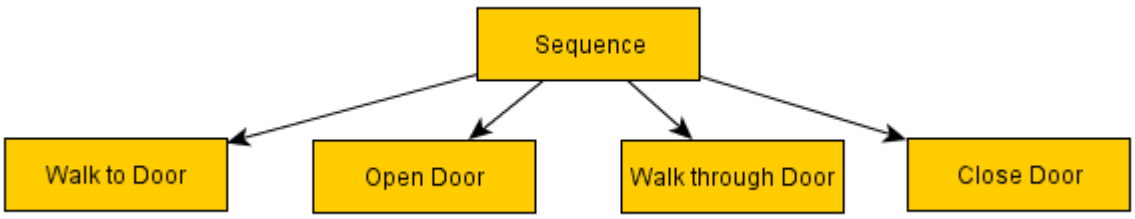
Πηγή: http://gamasutra.com

3.3.1 Σύνθετοι Κόμβοι (Composite Nodes)

Ένας σύνθετος κόμβος μπορεί να έχει ένα ή περισσότερα παιδιά. Δεν διατίθεται κάποια συγκεκριμένη προσπέλαση των child nodes του, καθότι αυτό προσδιορίζεται από την ίδια τη φύση του κόμβου. Αν ο composite node έχει γονέα, θα του στείλει μήνυμα επιτυχίας ή αποτυχίας αφού εκτελεστούν τα παιδιά του, ενώ κατά τη διάρκεια της εκτέλεσης αυτών επιστρέφει πάντα μήνυμα “σε εκτέλεση”.

Μπορεί να αναλυθεί σε υποκατηγορίες, οι πιο συχνές εκ των οποίων είναι:

- **Κόμβος ακολουθίας (Sequence Node)**: η εκτέλεση στους child nodes γίνεται ακολουθιακά. Αν οποιαδήποτε στιγμή εμφανιστεί μήνυμα failure, αυτόματα και όλος ο sequence node επιστρέφει μήνυμα failure. Αν και το τελευταίο παιδί επιστρέψει μήνυμα success, τότε ο sequence node επιστρέφει κι αυτός μήνυμα success. 'Ενα παράδειγμα κόμβου ακολουθίας φαίνεται στην εικόνα 9.



Εικόνα 9: Παράδειγμα κόμβου ακολουθίας

Πηγή: http://gamasutra.com

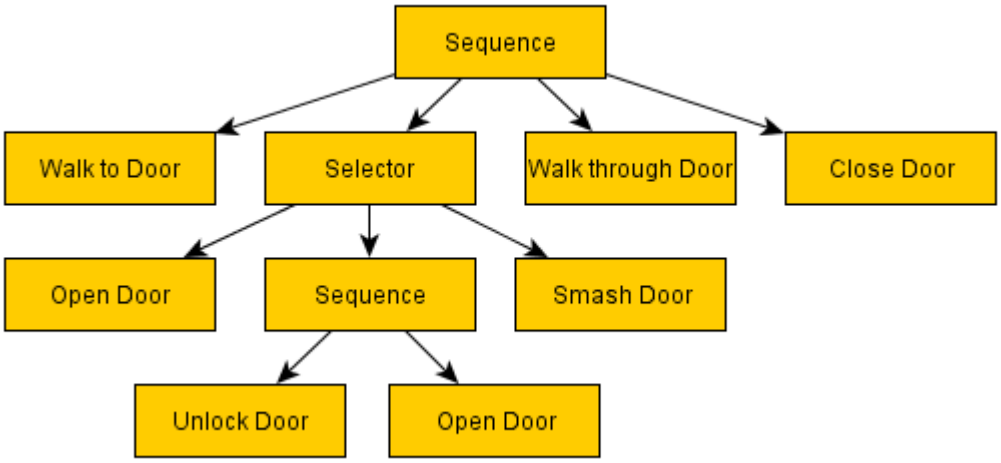
Οι ενέργειες εκτελούνται σειριακά, δηλαδή με φορά από αριστερά προς τα αριστερά. Αν έστω και ένας κόμβος επιστρέψει μήνυμα αποτυχίας, τότε ο κόμβος ακολουθίας επιστρέφει κι αυτός μήνυμα επιτυχίας, δηλαδή ο πράκτορας δεν θα προχωρήσει μέσα από την πόρτα. Ο ψευδοκώδικας (pseudocode) για κόμβο ακολουθίας με N κόμβους παιδιά έχει τη μορφή στο σχήμα 1.

```
for i = 1 to N:
    state = Tick(child[i])

    if state != SUCCESS:
        return state

return SUCCESS
```

Σχήμα 1: Ψευδοκώδικας κόμβου ακολουθίας BT

Πηγή: http://guineashots.com/

- **Κόμβος διαλέκτης (Selector Node)**: αν οποιοσδήποτε εκ των child nodes επιστρέψει μήνυμα success, τότε ο selector node επιστρέφει κι αυτός μήνυμα success, χωρίς να προσπελάσει τα (τυχόν) υπολοίποντα παιδιά του. Η γραφική αναπαράσταση ενός κόμβου διαλέκτη φάινεται στην εικόνα 10 με βάση το προηγούμενο παράδειγμα.



Εικόνα 10: Παράδειγμα κόμβου διαλέκτη

Πηγή: http://gamasutra.com

Ο πράκτορας έχει την επιλογή αν δει κάποια πόρτα να διαλέξει είτε να να την ανοίξει, είτε να την ξεκλειδώσει και μετά να την ανοίξει (σε περίπτωση που η πρώτη επιλογή αποτύχει), ή να τη σπάσει. Ουσιαστικά ο προηγούμενος κόμβος “Open Door” αντικαταστάθηκε με περισσότερες επιλογές για τον πράκτορα. Ο ψευδοκώδικας για έναν κόμβο διαλέκτη δίνεται στο σχήμα 2.

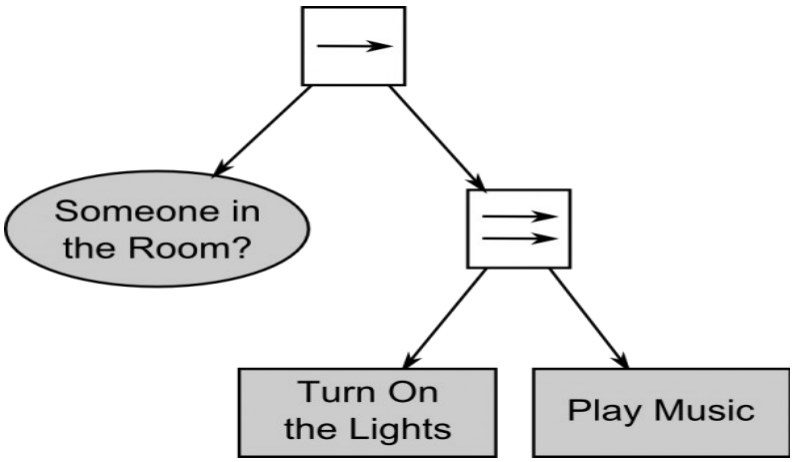
```
for i = 1 to N:
    state = Tick(child[i])

    if state != FAILURE:
        return state

return FAILURE
```

Σχήμα 2: Ψευδοκώδικας κόμβου διαλέκτη BT

Πηγή: http://guineashots.com/

- **Κόμβος τυχαίου διαλέκτη (Random Selector Node)**: μία απλή παραλλαγή του κόμβου διαλέκτη που μπορεί να κάνει έναν πράκτορα πιο ενδιαφέροντα και διαφορετικό. Ενώ ο κλασικός διαλέκτης εκτελεί κάθε ένα από τα παιδιά του σε αυστηρή σειρά που έχει οριστεί από τον σχεδιαστή αρχικά, αυτή εδώ η παραλλαγή αυτού του κόμβου εκτελεί τα παιδιά σε τυχαία σειρά.
- **Κόμβος τυχαίας ακολουθίας (Random Sequence Node)**: ομοίως με τον κόμβο τυχαίου διαλέκτη, ένας κόμβος τυχαίας ακολουθίας εκτελεί τα παιδιά του με τυχαία σειρά και όχι σειριακά όπως ένας κανονικός κόμβος ακολουθίας.
- **Κόμβος παραλληλίας (Parallel Node)**: γίνεται ταυτόχρονη εκτέλεση όλων των child nodes του και επιστρέφει μήνυμα success αν έχει επιστρέψει success ο αριθμός των παιδιών του που έχουν επιστρέψει με τη σειρά τους μήνυμα success, εφόσον όμως ο αριθμός αυτός είναι μεγαλύτερος από μία σταθερά S (κατώφλι επιτυχίας) ορισμένη σε κάθε parallel node. Ομοίως επιστρέφει μήνυμα failure αν ο αριθμός των παιδιών με μήνυμα failure είναι μεγαλύτερος από μία αντίστοιχη σταθερά F (κατώφλι αποτυχίας). Η ουσιαστική συμπεριφορά αυτού του κόμβου εξαρτάται από την πολιτική του:
    - *Sequence policy*: στην ακολουθιακή πολιτική, ο κόμβος παραλληλίας αποτυγχάνει αν έστω και ένα παιδί αποτύχει. Αν όλα τα παιδιά επιτύχουν, τότε αυτός ο κόμβος επιτυγχάνει. Αποτελεί την πιο συχνή πολιτική.
    - *Selector policy*: στη διαλεκτική πολιτική, ο κόμβος παραλληλίας επιτυγχάνει αν έστω και ένα παιδί επιστρέψει μήνυμα επιτυχίας, αν όλα τα παιδιά αποτύχουν τότε αποτυγχάνει και ο κόμβος.

Ένα ενδεικτικό παράδειγμα για κόμβο παραλληλίας, είναι το παράδειγμα ενός έξυπνου σπιτιού, που όποτε ανιχνεύει ανθρώπινη παρουσία (γεγονός που προκύπτει από την εκτέλεση του αρχικού κόμβου ακολουθίας), ανάβει τα φώτα και παράλληλα παίζει μουσική (εικόνα 11).



Εικόνα 11: Παράδειγμα κόμβου παραλληλίας

Πηγή: http://guineashots.com/

Ο ψευδοκώδικας για τον κόμβο παραλληλίας φαίνεται στο σχήμα 3.

```
for i = 1 to N:
    state_i = Tick(child[i])

if nSucc(state) >= S:
    return SUCCESS
else if nFail(state) >= F:
    return FAILURE
else
    return RUNNING
```

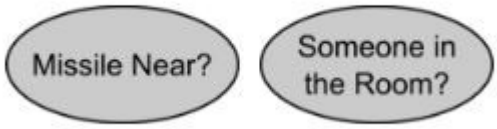
Σχήμα 3: Ψευδοκώδικας κόμβου παραλληλίας BT

Πηγή: http://guineashots.com/

3.3.2 Κόμβοι Φύλλα (Leaf Nodes)

Ένας τέτοιος κόμβος αποτελεί το βασικό θεμελιακό στοιχείο ενός BT. Δεν έχουν child nodes και επιστρέφουν τα ίδια μηνύματα όπως κάθε κόμβος BT (success, failure, running). Μπορούμε να τα διαχωρίσουμε σε δύο κατηγορίες, τους κόμβους φύλλα “συνθήκες” (condition leaf nodes) και τους κόμβους φύλλα “ενέργειες” (action leaf nodes). [8]

Condition leaf: Ένας condition leaf node αναλαμβάνει όπως προσδιορίζεται και από την ονομασία του, να ελέγξει την ικανοποίηση ή μη μίας συνθήκης, με την προϋπόθεση ότι διαθέτει μία μεταβλητή ως στόχο ή κάποια εσωτερική μεταβλητή, καθώς και ένα κριτήριο στο οποίο θα βασιστεί η απόφαση. Αυτοί οι κόμβοι δεν επιστρέφουν running, παρά μόνο success ή failure. Μπορούμε πιο απλά επομένως, να θεωρούμε πως ένας κόμβος φύλλου συνθήκης, αναπαριστάται ως μία ερώτηση που μπορεί να απαντηθεί μόνο με ναι ή όχι.



Εικόνα 12: Παράδειγμα κόμβου φύλλου συνθήκης

Πηγή: http://guineashots.com/

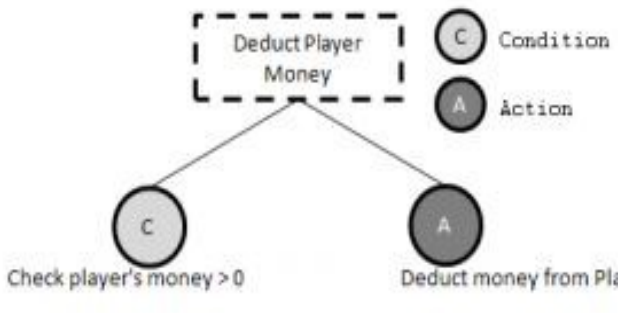
Action leaf: Ένας action leaf node εκτελεί υπολογισμούς για να αλλάξει την κατάσταση στην οποία βρίσκεται ο πράκτορας (agent). Το ποιες ενέργειες θα υλοποιηθούν εξαρτάται από τη φύση του πράκτορα, δεν υπόκεινται δηλαδή σε κάποιο συγκεκριμένο πρότυπο. Ένας τέτοιος κόμβος έχει τόσο εσωτερική όσο και εξωτερική δράση, μπορεί για παράδειγμα να οδηγεί σε κάποιες αλλαγές του περιβάλλοντος του παιχνιδιού (game environment) ως αποτέλεσμα των αλλαγών που υπέστη ο agent, αλλά μπορεί επίσης να αναλάβει την αποθήκευση αρχείων ή την αλλαγή εσωτερικών μεταβλητών. Αν η ενέργεια (action) ολοκληρωθεί, τότε ο κόμβος επιστρέφει success. Στην αντίθετη περίπτωση επιστρέφει failure, ενώ κατά τη διάρκεια εκτέλεσής του επιστρέφει running.[10]

Εικόνα 13: Παράδειγμα κόμβου φύλλου ενέργειας

Πηγή: http://guineashots.com/

Για παράδειγμα, μπορούμε να ορίσουμε μία συμπεριφορά η οποία θα υλοποιεί τη δράση “απόσπασε χρήματα από τον παίκτη”. Αυτή η δράση μπορεί να αποσυντεθεί σε δύο κόμβους φύλλα, ο ένας θα είναι κόμβος συνθήκης και ο άλλος κόμβος ενέργειας. Ο κόμβος συνθήκης θα ελέγχει αν ο παίκτης διαθέτει επαρκή χρήματα και ο κόμβος ενέργειας θα εκτελεί τη διαδικασία απόσπασής τους (αν αυτά υπάρχουν). Η γραφική αναπαράσταση του παραδείγματος φαίνεται στην εικόνα 14.

[10] http://guineashots.com/2014/08/10/an-introduction-to-behavior-trees-part-2/



Εικόνα 14: Παράδειγμα συμπεριφοράς με κόμβο συνθήκης και ενέργειας

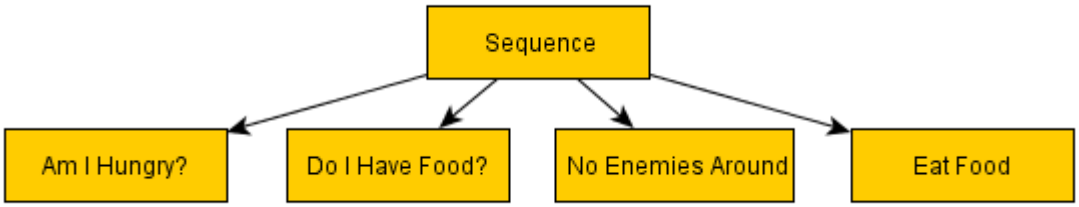
Πηγή: http://www.doc.ic.ac.uk/

3.3.3 Κόμβοι Διακοσμητές (Decorator Nodes)

Σε αντίθεση με τους composite nodes, αυτοί οι κόμβοι μπορούν να έχουν μόνο έναν child node. Η λειτουργία τους έγκειται είτε στην αναστροφή του αποτελέσματος που επιστρέφει ο child node, είτε να τερματίζουν τη λειτουργία του, είτε να επαναλάβουν την εκτέλεσή του.

Μπορεί να αναλυθεί και αυτό το είδος σε υποκατηγορίες, οι πιο συχνές εκ των οποίων είναι:

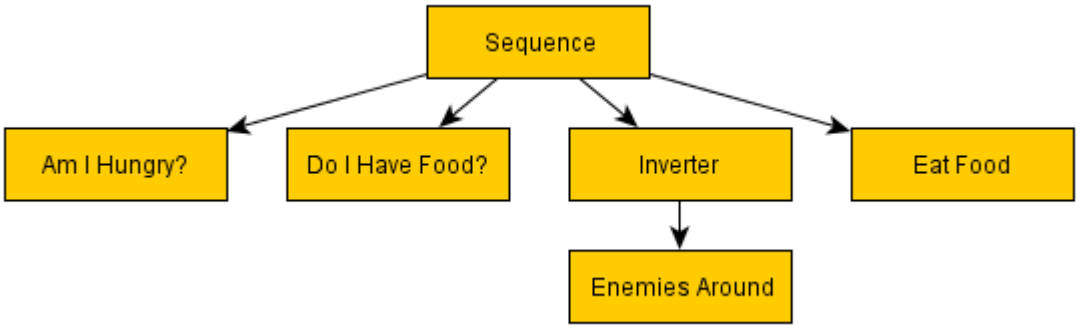
- **Αναστροφέας (Inverter)**: αναστρέφουν το αποτέλεσμα που επέστρεψε ο child node και στέλνουν το νέο μήνυμα στον parent node. Για παράδειγμα, έστω ότι έχουμε το πρόβλημα της εικόνας 15 που καλείται να αντιμετωπίσει ο πράκτορας.



Εικόνα 15: Παράδειγμα συμπεριφοράς

Πηγή: http://gamasutra.com

Ο κόμβος “No Enemies Around” μπορεί να αντικατασταθεί με έναν κόμβο αντιστροφέα όπως φαίνεται στην εικόνα 16.



Εικόνα 16: Παράδειγμα συμπεριφοράς με κόμβο αντιστροφέα

Πηγή: http://gamasutra.com

Ο ψευδοκώδικας για τον κόμβο αντιστροφέα φαίνεται στο σχήμα 4.

```
state = Tick(child)

if state == SUCCESS:
    return FAILURE
else if state == FAILURE:
    return SUCCESS
else
    return state
```

Σχήμα 4: Ψευδοκώδικας για τον κόμβο αντιστροφέα BT

Πηγή: http://guineashots.com/

- **Επιτυγχάνων (Succeeder)**: ανεξάρτητα από το μήνυμα επιστροφής του child node, επιστρέφουν στον parent node πάντα μήνυμα success.

```
state = Tick(child)

return SUCCESS
```

Σχήμα 5: Ψευδοκώδικας για τον επιτυγχάνοντα κόμβο BT

Πηγή: http://guineashots.com/

- **Επαναληπτικός (Repeater)**: επαναλαμβάνει την εκτέλεση του child node έναν προκαθορισμένο αριθμό φορών.

```
while j < M:
   state = Tick(child)
   if state != SUCCESS and state != FAILURE:
       return state

   j++

return SUCCESS
```

Σχήμα 6: Ψευδοκώδικας για τον επαναληπτικό κόμβο BT

Πηγή: http://guineashots.com/

- **Επαναληπτικός έως αποτυχίας (Repeat Until Fail)**: όπως υποδεικνύει το όνομα, αυτός ο τύπος κόμβου επαναλαμβάνει την εκτέλεση του child node μέχρι αυτό να επιστρέψει εν τέλει μήνυμα failure, ώστε μετέπειτα να περαστεί μήνυμα success στον parent node.

```
repeat
    state = Tick(child)
    if state != SUCCESS and state != FAILURE:
       return state

until state == FAILURE

return SUCCESS
```

Σχήμα 7: Ψευδοκώδικας για τον επαναληπτικό έως αποτυχίας κόμβο BT

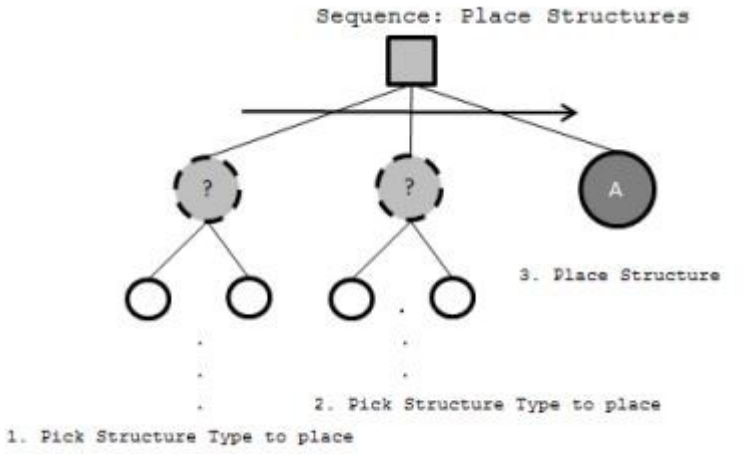
Πηγή: http://guineashots.com/

3.3.3.1 Lookup Decorator - Μία ιδιαίτερη περίπτωση

Από την περιγραφή των κόμβων διακοσμητών, ένα αξιοπρόσεκτο σημείο είναι ότι ένας κόμβος διακοσμητής δεν εμφανίζεται ποτέ ως κόμβος φύλλο ενός BT, αφού η λειτουργία τους εξαρτάται κατ' ουσίαν στις ενέργειες των κόμβων παιδιών τους. Υπάρχει όμως ένας κόμβος διακοσμητής που εμφανίζεται πάντα ως κόμβος φύλλο, και αυτός είναι ο Διακοσμητής Αναζήτησης (Lookup Decorator ή αλλιώς Lookups).

