ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΝΑΥΠΗΓΩΝ ΜΗΧΑΝΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΤΟΜΕΑΣ ΜΕΛΕΤΗΣ ΠΛΟΙΟΥ ΚΑΙ ΘΑΛΑΣΣΙΩΝ ΜΕΤΑΦΟΡΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΕΛΕΤΗΣ ΠΛΟΙΟΥ

ΔΙΑΠΑΝΕΠΙΣΤΗΜΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ :
«ΝΑΥΤΙΚΗ ΚΑΙ ΘΑΛΑΣΣΙΑ ΤΕΧΝΟΛΟΓΙΑ ΚΑΙ ΕΠΙΣΤΗΜΗ»

Προσομοίωση Εκκένωσης Πλήθους, Επιβατηγού - Οχηματαγωγού Πλοίου, στο Περιβάλλον Unity3D

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Ι. Καραφωτιάς

Επιβλέπων : Αλέξανδρος Γκίνης
Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Μάιος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΝΑΥΠΗΓΩΝ ΜΗΧΑΝΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΤΟΜΕΑΣ ΜΕΛΕΤΗΣ ΠΛΟΙΟΥ ΚΑΙ ΘΑΛΑΣΣΙΩΝ ΜΕΤΑΦΟΡΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΕΛΕΤΗΣ ΠΛΟΙΟΥ

ΔΙΑΠΑΝΕΠΙΣΤΗΜΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ :
«ΝΑΥΤΙΚΗ ΚΑΙ ΘΑΛΑΣΣΙΑ ΤΕΧΝΟΛΟΓΙΑ ΚΑΙ ΕΠΙΣΤΗΜΗ»

Προσομοίωση Εκκένωσης Πλήθους, Επιβατηγού - Οχηματαγωγού Πλοίου, στο Περιβάλλον Unity3D

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Ι. Καραφωτιάς

Επιβλέπων : Αλέξανδρος Γκίνης
Επ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 27η Μαΐου 2015.

……..	……..	……..
Α. Γκίνης	Γ. Ζαραφωνίτης	Κ. Κώστας
Επ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Εφαρ. Τ.Ε.Ι. Αθήνας

Αθήνα, Μάιος 2015

.....
Γεώργιος Ι. Καραφωτιάς
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Γεώργιος Ι. Καραφωτιάς, 2015
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σκοπός της διπλωματικής εργασίας είναι η δημιουργία ενός προσομοιωτή της διαδικασίας εκκένωσης ενός επιβατηγού πλοίου σε έκτακτες καταστάσεις, κυρίως λόγω εσωτερικών απειλών. Ο προσομοιωτής θα αναπτυχθεί με τη βοήθεια του λογισμικού πακέτου Unity3D, το οποίο περιλαμβάνει την rendering μηχανή και το αναπτυξιακό περιβάλλον της τριδιάστατης εικονικής σκηνής. Ο προγραμματισμός της τεχνητής νοημοσύνης των αυτόνομων χαρακτήρων, που θα αναπαριστούν το πλήθος, θα χωρισθεί σε δύο μέρη : το πρώτο θα ελέγχει την κίνησή τους χρησιμοποιώντας το ενσωματωμένο σύστημα εύρεσης διαδρομής και το δεύτερο τους τρόπους και κανόνες αλληλεπίδρασης μεταξύ τους - το οποίο θα πραγματοποιηθεί με γραφή συγκεκριμένου κώδικα. Το αποτέλεσμα θα είναι η δημιουργία ενός μοντέλου πλήθους με ρεαλιστικά συστήματα κατεύθυνσης/κίνησης και συγκρούσεων.

Το μοντέλο αυτό θα χρησιμοποιηθεί στην προηγμένη μέθοδο ανάλυσης εκκένωσης που έχει υιοθετήσει η Maritime Safety Committee (MSC) και περιγράφεται στο «Interim guidelines for evacuation analysis for new and existing passenger ships». Οι κατευθυντήριες αυτές γραμμές προήλθαν από τους κανονισμούς του International Maritime Organization (IMO) για την ασφάλεια στα επιβατηγά πλοία και επιβάλλουν τη χρήση της μοντελοποίησης της εκκένωσης για την αξιολόγηση των διαδρομών διαφυγής από νωρίς στην διαδικασία σχεδιασμού του πλοίου. Η ανάγκη για αύξηση της ασφάλειας των επιβατών σε έκτακτες καταστάσεις προέκυψε λόγω σειράς θανάσιμων ατυχημάτων με πολλαπλές απώλειες σε επιβατηγά πλοία αλλά και από πρόσφατες ναυπηγήσεις cruise liners μερικών χιλιάδων επιβατών. Έτσι, η χρήση του μοντέλου θα διευκολύνει τον σχεδιασμό του νέου πλοίου καθώς θα μπορεί να επιδείξει διάφορες πληροφορίες στον ναυπηγό μηχανικό, όπως ο συνολικός χρόνος εκκένωσης, τις διαδρομές που ακολουθήθηκαν και τις περιοχές αυξημένης κυκλοφοριακής συμφόρησης.

Λέξεις Κλειδιά

Προσομοίωση, Μοντελοποίηση, Εκκένωση, Επιβατηγό Πλοίο, Τεχνητή Νοημοσύνη, Αυτόνομος Χαρακτήρας, Μοντέλο Πλήθους, Κυκλοφοριακή Συμφόρηση, MSC, IMO, Unity3D

Abstract

The objective of this thesis is the creation of a simulator of a passenger ship's evacuation procedure after an emergency, usually caused by internal threats. The simulator will be developed on the Unity3D software package, which includes a 3D rendering engine and a development environment for the virtual scene. The autonomous characters (agents), which all together represent a crowd, will have artificial intelligence that is programmed in two parts : the first will control their movement using the embedded, in Unity3D, path navigation system and the second the ways and rules that the agents can interact with each other. The final outcome will be the creation of a crowd model with realistic movement and collision systems.

This model will be used in the advanced evacuation analysis method that is authored by Maritime Safety Committee (MSC) and further described in «Interim guidelines for evacuation analysis for new and existing passenger ships». These guidelines derived from International Maritime Organization's (IMO) regulations about safety in passenger ships and impose the use of evacuation modelling to evaluate the escape routes in the ship design process. The need for increased measures concerning the passengers' safety in real emergency conditions originated from a series of multiple death accidents involving passenger ships and from recent shipbuilding of new, several thousand passenger, cruise liners. So, the model will facilitate the design of a new ship since it can display various information to the naval architect, such as the total evacuation time, the paths that were followed and the congestion points.

Keywords

Simulation, Modelling, Evacuatioon, Passenger Ship, Atificial Intelligence, Autonomous Character, Agent, Crowd Model, Congestion, MSC, IMO, Unity3D

Περιεχόμενα

Περίληψη **5**

Abstract **6**

Περιεχόμενα **7**

Κατάλογος Πινάκων **8**

Κατάλογος Σχημάτων **9**

Κεφάλαιο 1ο) ΕΙΣΑΓΩΓΗ **11**

1.1) ΣΥΝΤΟΜΗ ΠΕΡΙΓΡΑΦΗ ΤΗΣ MSC 12
1.2) ΑΣΦΑΛΕΙΑ ΣΤΑ ΕΠΙΒΑΤΗΓΑ ΠΛΟΙΑ 13

Κεφάλαιο 2ο) ΛΟΓΙΣΜΙΚΟ **16**

2.1) AUTOCAD 16
2.1.1) Γενικά 16
2.1.2) Τρόπος λειτουργίας 16
2.1.3) Χρήση 18
2.2) 3D STUDIO MAX 20
2.2.1) Γενικά 20
2.2.2) Τρόπος λειτουργίας 20
2.2.3) Χρήση 22
2.3) UNITY3D 26
2.3.1) Γενικά 26
2.3.2) Τρόπος λειτουργίας 27
2.3.3) Χρήση 28

Κεφάλαιο 3ο) ΜΟΝΤΕΛΟ ΜΕΘΟΔΟΥ **30**

3.1) ΠΡΟΗΓΜΕΝΗ ΜΕΘΟΔΟΣ ΑΝΑΛΥΣΗΣ ΕΚΚΕΝΩΣΗΣ 30
3.1.1) Σκοπός προηγμένης μεθόδου 31
3.1.2) Υποθέσεις προηγμένης μεθόδου 31
3.1.3) Τρόπος υπολογισμού συνολικού χρόνου εκκένωσης 31
3.2) ΜΟΝΤΕΛΟΠΟΙΗΣΗ ΟΝΤΟΤΗΤΑΣ-AGENT 33
3.2.1) Παράμετροι Agent 33
3.2.2) Συμπεριφορές και κίνηση Agent 39
3.2.2.1) Συμπεριφορά Seek 39
3.2.2.2) Συμπεριφορά Arrive 40
3.2.2.3) Συμπεριφορά AvoidWall 41
3.2.2.4) Συμπεριφορά PushAgent 42
3.2.2.5) Συμπεριφορά AvoidNearest 43
3.2.2.6) Κίνηση Agent 44

Κεφάλαιο 4ο) ΥΛΟΠΟΙΗΣΗ ΠΡΟΣΟΜΟΙΩΣΗΣ ΣΤΟ UNITY3D **46**
4.1) ΚΕΝΤΡΙΚΗ ΣΚΗΝΗ 46
4.2) ΑΝΑΛΥΣΗ AGENT 55
4.3) ΑΝΑΛΥΣΗ AGENTMANAGER 64
4.4) ΟΛΟΚΛΗΡΩΣΗ ΠΡΟΣΟΜΟΙΩΣΗΣ 70
4.5) ΤΕΣΤ ΣΚΗΝΕΣ 72
Κεφάλαιο 5ο) ΣΥΜΠΕΡΑΣΜΑΤΑ **74**
Βιβλιογραφία **76**

Κατάλογος Πινάκων

Κεφάλαιο 2ο)
ΠΙΝΑΚΑΣ 2.1 : ΔΙΑΣΤΑΣΕΙΣ ΚΑΤΑΣΤΡΩΜΑΤΩΝ 19

Κεφάλαιο 3ο)
ΠΙΝΑΚΑΣ 3.1 : ΣΥΝΘΕΣΗ ΠΛΗΘΥΣΜΟΥ (ΗΛΙΚΙΑ ΚΑΙ ΦΥΛΟ) 37
ΠΙΝΑΚΑΣ 3.3 : ΜΕΓΙΣΤΗ ΤΑΧΥΤΗΤΑ ΣΕ ΕΠΙΠΕΔΟ ΕΔΑΦΟΣ 38
ΠΙΝΑΚΑΣ 3.3 : ΜΕΓΙΣΤΗ ΤΑΧΥΤΗΤΑ ΣΕ ΣΚΑΛΕΣ 38

Κεφάλαιο 4ο)
ΠΙΝΑΚΑΣ 4.1 : ΠΑΡΑΔΕΙΓΜΑ ΜΕΤΡΗΣΕΩΝ ΤΙΜΩΝ CONGESTION ΠΡΟΣ ΧΡΟΝΟ 71

Κατάλογος Εικόνων

Κεφάλαιο 2ο)

Εικονα 2.1 : Καταστρωμα γεφυρας – deck7 17
Εικονα 2.2 : Καταστρωμα επιβατων – deck6 18
Εικονα 2.3 : Καταστρωμα επιβιβασης – deck5 19
Εικονα 2.4 : Μοντελα deck7 και deck7_ground 20
Εικονα 2.5 : Ελεγχος σφαλματων του σχεδιου με τον μ/Σ Unwrap UVW 23
Εικονα 2.6 : Μοντελα deck6 και deck6_ground 24
Εικονα 2.7 : Μοντελα deck5 και deck5_ground 25
Εικονα 2.8 : Διεπαφη του Unity3D (edit mode) 27
Εικονα 2.9 : Διεπαφη του Unity3D (Play mode) 28

Κεφάλαιο 3ο)

Εικονα 3.1 : Συνολικος χρονος εκκενωση 32
Εικονα 3.2 : Παραμετροι Agent 34
Εικονα 3.3 : Γραφικο περιβαλλον αποδοσης τιμων στις παραμετρους του AgentManager 35
Εικονα 3.4 : Συμπεριφορα Seek 39
Εικονα 3.5 : Συμπεριφορα AvoidWall 41
Εικονα 3.6 : Συμπεριφορα PushAgent 42
Εικονα 3.7 : Συμπεριφορα AvoidNearest με ταχυτητες ιδιας κατευθυνησης 43
Εικονα 3.8 : Συμπεριφορα AvoidNearest με ταχυτητες αντιθετης κατευθυνησης 43

Κεφάλαιο 4ο)

Εικονα 4.1 : Δημιουργια NavMesh του Deck5 46
Εικονα 4.2 : Main camera culling του Deck6 47
Εικονα 4.3 : Μητρα συγκρουσεων μεταξυ Layers 49
Εικονα 4.4 : Deck7 και Deck7_ground με Glass Shader 50
Εικονα 4.5 : Deck7 και Deck7_ground με Transparent και Outline Shaders 51
Εικονα 4.6 : Editor Script για τοποθετηση Prefab : GeneratorAgent 52
Εικονα 4.7 : Generator Script για δημιουργια Agents 52
Εικονα 4.8 : Congestion Area 53
Εικονα 4.9 : Muster Station 54
Εικονα 4.10 : Debug πληροφοριες στο παραθυρο Scene 55
Εικονα 4.11 : Παραδειγμα “stuck” Agent 57
Εικονα 4.12 : Function : OffsetCorners 59
Εικονα 4.13 : Επεξηγηση Function : OffsetCorners 60
Εικονα 4.14 : Ο Agent ελεγχει αν μπορει να συντομευσει την πορεια του 62
Εικονα 4.15 : Αποσταση Agent απο πλησιεστερη ακρη του NavMesh 63
Εικονα 4.16 : AgentManager 68
Εικονα 4.17 : Ολοκληρωση προσομοιωσης 71
Εικονα 4.18 : Παραδειγμα Test Scene 73

Κεφάλαιο 1

Εισαγωγή

Η παρούσα διπλωματική εργασία προέκυψε ως φυσική συνέχεια της εξαμηνιαίας εργασίας που εκπονήσαμε στο πλαίσιο του μαθήματος «Στερεά Μοντελοποίηση : Ειδικά Κεφάλαια και Εφαρμογές στο Εικονικό Πλοίο» του διαπανεπιστημιακού προγράμματος μεταπτυχιακών σπουδών «Ναυτική και Θαλάσσια Τεχνολογία και Επιστήμη». Σε εκείνη την εργασία μελετήθηκαν βασικές αρχές και λειτουργίες της τεχνητής νοημοσύνης στο περιβάλλον Unity3D, με κυριότερες τις :

- Finite State Machine
- Pathfinding χρησιμοποιώντας τον αλγόριθμο A*
- Pathfinding χρησιμοποιώντας το σύστημα Nav Mesh
- Pathfollowing
- Obstacle Avoidance
- Flocking

Χρησιμοποιώντας αυτές τις τεχνικές και προσθέτοντας και άλλες, όπως διάφορες συμπεριφορές που θα αναλύσουμε στη συνέχεια, καταλήξαμε στη κατασκευή ενός μοντέλου για την ανάλυση της διαδικασίας εκκένωσης ενός επιβατηγού-οχηματαγωγού πλοίου που αποτελεί και το θέμα της μεταπτυχιακής διπλωματικής εργασίας. Θα προσπαθήσουμε να δείξουμε ότι το μοντέλο αυτό συνάδει με τις απαιτήσεις της προηγμένης μεθόδου της επιτροπής Maritime Safety Committee (MSC) του Διεθνούς Ναυτιλιακού Οργανισμού (International Maritime Organization – IMO). Πριν προχωρήσουμε στην περιγραφή του μοντέλου μας, ας δούμε πρώτα μια σύντομη περιγραφή της MSC και γενικά της ασφάλειας στα επιβατηγά πλοία.

1.1 Σύντομη περιγραφή της MSC

Η Maritime Safety Committee είναι θυγατρική του συμβουλίου του IMO [1]. Η επιτροπή αποτελείται από όλα τα κράτη μέλη και είναι το ανώτερο τεχνικό όργανο του οργανισμού. Οι λειτουργίες της είναι να εξετάζει κάθε θέμα που εμπίπτει στην δικαιοδοσία του οργανισμού σχετιζόμενο με :

- βοηθήματα ναυσιπλοΐας,
- κατασκευή και εξοπλισμό πλοίων,
- ασφάλεια επάνδρωσης,
- κανόνες για την πρόληψη συγκρούσεων,
- διαχείριση επικίνδυνων φορτίων,
- διαδικασίες και απαιτήσεις της ασφάλειας στη θάλασσα,
- υδρογραφικές πληροφορίες,
- ημερολόγιο πλοίου και αρχεία πλοήγησης,
- διερεύνηση θαλάσσιων ατυχημάτων,
- ανέλκυσης και διάσωσης,
- οποιοδήποτε άλλο θέμα που επηρεάζει την ασφάλεια στη θάλασσα.

Η Επιτροπή καλείται επίσης να παρέχει τους μηχανισμούς για την εκτέλεση οποιωνδήποτε καθηκόντων που της ανατίθενται από τη σύμβαση του IMO ή οποιοδήποτε καθήκον, εντός των προαναφερθέντων θεμάτων, που ενδέχεται να της ανατεθεί από κάποιο διεθνές όργανο αφού γίνει πρώτα αποδεκτό από τον οργανισμό. Έχει, ακόμα, την ευθύνη για την εξέταση και την υποβολή συστάσεων και οδηγιών σχετικά με την ασφάλεια για πιθανή έγκριση από τη συνέλευση του οργανισμού.

Η MSC [2], έχοντας ήδη εγκρίνει τις κατευθυντήριες γραμμές μιας απλοποιημένης ανάλυσης εκκένωσης RO-RO (Roll On-Roll Off) πλοίων ως έναν οδηγό για την υλοποίηση του κανονισμού II-2/28-1.3 της SOLAS (Safety Of Life At Sea), ζήτησε τον Μάιο του 1999 απ’την υποεπιτροπή υπεύθυνη για την προστασία από φωτιά να αναπτύξει κανόνες για την ανάλυση της εκκένωσης σε επιβατηγά πλοία εν γένει αλλά και σε ταχύπλοα επιβατηγά.

Τον Ιούνιο του 2001, μετά από τις συστάσεις της υποεπιτροπής, ενέκρινε τις κατευθυντήριες γραμμές για την απλοποιημένη μέθοδο ανάλυσης εκκένωσης ταχύπλοων επιβατηγών. Τον Μάιο του 2005 εξέτασε μια πρόταση της υποεπιτροπής και τελικά ενέκρινε καινούργιες κατευθυντήριες γραμμές που αντικατέστησαν τις προηγούμενες.

Ενδιάμεσα, τον Μάιο του 2002, έκανε δεκτές τις κατευθυντήριες γραμμές της υποεπιτροπής για ανάλυση εκκένωσης για νέα και υπάρχοντα επιβατηγά πλοία. Επίσης, κάλεσε τα κράτη μέλη να συλλέξουν και να υποβάλλουν στην υποεπιτροπή οτιδήποτε δεδομένα ή πληροφορίες έχουν από έρευνα, ανάπτυξη ή δοκιμές, ευρήματα για την ανθρώπινη συμπεριφορά που θα μπορούσαν να χρησιμοποιηθούν για να βελτιώσουν τις παρούσες κατευθυντήριες γραμμές.

Την σημερινή μορφή των κατευθυντήριων γραμμών έδωσε η MSC τον Οκτώβριο του 2007. Αφορούν νέα και υπάρχοντα επιβατηγά πλοία, συμπεριλαμβανομένων και των RO-RO. Υπάρχουν δύο διαφορετικές μέθοδοι για την ανάλυση της εκκένωσης :

- Η απλοποιημένη.
- Η προηγμένη.

Εμείς θα χρησιμοποιήσουμε την προηγμένη μέθοδο και θα την αναλύσουμε περισσότερο στο κεφάλαιο 3.

1.2 Ασφάλεια στα επιβατηγά πλοία

Στη ΑΕΝ Σχολή Πλοιάρχων γίνονται οι παρακάτω συστάσεις όσον αφορά θέματα ασφάλειας σε επιβατηγά πλοία [3] :

1. Η Σύμβαση του IMO για την Ασφάλεια της Ζωής στην Θάλασσα (Safety Of Life At Sea) απαιτεί επαρκή αριθμό εκπαιδευμένων ατόμων να βρίσκονται στο πλοίο για την συγκέντρωση και την αρωγή ανεκπαίδευτων ατόμων.
2. Το πλήρωμα, το οποίο είναι καθορισμένο από τους πίνακες διαιρέσεως να βοηθά τους επιβάτες σε περιπτώσεις εκτάκτων αναγκών, θα πρέπει να λαμβάνει επιπρόσθετη εκπαίδευση, προκειμένου να διασφαλίζεται ότι θα εκτελεί σωστά τα καθήκοντά του. Ο αριθμός των εκπαιδευμένων ατόμων θα πρέπει να περιλαμβάνεται στο έγγραφο ασφαλούς επανδρώσεως του πλοίου.
3. Όπου πραγματοποιείται εκπαίδευση σε μία σειρά εκπαιδευτικών μαθημάτων στην ξηρά, θα πρέπει να συμπληρώνεται από εκπαίδευση επί του πλοίου πριν από την ανάληψη των καθηκόντων που αναφέρονται στη 2. Η εκπαίδευση θα πρέπει να ικανοποιεί το κράτος σημαίας και θα πρέπει να προσδιορίζονται κάποια μέσα για την εξασφάλιση ότι τα μέλη του πληρώματος διατηρούν συνεχή επάρκεια γνώσεων μέσα από περιοδικές σειρές επιμορφωτικών μαθημάτων, γυμνάσια ή σχετική εργασιακή εμπειρία.
4. Οι επικοινωνιακές δεξιότητες των καθορισμένων ναυτικών θα πρέπει να επαρκούν, προκειμένου να βοηθούν τους επιβάτες κατά την διάρκεια μίας έκτακτης ανάγκης, λαμβάνοντας υπόψη τα ακόλουθα κριτήρια :
   α. την γλώσσα ή τις γλώσσες, οι οποίες είναι κατάλληλες για τις κυριότερες εθνικότητες επιβατών που μεταφέρονται σε μία συγκεκριμένη διαδρομή,
   β. την ικανότητα χρήσεως στοιχειώδους Αγγλικού λεξιλογίου για βασικές οδηγίες, η οποία να παρέχει τρόπο επικοινωνίας με έναν επιβάτη, που χρήζει βοήθειας, είτε ο επιβάτης και το μέλος του πληρώματος μιλούν μία κοινή γλώσσα είτε όχι,
   γ. την πιθανή ανάγκη επικοινωνίας κατά την διάρκεια ενός έκτακτου περιστατικού με άλλα μέσα (π.χ. μέσω επιδείξεως ή νευμάτων ή ζητώντας του να προσέξει

τον χώρο των οδηγιών, τους σταθμούς συγκεντρώσεως, τα μέσα διασώσεως ή τις διόδους εκκενώσεως) όταν η λεκτική επικοινωνία δεν εξυπηρετεί,

δ. την ευρύτητα με την οποία έχουν παρασχεθεί πλήρεις οδηγίες ασφαλείας στους επιβάτες στην μητρική τους γλώσσα ή γλώσσες,

ε. τις γλώσσες στις οποίες μπορούν να μεταδοθούν έκτακτες ανακοινώσεις κατά την διάρκεια μίας έκτακτης ανάγκης ή ενός γυμνασίου, προκειμένου να μεταφέρουν ζωτικές οδηγίες στους επιβάτες και να βοηθήσουν τα μέλη του πληρώματος στην αρωγή των επιβατών.

5. Η εκπαίδευση που παρέχεται βάσει της 2. θα πρέπει να περιλαμβάνει, αλλά όχι απαραίτητα να περιορίζεται, τα ακόλουθα θεωρητικά και πρακτικά αντικείμενα :

α. επίγνωση των σχεδίων των σωστικών συσκευών και των σχεδίων αντιμετωπίσεως πυρκαϊών και γνώση των πινάκων διαιρέσεως και των εκτάκτων οδηγιών συμπεριλαμβανομένων :

i. των γενικών συναγερμών και των διαδικασιών για την συγκέντρωση των επιβατών,

ii. των χώρων ευθύνης με έμφαση στους “τομείς που ανήκουν”.

β. την γενική διάταξη του πλοίου με ιδιαίτερη έμφαση στην θέση των σταθμών συγκεντρώσεως και επιβιβάσεως σε σωσίβιες λέμβους, στις προσβάσεις και στις διόδους διαφυγής,

γ. την θέση και την χρήση του εξοπλισμού εκτάκτων αναγκών σε σχέση με τα καθήκοντα της 2. με έμφαση στον «τομέα που ανήκουν» και τις διόδους διαφυγής από εκεί,

δ. την θέση των ατομικών σωσιβίων για ενήλικες και παιδιά,

ε. την θέση άλλων εφοδίων εκκενώσεως, π.χ. κουβέρτες, τα οποία πρέπει να μεταφερθούν στα σωσίβια σκάφη,

στ. στοιχειώδεις πρώτες βοήθειες και μεταφορά θυμάτων,

ζ. επικοινωνία :

i. χρήση συστημάτων ενδοεπικοινωνίας,

ii. σήμανση συναγερμού,

iii. ειδοποίηση των επιβατών,

iv. αναφορά και ειδοποίηση.

η. εκκένωση :

i. χρήση καταλόγων επιβατών ή καταμετρήσεις,

ii. συναγερμοί,

iii. συγκέντρωση - σημασία τηρήσεως τάξεως και διαδικασίες αποφυγής πανικού,

iv. έξοδοι κινδύνου,

v. εξοπλισμός εκκενώσεως,

vi. έλεγχος επιβατών στους διαδρόμους, στα κλιμακοστάσια και στα περάσματα,

vii. διατήρηση ελευθερίας διόδων διαφυγής από τυχόν εμπόδια,

viii. παροχή βοήθειας καθ’ οδόν προς τους σταθμούς συγκεντρώσεως και επιβιβάσεως σε σωσίβιες λέμβους,

ix. διαθέσιμες μέθοδοι για την απομάκρυνση ανθρώπων με αναπηρία και ατόμων που χρήζουν ιδιαίτερης βοήθειας,
x. περιορισμοί στην χρήση ανελκυστήρων,
xi. έρευνα στους χώρους ενδιαιτήσεως,
xii. εξασφάλιση ότι οι επιβάτες είναι κατάλληλα ενδεδυμένοι και ότι έχουν φορέσει σωστά τα ατομικά τους σωσίβια.

θ. περιπτώσεις πυρκαϊάς :
   i. ανίχνευση πυρκαϊάς και αρχικός περιορισμός,
   ii. σήμανση συναγερμού,
   iii. κίνδυνος εισπνοής καπνού,
   iv. αναπνευστική προστασία.

ι. περιπτώσεις εγκαταλείψεως του πλοίου:
   i. σωστή χρήση ατομικού σωστικού εξοπλισμού, π.χ. ατομικών σωσιβίων, στολών εμβαπτίσεως, κυκλικών σωσιβίων, φωτεινών σημάτων και καπνογόνων,
   ii. ανάγκη παροχής βοήθειας σε ιδιαίτερες περιπτώσεις.

ια. εξοικείωση μέσω επαναλαμβανόμενων οργανωμένων ξεναγήσεων στο πλοίο

ιβ. επαναλαμβανόμενη συμμετοχή σε γυμνάσια πυρκαϊάς και επιβιβάσεως σε σωσίβιες λέμβους, συμπεριλαμβανομένης της προσομοιώσεως μεταφοράς θυμάτων

ιγ. επαναλαμβανόμενη εξάσκηση στην χρήση του εξοπλισμού, όπως είναι η ένδυση ατομικών σωσιβίων και κατάλληλης προστατευτικής ενδυμασίας

ιδ. επαναλαμβανόμενη εξάσκηση στην χρήση των συστημάτων ενδοεπικοινωνίας, και

ιε. επαναλαμβανόμενη εξάσκηση στην εκκένωση.