Οι Lookups επιχειρούν να αυξήσουν τη δομοστοιχείωση (modularity) και επαναχρησιμοποίηση των BT, ψάχνοντας συγκεκριμένα έργα (tasks) υπο-συμπεριφορών, αντί να τα αναφέρουν ρητά μέσα στο δέντρο.

Για παράδειγμα, στην εικόνα 17 φαίνεται η εικονική αναπαράσταση μίας τέτοιας κατάστασης, που περιγράφει τη συμπεριφορά ενός πράκτορα σε ένα παιχνίδι όπου σχεδιάζει να τοποθετήσει κτίρια (structures) σε έναν δοθέν χάρτη.

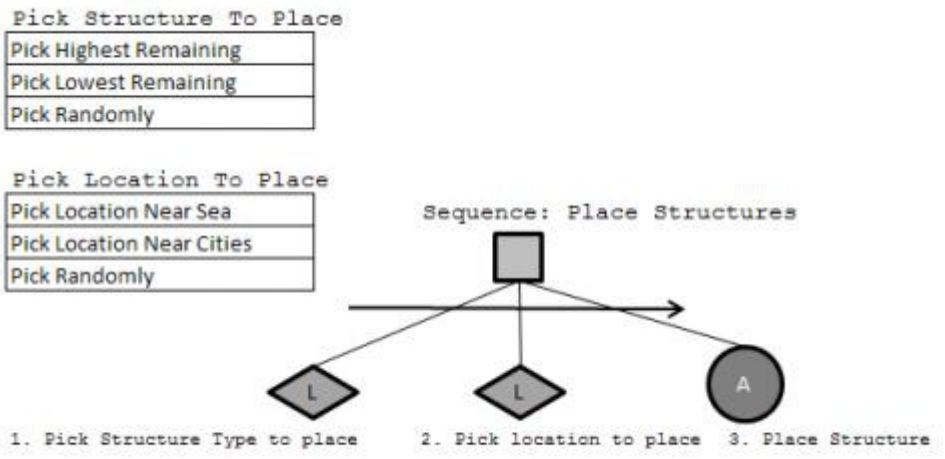


Εικόνα 17: Παράδειγμα συμπεριφοράς τοποθέτησης κτιρίου

Πηγή: http://www.doc.ic.ac.uk/

Η συμπεριφορά παίρνει τη μορφή κόμβου ακολουθίας (Sequence). Πρώτα επιλέγεται ο τύπος του κτιρίου προς τοποθέτηση (Pick Structure Type to place), έπειτα αποφασίζεται η τοποθεσία που αυτό θα τοποθετηθεί, ενώ τελικά ο πράκτορας τοποθετεί το κτίριο. Τα πρώτα δύο έργα περιλαμβάνουν κόμβους διαλεχτές (Selectors), καθώς δύναται να σχεδιαστεί ο πράκτορας με τέτοιο τρόπο ώστε να έχει διαφορετικές υλοποιήσεις ανάλογα τον τύπου του κτιρίου που καλείται να τοποθετήσει. Μια ευρετική σύμφωνα με το πλήθος των διαθέσιμων κτηρίων κάθε κατηγορίας θα καθόριζε την πρώτη επιλογή. Ομοίως για τη δεύτερη απόφαση που έχει να λάβει ο πράκτορας, μπορεί να αποφασίσει το μέρος όπου θα τοποθετήσει το κτίριο ανάλογα με το αν θα είναι κοντά σε θαλασσινή περιοχή, κοντά σε πόλη ή κοντά σε άλλα κτίρια.

Η ιδέα είναι να συσχετιστούν αυτές οι διαφορετικές υλοποιήσεις σε έναν πίνακα αναζήτησης (lookup table), ομαδοποιημένα από το υψηλού επιπέδου έργο που αυτά επιτυγχάνουν, και να γίνει αντικατάσταση των κόμβων παιδιών της κύριας ακολουθίας (Sequence) με Lookups, όπως φαίνεται στο σχήμα 18.



Εικόνα 18: Χρήση Lookup για διαμόρφωση συμπεριφοράς

Πηγή: http://www.doc.ic.ac.uk/

Ομαδοποιώντας αυτές τις υλοποιήσεις σε μία ιεραρχία πινάκων, οδηγούμαστε προς μια ιεραρχική αρχιτεκτονική προσανατολισμένη στο στόχο (hierarchical goal-oriented architecture). Αυτά τα υψηλού επιπέδου έργα μπορούν να αντιμετωπιστούν και ως υψηλού επιπέδου στόχοι, που το BT θέλει εν τέλει να ολοκληρώσει, επιτυγχάνοντάς το μέσω αποσύνθεσής τους σε μικρότερους υπό-στόχους. Αυτή η προσέγγιση μπορεί να επισημοποιηθεί με τη θεωρητική μεθοδολογία του σχεδιασμού επικεντρωμένου στη συμπεριφορά (Behavior Oriented Design).

3.3.3.2 Σχεδιασμός Επικεντρωμένος στη Συμπεριφορά (Behavior Oriented Design)

Ο BOD (Joanna Bryson) είναι μια μεθοδολογία ανάπτυξης που στοχεύει στο σχεδιασμό και την κατασκευή πλήρων πολύπλοκων πρακτόρων (complete complex agents - CCA) σε πλαίσιο δομοστοιχείων [17]. Η μεθοδολογία αυτή αποτελεί βελτίωση πάνω σε υπάρχουσες αρχιτεκτονικές για αυτούς τους CCA.

Το πιο ενδιαφέρον στοιχείο του BOD είναι η υψηλή αντιστοιχία με τα BT, γεγονός που δίνεται τη δυνατότητα σχεδιασμού BT που είναι ικανά να εκτελέσουν παρόμοιους τύπους αντιδραστικού σχεδιασμού (reactive planning), όπως επιτρέπει και ο BOD. [11]

3.3.3.2.1 Αντιδραστικός Σχεδιασμός (Reactive Planning)

Ο σχεδιασμός (Planning) περιλαμβάνει τον ορισμό της ακολουθίας γεγονότων που θα αποτελέσουν την ολοκλήρωση ενός στόχου. Πολλές από αυτές τις ενέργειες είναι συνήθως εν δυνάμει εκτελέσιμες σε σχέση με μία κατάσταση ή ένα συμβάν, αν και μπορεί να μην είναι λογικά ή φυσικά δυνατό για αυτές να εκτελεστούν μαζί.

Προκύπτει επομένως η απαίτηση της ρητής συγκεκριμενοποίησης των συνθηκών, για την οποία μπορεί η κάθε ενέργεια να εκφράζεται, που είναι η περίπτωση της αρχιτεκτονικής υπαγωγής (subsumption architecture) και της αρχιτεκτονικής δικτύου

[11]Year, F., Project, I., & Report, F. (2009). An A . I . Player for DEFCON : An Evolutionary Approach Using Behavior Trees.

πρακτόρων (agent network architecture). Αυτή η απαίτηση αυξάνεται όσο μεγαλώνει η πολυπλοκότητα του δοθέντος πεδίου με όλο και πιο πολλές συμπεριφορές να προστίθενται Ο αντιδραστικός σχεδιασμός προσπαθεί να αντιμετωπίσει αυτό το γεγονός περιγράφοντας με τη χρήση ακολουθιών συμβάντων, κάθε συμπεριφορά σε ακολουθίες δράσης-επιλογής (action-selection sequences). Αυτές οι ακολουθίες μπορούν να διακοπούν, αποτρέποντας την ολοκλήρωση μίας ακολουθίας ενεργειών, ως αποτέλεσμα δύο κατηγοριών συμβάντων:

1. Κάποιος συνδυασμός από συναγερμούς (alarms), αιτήσεις ή ευκαιρίες μπορεί να κάνει την επιδίωξη ενός διαφορετικού σχεδιασμού πιο σχετική.
2. Κάποιος συνδυασμός ευκαιριών ή δυσκολιών μπορεί να απαιτήσει τον επαναπροσδιορισμό της σειράς της τρέχουσας ακολουθίας.

Για να αντιμετωπίσει τις εμφανίσεις τέτοιων συμβάντων κατά τον ορισμό μιας ακολουθίας σε ένα αντιδραστικό σχέδιο, ο BOD εισάγει τη χρήση τριών τύπων στοιχείων σχεδίου (plan elements) ώστε να γίνει προσέγγιση αντιδραστικού σχεδιασμού. Για επεξήγηση αυτών των τριών τύπων, θα χρησιμοποιηθεί ένα παράδειγμα συμπεριφοράς μιας τεχνητής μαϊμούς.

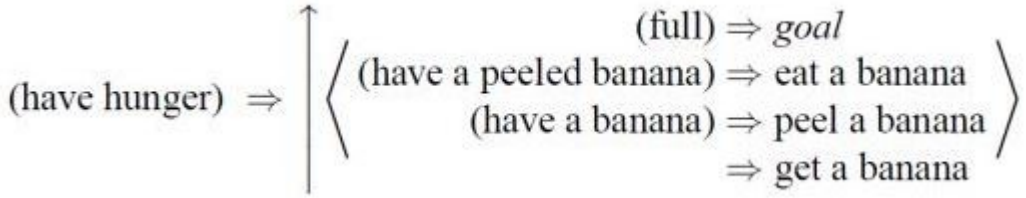
- Μοτίβο δράσης (action pattern): μια ακολουθία θεμελιακών στοιχείων, που μπορούν να είναι είτε ενέργειες είτε αισθητηριακά κατηγορήματα. Είναι χρήσιμα, επειδή επιτρέπουν έναν σχεδιαστή πρακτόρων να διατηρήσει το σύστημα όσο πιο απλό γίνεται, και να επιταχύνει βελτιστοποιήσεις στοιχείων που τρέχουν σε σειρά υπευθύνως.

$$\langle \text{get a banana} \rightarrow \text{peel a banana} \rightarrow \text{eat a banana} \rangle$$

Σχήμα 8: Παράδειγμα μοτίβου δράσης

Πηγή: http://www.doc.ic.ac.uk/

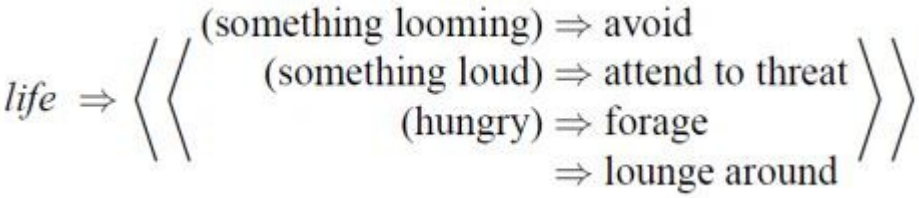
- Αρμοδιότητα (competence): αυτές κωδικοποιούν την προτεραιότητα στη θέση μίας προσωρινής παραγγελίας, όπως με τις φυσιολογικές ακολουθίες, και αυτό επιτρέπει μία επιπρόσθετη ευελιξία στην εκτεθειμένη συμπεριφορά. Στην παρακάτω εικόνα φαίνεται ο σχεδιασμός μιας αρμοδιότητας για την ίδια τεχνητή μαϊμού, αλλά με προτεραιότητες στις ενέργειές της οι οποίες εκτελούνται σύμφωνα με τη φορά του βέλους. Εδώ, η μαϊμού θα φάει μία ξεφλουδισμένη μπανάνα, αν της δοθεί μία. Αν της δοθεί μία μπανάνα, θα την ξεφλουδίσει, ή αν δεν της δοθεί τίποτα από τα παραπάνω θα αποπειραθεί να πάρει η ίδια μια μπανάνα. Η αρμοδιότητα τερματίζεται όταν είτε ο στόχος έχει επιτευχθεί, είτε αν είναι αδύνατη η ικανοποίηση οποιασδήποτε ενέργειας.



Σχήμα 9: Παράδειγμα αρμοδιότητας

Πηγή: http://www.doc.ic.ac.uk/

- Συλλογή κινήσεων (Drive Collection): παρέχει μία μορφή διαιτησίας (arbitrating) ανάμεσα σε στοιχεία σχεδίου ή στόχους, και χειρίζονται αλλαγές στο γενικό πλαίσιο που εμφανίζονται και στο περιβάλλον και εσωτερικά. Είναι σχεδιασμένη να μην τερματίζεται ποτέ και εκτελείται σε κάθε χτύπημα (tick) του κύκλου που κάνει το πρόγραμμα, όπου το κάθε στοιχείο της συλλογής κινήσεων με την υψηλότερη προτεραιότητα που ενεργοποιείται, περνάει τον έλεγχο για το επόμενο στοιχείο σχεδίου.

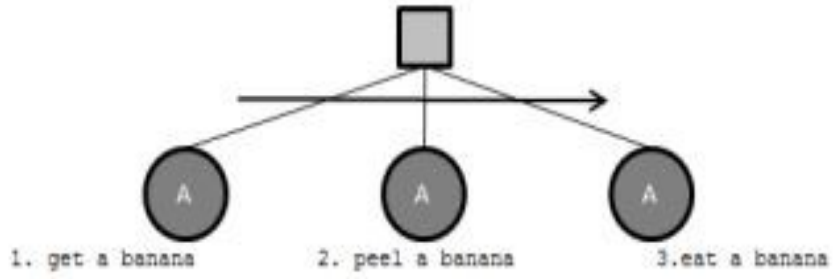


Σχήμα 10: Παράδειγμα συλλογής κινήσεων

Πηγή: http://www.doc.ic.ac.uk/

3.3.3.2.2 Αντιστοιχία με τα BT

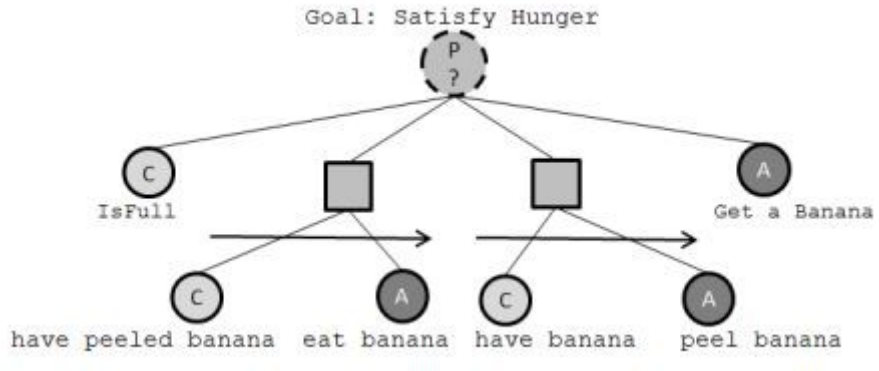
Με βάση το παράδειγμα που χρησιμοποιήθηκε με τη συμπεριφορά της τεχνητής μαϊμούς, διαπιστώνεται πως το μοτίβο δράσης μπορεί να βρεθεί σε αντιπαράθεση με κόμβο ακολουθίας ενός BT.



Εικόνα 19: Μοτίβο δράσης ως κόμβος ακολουθίας BT

Πηγή: http://www.doc.ic.ac.uk/

Στην εικόνα 20 φαίνεται ο αντίστοιχος κόμβος διαλέκτης για την αρμοδιότητα. Αξίζει εδώ να σημειωθεί, πως ο κόμβος διαλέκτης είναι στην πραγματικότητα ένας κόμβος διαλέκτης με προτεραιότητα, όπου τοποθετεί στην αριστερότερη θέση τις ενέργειες με μεγαλύτερη προτεραιότητα και κινείται προς τα δεξιά ακολουθώντας μια σειρά ενεργειών φθίνουσας προτεραιότητας.



Εικόνα 20: Αρμοδιότητα ως κόμβος διαλέκτης BT

Πηγή: http://www.doc.ic.ac.uk/

Περιγράφοντας τέτοιες αντιστοιχίες, είναι δυνατή η αναγνώριση της εφαρμογής των BT στη μεθοδολογία BOD. [12]

3.4 Αξιολόγηση δέντρου

Γενικά, η συμπεριφορά δεν αξιολογείται από τη ρίζα (root) του δέντρου σε κάθε ανανέωση. Αυτό θα είχε ως αποτέλεσμα το BT να συμπεριφέρεται πιο πολύ σαν ένα

[12] Year, F., Project, I., & Report, F. (2009). An A . I . Player for DEFCON : An Evolutionary Approach Using Behavior Trees.

Decision Tree, με ελάχιστη υποστήριξη για εκτέλεση και καταγραφή συμπεριφορών με την πάροδο του χρόνου.

Εντούτοις, η έννοια ενός δυναμικού κόμβου διαλέκτη και η ύπαρξη συνθηκών που εκτελούνται σε έναν κόμβο παραλληλίας μαζί με ολόκληρα υπόδενδρα, σημαίνει πως υπάρχει πάντα μια όψη της λογικής που είναι ενεργή ψηλά στο δέντρο, που ασχολείται με γεγονότα ειδικών περιπτώσεων. Αυτή η λογική εγκαθιδρύεται ως απαραίτητη από σχεδιαστές και διασκορπίζεται ελεύθερα γύρω από το δέντρο. [7]

3.5 Προηγμένες αρχιτεκτονικές

Γενικά, τα BT δεν υπόκεινται σε κάποια συγκεκριμένα κριτήρια ταξινόμησης, αλλά χαρακτηρίζονται από κάποια πρότυπα που μπορούν να τα διαχωρίσουν σε δύο κατηγορίες, τα BT πρώτης γενιάς (first-generation trees) και τα BT δεύτερης γενιάς (second-generation trees). Η βασική διαφορά αυτών των δύο κατηγοριών είναι η χρήση των δέντρων πρώτης γενιάς ως γενικό μοτίβο για σύνταξη κώδικα με C++, χωρίς ιδιαίτερη ικανότητα επαναχρησιμοποίησης που χαρακτηρίζεται κυρίως από απλές υλοποιήσεις. Στα δέντρα δεύτερης γενιάς δημιουργούνται ως μία ανεξάρτητη βιβλιοθήκη που μπορεί να χρησιμοποιηθεί σε οποιαδήποτε λογική παιχνιδιών.

Μία ισχυρή επέκταση στα δέντρα πρώτης γενιάς, είναι η δυνατότητα διαμοιρασμού δεδομένων ανάμεσα σε πολλά στιγμιότυπα, συγκεκριμένα στη δομή του δέντρου ή σε διαμοιραζόμενες παραμέτρους. Με αυτόν τον τρόπο μπορεί να μειωθεί σημαντικά η απαίτηση για μνήμη των συστημάτων που έχουν ως βάση BT, ειδικά σε σκηνές με μεγάλο αριθμό από πολύπλοκους χαρακτήρες.

Η πιο σημαντική απαίτηση για διαμοιρασμό δεδομένων είναι ο διαχωρισμός δύο βασικών εννοιών:

- **Κόμβοι (Nodes)** - εκφράζουν τα στατικά δεδομένα των BT δέντρων, για παράδειγμα τους δείκτες στους κόμβους-παιδιά (children) ενός σύνθετου κόμβου (composite node), ή σε κοινές παραμέτρους.
- **Εργασίες (Tasks)** - τα παροδικά δεδομένα κόμβου που απαιτούνται για την εκτέλει του κάθε BT κόμβου. Για παράδειγμα, ένας κόμβος ακολουθίας (sequence node) χρειάζεται έναν δείκτη για την τρέχουσα συμπεριφορά και οι ενέργειες συνήθως απαιτούν συμφραζόμενα (context). [23]

3.5.1 Βελτιώσεις στην απόδοση της πρόσβασης μνήμης

Ένα από τα σημαντικά μειονεκτήματα των δέντρων πρώτης γενιάς σε σύγχρονα μηχανήματα υπολογιστών (hardware), είναι τα μοτίβα πρόσβασης μνήμης που εκτίθενται όταν εκτελούνται τα BT. Αυτό αποτελεί κυρίως ζήτημα στο τωρινό console hardware που διαθέτει μειωμένη μνήμη προσκόμισης (memory prefetching).