6. Πριν από τον απόπλου του πλοίου, θα πρέπει να δίδονται οδηγίες προς τους επιβάτες σχετικά με τις διαδικασίες εκτάκτων αναγκών και εκκενώσεως του πλοίου.
7. Όπου είναι εφικτό, θα πρέπει να προβάλλεται βιντεοταινία σχετικά με την ασφάλεια αμέσως μετά την επιβίβαση των επιβατών.
8. Θα πρέπει να υπάρχουν τοποθετημένα σε κατάλληλο ύψος σαφή σήματα εκτάκτων αναγκών για την αρωγή των επιβατών, διατυπωμένα σε μία γλώσσα κατανοητή από την πλειονότητά τους, τα οποία θα υποδεικνύουν τις διαδρομές προς τους σταθμούς συγκεντρώσεως και την θέση των κιβωτίων με τα ατομικά σωσίβια. Γι’ αυτόν τον σκοπό θα πρέπει να χρησιμοποιούνται τα διεθνή σύμβολα του IMO.
9. Τα γυμνάσια επιβιβάσεως σε σωσίβιες λέμβους θα πρέπει να διεξάγονται σύμφωνα με την SOLAS. Οι άλλες σωστικές συσκευές και ο εξοπλισμός θα πρέπει να ελέγχονται τακτικά και να διατηρούνται σε καλή κατάσταση. Θα πρέπει πάντοτε να τηρούνται οι οδηγίες των κατασκευαστών αναφορικά με την συντήρηση και την αντικατάστασή τους.
10. Τα συστήματα συριγμών και επικοινωνιών θα πρέπει να δοκιμάζονται τακτικά και να διατηρούνται σε καλή λειτουργική κατάσταση.
11. Τα γυμνάσια και οι διαδικασίες για άνθρωπο στη θάλασσα θα πρέπει να διεξάγονται τακτικά.

Κεφάλαιο 2

Λογισμικό

Σε αυτό το κεφάλαιο θα περιγράψουμε περιληπτικά τα τρία διαφορετικά προγράμματα που χρησιμοποιήσαμε για την ολοκλήρωση της εργασίας. Για κάθε πρόγραμμα, θα αναφέρουμε γενικές πληροφορίες, θα δείξουμε τον τρόπο λειτουργίας και θα εξηγήσουμε πώς το αξιοποιήσαμε εμείς για την επίτευξη της δημιουργίας του μοντέλου προσομοίωσης της διαδικασίας εκκένωσης.

2.1 AutoCAD

2.1.1 Γενικά

Πρόκειται για το πιο γνωστό πρόγραμμα της εταιρείας Autodesk [4]. Είναι σχεδιαστικό λογισμικό (Computer-Aided Design, CAD) που υπάρχει από το 1982 και θεωρείται το πιο διαδεδομένο CAD πρόγραμμα παγκοσμίως. Χρησιμοποιείται κυρίως από μηχανικούς και γι'αυτό υπάρχουν εξειδικευμένες εκδόσεις του, όπως :

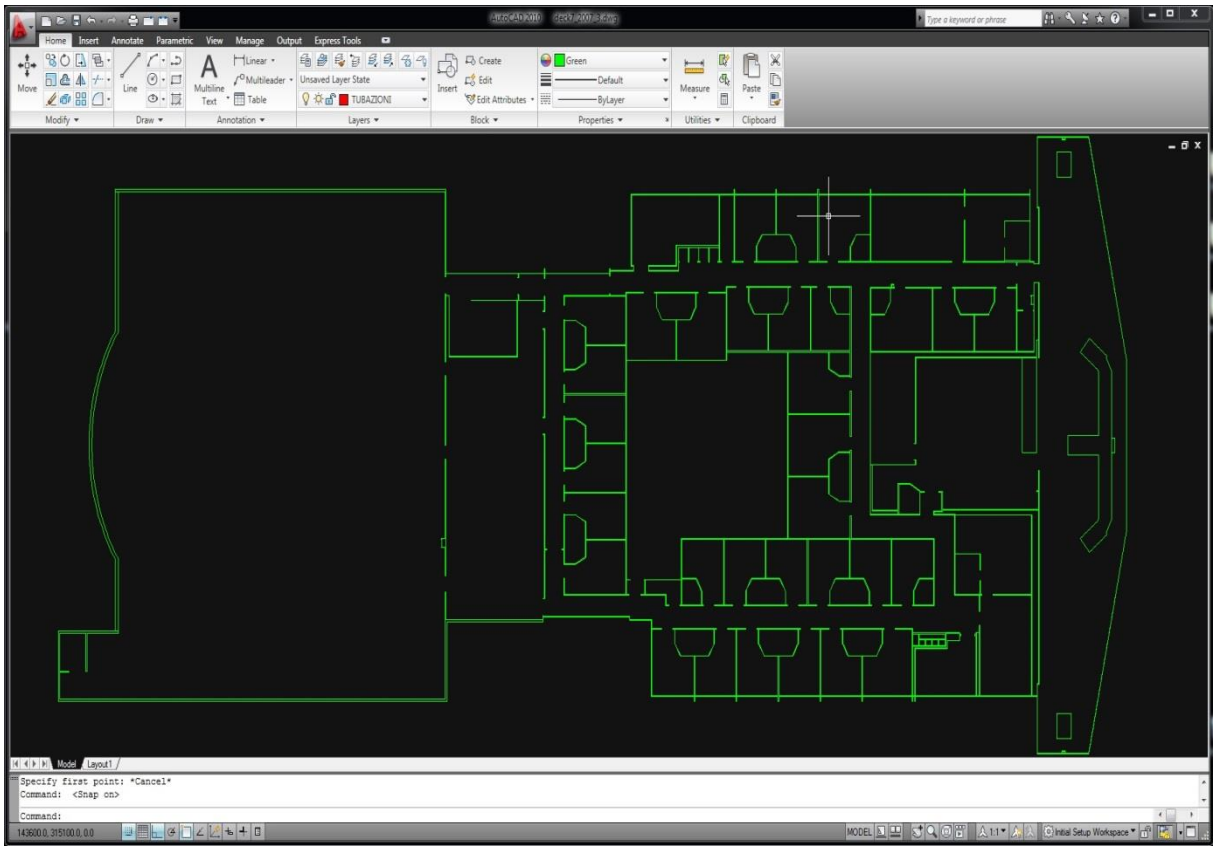
- Για αρχιτέκτονες μηχανικούς : AutoCAD Architecture.
- Για πολιτικούς μηχανικούς : AutoCAD Civil 3D.
- Για ηλεκτρολόγους μηχανικούς : AutoCAD Electrical.
- Για μηχανολόγους μηχανικούς : AutoCAD Mechanical.
- Για τοπογράφους μηχανικούς : AutoCAD Map 3D (GIS).
- Για μηχανικούς κατασκευής κτιρίων : AutoCAD MEP.
- Για μηχανικούς κατασκευής σωληνώσεων : AutoCAD P&ID.
- Για μηχανικούς κατασκευής εργοστασίων : AutoCAD Plant 3D.

2.1.2 Τρόπος λειτουργίας

Η σχεδίαση στο AutoCAD βασίζεται στη χρήση βασικών σχημάτων όπως γραμμές, πολύγωνα, κύκλοι και την τροποποίησή τους για να επιτύχουμε το ακριβές σχήμα που

θέλουμε. Η τροποποίηση γίνεται με μετασχηματισμούς, π.χ. : περιστροφή (rotate), μετατόπιση στο χώρο (offset), αλλαγή μεγέθους (scale), δημιουργία ειδώλου (mirror), έκταση προς κάποια κατεύθυνση (stretch), περικοπή (trim), ένωση με καμπύλη γραμμή (fillet), ένωση ομοίων αντικειμένων σε ένα (join), διάσπαση ενός αντικειμένου σε κάποιο σημείο (break), αντιστροφή της διεύθυνσης μιας ευθείας ή καμπύλης γραμμής (reverse) και άλλοι πολλοί δευτερεύοντες.

Επίσης, υπάρχουν πολλές βοηθητικές λειτουργίες που διευκολύνουν την εργασία μας όπως η δημιουργία και η επεξεργασία αντικειμένων ως μια ομάδα (group), η χρήση επεξηγηματικού κειμενου, η ένταξη αντικειμένων σε κάποιο επίπεδο (layer) ώστε να μπορούμε να εργαζόμαστε σε αυτό χωρίς να επηρεάζουμε τα αντικείμενα των υπολοίπων layers, η καταμέτρηση ακριβών διαστάσεων (distance) και ο προγραμματισμός διαφόρων πρόσθετων λειτουργιών στις γλώσσες : VBA, .NET, AutoLISP, Visual LISP, ObjectARX (βασισμένη στη C++).



Εικόνα 2.1 : Κατάστρωμα γέφυρας – deck7

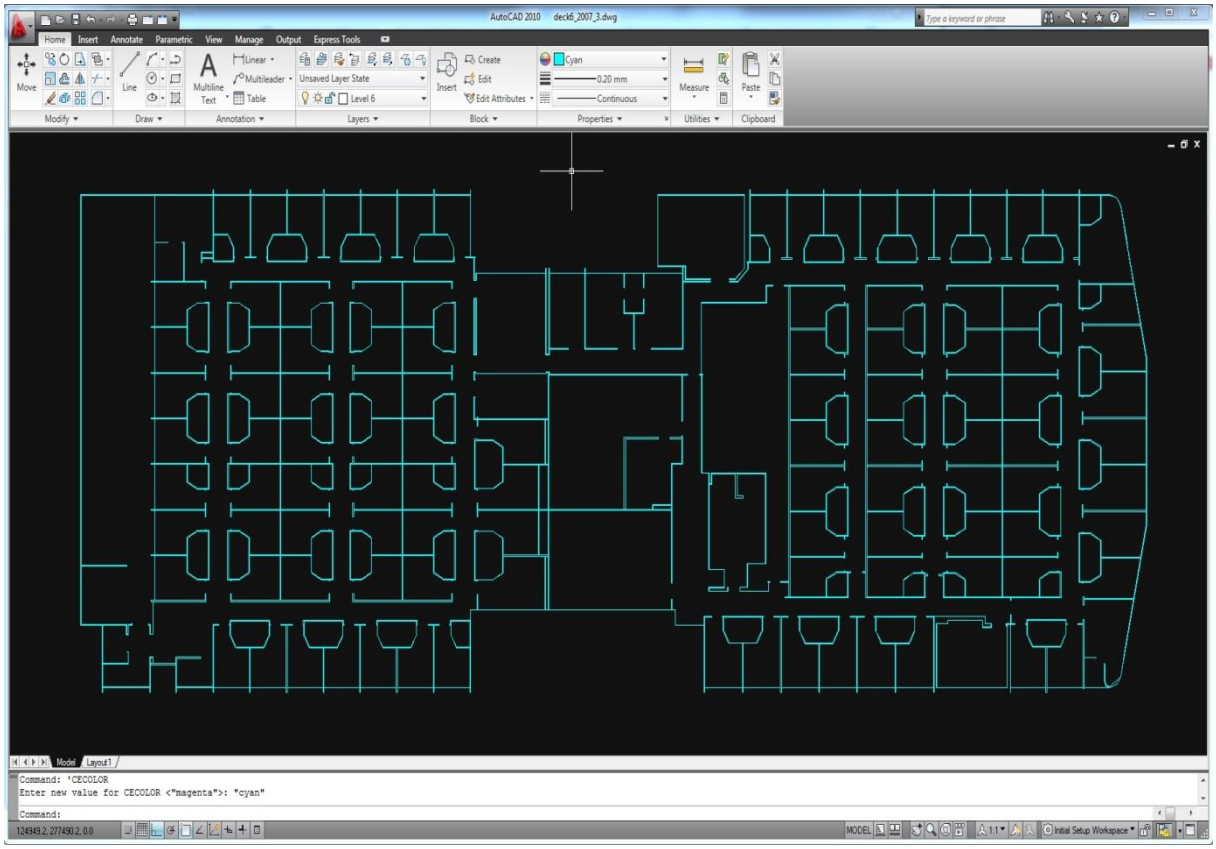
Στην Εικόνα 2.1 μπορούμε να δούμε τη διεπαφή του χρήστη (user interface). Η οθόνη ουσιαστικά χωρίζεται σε τρία μέρη. Το πάνω μέρος έχει όλες τις διαθέσιμες λειτουργίες, το μεσαίο και μεγαλύτερο είναι η περιοχή σχεδιασμού των αντικειμένων μας και στο κάτω μέρος βρίσκεται η κονσόλα στην οποία μπορούμε να δίνουμε απευθείας εντολές, π.χ. η εντολή line δημιουργεί μια γραμμή από τα δύο σημεία που θα ορίσουμε, ή να εισάγουμε τις παραμέτρους μιας εντολής. Κάτω από την κονσόλα υπάρχει μια γραμμή με βοηθητικές

λειτουργίες για την επιλογή και τοποθέτηση αντικειμένων αλλά και χειρισμού του πεδίου εργασίας μας, δηλαδή του κεντρικού μέρους, π.χ. η δυνατότητα να κάνουμε zoom/unzoom.