Η κύρια αιτία ύπαρξης αυτών των μοτίβων είναι η δυναμική δέσμευση μνήμης των κόμβων του BT. Χωρίς προσεκτική διαχείρηση, ο BT κόμβος μπορεί να είναι διασκορπισμένος μέσα στη μνήμη με αποτέλεσμα ελλείψεις στην κρυφή μνήμη (cache), ενώ η εκτέλεση του BT μεταφέρεται από έναν κόμβο σε έναν άλλο. Αλλάζοντας τη

διάταξη των BT κόμβων στη μνήμη, είναι πιθανό να υπάρξει σημαντική βελτίωση στην απόδοση των BT ως προς τη μνήμη. [13]

3.5.2 Βελτιστοποιήσεις

Η τεχνητή νοημοσύνη που αναπτύσσεται για ένα βιντεοπαιχνίδι, συνήθως αναλώνεται στην εκτέλεση παραπλήσιων ενεργειών και κάνοντας συντακτικό έλεγχο διαμοιραζόμενης λογικής. Συνεπώς, η πρώτη και πιθανότατα η πιο σημαντική αποδοτικότητα αποκτάται με διαχωρισμό των δεδομένων επίβλεψης κατάστασης (stateful data) από τα δεδομένα χωρίς επίβλεψη κατάστασης (stateless data) που διαθέτει ο πράκτορας. Η λογική δομή του αυτόνομου πράκτορα είναι αμετάβλητη, χωρίς επίβλεψη κατάστασης και στατική. Ένας ανεξάρτητος κόμβος όμως, μπορεί να θέλει να συλλάβει δεδομένα με επίβλεψη κατάστασης ώστε να διατηρήσει την εκτέλεση για οποιοδήποτε στιγμιότυπο ενός πράκτορα. Για να επιτύχει αυτόν τον διαχωρισμό, οι τιμές των μελών κλάσεων κάθε κόμβου θεωρούνται χωρίς επίβλεψη κατάστασης, και ο κόμβος θα προσδιορίσει δεδομένα επίβλεψης κατάστασης ανά πράκτορα ώστε να συλλάβει τιμές των μελών. Με αυτόν τον τρόπο, ακόμα και όσο νέοι πράκτορας πληθαίνουν, το ποσό της μνήμης που καταλαμβάνεται από τις δενδρικές δομές της τεχνητής νοημοσύνης παραμένει το ίδιο. [14]

3.5.3 Υβριδικά συστήματα

Τα BT αποτελούν μία ισχυρή τεχνική δημιουργίας ευανάγνωστου και ευέλικτου σχεδιασμού τεχνητής νοημοσύνης. Προσφέρουν πολλές δυνατότητες που από μόνες τους αφήνουν περιθώρια πειραματισμών σε μεγάλο βάθος. Μπορούν όμως να υπάρξουν και σε συνδυασμό με άλλες τεχνικές με σκοπό τη δημιουργία υβριδικών συστημάτων (hybrid systems) που θα υλοποιούν κάποια συμπεριφορά. Σε αυτήν την κατεύθυνση οδηγούμαστε λόγω της δυσκολίας υλοποίησης συμπεριφοράς που σχετίζεται με καταστάσεις με BT.

Μπορεί για παράδειγμα να υλοποιηθεί ένα υβριδικό σύστημα που αποτελείται από BT και χρησιμοποιούν μηχανή καταστάσεων (state machine) ώστε να φαίνεται κάθε στιγμή ποια συμπεριφορά εκτελείται. Ένας τέτοιος σχεδιασμός όμως επιφέρει επιπρόσθετα βάρη στους σχεδιαστές, πρόβλημα που μπορεί να επιλυθεί με τη δημιουργία BT που έχουν συμπεριφορά state machines, δηλαδή με δυνατότητα εντοπισμού σημαντικών γεγονότων που θα τερματίζουν το τρέχον subtree και θα ξεκινάνε την εκτέλεση ενός άλλου.

Τα BT έχουν περιορισμούς που κάνουν κάποιες υλοποιήσεις συμπεριφορών δύσκολες (όχι όμως ακατόρθωτες), όπως για παράδειγμα το σχεδιασμό ενός χαρακτήρα που αλλάζει συμπεριφορά με βάση εξωτερικούς παράγοντες και δεν αμφιταλαντεύεται απλά ανάμεσα σε κάποιες συγκεκριμένες συμπεριφορές. Τέτοιου είδους περιορισμοί μπορούν να επιλυθούν με υβριδικά συστήματα.

[13] Champandard, J. (2007, July 11). AiGameDev: Using Resource Allocators to Synchronize Behaviors

[14] https://gamedevdaily.io/advanced-behavior-tree-structures-4b9dc0516f92?gi=ef2b39861bc0

3.5.3.1 Υβριδικό Σύστημα με Συστήματα Σχεδιασμού (Behavior Tree/Planner Hybrid)

Η βασική προϋπόθεση αυτής της υβριδικής προσέγγισης είναι απλή: γίνεται συνδυασμός των πλεονεκτημάτων των BT και των Planners ώστε να παραχθεί ένα σύστημα τεχνητής νοημοσύνης που είναι ευέλικτο και ανθεκτικό σε αλλαγές σχεδιασμού, ενώ παράλληλα επιτρέπει πλήρη έλεγχο σε σχεδιαστές πάνω στη δομή των ενεργειών που είναι διαθέσιμη στον πράκτορα. Χρησιμοποιείται ένα μοντέλο κατάστασης κόσμου και ένας ευρετικός μηχανισμός, όπως και στους Planners. Ενώ όμως ένας planner θα χτίσει ακολουθίες από στοιχειώδεις ενέργειες δυναμικά και θα χρησιμοποιήσει τον ευρετικό μηχανισμό για να επιλέξει την καλύτερη, η υβριδική προσέγγιση χρησιμοποιεί τον ευρετικό μηχανισμό για να επιλέξει ανάμεσα από τα “κλαδιά” (branches) ενός προκατασκευασμένου BT.

Η χρήση του BT σε αυτήν την προσέγγιση, επιτρέπει στους σχεδιαστές να αποκτούν πλήρη έλεγχο στο ποιες ενέργειες είναι διαθέσιμες και είναι επίσης εύκολο να ξανασχεδιαστεί η δομή αυτών των ενεργειών ενώ γίνεται επανάληψη. Όμως τα BT είναι αρκετά ανθεκτικά σε αλλαγές σχεδιαστών, γιατί η αλλαγή στην εσωτερική δομή μιας ενέργειας πρέπει να έχει αντίκρυσμα σε κάποιους κόμβους-γονείς. Σε αυτό το σημείο εμπλέκεται ο Planner.

Για τους κόμβους φύλλα, το σημείο αυτό της προσομοίωσης απλά επιστρέφει την καταληκτική κατάσταση του κόσμου όπως συμβαίνει με τις στοιχειώδεις ενέργειες στους Planners. Οι σύνθετοι κόμβοι προσομοιώνουν κάθε μία από τις συμπεριφορές των παιδιών τους σε σειρά και έπειτα επιστρέφουν το συσσωρευμένο αποτέλεσμα. Οι κόμβοι διαλεχτές προσομοιώνουν τη συμπεριφορά για κάθε ένα από τα παιδιά τους ώστε να προσδιοριστεί ποια θα ήταν η πιθανή κατάσταση κόσμου στην επιλογή του εκάστοτε παιδιού. Ο κόμβος διαλεχτής τότε χρησιμοποιεί αυτές τις αξιολογήσεις για να προσδιορίσει ποιον κόμβο να επιλέξει, και επιστρέφει το αποτέλεσμα του κόμβου ως δικό του.

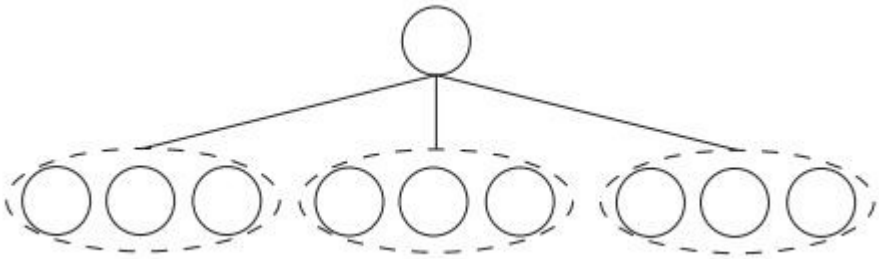
Αυτός ο υβριδικός σχεδιασμός επιτρέπει την κατασκευή BT που δεν γνωρίζουν τίποτα για την εσωτερική τους δομή. Είναι εφικτή η δημιουργία ενός κόμβου διαλεχτή επίθεσης που μπορεί να κατασκευαστεί από οποιοδήποτε αριθμό συμπεριφορών επίθεσης, και θα επιλέξει την πιο κατάλληλη σε σχέση με την τρέχουσα κατάσταση του κόσμου. Κάθε κόμβος διαλεχτής χρειάζεται να προσομοιώνει κάθε ένα από τα παιδιά του και να διαλέξει τελικά αυτό με την υψηλότερη αξιολόγηση μέσω του ευρετικού μηχανισμού. Αυτό επιτρέπει στους σχεδιαστές να τροποποιούν την εσωτερική δομή του δέντρου χωρίς να αλλάζουν παραπάνω κώδικα πιο ψηλά στην ιεραρχία. Επίσης επιτρέπει την αλλαγή της επίδρασης που έχει μία δράση κόμβου φύλλου στον κόσμο, χωρίς να υπάρχει όμως παράλληλα η ανησυχία της ανανέωσης ολόκληρου του δέντρου για να αντισταθμιστεί η οποιαδήποτε αλλαγή. Η δομή του BT επιτρέπει στους σχεδιαστές να έχουν πλήρη έλεγχο πάνω στις ικανότητες της τεχνητής νοημοσύνης, ενώ ο μηχανισμός των Planners χειρίζεται τον προσδιορισμό του πώς θα έπρεπε να είναι διαμορφωμένες οι ικανότητες της τεχνητής νοημοσύνης. [22]

3.6 Μάθηση (Learning) των Behavior Trees

3.6.1 Ορισμός της έννοιας του στόχου

Ένας στόχος G αποτελείται από μία σειρά P αποτελούμενη από n προϋποθέσεις και μια σειρά D από πλειάδες που αναπαριστούν την αποσύνθεση του στόχου. Ένας στόχος ικανοποιείται μόνο αν ικανοποιείται κάθε μία από τις n προϋποθέσεις της σειράς P. Η σειρά των αποσυνθέσεων D μπορεί να αναπαραστήσει ένα κλιμακωτό δέντρο από στόχους και τη δικιά τους αποσύνθεση.

Ένας στόχος δεν μπορεί να χρησιμοποιήσει έναν άλλο στόχο από τη δικιά του κατηγορία ως υπο-στόχο. Αυτό θα έκανε τον στόχο υψηλότερης κατηγορίας από τον κόμβο που θα είχε συμπεριλάβει. Το γεγονός ότι επιτρέπεται η αποσύνθεση ενός στόχου σε πολλαπλούς, επιτρέπει σε έναν προγραμματιστή βιντεοπαιχνιδιών να οδηγήσει τη σχεδίαση (planning) ανάμεσα σε κάποιους επιθυμητούς στόχους, αντί απλά να επιτρέπει στον σχεδιαστή να κάνει σχεδιασμό για έναν μόνο στόχο. [4]



Εικόνα 21: Παράδειγμα απλού στόχου με τρεις αποσυνθέσεις, εκ των οποίων η κάθε μία περιλαμβάνει τρεις στόχους

Πηγή: http://projekter.aau.dk/

3.6.2 Γενετικός Προγραμματισμός

Ο γενετικός προγραμματισμός συνίσταται σε έναν αλγόριθμο βελτιστοποίησης, που αντλείται από της τεχνικές εξέλιξης που συναντιούνται στη βιολογική εξέλιξη.

Στον γενετικό προγραμματισμό, μία λύση σε πρόβλημα είναι κωδικοποιημένη μέσα σε μία δενδρικά βασιζόμενη δομή, η οποία ενεργοποιείται από εξελικτικούς φορείς. Για παράδειγμα, ένα πρόβλημα μπορεί να είναι να χτιστεί μία μαθηματική έκφραση που υπολογίζει το π. Μία αναπαράσταση αυτού του προβλήματος θα μπορούσε να είναι ένα δέντρο έκφρασης. Σε αυτήν την αναπαράσταση, σταθεροί ή μεταβλητοί κόμβοι εισόδου θα βρίσκονταν στα φύλλα, και μαθηματικοί τελεστές όπως ο πολλαπλασιασμός, η διαίρεση ή τριγωνομετρικές συναρτήσεις θα βρίσκονταν σε εσωτερικούς κόμβους. Κατά τη διάρκεια της εξέλιξης, πιθανοί εξελικτικοί τελεστές θα μπορούσαν να ανασυνδυάσουν το δέντρο μεταστρέφοντας υπόδεντρα ή επεξεργάζοντας κόμβους. [2]

Ο γενετικός προγραμματισμός έχει χρησιμοποιηθεί από τους Hauptmann και Sipper για να αξιολογηθούν θέσεις στη σκακιέρα σε τελικά στάδια παιχνιδιού σκακιού όπου μένουν λίγα πιόνια πάνω της. Σημειώθηκε πως το γεγονός ότι ο γενετικός προγραμματισμός

επιτρέπει την ανθρώπινη γνώση να παρουσιάζεται εύκολα μέσω των συναρτήσεων και των τερματικών που χρησιμοποιούνται. Ο πράκτορας που χρησιμοποιούσε τα εξελιγμένα δέντρα αξιολόγησης, ήταν ικανός να φέρει ισοπαλία απέναντι σε μία από τις πιο νέες εξελιγμένες μηχανές σκακιού. [9]

3.6.3 Μετα-ευρετικός Μηχανισμός (Metaheuristic)

Μετα-ευρετικός μηχανισμός (metaheuristic) καλείται μία υψηλού επιπέδου διαδικασία που είναι σχεδιασμένη ώστε να βρει έναν ευρετικό μηχανισμό (heuristic) που θα προσφέρει μια καλή λύση σε ένα πρόβλημα βελτιστοποίησης, ειδικά όταν οι πληροφορίες που παρέχονται είναι ατελείς ή ελλιπείς.

Οι περισσότεροι μετα-ευρετικοί μηχανισμοί χαρακτηρίζονται από ορισμένες ιδιότητες:

- είναι στρατηγικές που οδηγούν την διαδικασία της αναζήτησης
- ο στόχος είναι να εξερευνηθεί αποτελεσματικά ο χώρος αναζήτησης για να βρεθούν οι σχεδόν ιδεατές λύσεις
- τεχνικές που απαρτίζουν τους μετα-ευρετικούς αλγορίθμους έχουν εύρος από απλές τοπικές διαδικασίες αναζήτησης μέχρι πολύπλοκες διαδικασίες μάθησης (learning)
- οι μετα-ευρετικοί αλγόριθμοι είναι συνήθως κατά προσέγγιση μη ντετερμινιστικοί
- δεν είναι οριζόμενες με βάση το πρόβλημα [13]

3.6.4 Σύνδεση με τα Behavior Trees

Μία προτεινόμενη προσέγγιση όπου θα συνδυάζονται τα BT μαζί με γενετικό προγραμματισμό και μετα-ευρετικό μηχανισμό, ξεκινάει με τον ορισμό των ενεργειών που μπορεί να εκτελέσει ο πράκτορας και ποιες συνθήκες μπορεί να παρατηρήσει στο περιβάλλον. Επίσης ορίζεται μία συνάρτηση καταλληλότητας (fitness function), η οποία δέχεται ως είσοδο ένα BT και δίνει ως αποτέλεσμα μία τιμή καταλληλότητας (fitness value) ανάλογα με το πόσο πιο κοντά είναι το BT στο να επιτύχει ένα δοσμένο στόχο. Μία εμπειρικά οριζόμενη μεταβλητή χρόνου t (σε δευτερόλεπτα) χρησιμοποιείται στη διαδικασία εκτέλεσης, όπου το BT εκτελείται διαρκώς αλλά η συνάρτηση καταλληλότητας αξιολογείται μόνο για τα περασμένα t δευτερόλεπτα. Η βαθμιαία αλλαγή στη συνάρτηση καταλληλότητας αξιολογείται ώστε να καθορίσει την πορεία του αλγορίθμου μάθησης (learning algorithm).

Στη συγκεκριμένη προσέγγιση ακολουθείται μία στρατηγική μετα-ευρετικής μάθησης (metaheuristic learning), όπου χρησιμοποιείται ένας άπληστος αλγόριθμος (greedy algorithm) αρχικά και μόλις αποτύχει, χρησιμοποιείται ο αλγόριθμος γενετικού προγραμματισμού. Αυτός ο αλγόριθμος επίσης θα χρησιμοποιηθεί όταν ο άπληστος αλγόριθμος δεν θα μπορεί να προσφέρει κάποιο αποτέλεσμα, ή όταν η πολυπλοκότητα της λύσης είναι αυξημένη. Αυτό το μείγμα μάθησης βασισμένο σε ευρετικές προσεγγίσεις υπήρξε ώστε να ελαχιστοποιήσει το χρόνο μάθησης σημαντικά σε σύγκριση με τη χρήση αποκλειστικά γενετικού προγραμματισμού, ενώ ταυτόχρονα επιτυγχάνεται ένα βέλτιστο BT που ικανοποιεί έναν δοθέν στόχο. [16]

3.6.5 Μάθηση Q

Η μάθηση Q είναι ένας αλγόριθμος ενισχυτικής μάθησης (reinforcement learning) που δημιουργεί και συντηρεί έναν πίνακα από τιμές που υπολογίζουν την χρησιμότητα της κάθε ενέργειας σε μία κατάσταση. Ο πράκτορας λαμβάνει μία συνάρτηση που συνδέει τις καταστάσεις με μια προκαθορισμένη ανταμοιβή. Ο αλγόριθμος έπειτα επιστρέφει ανταμοιβές στα ζευγάρια κατάστασης-ενέργειας, που οδηγούν σε καταστάσεις ανταμοιβών δημιουργώντας έτσι σταδιακά βελτιωμένες προβλέψεις χρησιμότητας. [19]

Μία κατάσταση μπορεί να αποτελείται από οποιαδήποτε διαμόρφωση μεταβλητών μέσα σε ένα περιβάλλον. Η μάθηση Q δεν είναι ένας αλγόριθμος βασισμένος σε ένα μοντέλο, και θα καταγράψει μόνο μία τιμή-Q για μία κατάσταση που έχει ήδη επισκεφθεί από τον πράκτορα κατά τη διάρκεια της εκπαίδευσης (training). Η μάθηση Q μπορεί επίσης να εφαρμοστεί ως ένα μοντέλο αντίληψης. Για παράδειγμα, αν οι αντιλήψεις αποτελούνταν αποκλειστικά από στοιχεία για την υγεία των πρακτόρων (health values), τίποτα δεν θα μπορούσε να γίνει γνωστό σχετικά με τις τοποθεσίες των πρακτόρων μέσα στο περιβάλλον (το οποίο είναι πραγματικά ένας πολύ σημαντικός παράγοντας).

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'}(Q(s', a')))$$

Σχήμα 11: Φόρμουλα ανανέωσης μάθησης Q

Πηγή: http://openaccess.city.ac.uk/3000/

Στη φόρμουλα ανανέωσης της μάθησης Q:

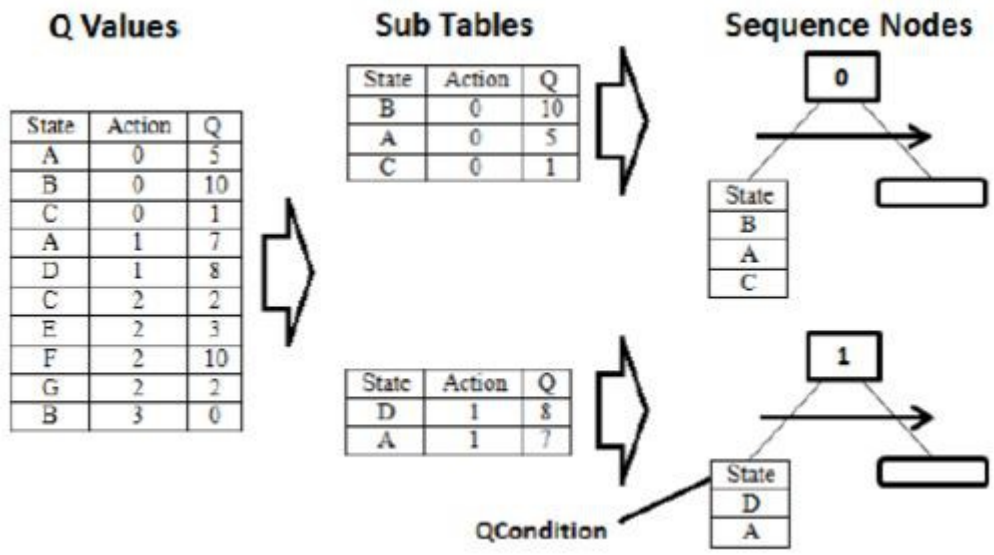
- Q(s, a) είναι η τιμή Q του τρέχοντος ζευγαριού κατάστασης-ενέργειας
- Q(s', a') είναι η τιμή Q του ζευγαριού διαδόχου κατάστασης-ενέργειας
- r είναι η ανταμοιβή που συνδέεται με την κατάσταση του διαδόχου
- α είναι η παράμετρος του ρυθμού εκμάθησης (learning rate parameter)
- γ είναι η παράμετρος του παράγοντα έκπτωσης (discount factor parameter) [18]

3.6.5.1 Ενσωμάτωση Μάθησης Q σε ένα BT

Σε ένα περιβάλλον μάθησης Q, οι κόμβοι ακολουθίας ενός BT σε βαθύ επίπεδο μπορούν να θεωρηθούν ως ενέργειες, επειδή ομαδοποιούν μαζί χαμηλότερου επιπέδου ενέργειες και τις εκτελούν με τη σειρά χωρίς διακοπή. Κάθε κόμβος ακολουθίας επίσης περιέχει κόμβους συνθήκης προς την αρχή της ακολουθίας.

Ο αλγόριθμος [19] ξεκινάει με ένα BT ως είσοδο. Το δέντρο αναλύεται για να βρεθούν οι κόμβοι ακολουθίας στο πιο βαθύ επίπεδο. Αυτοί οι κόμβοι αναγνωρίζονται ως ενέργειες για το στάδιο της ενισχυτικής μάθησης. Αυτές οι ενέργειες χρησιμοποιούνται σε μία offline φάση της μάθησης Q για να παράξουν έναν πίνακα με τιμές Q. Ο πίνακας μετά διαιρείται σε υποπίνακες ανά ενέργεια και οι καταστάσεις με την υψηλότερη τιμή για την ενέργεια εξάγονται σε κόμβους συνθήκης-Q μέσα στο BT. Οι κόμβοι συνθήκης μέσα στο BT που δόθηκε ως είσοδος αντικαθιστώνται από τους κόμβους συνθήκης-Q. Τέλος, η τοπολογία των BT αναδιοργανώνεται ταξινομώντας κάθε παιδί κάθε κόμβου ανάλογα

με την μέγιστη τιμή Q, γεγονός που παρέχει σε σχεδιαστές τεχνητής νοημοσύνης μια βελτιστοποιημένη μετάλλαξη των BT. [18]



Εικόνα 22: Εισαγωγή τιμών Q στο BT

Πηγή: http://openaccess.city.ac.uk/3000/

3.6.6 Εξελίσσοντας τα Behavior Trees

Τα BT είναι μία δομή με συνθήκες, σύνθετους κόμβους και ενέργειες. Διαθέτουν επίσης εσωτερικούς κόμβους που υπαγορεύουν τη ροή της εκτέλεσης, όπως επίσης και κόμβους που παρέχουν έλεγχο ή ανίχνευση πληροφοριών. Τοιουτοτρόπως, τα BT μπορούν να εξελιχτούν με τη χρήση τεχνικών γενετικού προγραμματισμού.

Μία εφαρμογή των εξελισσόμενων BT είναι στον ευθύ έλεγχο χαρακτήρα [1]. Σε αυτήν την περίπτωση, το BT κατευθύνει έναν χαρακτήρα να τρέχει και να κάνει άλματα σε ένα επίπεδο που είναι βασισμένο στο είδος “platforming” (πλατφόρμα) των βιντεοπαιχνιδιών. Αυτός ο έλεγχος χαρακτήρα κατορθώνεται μέσω της δημιουργίας ενός χάρτη του κόσμου πάνω στον οποίο θα υποβληθούν ερωτήσεις μέσω κόμβων συνθηκών (condition nodes), όπως και κόμβους ενεργειών (action nodes) που μιμούνται τα κουμπιά που πατάει ο ανθρώπινος παίκτης. Μία συντακτική γραμματική δημιουργήθηκε με ήδη ορισμένα μοτίβα ώστε να ρυθμιστούν τα υπόδεντρα που δημιουργήθηκαν κατά την εξέλιξη και να αποφευχθούν μη έγκυρα δέντρα. Οι συγγραφείς επίσης τονίζουν το γεγονός ότι τα BT που υπήρξαν ως αποτέλεσμα είναι αναγνώσιμα από τον άνθρωπο, επιτρέποντας έτσι αναλυτικό έλεγχο, όπως επιθυμεί η βιομηχανία βιντεοπαιχνιδιών. [15]

3.7 Μέλλον των Behavior Trees

Με τη διαρκή εξέλιξη των βιντεοπαιχνιδιών και των απαιτήσεων για ολοένα και περισσότερους μη-ανθρώπινους παίκτες μέσα σε ένα παιχνίδι, όλες οι τρέχουσες τεχνικές σχεδιασμού τεχνητής νοημοσύνης μπαίνουν στο μικροσκόπιο ώστε να

εντοπιστούν ελλείψεις και μειονεκτήματα και να εξαλειφθούν και να αντικατασταθούν καταλλήλως. Αυτή τη στιγμή, η βιομηχανία των βιντεοπαιχνιδιών χρησιμοποιεί κατά κόρον ΒΤ, που όμως δείχνει αδυναμία ικανοποίησης κάποιων πιο πολύπλοκων χαρακτηριστικών τεχνητής νοημοσύνης. Το τρέχον φλέγον θέμα είναι η δημιουργία ρεαλιστικής τεχνητής νοημοσύνης που βασίζεται σε ατελείς πληροφορίες, κάτι που τα ΒΤ δεν μπορούν να ικανοποιήσουν.

3.7.1 Προβλήματα των ΒΤ στη σύγχρονη βιομηχανία βιντεοπαιχνιδιών

Οι απαιτήσεις πλέον απαιτούν το χειρισμό μεγάλου αριθμού παραγόντων εισαγωγής, που χρησιμοποιούνται για να γίνει επιλογή ανάμεσα από έναν εξίσου μεγάλο αριθμό συμπεριφορών, κάνοντας τις υπάρχουσες τεχνολογίες τεχνητής νοημοσύνης να δείχνουν τη φθορά και τις βλάβες τους. Γίνεται αδύνατο για προγραμματιστές τεχνητής νοημοσύνης να σχεδιάσουν κάθε δυνατό σενάριο μέσα σε μία μηχανή καταστάσεων ή ένα δέντρο, και γίνεται αδύνατο να σχεδιαστεί ένα επαρκές ποσό από δοκιμές που θα επιβεβαιώνουν ότι ο πράκτορας φέρεται όπως είναι το προβλεπόμενο όταν τοποθετείται μπροστά σε παίκτες - ανθρώπους.

Όπως προαναφέρθηκε, τα ΒΤ χωλαίνουν στο να χειριστούν πιο πολύπλοκες συμπεριφορές, γεγονός που αποτελεί την κύρια πρόκληση στο σχεδιασμό τεχνητής νοημοσύνης αιχμής. Τα προβλήματα που παρουσιάζει ένα ΒΤ προέρχονται από την δυνατότητα που προσφέρουν στο χειρισμό πρακτόρων όταν αυτοί αυξάνονται σε πλήθος και πολυπλοκότητα.

Όταν στη διαδικασία ανάπτυξης βιντεοπαιχνιδιών αντιμετωπίζεται το πρόβλημα της επέκτασης της τεχνητής νοημοσύνης, ο αριθμός των σφαλμάτων και ο χρόνος που απαιτείται για να διορθωθούν αυξάνεται εκθετικά. Συνεπώς όταν ο προγραμματισμός ενός παιχνιδιού μένει πίσω από το σχεδιασμό του, το τελικό παιχνίδι καταλήγει να έχει πολύ λιγότερο περιεχόμενο και το γενικό του πλαίσιο γίνεται τετριμμένο. Κατά συνέπεια, πολλά παιχνίδια δεν επενδύουν πόρους στην αντιμετώπιση αυτών των δύσκολων προβλημάτων και καταλήγουν να υλοποιούν κοινότοπη τεχνητή νοημοσύνη που έχει ήδη κυκλοφορήσει και είναι ήδη προβλέψιμη στην αγορά.

Είναι επομένως αναγκαία για την μετέπειτα πορεία της βιομηχανίας των βιντεοπαιχνιδιών, μία νέα, καλύτερη και εύρωστη τεχνική στην τεχνητή νοημοσύνη του σήμερα.

4. ΠΑΙΧΝΙΔΙ CAPTURE THE FLAG ΣΕ UNITY3D

4.1 Περιγραφή Unity3D

Η μηχανή παιχνιδιών Unity3D είναι μία μηχανή παιχνιδιών που προσφέρει υποστήριξη σε πολλές πλατφόρμες και δίνει επίσης τη δυνατότητα της εύκολης μετάβασης μίας εφαρμογής από τη μία πλατφόρμα σε μια άλλη. Ταυτρόχρονα, προσφέρει μια πληθώρα εργαλείων οπτικού σχεδιασμού που απλοποιεί τη διαδικασία δημιουργίας ενός εικονικού κόσμου και της οπτικοποίησης ή επεξεργασίας των οντοτήτων που τον αποτελούν, ενώ η εκτέλεση του παιχνιδιού είναι δυνατή μέσα από το ίδιο το περιβάλλον.

Η πλατφόρμα Unity3D δίνει την ευχέρεια στον προγραμματιστή να επιλέξει αν θα αναπτύξει την εφαρμογή του σε μία διερμηνευμένη γλώσσα προγραμματισμού, τη Javascript, ή θα προχωρήσει στην ανάπτυξη με μία αντικειμενοστρεφή γλώσσα, τη C#. Και οι δύο αυτές εναλλακτικές ενδείκνυνται για προγραμματιστές χωρίς μεγάλη πείρα στον προγραμματισμό, αφού υπάρχει ακμάζουσα και υποστηρικτική κοινότητα γύρω από την πλατφόρμα Unity3D.

Είναι εύκολη στη χρήση ως μηχανή παιχνιδιών, και προσφέρει άμεσα οπτικοποιημένα αποτελέσματα ενώ γράφεται ο κώδικας στον συντάκτη κώδικα (editor), χωρίς καθυστερήσεις μεταγλώττισης (compile) και χτισίματος (build), αφού είναι ενέργειες που εκτελούνται αυτόματα.

Η χρήση της είναι δωρεάν χωρίς να στερείται χαρακτηριστικών έναντι της επί πληρωμής έκδοσής της, και επίσης διαθέτει ένα κατάστημα επιπλέον χαρακτηριστικών για χρήση, πολλά εκ των οποίων είναι και αυτά δωρεάν.

Τα παραπάνω χαρακτηριστικά καθιστούν τη μηχανή παιχνιδιών Unity3D την ιδανική πλατφόρμα να ξεκινήσει ένας καινούριος στο χώρο προγραμματιστής την επαφή του με την ανάπτυξη παιχνιδιών, βλέποντας άμεσα αποτελέσματα της δουλειάς του και βρίσκοντας αμέριστη βοήθεια μέσω των κοινοτήτων της πλατφόρμας στο διαδίκτυο.

4.2 Περιγραφή παιχνιδιού Capture The Flag (CTF)

Υπάρχουν δύο ομάδες σε μία αρένα που διαθέτουν η κάθε μία από μία σημαία και από 4 παίκτες. Οι δύο ομάδες, χωριζόμενες ως ομάδες Right και Left, προσπαθούν να πιάσουν τη σημαία της αντίπαλης ομάδας και να επιστρέψουν στη δικιά τους αρένα. Δηλαδή, ένας παίκτης της Right ομάδας θέλει να πιάσει τη σημαία που βρίσκεται στην αρένα της Left ομάδας και να την επιστρέψει στη δικιά του βάση. Οι παίκτες που βρίσκονται στην αρένα τους μπορούν να “αιχμαλωτίσουν” τους εχθρικούς παίκτες, στέλνοντάς τους πίσω στη βάση τους.

4.3 Επιλογή CTF παιχνιδιού ως πεδίο δοκιμών

Η ανάπτυξη ενός παιχνιδιού CTF ως πεδίο δοκιμών για BT αρχιτεκτονικές επιλέχθηκε γιατί προσφέρει τόσο δυνατότητες απλών όσο και πιο σύνθετων υλοποιήσεων συμπεριφορών. Η ανάπτυξη που περιγράφηκε, ακολούθησε μια απλή δομή που όμως αφήνει περιθώρια για πολλές επεκτάσεις που μπορούν να ενσωματώσουν στοιχεία από διάφορες τεχνικές.

Ένα CTF παιχνίδι είναι μία ιδανική επιλογή για έλεγχο συμπεριφορών όπως η αμυντική ή η επιθετική τακτική, καθώς και η ιεράρχηση στόχων. Για παράδειγμα, με τον τρόπο που γίνεται η αξιολόγηση του CTF BT, το κυνήγι και η αρπαγή της σημαίας είναι πάντα αυτό που πρέπει να έχει ως πρώτη προτεραιότητα ο πράκτορας. Είναι λοιπόν ένα πεδίο δοκιμών που επιτρέπει πειραματισμούς τόσο ως προς τη συμπεριφορά ενός πράκτορα μεμονωμένα, είτε την παρακολούθηση ενός πράκτορα ως μέρος μίας ομάδας με έναν κοινό στόχο.

4.4 Αρχιτεκτονικές Μαυροπίνακα (Blackboard Architectures)

Ένα σύστημα μαυροπίνακα δεν είναι από μόνο του ένα εργαλείο δημιουργίας αποφάσεων, αλλά αποτελεί έναν μηχανισμό που ελέγχει τις ενέργειες από αρκετούς δημιουργούς αποφάσεων. Ως μοτίβο σχεδίασης χαρακτηρίζεται από ευελιξία που το κάνει ελκυστικό στους σχεδιαστές.

Η μορφή του μαυροπίνακα εξαρτάται από την εφαρμογή. Οι αρχιτεκτονικές μαυροπίνακα έχουν ποικίλες χρήσεις, μπορούν να χρησιμοποιηθούν για παράδειγμα στην πηδαλιούχηση (steering) χαρακτήρων. Σε αυτήν την περίπτωση, ο μαυροπίνακας θα περιέχει τριών διαστάσεων (3D) τοποθεσίες ή και συνδυασμούς κινήσεων. Χρησιμοποιούμενος ως αρχιτεκτονική δημιουργίας αποφάσεων, μπορεί να περιέχει και πληροφορίες σχετικά με την τρέχουσα κατάσταση του παιχνιδιού, την τοποθεσία των εχθρών ή των πόρων, και την εσωτερική κατάσταση ενός χαρακτήρα.

Υπάρχουν γενικά χαρακτηριστικά, που οδηγούν κατά ένα τρόπο προς την μορφοποίηση μιας γενικής γλώσσας μαυροπίνακα. Επειδή ο στόχος είναι να επιτρέπονται διαφορετικά κομμάτια κώδικα να επικοινωνούν μεταξύ τους απρόσκοπτα, η πληροφορία στον μαυροπίνακα χρειάζεται τουλάχιστον τρία μέρη: τιμή, ταυτοποίηση τύπου και σημασιολογίας. Ο μαυροπίνακας τυπικά θα κληθεί να αντιμετωπίσει μία ποικιλία από διαφορετικούς τύπους δεδομένων, συμπεριλαμβάνοντας και δομές (structures). Αυτά τα δεδομένα μπορεί για παράδειγμα να εκπροσωπούν την τιμή της υγείας (health value) ενός πράκτορα ως ακέραια τιμή ή την τοποθεσία του εκφραζόμενη σε τρισδιάστατα διανύσματα.

Επειδή τα δεδομένα στον μαυροπίνακα μπορούν να είναι διαφορετικών τύπων, το περιεχόμενό τους πρέπει να μπορεί να είναι αναγνωρίσιμο. Αυτό μπορεί να είναι κώδικας απλού τύπου, σχεδιασμένος να αφήνει στον σχεδιαστή την ελευθερία χρήσης του κατάλληλου τύπου για τα δεδομένα (στην C/C++ αυτό γίνεται συνήθως με διαμόρφωση του τύπου (typecasting) της τιμής στον κατάλληλο τύπο). [5]

4.5 Δομή του CTF BT

Το δέντρο συμπεριφοράς που αναπτύχθηκε για το CTF παιχνίδι, αποτελείται από κλάσεις που αναπαριστούν τους κύριους κόμβους του BT, όπως σύνθετοι κόμβοι και κόμβοι διακοσμητές, κλάσεις που αποτελούν τους κόμβους των φύλλων του δέντρου και κλάσεις που επισυνάπτονται στον παίκτη, στους παίκτες - ρομπότ (bots), στην αρένα, στις σημαίες και στην κεντρική δομή του δέντρου που απαντάται σε κάθε bot. Ο πηγαίος κώδικας για το δέντρο συμπεριφοράς και τους κόμβους φύλλα που αποτελούν το παιχνίδι CTF, μπορεί να προσπελαστεί από το σύνδεσμο:

https://github.com/Raniato/BehaviorTrees.

Συγκεκριμένα οι κλάσεις των κύριων κόμβων είναι:

- BTree : όπου γίνεται καθορισμός του τι είναι η ρίζα (root) ενός δέντρου και τι blackboard data (δεδομένα μαυροπίνακα)
- Node : γενική μορφή κόμβου του BT, ορίζει τις πιθανές μορφές του αποτελέσματος (επιτυχία, αποτυχία, σε εξέλιξη = SUCCESS, FAIL, PROCESSING)
- Composite : σύνθετος κόμβος που κληρονομεί το Node
- Decorator : κόμβος διακοσμητής που κληρονομεί το Node
- Inverter : αντιστρέφει το αποτέλεσμα εκτέλεσης, κληρονομεί το Decorator
- Parallel : επιστρέφει μήνυμα επιτυχίας αν επιστρέψουν και όλα τα παιδιά επιτυχία, αν έστω και ένα επιστρέψει αποτυχία τότε επιστρέφει όλος ο κόμβος αποτυχία, κληρονομεί το Composite
- ParallelSelector : αν έστω και ένα παιδί επιστρέψει μήνυμα επιτυχίας, το ParallelSelector τερματίζει την εκτέλεση όλων των παιδιών και επιστρέφει επιτυχία. Ομοίως αν έστω και ένα παιδί επιστρέψει μήνυμα επιτυχίας, τερματίζονται όλα τα παιδιά και επιστρέφεται μήνυμα αποτυχίας. Κληρονομεί το Composite
- RepeatUntilFail : επαναλαμβάνει μία συμπεριφορά μέχρι αυτή να επιστρέψει μήνυμα αποτυχίας, κληρονομεί το Decorator
- Selector : αν ένας κόμβος επιστρέψει success, τότε δεν γίνεται προσπέλαση των υπόλοιπων παιδιών του και επιστρέφει μήνυμα επιτυχίας. Κληρονομεί το Decorator
- Succeeder : επιστρέφει πάντα μήνυμα επιτυχίας. Κληρονομεί το Decorator
- Sequence : κόμβος ακολουθίας με σειριακή προσπέλαση των παιδιών του, κληρονομεί το Composite
- Leaf : κόμβος φύλλο του δέντρου, κληρονομεί το Node

Η κλάση BTree υλοποιεί όπως προαναφέρθηκε, συναρτήσεις που ορίζουν τη ρίζα και τα δεδομένα μαυροπίνακα του δέντρου. Ο χειρισμός των δεδομένων μαυροπίνακα γίνεται μέσω των συναρτήσεων GetValue(string name) και SetValue(string name), όπου ο χρήστης μπορεί να χρησιμοποιήσει οπουδήποτε στο πρόγραμμα με τις κατάλληλες κλήσεις και να προσθέσει μία τιμή για μία δεδομένη μεταβλητή του δέντρου, ή να λάβει και να ελέγξει την τρέχουσα τιμή κάποιας μεταβλητής ώστε να καθορίσει τη συμπεριφορά. Για παράδειγμα σε κάθε φύλλο κόμβο γίνεται GetValue στα δεδομένα μαυροπίνακα ώστε να γίνει έλεγχος της κατάστασης του πράκτορα και να αποφασιστεί επιστροφή επιτυχούς (SUCCESS) ή ανεπιτυχούς (FAIL) αποτελέσματος.

Στα δεδομένα μαυροπίνακα που έχω ορίσει στην κλάση BehaviorTreeScript όπου γίνεται το τελικό στήσιμο του δέντρου, έχουν οριστεί δεδομένα μαυροπίνακα που καλύπτουν ένα εύρος καταστάσεων. Το δέντρο που ορίζεται σε αυτήν την κλάση, καθορίζει για κάθε ρομπότ (bot):

- την ομάδα του

- την αναφορά αντικειμένου παιχνιδιού (GameObject) της σημαίας που πρέπει να πιάσει η ομάδα του
- το αν έχει τη δεδομένη στιγμή τη σημαία
- την τοποθεσία του
- την τοποθεσία όπου θα επανεμφανιστεί αν τον αιχμαλωτίσει ένα εχθρικό ρομπότ
- το αν είναι αιχμαλωτισμένο
- την περιοχή στην αρένα που του θεωρείται ως βάση του
- το τρέχον αντικείμενο που καλείται να αναζητήσει
- τους παράγοντες που επηρεάζουν την ταχύτητα που κινείται
- τους εχθρούς που είναι κοντά του
- τους συμμάχους του
- την τρέχουσα κατάστασή του, η οποία μπορεί να είναι DEFENDING (ακολουθεί τακτική άμυνας), GOING_FOR_THE_FLAG (είναι αυτό το ρομπότ που θα πάει να πιάσει τη σημαία και όχι άλλο), TAGGED (αιχμαλωτισμένο) και RETRIEVING_THE_FLAG (επιστροφή της σημαίας πίσω στη βάση).