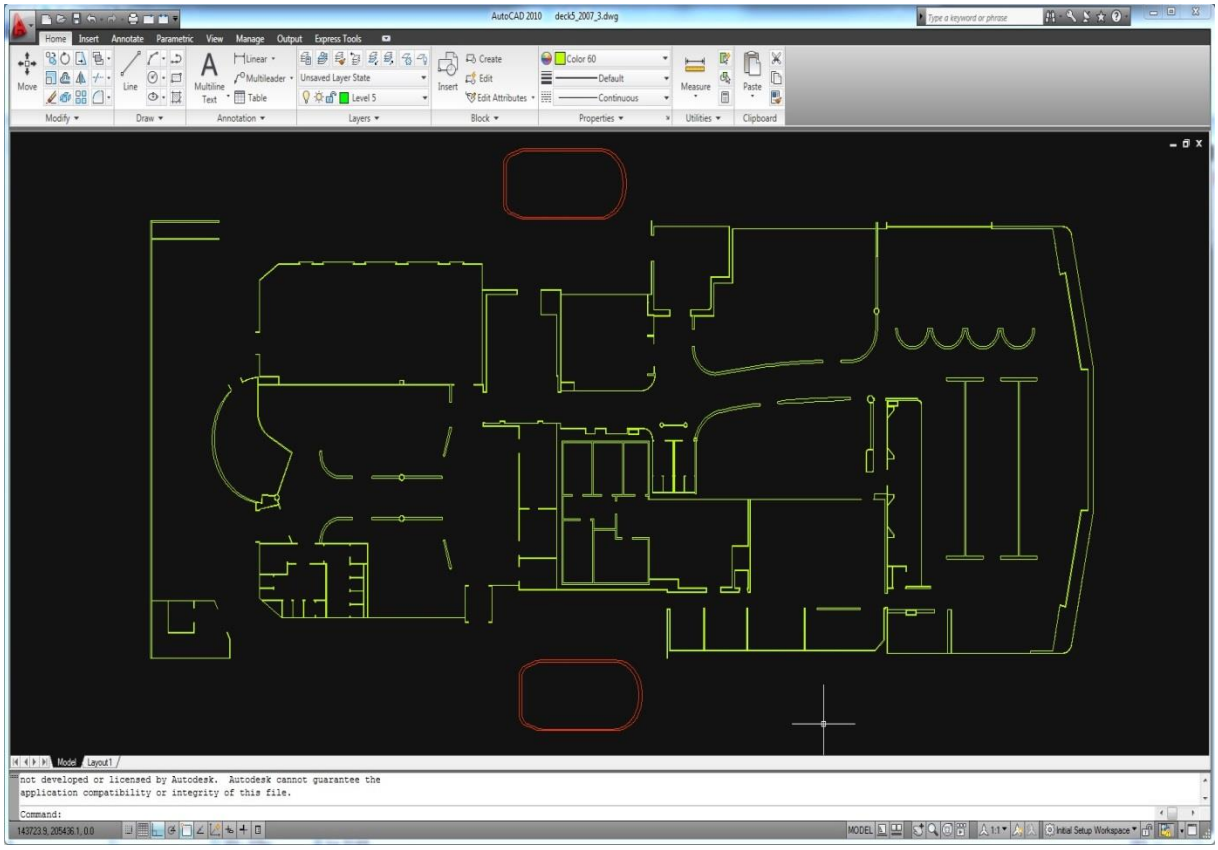
2.1.3 Χρήση

Εμείς δουλέψαμε με την γενική έκδοση AutoCAD 2010. Σχεδιάσαμε τρία καταστρώματα ενός επιβατηγού-οχηματαγωγού πλοίου τα οποία χρησιμοποιήσαμε στην προηγμένη μέθοδο ανάλυσης της διαδικασίας εκκένωσης. Στην Εικόνα 2.1 φαίνεται το σχέδιο του καταστρώματος γέφυρας (bridge deck ή deck7 όπως έχουμε ονοματίσει το παραχθέντα τριδιάστατο αντικείμενο στο λογισμικό Unity3D). Θα εξηγήσουμε τη διαδικασία μετατροπής σε τριδιάστατο αντικείμενο στη παράγραφο §2.2.3.

Στις παρακάτω εικόνες 2.2 και 2.3 βλέπουμε τα σχέδια των καταστρωμάτων επιβατών (passenger deck – deck6) και επιβίβασης (embarkation deck – deck5) αντίστοιχα.



Εικόνα 2.2 : Κατάστρωμα επιβατών – deck6



Εικόνα 2.3 : Κατάστρωμα επιβίβασης – deck5

Δουλέψαμε κυρίως με αρχεία DWG, που είναι τα συνηθέστερα στο AutoCAD. Τα αρχεία αυτά θα εισάγουμε στην συνέχεια στο λογισμικό 3D Studio Max για να φτιάξουμε τριδιάστατα μοντέλα της τοπολογίας των καταστρωμάτων αλλά και τα μοντέλα των πατωμάτων των καταστρωμάτων. Σε αυτά τα μοντέλα θα κινούνται ύστερα οι Agents στο λογισμικό Unity3D για την προσομοίωση της εκκένωσης.

Οι διαστάσεις των τριών καταστρωμάτων είναι :

	Μήκος	**Πλάτος**	**Ύψος**
deck5	72.438m	27.147m	3.200m
deck6	71.084m	26.740m	2.700m
deck7	70.316m	32.503m	2.750m

Πίνακας 2.1 : Διαστάσεις καταστρωμάτων

Η ακολουθία των εργασιών, δηλαδή ότι ξεκινάμε από κάποιο σχέδιο AutoCAD χωρίς περιττές γραμμές, έπειτα φτιάχνουμε το ανάλογο τριδιάστατο μοντέλο και τέλος πραγματοποιούμε την εξομοίωση σε κάποια 3D rendering μηχανή εμπνεύστηκε από εδώ : [6].

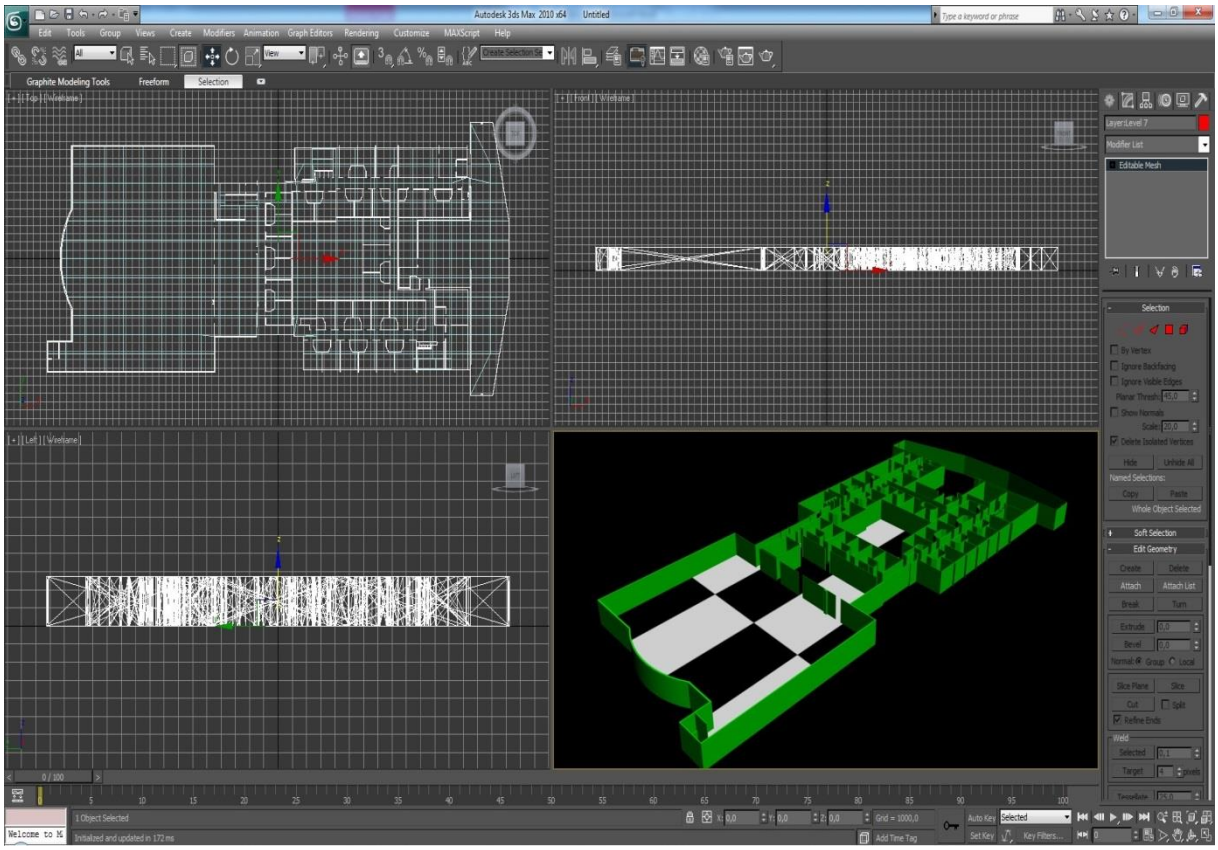
2.2 3D Studio Max

2.2.1 Γενικά

Για την κατασκευή των τριδιάστατων μοντέλων των καταστρωμάτων και των πατωμάτων τους, θα χρησιμοποιήσουμε το 3D Studio Max της εταιρείας Autodesk [5]. Το 3D Studio ειδικεύεται στη δημιουργία τριδιάστατων αντικειμένων, χαρακτήρων, περιβαλλόντων και γενικά λεπτομερών τριδιάστατων σκηνών συμπεριλαμβανομένων και των φωτεινών πηγών, καμερών και φωτοσκιάσεών τους. Ακόμα, μπορούμε να εφαρμόζουμε υφές (textures) στα αντικείμενα και να προσδιορίζουμε τον τρόπο κίνησης τους (animation). Το πιο σημαντικό χαρακτηριστικό του 3D Studio είναι η δημιουργία πολύπλοκων αντικειμένων και η εφαρμογή ποικίλων μετασχηματισμών σε αυτά.

2.2.2 Τρόπος λειτουργίας

Στην Εικόνα 2.4 φαίνεται η διεπαφή του 3D Studio Max και συγκεκριμένα τα μοντέλα deck7 και deck7_ground :



Εικόνα 2.4 : Μοντέλα deck7 και deck7_ground

Βλέπουμε την κάτοψη, πλάγια όψη, πρόσοψη και την προοπτική όψη του υπό κατασκευή αντικειμένου, έχοντας έτσι καλύτερη τριδιάστατη αντίληψη της σκηνής. Στο πάνω μέρος, βρίσκονται τα εργαλεία που χρησιμοποιούνται πιο συχνά, όπως επιλογή αντικειμένων ή ομάδα από αυτά, μετασχηματισμοί θέσης, περιστροφής, μεγέθους, αναστροφής και αντιστροφής, διορθωτές (editors) ονόματος, επιπέδων (layers), καμπύλων (curves) και υλικών (material). Στο κάτω μέρος υπάρχει η κλίμακα των καρέ (frames) με την οποία αλλάζουμε τα χαρακτηριστικά του αντικειμένου σε κάθε καρέ, δίνοντας του έτσι την ψευδαίσθηση της κίνησης. Στο δεξί μέρος βρίσκονται οι πιο σημαντικές λειτουργίες δημιουργίας και μορφοποίησης των αντικειμένων. Παραδείγματος χάρη, μπορούμε να φτιάξουμε πολλά στοιχειώδη τριδιάστατα αντικείμενα (σφαίρες, κύβοι, κύλινδροι, πυραμίδες, σπείρες, κώνοι, σωλήνες), εκτεταμένα αντικείμενα (άξονες τροχών, άτρακτοι, κύβοι και κύλινδροι στρογγυλεμένων άκρων, πρίσματα, κάψουλες, δεσμοί σπειρών, πολύεδρα) και να εφάρμοσουμε πράξεις συνόλων σε αυτά (ένωση, τομή, αφαίρεση A-B και B-A) και μετασχηματισμούς για να δημιουργήσουμε καινούργια πολυπλοκότερα αντικείμενα. Οι μετασχηματισμοί εφάρμοζονται σε ολόκληρα τα αντικείμενα ή μόνο σε μέρη τους και είναι τριών ειδών :

- Πλέγματος (mesh).
- Καμπύλης (spline).
- Πολυγώνων (polygon).

Οι πιο σημαντικοί είναι : λύγισης (bend), κλίσης μη κάθετης επιφάνειας (bevel), διατομής (cross section), προεξοχών (extrude), κάμψης (flex), τήξης (melt), καθρεφτισμού (mirror), θορύβου (noise), κανονικοποίησης (normalize), προβολής (projection), ρυτίδων (ripple), κελύφους (shell), ασυμμετρίας (skew), μορφοποίησης του δέρματος (skin morph), τεμαχισμού (slice), σφαιρικοποίησης (spherify), συμπίεσης (squeeze), έκτασης (stretch), υποδιαίρεσης (subdivide), αντικατάστασης (substitute), σάρωσης (sweep), συμμετρίας (symmetry), σταδιακής μείωσης (taper), διακόσμησης με μωσαϊκό (tessellate), αφαίρεση ανεπιθύμητων ακρών (trim), συστροφής (twist), κυματισμού (wave). Μπορούμε να εφαρμόσουμε μεγάλο πλήθος μετασχηματισμών σε ένα αντικείμενο και αυτοί τοποθετούνται με μορφή στοίβας, στην οποία μπορούμε να αλλάζουμε και τη σειρά τους.

Επίσης, στο δεξί μέρος βρίσκονται και οι λειτουργίες επιλογής χρωμάτων των αντικειμένων, δημιουργίας συστημάτων σωματιδίων (particle system) για να εξομοιώσουμε π.χ. ένα σύστημα αντικειμένων φωτιάς-καπνού, δυναμικών αντικειμένων (dynamic object) που κινούνται σαν να επιδρούν πάνω τους οι φυσικές δυνάμεις του πραγματικού κόσμου (π.χ. η βαρύτητα), φωτεινών πηγών και καμερών αλλά και τοποθέτησης των σκιών τους. Αξίζει να αναφερθούμε και στη δυνατότητα κατασκευής λεπτομερέστατων διπόδων (biped) χαρακτήρων. Αρχικά φτιάχνουμε το πλέγμα το οποίο αποτελείται απο αυτόνομα μέρη που αντιστοιχούν στα διαφορετικά μέλη του σώματος, έτσι ώστε να είναι δυνατή η μετακίνηση π.χ. μόνο του ενός χεριού αλλά και η γενικότερη κίνηση όλου του σώματος να είναι πιο φυσική. Έπειτα, μπορούμε να τοποθετήσουμε διάφορα materials τα οποίο προσδιορίζουν το πώς η επιφάνεια αντανακλά το φως της φωτεινής πηγή της σκηνής. Συνήθως, τα material έχουν και κάποιο εφαρμοσμένο texture για να “ντύνεται” με αυτό ο χαρακτήρας.

Το 3D Studio μπορεί και εξάγει τη σκηνή που έχουμε φτιάξει σε διαφορετικού είδους αρχεία, οπότε μπορούμε ύστερα να την εισάγουμε και χρησιμοποιήσουμε σε άλλα προγράμματα όπως το Quest3D. Μία ομοιότητα με το Quest3D είναι και η δυνατότητα εξαγωγής της σκηνής σε εκτελέσιμο αρχείο το οποίο μπορούμε να τρέξουμε για να δούμε τη σκηνή σε κάποιον ηλεκτρονικό υπολογιστή που δεν έχει εγκατεστημένο το 3D Studio Max. Τέλος, το 3D Studio, όπως και άλλα προγράμματα της εταιρείας Autodesk, μπορεί και χρησιμοποιεί το σύστημα Autodesk Backburner. Πρόκειται για ένα *σύστημα κατανεμημένης ουράς* (Distributed Queueing System – DQS) στο οποίο συμμετέχουν υπολογιστές διαφόρων λειτουργικών συστημάτων (Linux, Windows, Irix) και εργάζονται συλλογικά στο ίδιο δίκτυο. Επειδή η πολυπλοκότητα μιας σκηνής στο 3D Studio μπορεί είναι πολύ μεγάλη λόγω των πολλών αντικειμένων και λεπτομερειών που μπορεί να περιέχει, το rendering της σκηνής από ένα μόνο υπολογιστή μπορεί να χρειαστεί πολύ ώρα. Έτσι, χρησιμοποιώντας το DQS, πολλοί υπολογιστές στο ίδιο δίκτυο συνδράμουν στην επίτευξη μιας κοινής εργασίας, π.χ. το rendering, σε λιγότερο χρόνο από όσο θα έκανε ένας μόνο υπολογιστής. Το DQS απαρτίζεται από :

- Έναν υπολογιστή (ονομάζεται Render Client) του οποίου κάποια εφαρμογή στέλνει τη ζητούμενη εργασία (rendering) στους render nodes.
- Render Nodes είναι ένας ή περισσότεροι υπολογιστές που αναλαμβάνουν την πραγματοποίηση της εργασίας.
- Ένας υπολογιστής (Backburner Manager) που είναι υπεύθυνος για την διανομή και διαχείριση των εργασιών που εκτελούν οι Render Nodes.
- Ένας ή περισσότεροι υπολογιστές (Backburner Monitor) οι οποίοι παρακολουθούν τις εκτελούμενες εργασίες στους Render Nodes.

Τα τέσσερα μέρη επικοινωνούν μεταξύ τους με το πρωτόκολο TCP/IP.

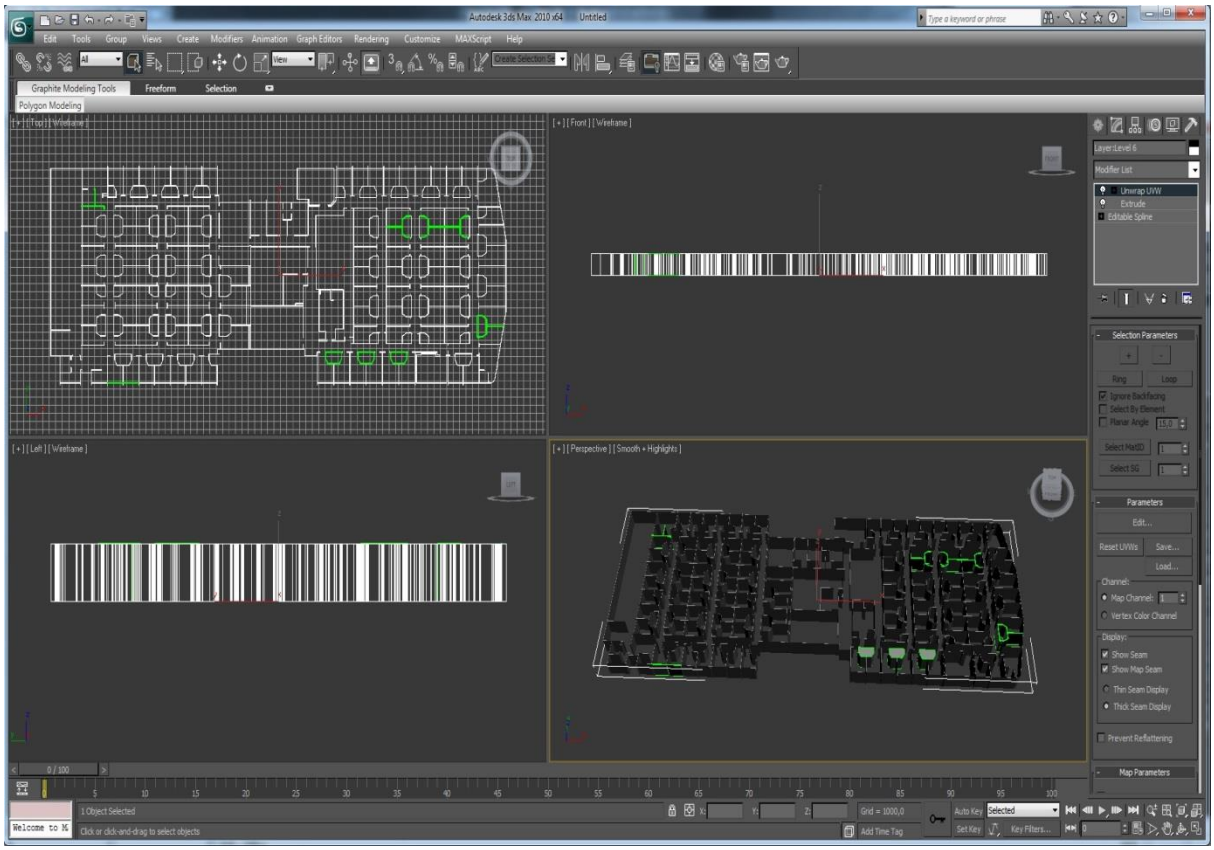
2.2.3 Χρήση

Θα περιγράψουμε σύντομα τα βήματα της διαδικασία δημιουργίας των μοντέλων των καταστρωμάτων στο 3D Studio Max 2010 από το διδιάστατο σχέδιο του AutoCAD :

- Δημιουργία σχεδίου στο AutoCAD και αποθήκευση του σε αρχείο DWG.
- Στο 3D Studio κάνουμε import το αρχείο DWG. Στο παράθυρο διαλόγου που ανοίγει προσέχουμε να είναι επιλεγμένες όλες οι επιλογές στον τομέα Geometry Options και κυρίως η Cap closed spline, ώστε όταν κάνουμε στη συνέχεια Extrude, οι τοίχοι που θα δημιουργηθούν θα κλείσουν από τη πάνω και κάτω μεριά τους.
- Επιλέγουμε το διδιάστατο μοντέλο που είναι τώρα ένα Editable Spline και του εφαρμόζουμε τον μετασχηματισμό Extrude οπότε οι γραμμές θα ανυψωθούν και θα μετατραπούν σε τριδιάστατους τοίχους.
- Κάνουμε Collapse τον Extrude (το μοντέλο μετατρέπεται σε Editable Mesh) και προσθέτουμε τον μετασχηματισμό Unwrap UVW ο οποίος προετοιμάζει τα faces για

να δεχθούν textures αλλά και δείχνει σφάλματα (πράσινο χρώμα) του σχεδίου που έχουν προκύψει από πολύγωνα που δεν έχουν κλείσει σωστά (Εικόνα 2.5). Επίσης, ενεργοποιούμε την επιλογή Backface Cull στα Object Properties (δεξί click) του μοντέλου στο tab General. Έτσι, θα φανεί αν κάποια faces έχουν λάθος προσανατολισμό (normal) οπότε θα πρέπει να επιλεχθούν και να γίνουν Flip.

- Κάνουμε Collapse τον Unwrap UVW και προσθέτουμε τον μετασχηματισμό UVW Map. Στις παραμέτρους επιλέγουμε Planar Mapping.
- Κάνουμε Collapse τον UVW Map. Προσέχουμε το μοντέλο να βρίσκεται στη θέση : (0, 0, 0) και όλα τα μέρη του να βρίσκονται στο ίδιο layer και να έχουν το ίδιο material.
- Με επιλεγμένο το μοντέλο, κάνουμε Export Selected και επιλέγουμε αρχείο μορφής FBX.
- Εισάγουμε (import) το αρχείο FBX στο λογισμικό Unity3D όπου είναι πλέον έτοιμο να το χρησιμοποιήσουμε στη σκηνή μας. Προσέχουμε στο Rig του μοντέλου ο Scale Factor να είναι 0.001 ώστε τα mm στο 3D Studio να αντιστοιχούν σε m στο Unity3D.

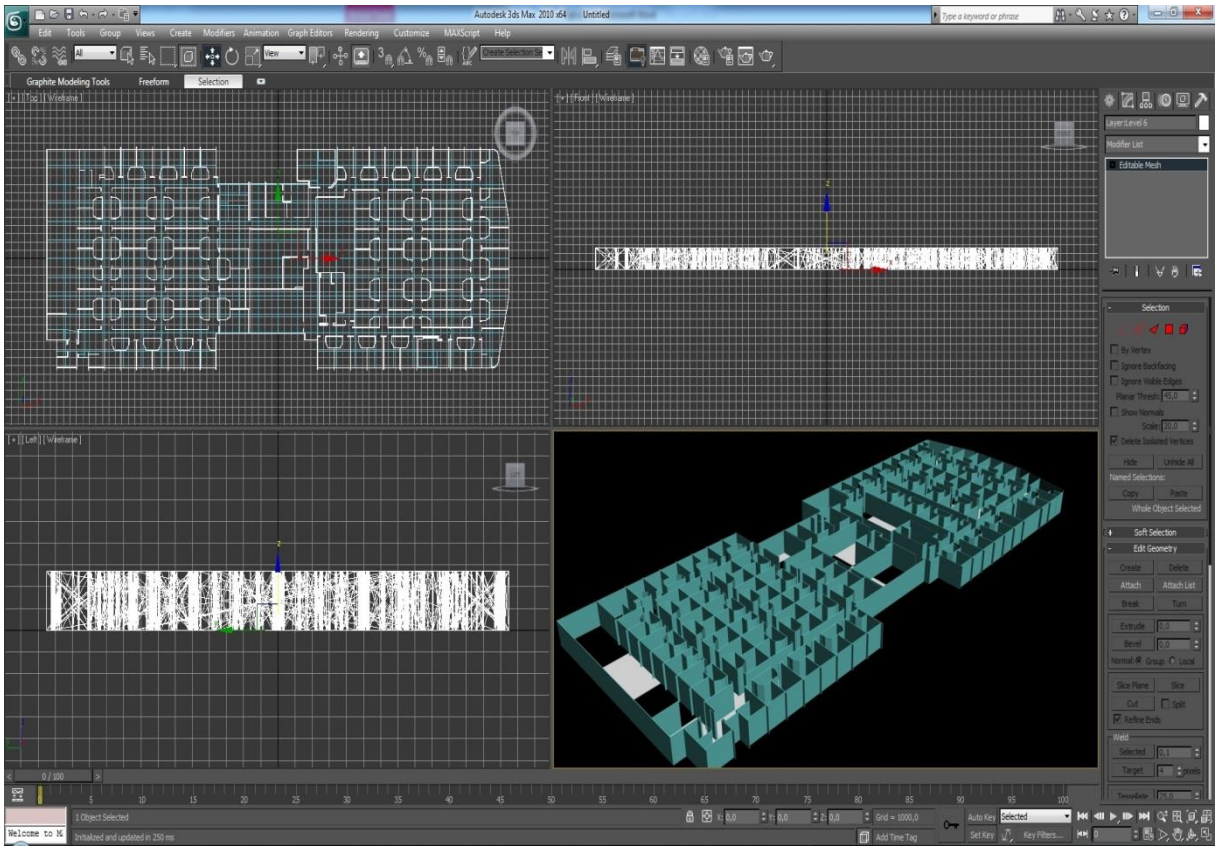


Εικόνα 2.5 : Έλεγχος σφαλμάτων του σχεδίου με τον μετασχηματισμό Unwrap UVW

Τα ύψη που θέτουμε στον μετασχηματισμό Extrude είναι 3200mm για το deck5, 2700mm για το deck6 και 2750mm για το deck7 (Πίνακας 1).

Για τα μοντέλα των πατωμάτων εκτελούμε τα ίδια βήματα εκτός απ’τον μετασχηματισμό Extrude αφού δεν έχουν ύψος ως απλά επίπεδα. Ξεκινήσαμε από επίπεδα τα οποία μετακινώντας τα vertices τους φέραμε στην επιθυμητή μορφή. Ακόμα, δημιουργήσαμε τρύπες ώστε να συνδέονται τα decks με τις σκάλες

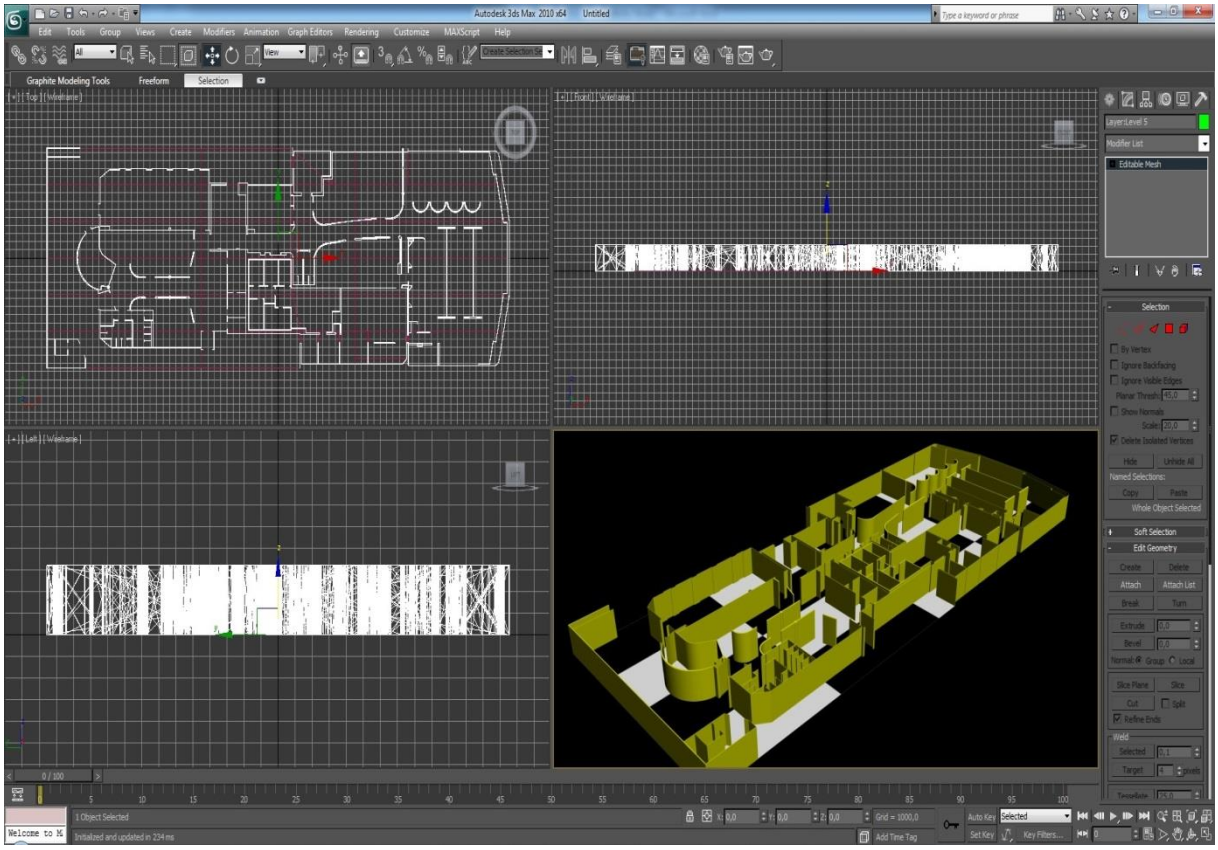
Για χάρη πληρότητας παραθέτουμε και τις εικόνες των μοντέλων των υπολοίπων καταστρωμάτων και των πατωμάτων τους.