Στους κόμβους φύλλα (leaf nodes) περιλαμβάνονται όλες οι συμπεριφορές που καλείται να υλοποιήσει κάθε ρομπότ στο παιχνίδι. Σύμφωνα με τους κανόνες που θεσπίστηκαν παραπάνω, αυτές οι ενέργειες διαχωρίστηκαν στις εξής συμπεριφορές που ορίζουν:

- CaptureFlag : την αιχμαλωσία της σημαίας
- ChaseEnemy : το κυνήγι ενός εχθρικού ρομπότ (ή του παίκτη - ανθρώπου)
- CatchEnemy : την σύλληψη ενός εχθρού
- CheckFlagCaptivity : τον έλεγχο για το αν η σημαία έχει ήδη συλληφθεί
- Defend : την άμυνα στη βάση
- GoForFlag : την κίνηση ενός ρομπότ να πάει για τη σημαία, αν δεν πάει κανένας άλλος
- ReleaseFlag : την ελευθέρωση της σημαίας σε περίπτωση που ο παίκτης που την είχε συλλήφθηκε
- Respawn : την επιστροφή του παίκτη που συνελήφθη από εχθρική μονάδα, πίσω στη βάση του
- ScorePoint : την εμφάνιση της ομάδας που κέρδισε το γύρο
- Seek : την πρόοδο που κάνει ένα ρομπότ ψάχνοντας το αντικείμενο που του έχει ανατεθεί
- SeekEnemyFlag : την αναζήτηση για τη σημαία που πρέπει να πιάσει η ομάδα του ρομπότ
- Tagged : την τοποθέτηση σε κατάσταση TAGGED που αργότερα οδηγεί στη συμπεριφορά Respawn

4.5.1 BT σε άλλα πεδία δοκιμών

Σε αυτό το κεφάλαιο περιγράφεται η υλοποίηση για ένα BT πάνω σε πεδίο δοκιμών CTF, αλλά τα BT σαφώς μπορούν να χρησιμοποιηθούν σε πολλά διαφορετικά πεδία, όπως για παράδειγμα ένα Real Time Strategy (RTS) παιχνίδι, που αφήνει και πιο πολλά περιθώρια ανάπτυξης τακτικών ομάδας (squad tactics).

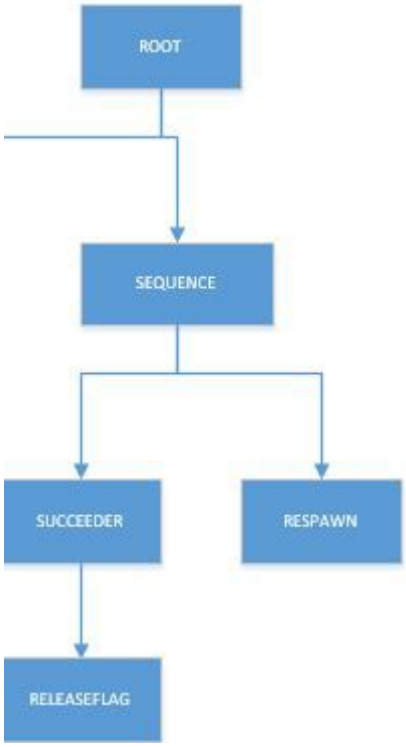
Σε κάθε περίπτωση, ένα BT δεν χρειάζεται να φτιάχνεται από το μηδέν, καθώς η κύρια δομή του είναι προσαρμοζόμενη σε μια γενική μορφή που επιτρέπει την επαναχρησιμοποίησή της. Ο κόμβος της ρίζας μπορεί να είναι στις περισσότερες περιπτώσεις ένας κόμβος διαλέκτης, και η κύρια μορφή κόμβου που χρησιμοποιείται ως απόρροια της κοινής λογικής κατά τη διάρκεια ενός σχεδιασμού παιχνιδιού, είναι αυτή του κόμβου ακολουθίας. Συνήθως οι πράκτορες καλούνται να εκτελέσουν μια σειριακή ροή ενεργειών, η οποία θα αποτελείται από κάποιους κόμβους παραλληλίας, περισσότερους κόμβους διαλέκτες ώστε να διαφαίνεται μια σαφής αλληλεπίδραση με το περιβάλλον και την κατάλληλη χρήση των κόμβων αντιστροφής, επιτυχίας και επανάληψης μέχρι αποτυχίας.

Από ένα πεδίο δοκιμών σε ένα άλλο, ακόμα και αν η λογική του παιχνιδιού καθεαυτού είναι αρκετά διαφορετική, η δομή ενός BT είναι εύκολα προσαρμοζόμενη και μπορούν εύκολα να αφαιρεθούν ενέργειες και να προστεθούν κάποιες άλλες χωρίς εμφάνιση σφαλμάτων. Σε ένα πεδίο δοκιμών όπου οι πράκτορες χωρίζονται σε κατηγορίες που καλούνται να εκτελέσουν διαφορετικά σύνολα ενεργειών, τότε η χρήση BT διευκολύνει πολύ την ανάπτυξη του παιχνιδιού. Η κεντρική λογική μπορεί να μείνει η ίδια, δηλαδή να παραμείνουν αναλλοίωτη κόμβοι παραλληλίας, κόμβοι διαλέκτες και κόμβοι ακολουθίας που διαθέτουν πολλούς κόμβους φύλλα ως παιδιά, και να αλλάξει μόνο ο κώδικας που χαρακτηρίζει τα παιδιά φύλλα. Ακόμα και με τόσες μικρές αλλαγές, μπορούν να προκύψουν πράκτορες που υλοποιούν τελείως διαφορετικές ενέργειες και συμπεριφορές.

4.6 Υπόδενδρα (Subtrees) της δομής BT του CTF

Η κεντρική δομή του BT που ακολουθεί το κάθε ρομπότ, είναι καθορισμένη και αναπτυγμένη στην κλάση BehaviorTreeScript, η οποία θα παρουσιαστεί σχηματικά. Κατά το σχεδιασμό, το κυρίως δέντρο είναι πιο εύκολο να χωριστεί σε μικρότερα υπόδενδρα (subtrees), γεγονός που καθιστά πιο σαφή την μετέπειτα ανάπτυξή του.

4.6.1 Υπόδενδρο Respawn



Εικόνα 23: Υπόδενδρο Respawn

Ο κόμβος της ρίζας είναι γονέας μίας ακολουθίας που εξασφαλίζει πως όταν ένα ρομπότ είναι TAGGED (συνελήφθη από κάποιον εχθρό), τότε θα απελευθερώνει τη σημαία και θα κάνει Respawn πίσω στη βάση του. Σημαντική σημείωση πως ο κόμβος της ρίζας είναι ένας κόμβος διαλέκτης, δηλαδή θα κάνει την επιλογή της εκτέλεσης ανάμεσα στα δύο κύρια υπόδενδρα, αυτό που εξετάζεται τώρα και στο επόμενο (άμυνας/επίθεσης ή αιχμαλωσίας της σημαίας).

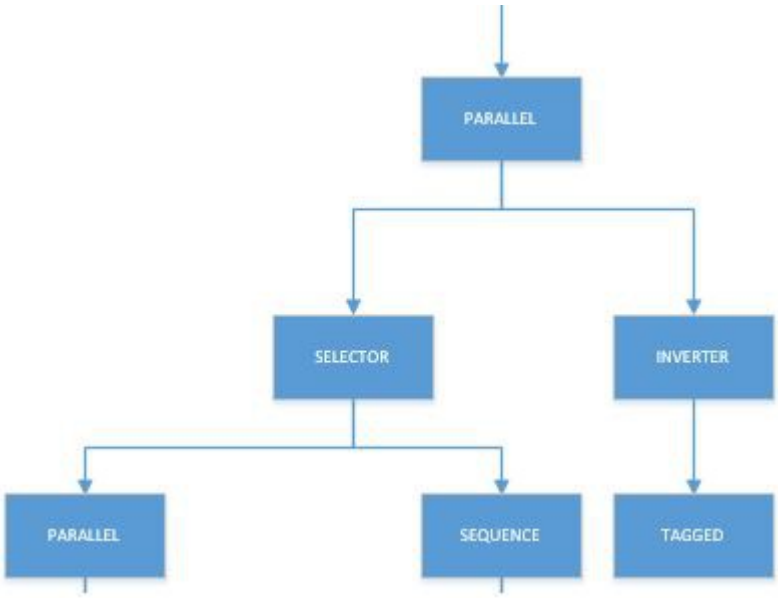
Αν το ρομπότ (bot) επιστρέψει μήνυμα αποτυχίας στο άλλο υπόδενδρο, τότε προχωράει στην εκτέλεση του υπόδενδρου Respawn. Όπως έχει ήδη αναφερθεί, ο κόμβος διαλέκτης (selector) σε αντίθεση με τον τυχαίο κόμβο διαλέκτη (random selector), θα εκτελέσει τα παιδιά του με αυστηρή σειρά. Άρα όσο το άλλο υπόδενδρο του ROOT επιστρέφει μήνυμα επιτυχίας (όπως φαίνεται παρακάτω, στηρίζεται σε έναν κόμβο παραλληλίας), το υπόδενδρο Respawn δεν θα εκτελείται καθόλου.

Είναι προφανές λοιπόν, πως αυτό το υπόδενδρο χειρίζεται την κατάσταση όπου το ρομπότ έχει γίνει αντιληπτό και έχει αιχμαλωτιστεί από ένα εχθρικό ρομπότ, ενώ το άλλο υπόδενδρο χειρίζεται τις συμπεριφορές που μπορεί να υλοποιήσει ο πράκτορας ενώ έχει ελευθερία κινήσεων και δεν καλείται να ξεκινήσει πάλι από τη βάση του.

Η συμπεριφορά ReleaseFlag, επιστρέφει την πιασμένη από ρομπότ σημαία στην αρχική της θέση. Όμως αν ο πράκτορας που καλείται να υλοποιήσει τη συμπεριφορά που περιγράφεται σε αυτόν τον κόμβο φύλλου δεν έχει τη σημαία τη δεδομένη στιγμή, τότε η συμπεριφορά ReleaseFlag θα γυρίσει μήνυμα αποτυχίας. Όμως ως παιδί ενός κόμβου ακολουθίας, η επιστροφή μηνύματος αποτυχίας θα είχε ως αποτέλεσμα την

αποτυχία ολόκληρου του κόμβου ακολουθίας, άρα ο κόμβος διαλέκτης ROOT δεν θα είχε τη δυνατότητα να επιλέξει ένα παιδί. Για αυτό το λόγο χρησιμοποιείται ο επιτυγχάνων κόμβος, που ουσιαστικά θα διαμορφώσει το αποτέλεσμα του ReleaseFlag ώστε να διατηρηθεί η ομαλή εκτέλεση της ακολουθίας και τελικά, του κόμβου ROOT.

4.6.2 Υπόδενδρο άμυνας/επίθεσης ή αιχμαλωσίας της σημαίας

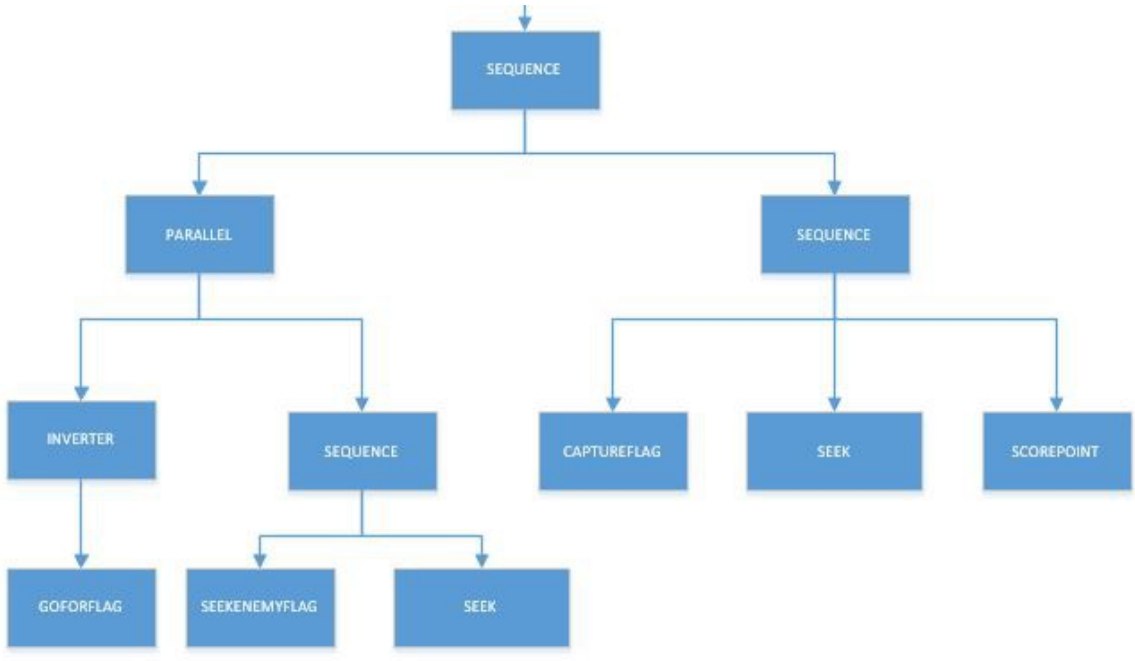


Εικόνα 24: Υπόδενδρο άμυνας/επίθεσης ή αιχμαλωσίας της σημαίας

Ο εικονιζόμενος αρχικός κόμβος παραλληλίας είναι παιδί της ρίζας του δέντρου, εξασφαλίζει πως το ρομπότ δεν είναι TAGGED αλλά έχει ελευθερία κινήσεων, και καλείται να επιλέξει αμυντική ή επιθετική τακτική.

Ο κόμβος παραλληλίας εξασφαλίζει την παράλληλη εκτέλεση όλων των παιδιών του, επομένως αν σε οποιαδήποτε στιγμή ο κόμβος TAGGED επιστρέψει θετικό αποτέλεσμα, θα γίνει αποτέλεσμα αποτυχίας μέσω του κόμβου αντιστροφής INVERTER και το ρομπότ θα ξεφύγει τελείως από αυτό το υπόδενδρο, αφου τελικά ο κεντρικός κόμβος παραλληλίας θα επιστρέψει μήνυμα αποτυχίας. Άρα σειρά εκτέλεσης έχει το υπόδενδρο που αναλύθηκε προηγουμένως (Respawn subtree) και παρουσιάζει τη συμπεριφορά που ακολουθεί ο πράκτορας αν έχει βρεθεί σε αιχμαλωσία (TAGGED).

Παρακάτω αναπαριστώνται και περιγράφονται τα δύο κύρια υπόδενδρα του BT που αφορούν τις συμπεριφορές που καλείται να υλοποιήσει ο πράκτορας ενώ δεν είναι TAGGED, το ένα έχει γονέα τον κόμβο παραλληλίας και το άλλο τον κόμβο ακολουθίας (στο σχήμα κάτω αριστερά).



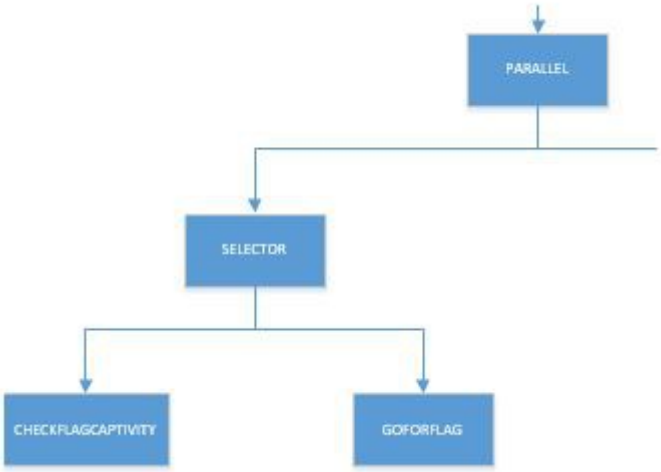
Εικόνα 25: Υπόδενδρο σύλληψης σημαίας

4.6.3 Υπόδενδρο σύλληψης σημαίας

Μέσω αυτής της ακολουθίας επιλέγεται η τακτική που θέτει ως επίκεντρο την αιχμαλώτιση της σημαίας. Ο κόμβος φύλλο GOFORFLAG ελέγχει αν οποιοσδήποτε σύμμαχος του ρομπότ πηγαίνει προς τη σημαία, έτσι ώστε αν δεν πηγαίνει, ο κόμβος αντιστροφέας θα επιστρέψει μήνυμα επιτυχίας και μέσω της ακολουθιακής εκτέλεσης των κόμβων SEEKENEMYFLAG και SEEK, ο κόμβος παραλληλίας θα επιστρέψει μήνυμα επιτυχίας αν το ρομπότ είναι στο κυνήγι για τη σημαία. Όλο το υπόδενδρο θα επιστρέψει μήνυμα επιτυχίας αν τελικά το ρομπότ καταφέρει να αιχμαλωτίσει τη σημαία που έψαχνε, αναζητήσει το κοντινότερο σημείο της βάσης του ώστε να την πάει εκεί και τελικά, μέσω του SCOREPOINT να δώσει τη νίκη στην ομάδα του και να γίνει επαναφορά του αρχικού επιπέδου του παιχνιδιού.

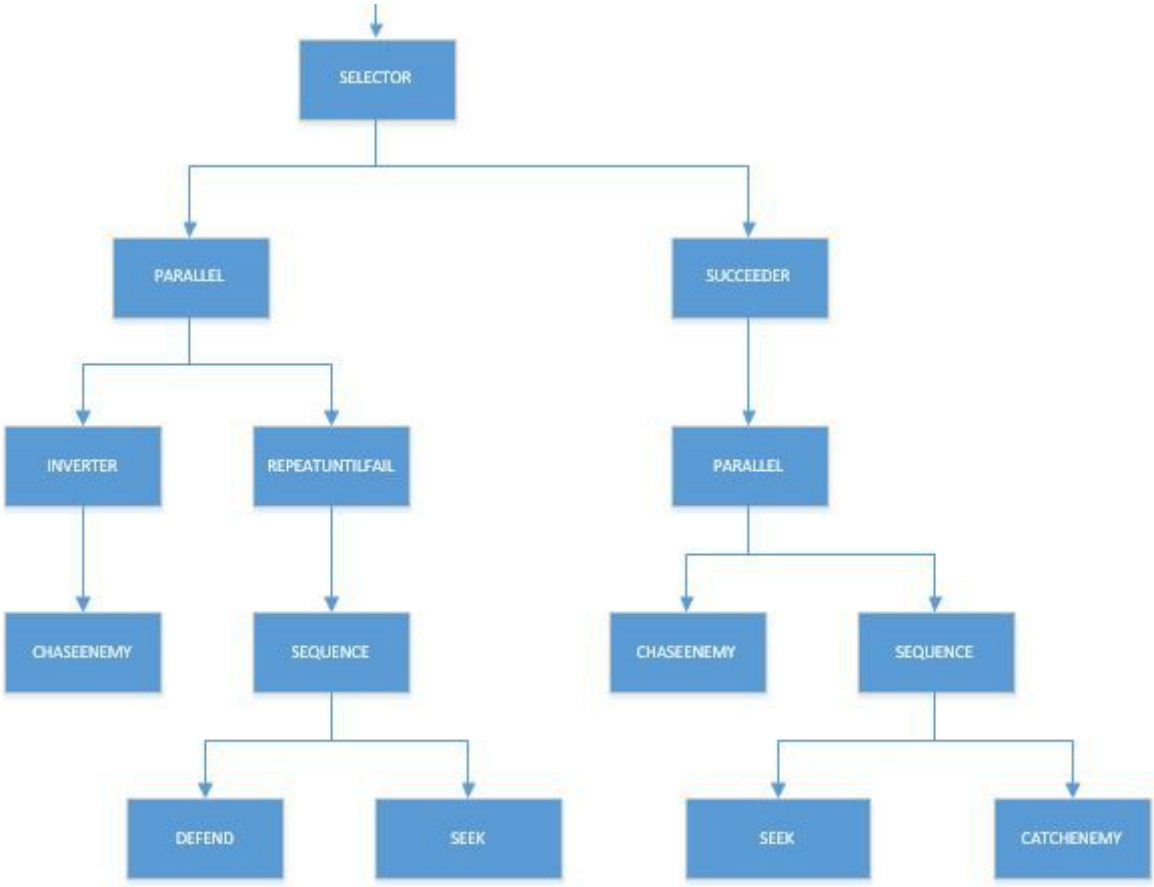
4.6.4 Υπόδενδρο επίθεσης ή άμυνας

Ο κόμβος παραλληλίας από το υπόδενδρο άμυνας/επίθεσης ή αιχμαλωσίας της σημαίας, κάτω αριστερά στην εικόνα 24, έχει δύο παιδιά SELECTORS.



Εικόνα 26: Υπόδενδρο επίθεσης ή άμυνας – έλεγχος αιχμαλωσίας σημαίας

Στο πρώτο παιδί - SELECTOR, η δομή ελέγχει αν η σημαία είναι ήδη υπό αιχμαλωσία. Αν η σημαία είναι ελεύθερη, δηλαδή ο κόμβος CHECKFLAGCAPTIVITY επιστρέψει αποτυχία και δεν πηγαίνει κανένας συμπαίκτης για να την αιχμαλωτίσει, γεγονός που ελέγχεται από το φύλλο GOFORFLAG, τότε ο κόμβος παραλληλίας επιστρέφει μήνυμα αποτυχίας και εκτελείται το υπόδενδρο που αναλύθηκε προηγουμένως (υπόδενδρο σύλληψης σημαίας).