Εικόνα 2.6 : Μοντέλα deck6 και deck6_ground

Στα μοντέλα των πατωμάτων έχουμε τοποθετήσει ένα material με ένα texture μορφής σκακιέρας (Diffuse : Checker στον Material Editor). Αυτό συνηθίζεται να γίνεται ως τεστ για να διαπιστώνουμε αν το UVW Mapping είναι της μορφής που θέλουμε. Εδώ είναι ένα απλό Planar Mapping.

Προφανώς, μετά στο Unity3D μπορούμε να αλλάξουμε το checker με οποιαδήποτε άλλο texture θέλουμε για να επιτύχουμε το επιθυμητό οπτικό αποτέλεσμα.



Εικόνα 2.7 : Μοντέλα deck5 και deck5_ground

2.3 Unity3D

2.3.1 Γενικά

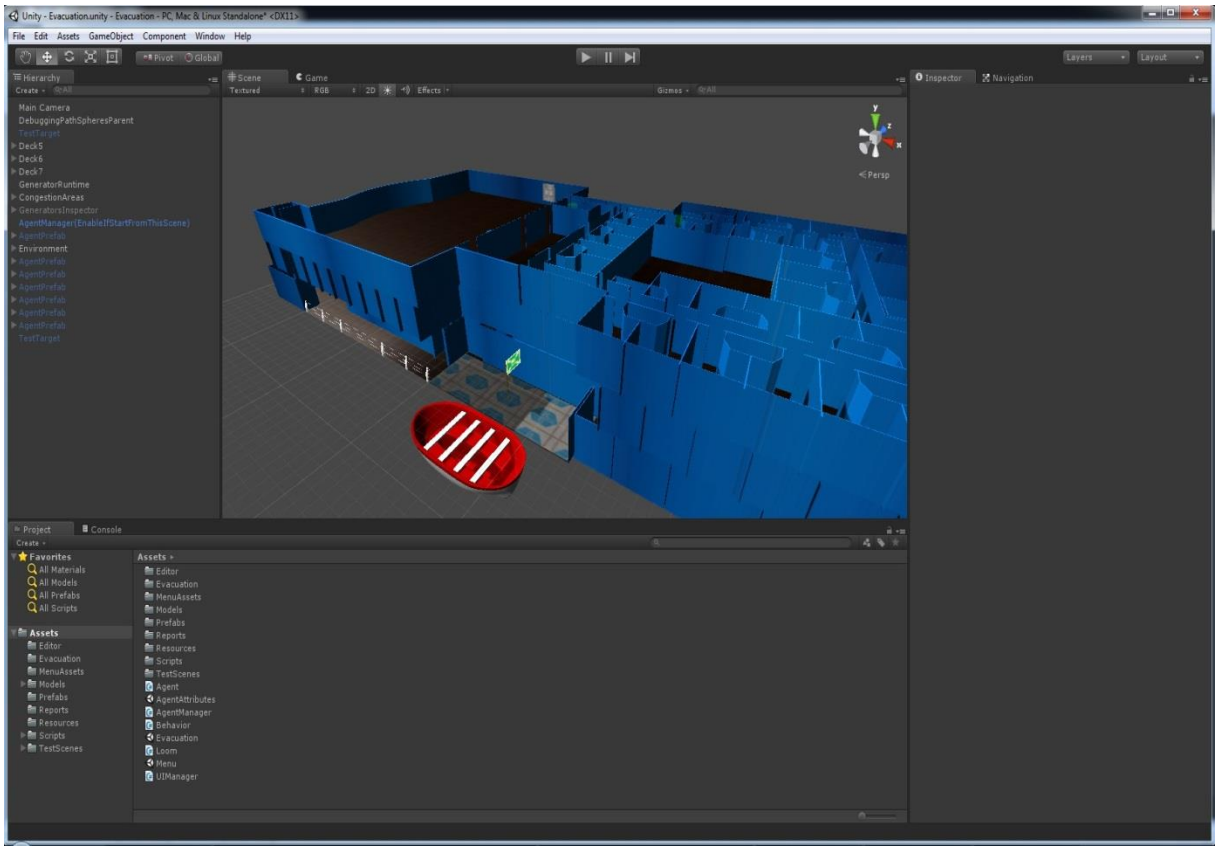
Το λογισμικό Unity3D (ή Unity για συντομία) έχει αναπτυχθεί από την εταιρεία Unity Technologies και περιλαμβάνει μια πλήρη τριδιάστατη rendering μηχανή καθώς και ένα ενσωματωμένο αναπτυξιακό περιβάλλον (Integrated Development Environment ή IDE) με το οποίο ο χρήστης μπορεί να δημιουργεί και να εξελίσσει διδιάστατες ή και τρισδιάστατες εφαρμογές [7]. Για την υλοποίηση των δυνατοτήτων των εφαρμογών χρησιμοποιούνται Scripts που αναπτύσσονται, από προεπιλογή, στο πολλαπλών πλατφόρμων (cross-platform) IDE : MonoDevelop, αλλά μπορούν να επιλεχθούν και άλλα IDEs όπως το Microsoft Visual Studio. Υποστηρίζονται οι εξής γλώσσες : Boo, C# και JavaScript. Τα μεγαλύτερα πλεονεκτήματα του Unity θεωρούνται :

- Ο μικρός χρόνος που απαιτείται για την εκμάθηση των βασικών λειτουργιών.
- Η ευκολία ανάπτυξης πολύπλοκων και καλαίσθητων τρισδιάστατων σκηνών.
- Η ύπαρξη πολλών έτοιμων συστημάτων όπως : Physics System (το γνωστό PhysX), Collision System, Graphical User Interface, Terrain Creator & Editor, Particle System, NavMesh, Lighting & Shadows, Animation, κτλ, τα οποία μπορούν να ενσωματωθούν ομαλά στο κυρίως πρόγραμμα.
- Η χρήση Scripts για την ανάπτυξη και παραμετροποίηση των εφαρμογών.
- Η δυνατότητα εξαγωγής της εφαρμογής σε 21 διαφορετικές πλατφόρμες με κυριότερες τις εξής : Pc, Mac, Linux, Web Player, Android, iOS, Windows Phone 8, BlackBerry 10, PS4, Xbox 360, Wii U και Oculus Rift, χωρίς ο χρήστης να χρειάζεται να ξέρει τα ξεχωριστά βασικά IDEs της κάθε πλατφόρμας π.χ. DirectX, OpenGL, OpenGL ES κτλ.

Αυτή τη στιγμή έχει φτάσει στην έκδοση 5.0.1. Υπάρχουν δύο εκδόσεις του Unity, η personal που διατίθεται χωρίς κάποιο κόστος και η professional που έχει περισσότερα features όπως η ανάπτυξη εφαρμογών σε cloud και team licence αλλά με κόστος.

2.3.2 Τρόπος λειτουργίας

Στην Εικόνα 2.8 φαίνεται η διεπαφή του Unity, όταν βρίσκεται σε κατάσταση επεξεργασίας (edit mode) :



Εικόνα 2.8 : Διεπαφή του Unity3D (edit mode)

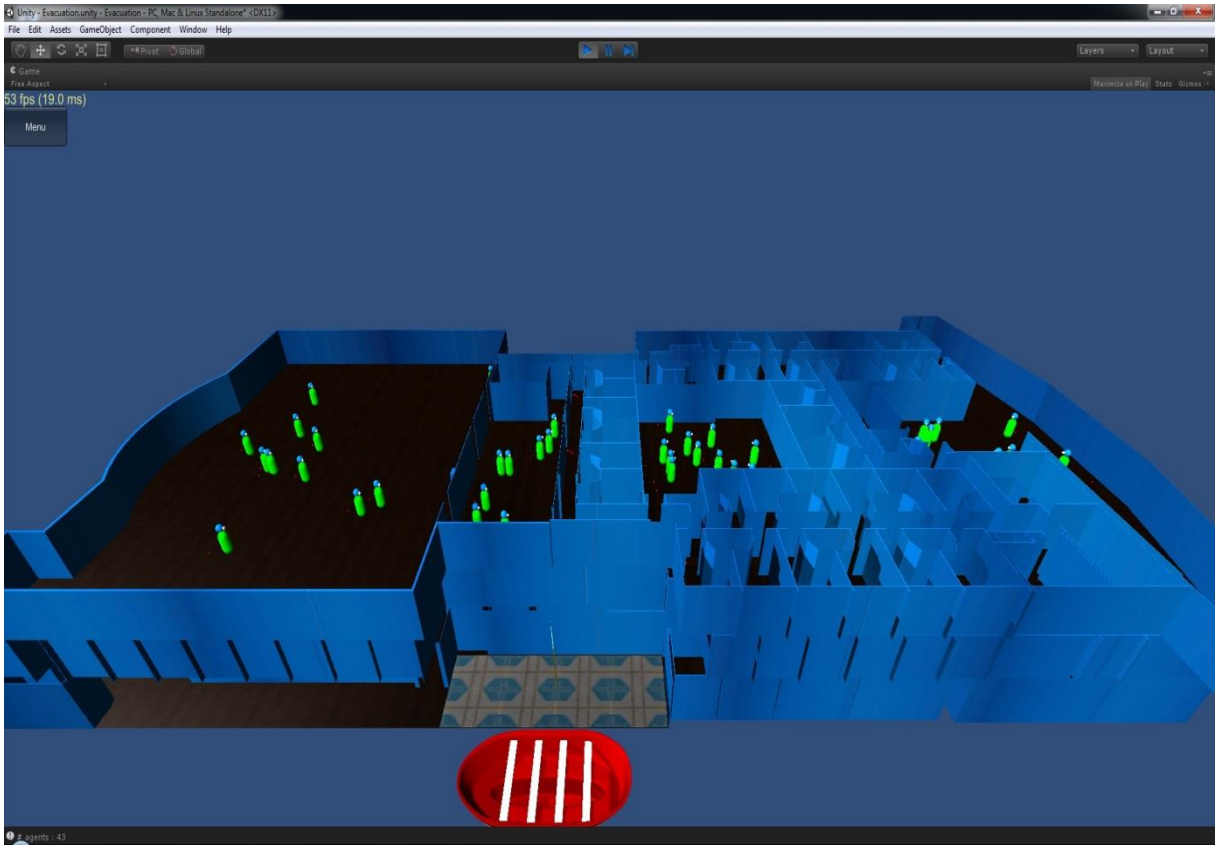
Βλέπουμε ότι ουσιαστικά χωρίζεται σε τέσσερα παράθυρα. Στο κάτω (Project) υπάρχουν όλα τα αρχεία (assets) που ανήκουν στην εφαρμογή που αναπτύσσουμε. Παραδείγματα assets είναι τα μοντέλα, εικόνες, ήχοι και μουσικές, οι σκηνές που αποτελούν την εφαρμογή, τα διάφορα Scripts που έχουμε αναπτύξει κτλ.

Από την συλλογή όλων των assets του project βλέπουμε ποια ακριβώς χρησιμοποιούμε στην εκάστοτε σκηνή στο αριστερό παράθυρο, το Hierarchy. Δηλαδή στο Hierarchy υπάρχουν όλα τα αντικείμενα από τα οποία αποτελείται η συγκεκριμένη σκηνή. Αυτά που φαίνονται με σκούρο μπλε χρώμα είναι αντικείμενα που έχουν προκύψει από prefabs. Prefabs είναι σαν καλούπια αντικειμένων, από τα οποία αποκτούμε όμοια αντικείμενα με κοινές ιδιότητες. Βέβαια, μπορούμε να δημιουργήσουμε ένα αντικείμενο από κάποιο prefab και έπειτα να του αλλάξουμε τις ιδιότητές του.

Αυτό γίνεται στο δεξιό παράθυρο, το Inspector. Έχοντας επιλέξει την κεντρική κάμερα (Main Camera) στο παράθυρο Hierarchy, βλέπουμε τις ιδιότητές της στο Inspector. Έτσι, εδώ

μπορούμε να αλλάξουμε τιμή σε κάποια ιδιότητα, π.χ. τις μοίρες του πεδίου όρασης (Field Of View, FOV). Ομοίως και για όλα τα υπόλοιπα αντικείμενα στο Hierarchy. Ας τονίσουμε εδώ ότι στον Inspector διακρίνονται μόνο οι δημόσιες (public) μεταβλητές των Scripts μας.

Στο κεντρικό παράθυρο (Scene) γίνεται το rendering της σκηνής. Εδώ θα φαίνονται και οι αλλαγές που κάνουμε στις χωρικές ιδιότητες των αντικειμένων, π.χ. μετακινώντας τα στα χώρο ή μεγενθύνοντας μια διάστασή τους. Όταν πατήσουμε το button Play στο πάνω μέρος της οθόνης τότε το παράθυρο Scene αντικαθίσταται απ'το παράθυρο Game όπου "τρέχει" κανονικά η εφαρμογή μας. Το button Pause την σταματάει προσωρινά μέχρι να πατήσουμε ξανά το Play και το δεξιότερο button προχωράει τον χρόνο κατά ένα frame. Στην Εικόνα 2.9 έχουμε πατήσει το Play και το Unity βρίσκεται σε κατάστασης αναπαραγωγής (play mode). Διακρίνονται και οι επιβάτες που έχουν δημιουργηθεί και περπατάνε στο κατάστρωμα.



Εικόνα 2.9 : Διεπαφή του Unity3D (play mode)

Γενικά, η δημιουργία εφαρμογών στο Unity βασίζεται στις δύο κεντρικές έννοιες Game Object και Component. Game Object είναι το κάθε αντικείμενο της σκηνής μας. Μερικά Game Objects έχουν φυσική υπόσταση, δηλαδή γραφική αναπάρασταση (mesh) στη σκηνή οπότε είναι και ορατά, όπως το μοντέλο ενός καταστρώματος, ενώ άλλα Game Objects δεν αναπαρίστανται γραφικά στη σκηνή, καθώς επιτελούν κάποια λειτουργία όπου δεν είναι απαραίτητη η οπτικοποίησή τους, π.χ. το Game Object που είναι υπεύθυνο για τις κινήσεις όλων των άλλων Game Objects που αντιπροσωπεύουν τους κινούμενους χαρακτήρες (Agents). Επίσης, κάποιο Game Object μπορεί να γίνει "παιδί" κάποιου άλλου για να κληρονομεί την μεταφορά, περιστροφή και κλίμακα του πατρικού αντικειμένου.

Η δεύτερη βασική έννοια, το Component, δίνει στα Game Objects τις ιδιότητες που θέλουμε να έχουν. Όλα τα Game Objects έχουν τουλάχιστον το Component Transform το οποίο αποτελείται από τρία τριδιάστατα διανύσματα (Vector3) : Θέση (Position), Περιστροφή (Rotation) και Κλίμακα (Scale). Έτσι, αν θέλουμε κάποιο Game Object να λειτουργεί ως μια φωτεινή πηγή στη σκηνή, του προσθέτουμε ένα Light Component και δίνουμε έπειτα τιμές στις διάφορες παραμέτρους, π.χ. το είδος της φωτεινή πηγής να είναι Directional. Ένα άλλο παράδειγμα είναι αν θέλουμε το Game Object μας να έχει πιο φυσική κίνηση και να αντιδρά σε συγκρούσεις με άλλα Game Objects που συναντά, του προσθέτουμε δύο Components : Rigidbody και κάποιας μορφής (π.χ. Capsule) Collider.

Όπως είπαμε και στη §2.2.3, ένα από τα μεγαλύτερα προτερήματα του Unity είναι και η δυνατότητα που έχουμε να δημιουργούμε εμείς τις λειτουργίες που θα θέλαμε για κάποιο Game Object αν αυτές δεν υπάρχουν ήδη έτοιμες. Δηλαδή, μπορούμε να προγραμματίσουμε σε κάποια γλώσσα αυτές τις λειτουργίες και να τις αποθηκεύσουμε σε κάποιο Script. Έπειτα, προσθέτουμε αυτό το Script ως Component στο Game Object που θέλουμε, με αποτέλεσμα αυτό να αποκτά τις ζητούμενες λειτουργίες.

2.3.3 Χρήση

Η διαδικασία εισαγωγής στο Unity των τριδιάστατων μοντέλων των καταστρωμάτων και των πατωμάτων τους παρουσιάστηκε στην §2.2.3. Η χρήση τους μέσα στο περιβάλλον του Unity αλλά και η περιγραφή για το πώς μοντελοποιήσαμε ένα πλήθος επιβατών και προσομοιώσαμε τη διαδικασία εκκένωσης στο Unity είναι τα κεντρικά θέματα του επόμενου κεφαλαίου.

Κεφάλαιο 3

Μοντέλο μεθόδου

Σε αυτό το κεφάλαιο θα ασχοληθούμε αρχικά με την προηγμένη μέθοδο ανάλυσης εκκένωσης αναλύοντας το σκοπό της, τις υποθέσεις της και τον τρόπο υπολογισμού του συνολικού χρόνου εκκένωσης.

Στη συνέχεια θα αναπτύξουμε το πώς μοντελοποιήσαμε τους επιβάτες του πλοίου ως αυτόνομες, κινούμενες και με τεχνητή νοημοσύνη οντότητες. Για συντομία, θα τις αποκαλούμε Agents.

Στο τέλος, θα περιγράψουμε τη συσχέτιση των φυσικών νόμων κίνησης και του κώδικά μας.

3.1 Προηγμένη μέθοδος ανάλυσης εκκένωσης

Όπως είδαμε και στην §1.1, η Maritime Safety Committee (MSC) του International Maritime Organization (IMO) στο «Interim guidelines for evacuation analysis for new and existing passenger ships» προτείνει δύο διαφορετικές μεθόδους για την ανάλυση της διαδικασίας εκκένωσης. Η πρώτη (απλοποιημένη) θεωρεί το πλήθος των επιβατών ως ρευστό την κίνηση του οποίου ορίζουν οι εξισώσεις Navier-Stokes. Η ακρίβεια αυτής της μεθόδου μειώνεται όσο αυξάνεται η πολυπλοκότητα του πλοίου, δηλαδή όταν αυξάνεται ο αριθμός των διαφορετικών τύπων επιβατών και χώρων διαμονής, ο αριθμός των καταστρωμάτων και των σκαλών. Για αυτό το λόγο αλλά και επειδή είναι εύκολη στην ανάπτυξη, χρησιμοποιείται κυρίως μόνο στα αρχικά στάδια της σχεδίασης ενός νέου πλοίου για να δώσει μια προσέγγιση της αναμενόμενης απόδοσης της εκκένωσης.

Η προηγμένη μέθοδος συνήθως ταυτίζεται με προσομοίωση της εκκένωσης σε ηλεκτρονικό υπολογιστή, έχοντας όμως μια πιο μικροσκοπική αντίληψη των ανθρώπων (επιβάτες και πλήρωμα) [2]. Δηλαδή, ο κάθε άνθρωπος έχει θεωρηθεί ως μια ξεχωριστή οντότητα με πληροφορίες για την τοπολογία του μοντέλου του πλοίου και εξετάζεται η αλληλεπίδραση τόσο μεταξύ των οντοτήτων όσο και με το μοντελοποιημένο πλοίο. Θα την αναλύσουμε περισσότερο στις αμέσως επόμενες παραγράφους.

Ας τονίσουμε εδώ ότι οι απαιτήσεις της MSC δίνονται με την μορφή guidelines, δηλαδή είναι πιο πολύ κατευθυντήριες γραμμές παρά επιβαλλόμενοι κανονισμοί.

3.1.1 Σκοπός προηγμένης μεθόδου

Ο σκοπός της προηγμένης μεθόδου είναι να :

- Αναγνωρίσει και εξαλείψει, όσο είναι πρακτικά δυνατόν, τις περιοχές αυξημένης κυκλοφοριακής συμφόρησης που μπορεί να αναπτυχθούν κατά τη διάρκεια εκκένωσης.
- Επιδείξει ότι οι μηχανισμοί διαφυγής είναι ικανοποιητικά ευέλικτοι ώστε να καλύψουν και την πιθανότητα συγκεκριμένοι οδοί διαφυγής, σταθμοί συγκέντρωσης, σταθμοί επιβίβασης ή σωσίβιες λέμβη να μην είναι διαθέσιμοι λόγω κάποιου ατυχήματος.

3.1.2 Υποθέσεις προηγμένης μεθόδου

Έχουν γίνει οι εξής υποθέσεις :

- Οι επιβάτες και το πλήρωμα αναπαριστώνται ως μοναδικές οντότητες με συγκεκριμένες ιδιότητες και χρόνους απόκρισης.
- Οι επιβάτες και το πλήρωμα θα χρησιμοποιήσουν τις κύριες οδούς διαφυγής.
- Οι μηχανισμοί διαφυγής είναι 100% διαθέσιμοι.
- Το πλήρωμα θα βρίσκεται στις σωστές θέσεις για να βοηθήσουν τους επιβάτες.
- Οι επιβάτες ακολουθούν το σύστημα σήμανσης και τις οδηγίες του πληρώματος για τη δημιουργία της πορείας τους προς τους σταθμούς συγκεντρώσεως.
- Ο καπνός, η θερμότητα και τα τοξικά προϊόντα πυρκαϊάς δεν επηρεάζουν την απόδοση του πληρώματος και των επιβατών.
- Δεν υπάρχει συμπεριφορά οικογενειακής ομάδας.
- Η κίνηση και η κλίση του πλοίου δεν υπολογίζονται.

3.1.3 Τρόπος υπολογισμού συνολικού χρόνου εκκένωσης

Ο συνολικός χρόνος εκκένωσης υπολογίζεται απ’ τον τύπο :

$$T_{o\lambda} = 1.25 * T + \frac{2}{3} * (E + L) \tag{3.1}$$

Όπου **T** ο χρόνος μετακίνησης (Travel time), το 1.25 είναι ο παράγοντας ασφάλειας (safety factor), **E** είναι ο χρόνος επιβίβασης (Embarkation time) και **L** ο χρόνος καθέλκυσης (Launch time). Σύμφωνα με τις οδηγίες, θα πρέπει να ισχύει :

$$T_{o\lambda} \leq n \iff 1.25 * T + \frac{2}{3} * (E + L) \leq n \tag{3.2}$$

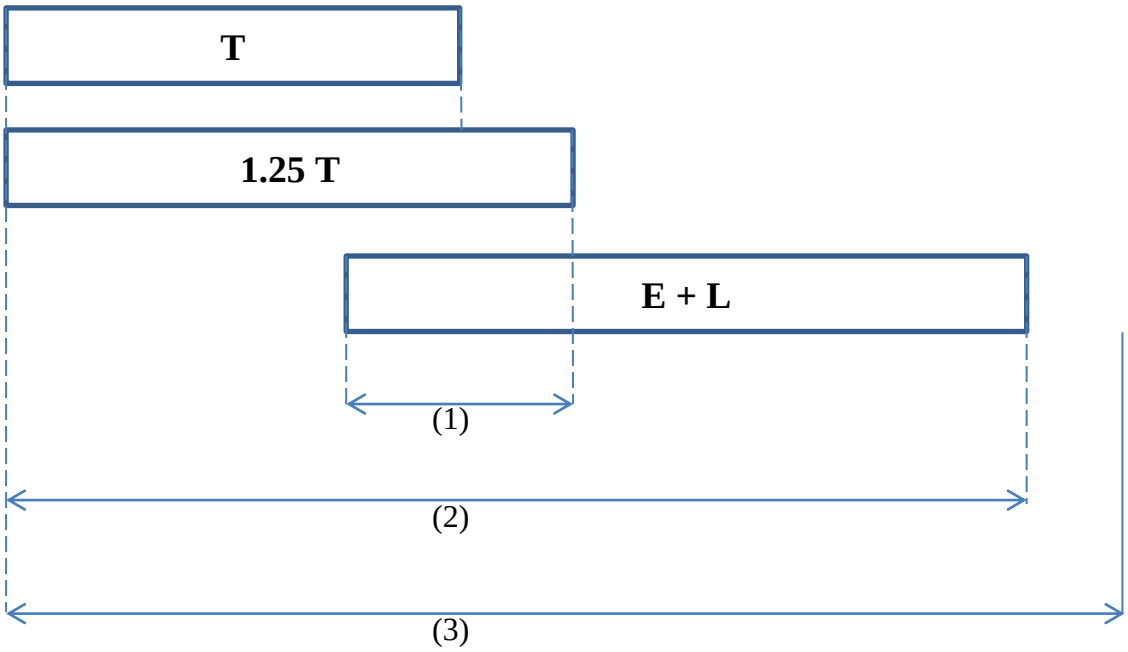
Όπου **n** είναι ο μέγιστος επιτρεπτός χρόνος εκκένωσης και προκύπτει ως :

$$n = \begin{cases} 60\ min,\ \ για\ Ro - Ro\ επιβατηγά\ πλοία \\ 60\ min, για\ επιβατηγά\ πλοία\ (εκτός\ Ro - Ro) με\ όχι\ περισσότερες\ από\ 3\ κεντρικές\ κάθετες\ ζώνες \\ 80\ min, για\ επιβατηγά\ πλοία\ (εκτός\ Ro - Ro) με\ περισσότερες\ από\ 3\ κεντρικές\ κάθετες\ ζώνες \end{cases} \quad (3.3)$$

Επίσης θα πρέπει να ισχύει :

$$E + L\ \leq 30\ min \quad (3.4)$$

Οι χρονικές απαιτήσεις των (3.2) και (3.3) σχηματικά :



Εικόνα 3.1 : Συνολικός χρόνος εκκένωσης

Όπου :

(1) είναι ο χρόνος επικάλυψης (Overlap time) και ισούται με (E + L) / 3,

(2) είναι ο μετρούμενος συνολικός χρόνος εκκένωσης (Calculated evacuation time),

(3) είναι ο μέγιστος επιτρεπτός χρόνος εκκένωσης, n, που υπολογίζεται απ'την (3.3)