Εικόνα 27: Υπόδενδρο επίθεσης ή άμυνας - δεύτερο μέρος

Το δεύτερο παιδί - SELECTOR του κόμβου παραλληλίας υπό ανάλυση αποτελείται από δύο μικρά υπόδενδρα. Στο πρώτο (αριστερά), αν κριθεί από τον κόμβο CHASEENEMY ότι το ρομπότ δεν χρειάζεται να κυνηγήσει κάποιον εχθρό, ο κόμβος αντιστροφέας επιστρέφει μήνυμα επιτυχίας και έτσι ο κεντρικός κόμβος παραλληλίας εκτελεί τη συμπεριφορά DEFEND μέχρι να κριθεί ότι ένας εχθρός πρέπει να κυνηγηθεί, ή αν επιστρέψει κάποιο μήνυμα λάθους ο κόμβος DEFEND.

Στο δεξιά υπόδενδρο θα υπάρχει πάντα μήνυμα επιτυχίας λόγω του επιτυγχάνων κόμβου SUCCEEDER. Αν το αριστερά υπόδενδρο επιστρέψει αποτυχία, γεγονός που όπως είδαμε προκύπτει αν αποφασιστεί κυνήγι σε έναν εχθρό, τότε ο κεντρικός κόμβος διαλέκτης SELECTOR θα προχωρήσει στην εκτέλεση της συμπεριφοράς CATCHENEMY.

4.7 Επιλογές υλοποίησης

Η δομή που έχει αναπτυχθεί είναι μια πολύ απλή υλοποίηση ενός πολύ κλασικού παιχνιδιού CTF με απλοποίηση των κανόνων, που αφήνει πολλά περιθώρια επέκτασης. Επιλέχθηκε αυτήν την υλοποίηση μετά από εισαγωγές και εξαγωγές διάφορων συμπεριφορών και έγινε θεώρηση πως η καταληκτική υλοποίηση είναι ένας πολύ καλό εναρκτήριο μέσο για κάποιον που δεν έχει ασχοληθεί ξανά με ανάπτυξη τεχνητής νοημοσύνης σε περιβάλλον Unity 3D.

Μπορεί κανείς να ξεκινήσει με αυτήν τη βιβλιοθήκη χωρίς ιδιαίτερες γνώσεις πάνω στην τεχνητή νοημοσύνη και τα BT και να μπορέσει αφού κατανοήσει σε βάθος τη δομή και τη φιλοσοφία τους, να επεκτείνει την παρούσα βιβλιοθήκη όπως επιθυμεί, αφού οι βασικές ενέργειες είναι ήδη υλοποιημένες. Διαβάζοντας την ανάλυση των υπόδενδρων που χρησιμοποιήθηκαν και τη λειτουργία των βασικών κόμβων ενός BT, μπορεί ο χρήστης να καταλάβει ακριβώς και με απτά παραδείγματα που μπορεί ο ίδιος να εκτελέσει, πώς ακριβώς είναι ένα BT σε λειτουργία.

5. ΕΠΙΛΟΓΟΣ

5.1 Συμπεράσματα

Η επιλογή υλοποίησης ενός παιχνιδιού CTF αποτέλεσε το πιο απλό πεδίο πειραματισμών για να γίνει μια εισαγωγή στη χρήση απλών και ευέλικτων ΒΤ. Οι ενέργειες που έχουν να εκτελέσουν οι πράκτορες είναι περιορισμένες, αλλά μέσω της αρχιτεκτονικής των ΒΤ υπάρχουν πολλές διαφορετικές προσεγγίσεις που μπορούν να αποφέρουν το ίδιο αποτέλεσμα.

Η υλοποίηση που διάλεξα εγώ να πραγματοποιήσω είναι αρκετά απλοϊκή, γεγονός που την καθιστά ευανάγνωστη και κατανοητή σε κάποιον που δεν έχει ασχοληθεί ή έχει ασχοληθεί αποκλειστικά επιδερμικά με τεχνικές τεχνητής νοημοσύνης. Γίνεται εισαγωγή και ορισμός τεχνικών και μεθόδων υλοποίησης πρακτόρων, πλεονεκτήματα και μειονεκτήματα της καθεμίας καθώς και εκτενείς συγκρίσεις.

Τα ΒΤ χρησιμοποιούνται ευρέως στη βιομηχανία βιντεοπαιχνιδιών και τα θεωρώ ως την πιο βασική δομή που πρέπει να γνωρίζει σήμερα ο οποιοσδήποτε ενδιαφέρεται και θέλει να ασχοληθεί με τον τομέα. Η τεχνητή νοημοσύνη όπως έχει προαναφερθεί, ακμάζει στη βιομηχανία βιντεοπαιχνιδιών και οι τετριμμένες συμπεριφορές γίνονται σιγά σιγά κομμάτι του παρελθόντος. Πηγάζει επομένως η αναγκαιότητα καινοτόμων ιδεών και η διαρκής ανάπτυξη της νοημοσύνης που διαθέτει ένας πράκτορας, η οποία πλέον δεν μπορεί να υλοποιηθεί από τις παλαιότερες και πιο συχνά χρησιμοποιημένες τεχνικές.

Σε έναν τομέα που συνεχώς αναπτύσσεται, πρέπει και εμείς ως ενδιαφερόμενοι να προσαρμοζόμαστε στις όποιες αλλαγές, και να μην αφήνουμε την εξέλιξη να μας προσπερνά, ειδικά όταν αφορά ένα πεδίο σαν την τεχνητή νοημοσύνη, είτε σε ακαδημαϊκό είτε σε βιομηχανικό επίπεδο.

5.2 Δυνατότητες ανάπτυξης

Εξαιτίας της ευελιξίας της δομής τους και τις δυνατότητες που προσφέρουν τα ΒΤ, είναι εύκολο να προσθέσει ή και να αφαιρέσει κανείς κόμβους από την προαναφερθείσα δομή ώστε να καταλήξει σε έναν πράκτορα που θα εκτελεί περισσότερες λειτουργίες. Οι δυνατότητες που δίνονται από τα ΒΤ καθιστούν εφικτή την προσθήκη περισσότερων ενεργειών, όπως για παράδειγμα μια πιο ανεπτυγμένη τακτική άμυνας ή κάποιες δυνατότητες συνεργασιακής επίθεσης των πρακτόρων που θα συντονιζόταν μέσω των κοινών τους δεδομένων.

Αυτή η τακτική οδηγεί στην ανάπτυξη και την επικείμενη συνένωση της παρούσας δομής με έννοιες όπως τακτικές ομάδας (squad tactics). Μέσω αυτής της έννοιας μπορούν να δημιουργηθούν πράκτορες που θα μπορούν να κινούνται πιο στρατηγικά και σε συγκεκριμένες διαδρομές, αφού τώρα ο πράκτορας ακολουθεί τυχαία διαδρομή κάθε φορά αρκεί να βρίσκεται μέσα στη βάση του.

Για την υλοποίηση όμως μίας τέτοιας λογικής, το ιδεατό θα ήταν το παιχνίδι να μεταφερθεί από την δύο διαστάσεων μορφή που έχει τώρα σε μία 3D υλοποίηση που θα μπορεί να προσφέρει πιο πολλές ευκαιρίες για αποτελεσματική εύρεση βέλτιστης διαδρομής (pathfinding). Στην τρέχουσα έκδοση του παιχνιδιού, ο πράκτορας διαθέτει

ένα Rigidbody και κινείται με βάση την επίδραση δύναμης ορισμένης από τον χρήστη, ενώ μια ιδεατή έκδοσή του θα μπορούσε να αξιοποιεί πλήρως τον αλγόριθμο A* για εύρεση διαδρομών σε 3D περιβάλλον, αφήνοντας έτσι τρομερά περιθώρια εξέλιξης και ανάπτυξης ενός αυτόνομου πράκτορα με πιο ρεαλιστική συμπεριφορά σε συνδυασμό με τακτικές ομάδας και σύστημα κάλυψης.

Με αυτές τις προσθήκες και τροποποιήσεις καθώς και την ύπαρξη εμποδίων που θα εμποδίζουν την ορατότητα, θα μπορεί ο πράκτορας να βρίσκεται σε θέση να εξάγει συμπεράσματα για συγκεκριμένες τοποθεσίες, να έχει τη δυνατότητα να κρύβεται πίσω από εμπόδια και πιθανόν να αναπτυχθεί και μια τακτική μάχης (combat) όπου τα ρομπότ θα έχουν στη διάθεσή τους χαρακτηριστικά όπως επίθεση (attack), ζωή (health) και πυρομαχικά (ammo). Με αυτόν τον τρόπο οδηγούμαστε σε μια υλοποίηση που μπορεί να προσφέρει μάχες από απόσταση (ranged combats) όπου τα bots θα μπορούν να βρίσκουν καταφύγιο σε συγκεκριμένα σημεία του χάρτη ώστε να ανεφοδιαστούν με πυρομαχικά ή να ανεβάσουν τους πόντους της ζωής τους.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Action leaf node	Κόμβος φύλλο ενέργειας
Action pattern	Μοτίβο δράσης
Action selection sequence	Ακολουθία δράσης επιλογής
Agent network architecture	Αρχιτεκτονική δικτύου πρακτόρων
Alarm	Συναγερμός
Arbitrating	Διαιτησία
Behavior Oriented Design	Σχεδιασμός επικεντρωμένος στη συμπεριφορά
Behavior Trees	Δέντρα συμπεριφοράς
Blackboard data	Δεδομένα μαυροπίνακα
Bot	Ρομπότ
Bugs	Σφάλματα
Build	Χτίσιμο
Cache	Κρυφή μνήμη
Callback functions	Συναρτήσεις επανάκλησης
Child nodes	Κόμβοι παιδιά
Classifiers	Ταξινομητές
Competence	Αρμοδιότητα
Compile	Μεταγλώττιση
Complete complex agents	Πλήρεις πολύπλοκοι πράκτορες
Composite node	Σύνθετος κόμβος
Condition leaf node	Κόμβος φύλλο συνθήκης

Context	Πλαίσιο αναφοράς
Current state	Τρέχουσα κατάσταση
Deadlocks	Αδιέξοδα
Decision tree	Δέντρο απόφασης
Decorator node	Κόμβος διακοσμητής
Drive collection	Συλλογή κινήσεων
Engine	Μηχανή
Event loop	Βρόχος συμβάντων
Failure	Αποτυχία
Finite state machines	Μηχανές πεπερασμένων καταστάσεων
First-come first-served system	Σύστημα εξυπηρέτησης με σειρά προτεραιότητας
First generation trees	Δέντρα πρώτης γενιάς
Fitness function	Συνάρτηση καταλληλότητας
Fuzzy terms	Ασαφείς όροι
Game environment	Περιβάλλον παιχνιδιού
Goal heuristic	Ευρετικός μηχανισμός για στόχο
Greedy algorithm	Άπληστος αλγόριθμος
Hard coded logic	Σκληρά κωδικοποιημένη λογική
Hardware	Μηχανήματα υπολογιστών
Heuristic	Ευρετικός μηχανισμός
Hierarchical finite state machines	Ιεραρχικές μηχανές πεπερασμένων΄καταστάσεων
Hierarchical goal-oriented architecture	Εραρχική αρχιτεκτονική προσανατολισμένη στο στόχο

Hybrid systems	Υβριδικά συστήματα
Intelligent agent	Έξυπνος πράκτορας
Inverter node	Κόμβος αντιστροφέας
Large-scale software-integrated system	Πληροφοριακό σύστημα μεγάλης κλίμακας
Leaf node	Κόμβος φύλλο
Learning	Μάθηση
Learning algorithm	Αλγόριθμος μάθησης
Leverage	Μόχλευση
Low-level	Χαμηλού επιπέδου
Memory prefetching	Προσκόμιση μνήμης
Metaheuristic	Μετα-ευρετικός μηχανισμός
Metaheuristic learning	Μετα-ευρετική μάθηση
Modelling language	Γλώσσα μοντελισμού
Modularity	Δομοστοιχείωση
Natural language	Φυσική γλώσσα
Notation	Σημειογραφία
Parallel node	Κόμβος παραλληλίας
Parent nodes	Κόμβοι γονείς
Pathfinding	Εύρεση βέλτιστου μονοπατιού
Plan	Σχέδιο
Plan elements	Στοιχεία σχεδίου
planning	Σχεδιασμός

Random selector node	Κόμβος τυχαίου διαλέκτη
React tree	Δέντρο αντίδρασης
Reactive mode	Αντιδραστική λειτουργία
Refactoring	Αναπαραγοντοποίηση
Reinforcement learning	Ενισχυτική μάθηση
Repeat until fail node	Επαναληπτικός κόμβος έως αποτυχίας
Repeater node	Επαναληπτικός κόμβος
Root	Ρίζα
Running	Σε εκτέλεση
Scripts	Δέσμες ενεργειών
Selector node	Κόμβος διαλέκτης
Sequence node	Κόμβος ακολουθίας
Squad tactics	Τακτικές ομάδων
State	Κατάσταση
Stateful data	Δεδομένα επίβλεψης κατάστασης
Steering	Πηδαλιούχηση
Structures	Δομές
Subroutine	Υπορουτίνα
Subsumption architecture	Αρχιτεκτονική υπαγωγής
Succeeder node	Επιτυγχάνων κόμβος
Success	Επιτυχία
Tagged	Αιχμαλωτισμένος
Testing	Δοκιμές

Traceable	Ανιχνεύσιμος
Training	Εκπαίδευση
Transition	Μετάβαση
Typecasting	Διαμόρφωση τύπου
Subtree	Υπόδενδρο

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

BT	Behavior Trees
DT	Decision Trees
FSM	Finite State Machines
GDC	Game Developers Conference
IGDA	International Game Developer Association

ΠΑΡΑΡΤΗΜΑ Ι

Πηγαίος κώδικας

Ο πηγαίος κώδικας της βιβλιοθήκης μπορεί να προσπελαστεί μέσω του συνδέσμου https://github.com/Raniato/BehaviorTrees.

AreaDeterminator.cs

```
using UnityEngine;
using System.Collections;

/* @class AreaDeterminator
 * @brief This is the script that determines if a bot(or the human player) is in its base.
 */

public class AreaDeterminator : MonoBehaviour {

    // Use this for initialization
    void Start () {}

    // Update is called once per frame
    void Update () {}

    //In right base
    void OnTriggerEnter(Collider otherPlayer)
    {
        if((otherPlayer.gameObject.tag == "Bot"))
            otherPlayer.gameObject.GetComponent<CTF.BotScript>().botTree.setValue(CTF.TreeData.IN_ARENA,
CTF.BotScript.TeamID.RIGHT);
        if ((otherPlayer.gameObject.tag == "HumanPlayer"))
          otherPlayer.gameObject.GetComponent<HumanPlayerScript> ().inBase = CTF.BotScript.TeamID.RIGHT;
    }

  void OnTriggerExit(Collider otherPlayer)
  {
    if ((otherPlayer.gameObject.tag == "Bot"))
            otherPlayer.gameObject.GetComponent<CTF.BotScript>().botTree.setValue(CTF.TreeData.IN_ARENA,
CTF.BotScript.TeamID.LEFT);
        if((otherPlayer.gameObject.tag == "HumanPlayer"))
          otherPlayer.gameObject.GetComponent<HumanPlayerScript>().inBase = CTF.BotScript.TeamID.LEFT;
  }
}
```

BehaviorTree.cs

```
using System;
using System.Collections.Generic;
using UnityEngine;

/* @class BehaviorTree
 * @brief Behavior Tree class, including routines for getValue, setRoot, setValue.
 */

namespace BTnamespace
{
    [System.Serializable]
    public class BehaviorTree
    {
        private Node _root;
    private Dictionary<string, object> _blackboardData = new Dictionary<string, object>();

    public object getValue(string name)
        {
            return _blackboardData [name];
        }

    public void setValue(string name, object value)
        {
            _blackboardData [name] = value;
        }

        public void setRoot(Node root)
        {
            _root = root;
        }

        public void Initialize()
        {
            _root.Initialize( this );
        }

        public Outcome Process()
        {
            return _root.Process( this );
        }
    }
}
```

BehaviorTreeScript.cs

```
using System;
using System.Collections.Generic;
using UnityEngine;

using BTnamespace;

/* @class BehaviorTreeScript
 * @brief This is the script that composes the final behavior tree structure and defines the blackboard data.
 */

namespace CTF
{
    //Blackboard Data
  public abstract class TreeData
  {
    public const string MY_TEAMID = "MyTeamID";
    public const string ENEMY_FLAG = "EnemyFlag";
    public const string HAS_FLAG = "HasFlag";
    public const string MY_TRANSFORM = "MyTransform";
    public const string ALL_TEAM_TILES = "TeamTiles";
    public const string RESPAWN_POSITION = "RespawnPosition";
    public const string IS_TAGGED = "IsTagged";
    public const string TAGGED_TEAMMATE = "TaggedTeammate";
    public const string SEEK_THIS = "SeekThis";
    public const string IN_ARENA = "InArena";
    public const string STEER = "Steer";
    public const string VELOCITY = "Velocity";
    public const string ENEMIES_NEARBY = "EnemiesNearby";
    public const string ALLIES = "Allies";
    public const string ENEMIES = "Enemies";
    public const string STATE = "State";
  }

    [System.Serializable]
  public class BehaviorTreeScript : BehaviorTree
  {
    public BehaviorTreeScript()
    {
      //Defence

      ChaseEnemy chaseEnemy = new ChaseEnemy();
      Inverter dontChaseEnemy = new Inverter();
      dontChaseEnemy.addChild(chaseEnemy);
      Defend defend = new Defend();
      Seek seekObject = new Seek();
```

```
Sequence Defence = new Sequence();
Defence.addChild(defend);
Defence.addChild(seekObject);
RepeatUntilFail defendUntilFail = new RepeatUntilFail();
defendUntilFail.addChild(Defence);
Parallel DefendCheckEnemy = new Parallel();
DefendCheckEnemy.addChild(dontChaseEnemy);
DefendCheckEnemy.addChild(defendUntilFail);

//Chase enemy

Seek seekObjectEnemy = new Seek();
CatchEnemy catchEnemy = new CatchEnemy();
Sequence seekCatch = new Sequence();
seekCatch.addChild(seekObjectEnemy);
seekCatch.addChild(catchEnemy);
ChaseEnemy enemyInSight = new ChaseEnemy();
Parallel seekAndCatch = new Parallel();
seekAndCatch.addChild(enemyInSight);
seekAndCatch.addChild(seekCatch);
Succeeder alwaysCatch = new Succeeder();
alwaysCatch.addChild(seekAndCatch);
Selector DefendSelector = new Selector();
DefendSelector.addChild(DefendCheckEnemy);
DefendSelector.addChild(alwaysCatch);

//defend depending on flag captivity status

CheckFlagCaptivity flagCaptured = new CheckFlagCaptivity();
GoForFlag anyoneGoingForFlag = new GoForFlag();
Selector anyoneOffensive = new Selector();
anyoneOffensive.addChild(flagCaptured);
anyoneOffensive.addChild(anyoneGoingForFlag);
Parallel DefenceParallel = new Parallel();
DefenceParallel.addChild(anyoneOffensive);
DefenceParallel.addChild(DefendSelector);

//offence

SeekEnemyFlag findEnemyFlag = new SeekEnemyFlag();
Seek seekEnemyFlag = new Seek();

Sequence findSeekEnemyFlag = new Sequence();
findSeekEnemyFlag.addChild(findEnemyFlag);
findSeekEnemyFlag.addChild(seekEnemyFlag);

GoForFlag anyoneFetchingFlag = new GoForFlag();
```

```
Inverter nooneFetchingFlag = new Inverter();
nooneFetchingFlag.addChild(anyoneFetchingFlag);

Parallel findFlagParallel = new Parallel();
findFlagParallel.addChild(nooneFetchingFlag);
findFlagParallel.addChild(findSeekEnemyFlag);

//bring the flag back

CaptureFlag captureFlag = new CaptureFlag();
Seek findIDTile = new Seek();
ScorePoint score = new ScorePoint();
Sequence captureReturnScore = new Sequence();
captureReturnScore.addChild(captureFlag);
captureReturnScore.addChild(findIDTile);
captureReturnScore.addChild(score);
Sequence retrieveFlagParallel = new Sequence();
retrieveFlagParallel.addChild(findFlagParallel);
retrieveFlagParallel.addChild(captureReturnScore);
Selector seekFlagSelector = new Selector();
seekFlagSelector.addChild(retrieveFlagParallel);

//Defence or Offence?

Selector DefenceOffence = new Selector();
DefenceOffence.addChild(DefenceParallel);
DefenceOffence.addChild(seekFlagSelector);

Tagged tagged = new Tagged();
Inverter notTagged = new Inverter();
notTagged.addChild(tagged);

Parallel defendOrOffend = new Parallel();
defendOrOffend.addChild(notTagged);
defendOrOffend.addChild(DefenceOffence);

//respawn

ReleaseFlag releaseFlag = new ReleaseFlag();
Succeeder alwaysReleaseFlag = new Succeeder();
alwaysReleaseFlag.addChild(releaseFlag);

Respawn respawn = new Respawn();

Sequence releaseFlagRespawn = new Sequence();
releaseFlagRespawn.addChild(alwaysReleaseFlag);
releaseFlagRespawn.addChild(respawn);
```

```
            Selector root = new Selector();
            root.addChild(defendOrOffend);
            root.addChild(releaseFlagRespawn);

            setRoot(root);
        }
    }
}
```

BotScript.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using BTnamespace;

/* @class BotScript
 * @brief This is the script that Initializes all information about a bot, regarding its team and its team's tiles, flag position etc.
 * One of the most valuable aspects implemented is the way the bot is keeping track of the enemies that are within its vicinity.
 */

namespace CTF
{
    public class BotScript : MonoBehaviour
    {
        public enum TeamID
        {
            RIGHT,
            LEFT
        }

        public enum broadcastMessage
        {
            DEFENDING,
            GOING_TO_FLAG,
            TAGGED,
            RETRIEVING_FLAG
        }

            public bool hasFlag;
            public bool isTagged;
            public broadcastMessage currentBroadcast;
            public float steer;
            public float velocity;
            public TeamID myTeamID;
```

```
        public BehaviorTreeScript botTree;
        public Dictionary<GameObject, broadcastMessage> myState = new Dictionary<GameObject,
broadcastMessage>();

    // Initializeialize everything concerning the bot. Team tiles, flag position, allies, respawn position
    void Start()
    {
      botTree = new BehaviorTreeScript();
      List<GameObject> enemiesNearby = new List<GameObject>();
      List<GameObject> enemies = new List<GameObject>();
      GameObject[] allPlayers = GameObject.FindGameObjectsWithTag("Bot");
      GameObject HumanPlayer = GameObject.FindGameObjectWithTag("HumanPlayer");
      List<GameObject> allies = new List<GameObject>();
      for (int i = 0; i < allPlayers.Length; i++){
     if ((allPlayers[i] != this.gameObject) && (allPlayers[i].GetComponent<BotScript>().myTeamID == myTeamID))
          allies.Add(allPlayers[i]);
      if((allPlayers[i]!=this.gameObject) && (allPlayers[i].GetComponent<BotScript>().myTeamID!=myTeamID))
                  enemies.Add(allPlayers[i]);
      }

      if (HumanPlayer.GetComponent<HumanPlayerScript> ().myTeamID == myTeamID)
              allies.Add (HumanPlayer);
          else
              enemies.Add (HumanPlayer);

      currentBroadcast = broadcastMessage.DEFENDING;

      botTree.setValue(TreeData.ALLIES, allies);
      botTree.setValue(TreeData.STATE, myState);
      GameObject seekThis = null;
      botTree.setValue(TreeData.SEEK_THIS, seekThis);
      botTree.setValue(TreeData.IN_ARENA, myTeamID);
      botTree.setValue (TreeData.ENEMIES, enemies);
      botTree.setValue(TreeData.ENEMIES_NEARBY, enemiesNearby);
      botTree.setValue(TreeData.MY_TEAMID, myTeamID);
      botTree.setValue(TreeData.STEER, steer);
      botTree.setValue(TreeData.VELOCITY, velocity);
      botTree.setValue(TreeData.HAS_FLAG, false);
      botTree.setValue(TreeData.IS_TAGGED, false);
      botTree.setValue(TreeData.MY_TRANSFORM, transform);
      if (myTeamID == TeamID.RIGHT){
        botTree.setValue(TreeData.ALL_TEAM_TILES, GameObject.FindGameObjectWithTag("RightTiles"));
        botTree.setValue(TreeData.ENEMY_FLAG, GameObject.Find("LeftFlag"));
 botTree.setValue(TreeData.RESPAWN_POSITION, GameObject.Find("RightRespawnNode").transform.position);
      }
      else{
        botTree.setValue(TreeData.ALL_TEAM_TILES, GameObject.FindGameObjectWithTag("LeftTiles"));
```

```
            botTree.setValue(TreeData.ENEMY_FLAG, GameObject.Find("RightFlag"));
 botTree.setValue(TreeData.RESPAWN_POSITION, GameObject.Find("LeftRespawnNode").transform.position);
        }

        botTree.Initialize();
        transform.rotation = Quaternion.Euler(0f, -90f, 90f);
    }

    // Update is called once per frame
    void Update()
    {
        hasFlag = (bool) botTree.getValue(TreeData.HAS_FLAG);
        isTagged = (bool) botTree.getValue(TreeData.IS_TAGGED);
        botTree.setValue(TreeData.MY_TRANSFORM, transform);
        if(Outcome.PROCESSING != botTree.Process())
            botTree.Initialize();
    }

    public void Message(GameObject PlayerObj, broadcastMessage PlayerMessage)
    {
        if (myState.ContainsKey(PlayerObj))
            myState[PlayerObj] = PlayerMessage;
        else
            myState.Add(PlayerObj, PlayerMessage);
                botTree.setValue(TreeData.STATE, myState);
    }

    public void BroadCastTeamMessage(broadcastMessage MyMessage)
    {
                currentBroadcast = MyMessage;
        List<GameObject> allies = botTree.getValue(TreeData.ALLIES) as List<GameObject>;

        for (int i = 0; i < allies.Count; i++){
                    if(allies[i].CompareTag("Bot"))
                allies[i].GetComponent<BotScript>().Message(gameObject, MyMessage);
        }
    }

            //add a nearby enemy on collision
    void OnTriggerEnter(Collider otherPlayer)
    {
                bool enemyFound = false;
              if ((otherPlayer.gameObject.tag == "Bot") && (otherPlayer.GetComponent<BotScript>().myTeamID !=
myTeamID))
                    enemyFound = true;
                                                    else if(otherPlayer.gameObject.tag == "HumanPlayer" &&
otherPlayer.GetComponent<HumanPlayerScript>().myTeamID != myTeamID)
```

```
                    enemyFound = true;
                if(enemyFound){
                        List <GameObject> enemiesNearby = botTree.getValue(TreeData.ENEMIES_NEARBY) as
List<GameObject>;
                    enemiesNearby.Add(otherPlayer.gameObject);
                    botTree.setValue(TreeData.ENEMIES_NEARBY, enemiesNearby);
                }
            }

            //remove the enemy while exiting collision
            void OnTriggerExit(Collider otherPlayer)
        {
                bool enemyFound = false;
                 if ((otherPlayer.gameObject.tag == "Bot") && (otherPlayer.GetComponent<BotScript>().myTeamID !=
myTeamID))
                    enemyFound = true;
                                                    else  if(otherPlayer.gameObject.tag  ==  "HumanPlayer"  &&
otherPlayer.GetComponent<HumanPlayerScript>().myTeamID != myTeamID)
                    enemyFound = true;

                if (enemyFound){
            GameObject CurrentSeekingObject = botTree.getValue(TreeData.SEEK_THIS) as GameObject;
            if (CurrentSeekingObject == otherPlayer.gameObject){
               CurrentSeekingObject = null;
               botTree.setValue(TreeData.SEEK_THIS, CurrentSeekingObject);
            }
                        List<GameObject>  enemiesNearby  =  botTree.getValue(TreeData.ENEMIES_NEARBY)  as
List<GameObject>;
            enemiesNearby.Remove(otherPlayer.gameObject);
            botTree.setValue(TreeData.ENEMIES_NEARBY, enemiesNearby);
          }
        }
      }
}
```

CaptureFlag.cs

```
using System;

using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class CaptureFlag
 * @brief In this class, bot is locating the shortest path through the tiles to reach its base while carrying the flag.
 */

namespace CTF
{
   class CaptureFlag : Leaf
   {
      GameObject enemyFlag;
      Transform myTransform;
      GameObject myTiles;

      public override void Initialize(BehaviorTree tree)
      {
         enemyFlag = tree.getValue(TreeData.ENEMY_FLAG) as GameObject;
         myTransform = (Transform) tree.getValue(TreeData.MY_TRANSFORM);
         myTiles = tree.getValue(TreeData.ALL_TEAM_TILES) as GameObject;
                  tree.setValue(TreeData.HAS_FLAG, true);
      }

      public override Outcome Process(BehaviorTree tree)
      {
          myTransform.gameObject.GetComponent<BotScript>().BroadCastTeamMessage(BotScript.broadcastMessa
ge.RETRIEVING_FLAG);
         enemyFlag.GetComponent<FlagScript>().Captured(myTransform.gameObject);
         GameObject seekTile;
         float closestDistance = float.MaxValue;
         int ClosestIndex = int.MaxValue;
         for (int i = 0; i < myTiles.transform.childCount; i++){
 loat tileDistance = (myTransform.transform.position - myTiles.transform.GetChild(i).transform.position).magnitude;
            if (closestDistance > tileDistance){    //get closer to the base
               closestDistance = tileDistance;
               ClosestIndex = i;
            }
         }
         seekTile = myTiles.transform.GetChild(ClosestIndex).gameObject;
```

```
            tree.setValue(TreeData.SEEK_THIS, seekTile);
            return Outcome.SUCCESS;
        }
    }
}
```

CatchEnemy.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class CatchEnemy
 * @brief In this class, bot is catching an enemy == tagging him.
 */

namespace CTF
{
    class CatchEnemy : Leaf
    {

        public override void Initialize(BehaviorTree tree)
        {}

        public override Outcome Process(BehaviorTree tree)
        {
            GameObject enemy = tree.getValue(TreeData.SEEK_THIS) as GameObject;
            if (enemy && enemy.CompareTag("Bot")){
                enemy.GetComponent<BotScript>().botTree.setValue(TreeData.IS_TAGGED, true);
                tree.setValue(TreeData.SEEK_THIS, null);              //I don't have anything to seek for now
                return Outcome.SUCCESS;
            }
             else if (enemy && enemy.CompareTag("HumanPlayer")){ //Same applies if the enemy-threat is the human
player
                enemy.GetComponent<HumanPlayerScript>().catchHuman();
                        return Outcome.SUCCESS;
            }
            else
                return Outcome.FAIL;
        }
    }
}
```

ChaseEnemy.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class ChaseEnemy
 * @brief In this class, bot is checking if it should chase an enemy. This decision is made by two facts:
 *  1)either an enemy has the flag, therefore he is a threat
 *      2)or an enemy is in my base and close to the bot that is taking this decision
 */

namespace CTF
{
    class ChaseEnemy : Leaf
    {
        BotScript.TeamID myTeamID;

        public override void Initialize(BehaviorTree tree)
        {
            myTeamID = (BotScript.TeamID) tree.getValue(TreeData.MY_TEAMID);
        }

        public override Outcome Process(BehaviorTree tree)
        {
            if (myTeamID == (BotScript.TeamID)tree.getValue(TreeData.IN_ARENA)){
         List<GameObject> enemies = tree.getValue(TreeData.ENEMIES_NEARBY) as List<GameObject>;
                if (enemies.Count > 0){
                    GameObject closestEnemy = null;
                    GameObject flagBearer = null;
                    float minDist = 5f;
                    float closestDist = 5f;
                    bool hasFlag = false;
                    Transform MyTransform = (Transform)tree.getValue(TreeData.MY_TRANSFORM);
                    for (int i = 0; i < enemies.Count; i++){
                        if (enemies[i].CompareTag("Bot")){
                            if((bool)enemies[i].GetComponent<BotScript>().botTree.getValue(TreeData.HAS_
FLAG)){ //if the enemy has the flag, mark him
                                flagBearer = enemies[i];
                                hasFlag = true;
                            }
    if (!(bool)enemies[i].GetComponent<BotScript>().botTree.getValue(TreeData.IS_TAGGED)){ //providing he is not
tagged, therefore not a threat
```

```
            Transform EnemyTransform =
            (Transform)enemies[i].GetComponent<BotScript>().botTree.getValue(TreeData.MY_TRANSFORM);
            float DistanceWithThisEnemy = (MyTransform.position - EnemyTransform.position).magnitude;
            if (closestDist > DistanceWithThisEnemy){
                    closestDist = DistanceWithThisEnemy;
                            closestEnemy = enemies[i];
             }
            }
        }
        else{   //if the human player is also one of my enemies, check his position and situation comparing to the flag
            if (!enemies[i].GetComponent<HumanPlayerScript>().isTagged){ //providing he is not tagged, therefore not a
threat
            float DistanceWithThisEnemy = (MyTransform.position - enemies[i].transform.position).magnitude;
            if (closestDist > DistanceWithThisEnemy){
                    closestDist = DistanceWithThisEnemy;
                    closestEnemy = enemies[i];
             }
            }
           }
          }
          if (closestDist <= minDist || hasFlag){ //decide chasing method depending on the previous "investigation"
             if(hasFlag)
                    tree.setValue(TreeData.SEEK_THIS, flagBearer);
             else
                    tree.setValue(TreeData.SEEK_THIS, closestEnemy);
              return Outcome.SUCCESS;
                                }

                            }
                    }
                    return Outcome.FAIL;
             }
        }
}
```

CheckFlagCaptivity.cs

```
using System;

using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class CheckFlagCaptivity
 * @brief In this class, bot is checking if enemy flag is under captivity.
 */

namespace CTF
{
   class CheckFlagCaptivity : Leaf
   {
      public override void Initialize(BehaviorTree tree)
      {}

      public override Outcome Process(BehaviorTree tree)
      {
            Dictionary<GameObject, BotScript.broadcastMessage> myTeamID = tree.getValue(TreeData.STATE) as
Dictionary<GameObject, BotScript.broadcastMessage>;

         foreach (KeyValuePair<GameObject, BotScript.broadcastMessage> retrievers in myTeamID){
            if (retrievers.Value == BotScript.broadcastMessage.RETRIEVING_FLAG)
               return Outcome.SUCCESS;
         }

         return Outcome.FAIL;
      }
   }
}
```

Composite.cs

```
using System;
using System.Collections.Generic;

/* @class Composite
 * @brief Class describing a composite node behavior.
 */

namespace BTnamespace
{
	public abstract class Composite : Node
	{
		protected List<Node> _children = new List<Node>();

		public virtual void addChild(Node child)
		{
			_children.Add (child);
		}

		public Node at (int index)
		{
			if(_children.Count <= index)
			  return null;
			return _children[index];
		}
	}
}
```

Decorator.cs

```
using System;
/* @class Decorator
 * @brief Class describing a decorator's structure.
 */

namespace BTnamespace
{
	public abstract class Decorator : Node
	{
		protected Node _child;

		public virtual void addChild( Node child )
		{
			_child = child;
		}

		public override void Initialize (BehaviorTree tree)
		{
			_child.Initialize (tree);
		}
	}
}
```

Defend.cs

```
using System;

using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class Defend
 * @brief In this class, bot is choosing random tiles to navigate to, as an implementation of a defensive behavior.
 */

namespace CTF
{
   class Defend : Leaf
   {
      GameObject myTiles;
      Transform MyTransfrom;
```

```
        public override void Initialize(BehaviorTree tree)
        {
            myTiles = tree.getValue(TreeData.ALL_TEAM_TILES) as GameObject;
            MyTransfrom = (Transform)tree.getValue(TreeData.MY_TRANSFORM);
        }

        public override Outcome Process(BehaviorTree tree)
        {
             MyTransfrom.gameObject.GetComponent<BotScript>().BroadCastTeamMessage(BotScript.broadcastMessa
ge.DEFENDING);
                    UnityEngine.Random.seed = (Guid.NewGuid().GetHashCode());
                    int randomTile;
                    if(myTiles.transform.childCount > 0)
                            randomTile  = UnityEngine.Random.Range(0, MaxCount);     //get a random tile from the ones I
am allowed to navigate to
            tree.setValue(TreeData.SEEK_THIS, myTiles.transform.GetChild(randomTile).gameObject);
            return Outcome.SUCCESS;
        }
    }
}
```

FlagScript.cs

```
using UnityEngine;
using System.Collections;

/* @class FlagScript
 * @brief This is the flag's class, containing its behavior depending on captivity status.
 */

public class FlagScript : MonoBehaviour {

        public bool isCaptured = false;
        public GameObject parent;
        public Vector3 flagPosition;

        // Use this for initialization
        void Start () {
                flagPosition = transform.position;
                parent = null;
        }

        //if the flag is captured, parent it to the capturer
        public void Captured(GameObject iParentItTo)
        {
                if(parent == null && isCaptured == false){
                        parent = iParentItTo;
                        isCaptured = true;
```

```
            }
        }

        public bool IsCaptured()
        {
            return isCaptured;
        }

        //if the flag is released, it returns to its prime position
        public void Release(GameObject currentParent)
        {
            if(isCaptured && currentParent == parent){
                isCaptured = false;
                parent = null;
                returnToBase();
            }
        }

        void returnToBase()
        {
            transform.position = flagPosition;
        }

        // Update is called once per frame
        void Update () {
            if(parent)
                transform.position = parent.transform.position;
        }
}
```

GoForFlag.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class GoForFlag
 * @brief In this class, bot is required to check if any of its teammates is going for the flag, compares distances and chooses accordingly.
 * If any teammate is going for the flag, class returns behavior result of success, otherwise behavior result of failure.
 */

namespace CTF
{
```

```
class GoForFlag : Leaf
{
    GameObject HumanPlayer = null;
    GameObject enemyFlag = null;
    BotScript.TeamID myTeamID;
    public override void Initialize(BehaviorTree tree)
    {
        List<GameObject> allies = tree.getValue(TreeData.ALLIES) as List<GameObject>;
        for (int i = 0; i < allies.Count; i++){            //construct a list with my bot allies and/or the human player
            if(allies[i].CompareTag("HumanPlayer"))
                HumanPlayer = allies[i];
        }
        enemyFlag = tree.getValue(TreeData.ENEMY_FLAG) as GameObject;
        myTeamID = (BotScript.TeamID)tree.getValue(TreeData.MY_TEAMID);

    }

    public override Outcome Process(BehaviorTree tree)
    {
        Transform MyTransform = (Transform)tree.getValue(TreeData.MY_TRANSFORM);

            if (HumanPlayer && HumanPlayer.GetComponent<HumanPlayerScript>().inBase != myTeamID && !
HumanPlayer.GetComponent<HumanPlayerScript>().isTagged){
            float theirDistance = (new Vector3(HumanPlayer.transform.position.x - enemyFlag.transform.position.x,
HumanPlayer.transform.position.y - enemyFlag.transform.position.y, 0.0f)).sqrMagnitude;
            float myDistance = (new Vector3(MyTransform.transform.position.x - enemyFlag.transform.position.x,
MyTransform.transform.position.y - enemyFlag.transform.position.y, 0.0f)).sqrMagnitude;
            if (theirDistance <= myDistance)       //compare distance I have with the distance my teammates do
                return Outcome.SUCCESS;
        }

        Dictionary<GameObject, BotScript.broadcastMessage> MymyTeamID = tree.getValue(TreeData.STATE) as
Dictionary<GameObject, BotScript.broadcastMessage>;
        foreach (KeyValuePair<GameObject, BotScript.broadcastMessage> teammatesMessage in MymyTeamID){
//check if any teammate is broadcasting "going" or "retrieving" the flag
                    if (teammatesMessage.Value == BotScript.broadcastMessage.GOING_TO_FLAG ||
teammatesMessage.Value == BotScript.broadcastMessage.RETRIEVING_FLAG)
            return Outcome.SUCCESS;
        }
        return Outcome.FAIL;
    }
}
}
```

HumanPlayerScript.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

/* @class HumanPlayerScript
 * @brief This is the script that determines the human player's interaction within the game.
 */

public class HumanPlayerScript : MonoBehaviour {

    public CTF.BotScript.TeamID myTeamID;
    public CTF.BotScript.TeamID inBase;
    public bool isTagged = false;
    public bool hasFlag = false;
    public Vector3 respawnPosition;
    public float velocity;
    List<GameObject> enemies = new List<GameObject>();
    List<GameObject> allies = new List<GameObject>();
    GameObject enemyFlag;

        // Use this for initialization
        void Start ()
    {
        GetComponent<Rigidbody>().AddForce(new Vector3(0.0f, 10.0f, 0.0f));
        if (GetComponent<Rigidbody>().velocity.magnitude > 0){
            float angle = Mathf.Atan2(GetComponent<Rigidbody>().velocity.y, GetComponent<Rigidbody>().velocity.x) *
Mathf.Rad2Deg;
            transform.rotation = Quaternion.Euler(angle, -90f, 90f);
        }
        GameObject[] allBots = GameObject.FindGameObjectsWithTag("Bot");
        for (int i = 0; i < allBots.Length; i++){
            if (allBots[i].GetComponent<CTF.BotScript>().myTeamID == myTeamID)
                allies.Add(allBots[i]);
            else
                enemies.Add(allBots[i]);
        }
        GameObject[] Flags = GameObject.FindGameObjectsWithTag("Flag");
        for (int i = 0; i < Flags.Length; i++){
            if (Flags[i].name == "LeftFlag")
                enemyFlag = Flags[i];
        }
        respawnPosition = GameObject.Find("RightRespawnNode").transform.position;
        }

        // Update is called once per frame
```

```
    void Update ()
  {
    if (!isTagged){
      if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow))
        GetComponent<Rigidbody>().AddForce(new Vector3(-10.0f, 0.0f, 0.0f));
      if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow))
        GetComponent<Rigidbody>().AddForce(new Vector3(10.0f, 0.0f, 0.0f));
      if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))
        GetComponent<Rigidbody>().AddForce(new Vector3(0.0f, 10.0f, 0.0f));
      if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))
        GetComponent<Rigidbody>().AddForce(new Vector3(0.0f, -10.0f, 0.0f));
      if (GetComponent<Rigidbody>().velocity.magnitude > 0){
        float angle = Mathf.Atan2(GetComponent<Rigidbody>().velocity.y, GetComponent<Rigidbody>().velocity.x)
* Mathf.Rad2Deg;
        transform.rotation = Quaternion.Euler(angle, -90f, 90f);
      }
        GetComponent<Rigidbody>().velocity = Vector3.ClampMagnitude(GetComponent<Rigidbody>().velocity,
velocity);
      if (inBase == myTeamID){
          for (int i = 0; i < enemies.Count; i++){
                                                                                  if ((!
(bool)enemies[i].GetComponent<CTF.BotScript>().botTree.getValue(CTF.TreeData.IS_TAGGED))
              && Collided(enemies[i].transform.position))
               enemies[i].GetComponent<CTF.BotScript>().botTree.setValue(CTF.TreeData.IS_TAG
GED, true);
          }
        }
        if (inBase != myTeamID){
          for (int i = 0; i < allies.Count; i++){
                                                                                  if
((bool)allies[i].GetComponent<CTF.BotScript>().botTree.getValue(CTF.TreeData.IS_TAGGED)
              && Collided(allies[i].transform.position))
               allies[i].GetComponent<CTF.BotScript>().botTree.setValue(CTF.TreeData.IS_TAGGE
D, false);
          }
          if (enemyFlag){
                                                  if(!enemyFlag.GetComponent<FlagScript>().IsCaptured() &&
Collided(enemyFlag.transform.position)){
              enemyFlag.GetComponent<FlagScript>().Captured(this.gameObject);
              hasFlag = true;
            }
          }
        }
      if (hasFlag)
        if (inBase == myTeamID)
          Application.LoadLevel(0);   //player has the flag, restart the scene
    }
```

```
        else{
                                gameObject.transform.position = new Vector3(respawnPosition.x, respawnPosition.y,
gameObject.transform.position.z);
                    isTagged=false;}

        }

    bool Collided(Vector3 Position)
    {
                float distance = (new Vector3(transform.position.x - Position.x, transform.position.y - Position.y,
0.0f)).magnitude;
        if (distance < 0.5f)
            return true;
        return false;
    }

    public void catchHuman()
    {
        isTagged = true;
        transform.position = new Vector3(respawnPosition.x, respawnPosition.y, transform.position.z);
    }

}
```

Inverter.cs

```
using System;
using UnityEngine;
using System.Collections.Generic;
using System.Linq;
using System.Text;

/* @class Inverter
 * @brief Class describing an inverter's behavior.
 * Whatever result the inverter's child returns, gets... inverted. Unless there is a PROCESSING
 * outcome, in that case it stays the same.
 */

namespace BTnamespace
{
    public class Inverter : Decorator
    {
        public override Outcome Process(BehaviorTree tree)
        {
            Outcome Result = _child.Process(tree);
            if (Result == Outcome.FAIL)
```

```
            return Outcome.SUCCESS;
        else if (Result == Outcome.SUCCESS)
            return Outcome.FAIL;
        else
            return Outcome.PROCESSING;
    }
  }
}
```

Leaf.cs

```
using System;
/* @class Leaf
 * @brief Class Leaf inherits Node class, doesn't have any properties of its own.
 */

namespace BTnamespace
{
    public abstract class Leaf : Node
    {}
}
```

Node.cs

```
using System;
/* @class Node
 * @brief Class Node contains functions about initializing and processing behaviors, as well as
 * defining the possible behavior results : success, fail, processing.
 */

namespace BTnamespace
{
    public enum Outcome
    {
        SUCCESS,
        FAIL,
        PROCESSING
    }

    public abstract class Node
    {
        public abstract void Initialize(BehaviorTree tree);
        public abstract Outcome Process(BehaviorTree tree);
    }
}
```

Parallel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

/* @class Parallel
 * @brief Class describing a parallel's behavior.
 * The parallel will return success if every child returns success. Even if only one child returns failure
 * then the parallel returns failure as well.
 */

namespace BTnamespace
{
    public class Parallel : Composite
    {
        private Outcome[] _childStatus;
        public override void Initialize(BehaviorTree tree)
        {
            _childStatus = new Outcome[_children.Count];
            for (int i = 0; i < _children.Count; i++){  _children[i].Initialize(tree);
                    _childStatus[i] = Outcome.PROCESSING;
            }
        }

        public override Outcome Process(BehaviorTree tree)
        {
            bool childInProcess = false;
            for(int i = 0; i < _children.Count; i++){
                        Outcome result = _children[i].Process(tree);
                        _childStatus[i] = result;
                        if (result == Outcome.FAIL) return Outcome.FAIL;
                        if(result == Outcome.PROCESSING)  childInProcess = true;
                        } if (childInProcess){
                        for(int i = 0; i < _children.Count; i++){
                              if(_childStatus[i] == Outcome.SUCCESS){
                                    _children[i].Initialize(tree);
                                    _childStatus[i] = Outcome.PROCESSING;
                              }
                        }
                        return Outcome.PROCESSING;
            }
                    return Outcome.SUCCESS;
        }
    }
}
```

ParallelSelector.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

/* @class ParallelSelector
 * @brief Class describing a parallel selector's behavior.
 * The parallel will return success if just one child returns success. The parallel selector then proceeds in terminating
 * every child. Same applies if just one child returns failure, the parallel selector will end every child in process and
 * then return failure as well.
 */

namespace BTnamespace
{
    public class ParallelSelector : Composite
    {
        private Outcome[] _childStatus;
        public override void Initialize(BehaviorTree tree)
        {
            _childStatus = new Outcome[_children.Count];

            for (int i = 0; i < _children.Count; i++){
                _children[i].Initialize(tree);
                _childStatus[i] = Outcome.PROCESSING;
            }
        }

        public override Outcome Process(BehaviorTree tree)
        {
            for (int i = 0; i < _children.Count; i++){
                if (_childStatus[i] == Outcome.PROCESSING){
                    Outcome result = _children[i].Process(tree);
                    _childStatus[i] = result;
                    if (result == Outcome.SUCCESS)
                                        return Outcome.SUCCESS;
                                    if(result == Outcome.FAIL)
                                        return Outcome.FAIL;
                }
            }
            return Outcome.PROCESSING;
        }
    }
}
```

ReleaseFlag.cs

```
using System;

using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class ReleaseFlag
 * @brief In this class, bot is required to drop the flag because it is currently tagged.
 * Results in failure if the bot doesn't currently possess the flag.
 */

namespace CTF
{
    class ReleaseFlag : Leaf
    {
        Transform MyTransform;
        public override void Initialize(BehaviorTree tree)
        {
            MyTransform = (Transform)tree.getValue(TreeData.MY_TRANSFORM);
        }

        public override Outcome Process(BehaviorTree tree)
        {
            bool hasFlag = (bool)tree.getValue(TreeData.HAS_FLAG);
            if (hasFlag){
                tree.setValue(TreeData.HAS_FLAG, false);
                tree.setValue(TreeData.SEEK_THIS, false);
                GameObject enemyFlag = tree.getValue(TreeData.ENEMY_FLAG) as GameObject;
                enemyFlag.GetComponent<FlagScript>().Release(MyTransform.gameObject);
                return Outcome.SUCCESS;
            }
            return Outcome.FAIL;
        }
    }
}
```

RepeatUntilFail.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

/* @class RepeatUntilFail
 * @brief Class describing a repeat until failure node's behavior.
 * This type of node has one child and executes it until the child returns a failure result.
 */

namespace BTnamespace
{
    public class RepeatUntilFail : Decorator
    {
        public override Outcome Process(BehaviorTree tree)
        {
                    Outcome Result = _child.Process(tree);
                    if ( Result != Outcome.FAIL){
                        if (Result == Outcome.SUCCESS)
                            _child.Initialize(tree);        //Begin again
                return Outcome.PROCESSING;
            }
            return Outcome.FAIL;
        }
    }
}
```

Respawn.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class Respawn
 * @brief In this class, bot has been caught by an enemy bot and is required to spawn again in its base.
 * If this class is accessed while a bot is not currently tagged, it returns failure.
 */

namespace CTF
{
    class Respawn : Leaf
```

```
    {
        GameObject myGameobject;
        bool isTagged = false;
        Transform myTransform;
        public override void Initialize(BehaviorTree tree)
        {
            myGameobject = ((Transform)tree.getValue(TreeData.MY_TRANSFORM)).gameObject;
            isTagged = (bool)tree.getValue(TreeData.IS_TAGGED);
            myTransform = (Transform)tree.getValue(TreeData.MY_TRANSFORM);
        }
        public override Outcome Process(BehaviorTree tree)
        {
            if (isTagged){
                tree.setValue(TreeData.SEEK_THIS, false);
                 myTransform.gameObject.GetComponent<BotScript>().BroadCastTeamMessage(BotScript.broadcastMes
sage.TAGGED);
                 Vector3 respawnPosition = (Vector3) tree.getValue(TreeData.RESPAWN_POSITION);   //locate respawn
position and go there instantly
                                myGameobject.transform.position  =  new  Vector3(respawnPosition.x,  respawnPosition.y,
myGameobject.transform.position.z);
                            tree.setValue(TreeData.IS_TAGGED,false);
                return Outcome.SUCCESS;
            }

            return Outcome.FAIL;
        }
    }
}
```

ScorePoint.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using BTnamespace;

/* @class ScorePoint
 * @brief In this class, bot has reached its base and is scoring the winning point.
 * This class cannot return a behavior result of FAIL.
 */

namespace CTF
{
    class ScorePoint : Leaf
    {
        BotScript.TeamID myTeamID;
```

```
        public override void Initialize(BehaviorTree tree)
        {
            myTeamID = (BotScript.TeamID)tree.getValue(TreeData.MY_TEAMID);
        }

        public override Outcome Process(BehaviorTree tree)
        {
            Debug.Log("score for team:"+myTeamID);
            Application.LoadLevel(0);          //score point - load level from the beginning
            return Outcome.SUCCESS;
        }
    }
}
```

Seek.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class Seek
 * @brief In this class, bot's velocity changes depending on the target that is
 *  currently being sought. Success is achieved if the target is reached.
 */

namespace CTF
{
    class Seek : Leaf
    {
        float steering;
        float velocity;
        Transform myTransform;
        GameObject seekNow;

        public override void Initialize(BehaviorTree tree)
        {
            steering = (float)tree.getValue(TreeData.STEER);
            velocity = (float)tree.getValue(TreeData.VELOCITY);
        }

        public override Outcome Process(BehaviorTree tree)
        {
            myTransform = (Transform)tree.getValue(TreeData.MY_TRANSFORM);
            seekNow = tree.getValue(TreeData.SEEK_THIS) as GameObject;
```

```
        float iVelFactor = velocity;
        if (seekNow){
            if (seekNow.gameObject.CompareTag("Flag"))          //going for flag = accelerate
                iVelFactor*= 2;
            else if((bool)tree.getValue(TreeData.HAS_FLAG)) //possesses flag = get half of normal speed
                iVelFactor /= 2;
            if (routineSeek(seekNow, iVelFactor))               //reached target
                return Outcome.SUCCESS;
            return Outcome.PROCESSING;                                  //still pursuing
        }
        else
            return Outcome.FAIL;
    }

        //routine to check if bot is close to the target it is seeking
    bool routineSeek(GameObject toSeek, float velocity)
    {
                                float distance = (new Vector3(myTransform.position.x - toSeek.transform.position.x,
myTransform.position.y - toSeek.transform.position.y, 0.0f)).magnitude;
        if (distance < 0.15f)
            return true;
                Vector3 Direction = (toSeek.transform.position - myTransform.position).normalized;
        Vector3 DesiredVelocity = Direction * velocity;
         Vector3 SteeringForce = (DesiredVelocity - myTransform.GetComponent<Rigidbody>().velocity).normalized *
steering;
        myTransform.GetComponent<Rigidbody>().AddForce(SteeringForce);
                                                        myTransform.GetComponent<Rigidbody>().velocity      =
Vector3.ClampMagnitude(myTransform.GetComponent<Rigidbody>().velocity, velocity);
        if (myTransform.GetComponent<Rigidbody>().velocity.magnitude > 0){
                                    float angle = Mathf.Atan2(myTransform.GetComponent<Rigidbody>().velocity.y,
myTransform.GetComponent<Rigidbody>().velocity.x) * Mathf.Rad2Deg;
            myTransform.transform.rotation = Quaternion.Euler(angle, -90f, 90f);
        }
        return false;
    }
  }
}
```

SeekEnemyFlag.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class SeekEnemyFlag
 * @brief In this class, the bot is seeking for the flag, after being the one chosen to go for it.
 */

namespace CTF
{
   class SeekEnemyFlag : Leaf
   {
            Transform myTransfrom;
      public override void Initialize(BehaviorTree tree)
      {
         myTransfrom = (Transform)tree.getValue(TreeData.MY_TRANSFORM);
          myTransfrom.gameObject.GetComponent<BotScript>().BroadCastTeamMessage(BotScript.broadcastMessa
ge.GOING_TO_FLAG);
      }

      public override Outcome Process(BehaviorTree tree)
      {
         GameObject EnemyFlag = tree.getValue(TreeData.ENEMY_FLAG) as GameObject;
         tree.setValue(TreeData.SEEK_THIS, EnemyFlag);
         return Outcome.SUCCESS;
      }
   }
}
```

Selector.cs

```
using System;
/* @class Selector
 * @brief Class describing a selector's behavior.
 * The selector will return a success result if any of its children returns
 * a success result.
 */

namespace BTnamespace
{
	public class Selector : Composite
	{
		private int _activeChildIndex = 0;

		public override void Initialize (BehaviorTree tree)
		{
			_activeChildIndex = 0;
			_children [_activeChildIndex].Initialize (tree);
		}

		public override Outcome Process (BehaviorTree tree)
		{
			Outcome result = _children[ _activeChildIndex ].Process (tree);
	    if (result == Outcome.FAIL){
	      ++_activeChildIndex;
	      if (_activeChildIndex >= _children.Count)
	        return Outcome.FAIL;
	      else{
	        _children[_activeChildIndex].Initialize(tree);
	        return Outcome.PROCESSING;
	      }
	    }
	    else
	      return result;
		  }
	}
}
```

Sequence.cs

```
using System;
/* @class Sequence
 * @brief Class describing a sequence's behavior.
 * This node runs its children in a sequential order. If a child returns
 * failure behavior result, then the sequence node returns failure as well.
 * If every child until the last one returns a success behavior result, then
 * the sequence node returns a success behavior result as well.
 */

namespace BTnamespace
{
	public class Sequence : Composite
	{
		private int _activeChildIndex = 0;

		public override void Initialize (BehaviorTree tree)
		{
			_activeChildIndex = 0;
			_children [_activeChildIndex].Initialize (tree);
		}

		public override Outcome Process (BehaviorTree tree)
		{
			Outcome result = _children[ _activeChildIndex ].Process (tree);
		if (result == Outcome.SUCCESS){
		++_activeChildIndex;
		if (_activeChildIndex >= _children.Count)
		   return Outcome.SUCCESS;
		else {
		   _children[_activeChildIndex].Initialize(tree);
		   return Outcome.PROCESSING;
		}
	   }
	   else
		   return result;
		}
	}
}
```

Suceeder.cs

```
using System;
using UnityEngine;
namespace BTnamespace
{
    public class Succeeder : Decorator
    {
        public override Outcome Process(BehaviorTree tree)
        {
            Outcome result = _child.Process(tree);
            if (result == Outcome.PROCESSING)
                return Outcome.PROCESSING;
            return Outcome.SUCCESS;
        }
    }
}
```

Tagged.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using UnityEngine;

using BTnamespace;

/* @class Tagged
 * @brief In this class, success result is given if the bot is tagged, failure message if otherwise.
 *      Hint: makes more sense as a class if combined with some further interaction from the teammates.
 *      For example, one of my first intentions was to make the teammate "dead" and not "tagged", therefore
 *      its teammates would have to decide who will go and resurrect him, in order to keep the team numbers even and
fair.
 */

namespace CTF
{
    class Tagged : Leaf
    {
        public override void Initialize(BehaviorTree tree)
        {

        }
        public override Outcome Process(BehaviorTree tree)
        {
```

```
            bool isTagged = (bool)tree.getValue(TreeData.IS_TAGGED);
            if (true == isTagged)
                return Outcome.SUCCESS;
            return Outcome.FAIL;
        }
    }
}
```

ΑΝΑΦΟΡΕΣ

[1] Colledanchise, M., & Ögren, P. (2014). How Behavior Trees modularize robustness and safety in hybrid systems. IEEE International Conference on Intelligent Robots and Systems, 1482–1488. http://doi.org/10.1109/IROS.2014.6942752

[2] Colledanchise, M., Parasuraman, R., & Ögren, P. (2015). Learning of Behavior Trees for Autonomous Agents. arXiv Preprint arXiv: …. Retrieved from http://arxiv.org/abs/1504.05811

[3] Ficocelli, L. (2007). Industry – Academia: How Can We Collaborate? Loading..., 1. Retrieved from http://journals.sfu.ca/loading/index.php/loading/article/view/10/21

[4] Larsen, S., & Groth, J. (2011). Extending Behavior Trees with Classical Planning, 72. Retrieved from http://projekter.aau.dk/projekter/files/52867041/ThesisFinal.pdf

[5] Millington, I., & Funge, J. D. (2009). Artificial intelligence for games. http://doi.org/10.1016/B978-0-12-374731-0.00001-3

[6] Nash, A., & Koenig, S. (2015). Game AI Pro 2: Collected Wisdom of Game AI Professionals. Retrieved from https://books.google.com/books?hl=en&lr=&id=7XV3CAAAQBAJ&pgis=1

[7] Oakes, B. J. (2013). Practical and theoretical issues of evolving behaviour trees for a turn-based game, (August).

[8] Perez, D., Nicolau, M., O'Neill, M., & Brabazon, A. (2011). Evolving behaviour trees for the Mario AI competition using grammatical evolution. Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6624 LNCS(PART 1), 123–132. http://doi.org/10.1007/978-3-642-20525-5_13

[9] Perez, D., Nicolau, M., O'Neill, M., & Brabazon, A. (2011). Reactiveness and navigation in computer games: Different needs, different approaches. 2011 IEEE Conference on Computational Intelligence and Games, CIG 2011, (July 2016), 273–280. http://doi.org/10.1109/CIG.2011.6032017

[10] Sithu Kyaw, A., Peters, C., & Naing Swe, T. (2013). Unity 4. x Game AI Programming. ISDN: 9781849693400

[11] Sukin, I. (2013). Game Development with Three.js. Retrieved from https://books.google.com/books?id=HY6kAQAAQBAJ&pgis=1

[12] Αναστασίου, Β. Μ., Διαμαντόπουλος, Π.,(2011). Μελέτη και υλοποίηση τεχνικών κατάστρωσης σχεδίου σε περιβάλλον Unity3D.

[13] Yang X-S, Chien SF, Ting TO. Computational Intelligence and Metaheuristic Algorithms with Applications. The Scientific World Journal. 2014;2014:425853. doi:10.1155/2014/425853.

[14] Division, P. (2002). Mark Griffiths The educational benefits of videogames Videogames have great positive potential in. Education and Health, 20(3), 47–51. Retrieved from http://www.sheu.org.uk/pubs/eh203mg.pdf

[15] Robertson, G., & Watson, I. (2015). Building Behavior Trees from Observations in Real-Time Strategy Games. Inista. http://doi.org/10.1109/INISTA.2015.7276774

[16] Year, F., Project, I., & Report, F. (2009). An A . I . Player for DEFCON : An Evolutionary Approach Using Behavior Trees.

[17] Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. Lecture Notes in Computer Science, 2592, January 2003.

[18] Child, C. H. T. & Dey, R. (2013). QL-BT: Enhancing Behaviour Tree Design and Implementation with Q-Learning. Computational Intelligence in Games (CIG), 2013 IEEE Conference on, pp. 275-282. doi: 10.1109/CIG.2013.6633623

[19] D. Xiaoqin et al., “Applying hierarchical reinforcement learning to computer games” in the IEEE Symp. on Computational Intelligence and Games, 2009, pp. 929–932.

[20] Palma, R., González-Calero, P. A., Gómez-Martín, M. A., & Gómez-Martín, P. P. (2011). Extending case-based planning with behavior trees. Proceedings of the 24th International Florida Artificial Intelligence Research Society, FLAIRS - 24, 407–412.

[21] Young, J. B. (2010, Fall). Mines.Humanoriented: Decision Tree Classifier. Retrieved: http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/lguo/decisionTree.html

[22] Hilburn, D., & Wars, K. S. (n.d.). Simulating Behavior Trees, 99-111

[23] Delmer, S. (2013). Behavior Trees for Hierarchical RTS AI, 1–10.