3.2 Μοντελοποίηση οντότητας – Agent

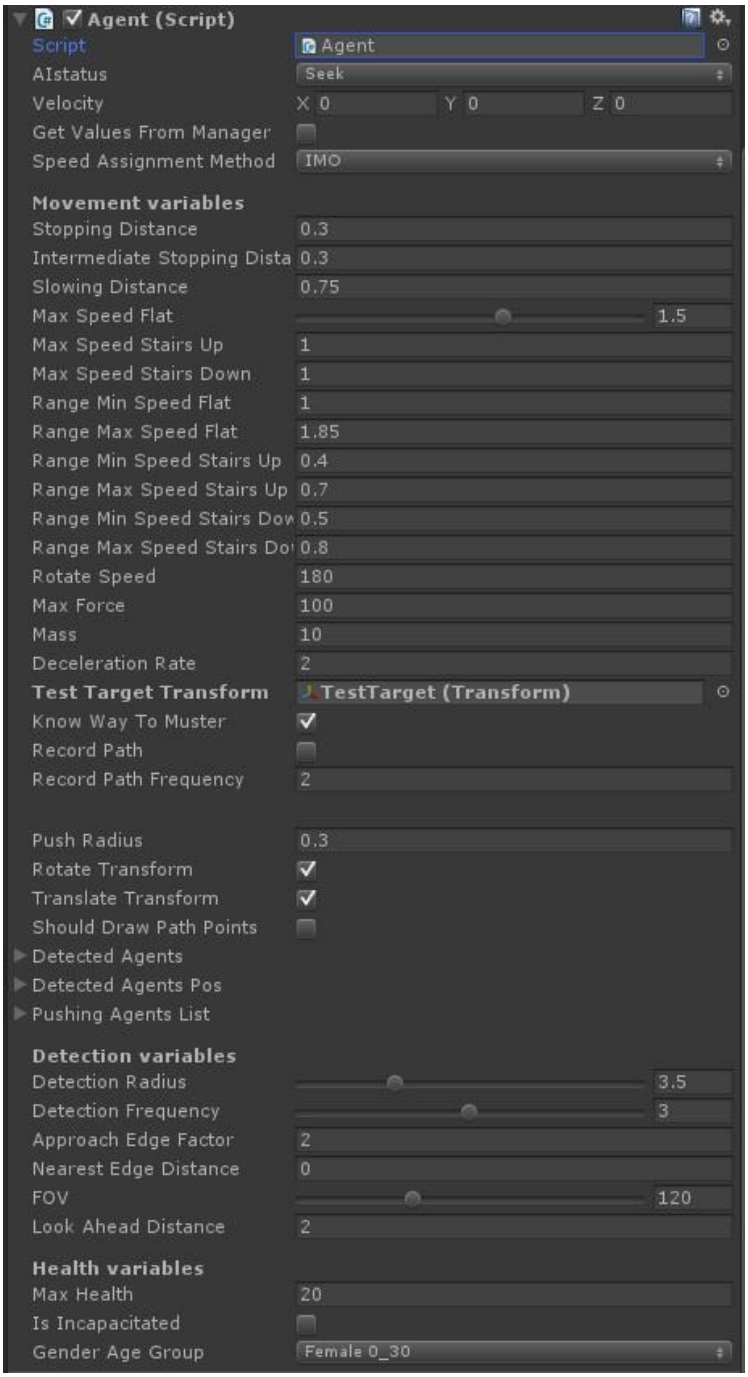
Την κάθε ανεξάρτητη οντότητα, της προσομοίωσης, που αναπαριστά άτομα συμμετέχοντες στην εκκένωση, θα αποκαλούμε Agent. Έτσι, ο Agent αποτελεί τη βάση της προσομοίωσής μας. Του έχουμε δώσει τέτοιες παραμέτρους ώστε να αποδίδει όσο το δυνατόν πιο πιστά την ανθρώπινη κίνηση. Επίσης, έχει ένα σύνολο συμπεριφορών που ορίζουν την γενική πορεία αλλά δημιουργούν τοπικές μικρές κινήσεις, π.χ. για να αποφύγει κάποιον άλλο κοντινό Agent.

Σύμφωνα με το MSC Circular 1238, Annex 2 (I.M.O. 2007), το μοντέλο του Agent θα πρέπει να έχει τουλάχιστον αυτά τα χαρακτηριστικά :

- Κάθε άνθρωπος αναπαριστάται ατομικά (από έναν Agent).
- Οι ιδιότητες κάθε Agent καθορίζονται από παραμέτρους, μερικές από τις οποίες είναι πιθανολογικές (θα αναλυθούν στην αμέσως επόμενη παράγραφο §3.2.1).
- Η πορεία κάθε Agent καταγράφεται (ο χρήστης της προσομοίωσης επιλέγει τη συχνότητα, δηλαδή πόσες φορές το δευτερόλεπτο γίνεται η καταγραφή της θέσης του Agent).
- Οι παράμετροι θα πρέπει να διαφέρουν ανάμεσα στα άτομα που συγκροτούν τον πληθυσμό της εκκένωσης (έχουμε ακολουθήσει τους πίνακες 3.1, 3.4 και 3.5 του MSC Circular 1238, Annex 2 (I.M.O. 2007) για τις ηλικιακές κατηγορίες και αντίστοιχες ταχύτητες των Agents).
- Οι βασικοί κανόνες για ατομικές αποφάσεις και κινήσεις είναι ίδιοι για όλους και περιγράφονται σε κάποιον καθολικό αλγόριθμο (έχουμε υιοθετήσει αυτήν την τεχνική και την περιγράφουμε στην παράγραφο §3.2.2).
- Η χρονική διαφορά ανάμεσα σε δύο κινήσεις οποιουδήποτε Agent στην προσομοίωση δε θα πρέπει να ξεπερνά το ένα δευτερόλεπτο προσομοιωμένου χρόνου (με τη βοήθεια του προγραμματισμού πολλαπλών νημάτων, multi-threading programming, καταφέραμε οι Agents να ελέγχουν την κίνησή τους κάθε frame. Έτσι, σε υπολογιστή που επιτυγχάνει μεγάλο αριθμό FPS, frames per second, π.χ. 60 fps, η χρονική διαφορά ανάμεσα σε δύο κινήσεις θα είναι 1 / 60 s ≈ 0.0167 s).

3.2.1 Παράμετροι Agent

Στην παρακάτω Εικόνα 3.2 διακρίνονται οι παράμετροι του Agent. Πάνω πάνω υπάρχει η boolean μεταβλητή GetValuesFromManager. Αν αυτή είναι true (που είναι και η default τιμή της) τότε ο Agent δεν λαμβάνει υπόψη του τις τιμές των παραμέτρων που του έχουν δοθεί στον Inspector της Εικόνας 3.2 αλλά θα λάβει τις τιμές απ’τον AgentManager που είναι ένα Game Object που ελέγχει όλους τους Agents της σκηνής.

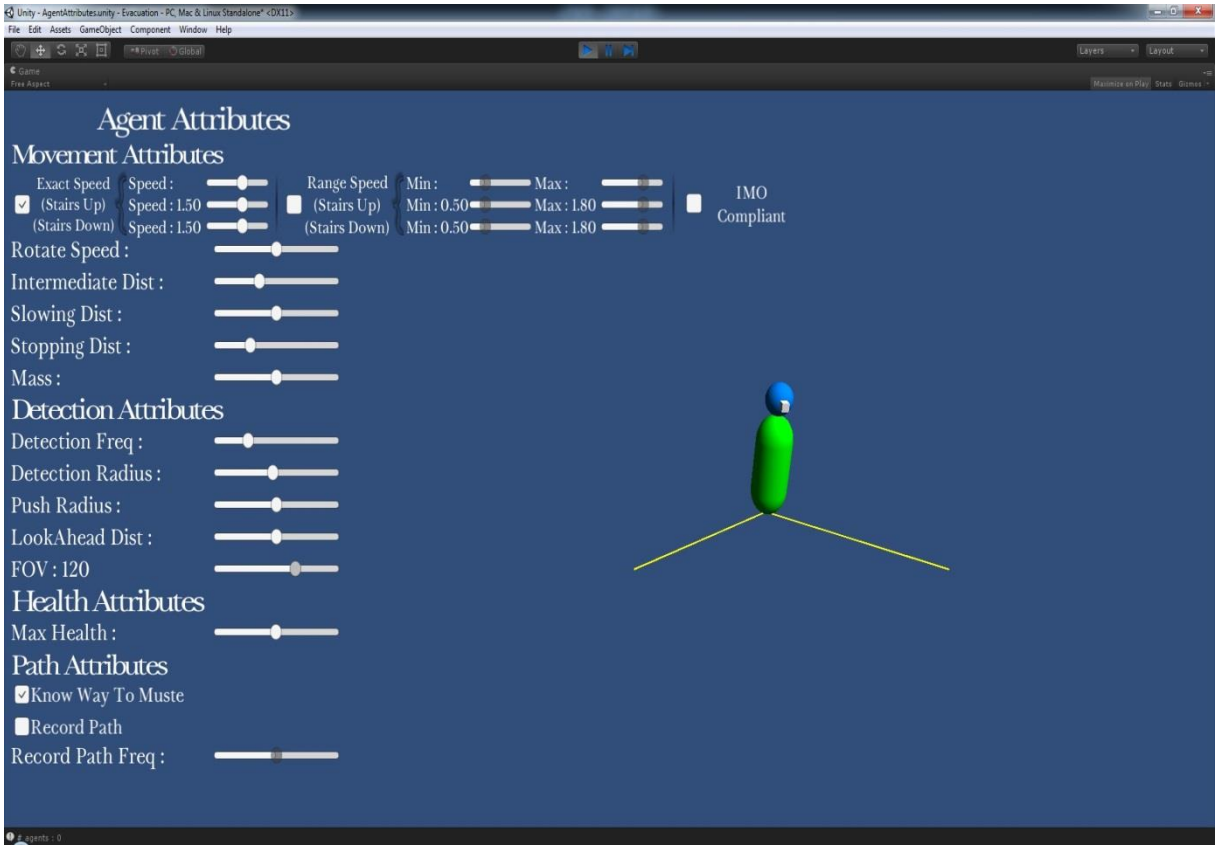


Εικόνα 3.2 : Παράμετροι Agent

Δηλαδή μέσω του AgentManager, ο χρήστης μπορεί να αλλάζει εύκολα τιμές στις παραμέτρους των Agents μαζικά. Αν, όμως, επιθυμεί να δώσει ξεχωριστές τιμές σε μερικούς συγκεκριμένους Agents, τότε απλά δίνει την τιμή false στην μεταβλητή GetValuesFromManager στα παράθυρα Inspector τους, οπότε αυτοί οι Agents θα χρησιμοποιήσουν τις τιμές που φαίνονται στον Inspector (Εικόνα 3.2) και τις οποίες μπορεί να παραμετροποιήσει για τον καθένα χωριστά.

Έχουμε φτιάξει, επίσης, μια σκηνή στην οποία μπορεί ο χρήστης να δίνει τιμές στις παραμέτρους του AgentManager σε γραφικό περιβάλλον για μεγαλύτερη ευκολία. Αυτό

διακρίνεται στην Εικόνα 3.3. Οι αλλαγές φαίνονται σε πραγματικό χρόνο. Μια άλλη εναλλακτική λύση είναι ο χρήστης να δώσει απευθείας τιμές στις παραμέτρους του AgentManager στον Inspector του.



Εικόνα 3.3 : Γραφικό περιβάλλον απόδοσης τιμών στις παραμέτρους του AgentManager

Άρα, τελικά, υπάρχουν τρεις διαφορετικοί τρόποι για να πάρει τιμές κάποιος Agent :

- Να θέσουμε τιμές σε κάθε Agent ξεχωριστά στον Inspector του και με GetValuesFromManager = false.
- Να θέσουμε τιμές στον Inspector του AgentManager τις οποίες θα "κληρονομήσουν" μετά όσοι Agents έχουν GetValuesFromManager = true.
- Να θέσουμε στο γραφικό περιβάλλον τις τιμές του AgentManager τις οποίες θα "κληρονομήσουν" μετά όσοι Agents έχουν GetValuesFromManager = true.

Σε κάθε περίπτωση, οι παράμετροι που χαρακτηρίζουν κάθε Agent είναι οι ίδιες, οπότε ας επεξηγήσουμε τι αντιπροσωπεύει η καθεμία :

- Αρχικά επιλέγεται ο τρόπος καθορισμού της μέγιστης ταχύτητας όταν ο Agent περπατάει σε επίπεδο έδαφος και όταν περπατάει στις σκάλες - ανεβαίνοντας και κατεβαίνοντας. Υπάρχουν τρεις περιπτώσεις :
  - i Ο χρήστης να θέσει επακριβώς την μέγιστη τιμή για τις τρεις ταχύτητες.
  - ii Ο χρήστης να θέσει ένα εύρος τιμών [min, max] και η μέγιστη τιμή να πάρει τυχαία μια τιμή μέσα σε αυτό το διάστημα.

iii Ο χρήστης να επιλέξει ότι θα συμβαδίσουμε με τις κατευθυντήριες γραμμές της MSC, οπότε οι μέγιστες τιμές φαίνονται στους πίνακες 3.1, 3.2 και 3.3 οι οποίοι προέκυψαν απ'τους πίνακες 3.1, 3.4 και 3.5 του Annex 2.

- Rotate Speed : Γωνιακή ταχύτητα περιστροφής.
- Intermediate Distance : Η απόσταση στην οποία πρέπει να πλησιάσει ο Agent οποιονδήποτε ενδιάμεσο κόμβο του path του, δηλαδή εκτός του τελευταίου, για να θεωρηθεί ότι τον "έφτασε" και να ξεκινήσει να κινείται προς τον επόμενο.
- Slowing Distance : Η απόσταση στην οποία πρέπει να πλησιάσει ο Agent τον τελευταίο κόμβο του path του για να αρχίσει να ελαττώνει την ταχύτητα του. Ουσιαστικά, σε αυτήν την απόσταση η συμπεριφορά κίνησης από Seek γίνεται Arrive. Θα αναλυθεί περισσότερο στη §3.2.2.2 και §4.2.
- Stopping Distance : Η απόσταση στην οποία πρέπει να πλησιάσει ο Agent τον τελευταίο κόμβο του path του για να θεωρήσει ότι έχει φτάσει τον στόχο του και ολοκληρώσει το path του.
- Mass : Μάζα.
- Detection Frequency : Η συχνότητα (πόσες φορές το δευτερόλεπτο) ο Agent δημιουργεί μια αόρατη σφαίρα του για να καταγράψει τους γειτονικούς του Agents. Θα αναλυθεί περισσότερο στη §4.2.
- Detection Radius : Η ακτίνα της προαναφερθείσας σφαίρας.
- Push Radius : Είναι η ακτίνα που ορίζει τα σώματα των Agents. Όταν δύο Agents βρεθούν σε απόσταση μικρότερη του 2 * Push Radius, τότε θεωρείται ότι έχει εισέρθει ο ένας στο χώρο του άλλου και αρχίζουν να απωθούνται μέχρι να βρεθούν πάλι σε απόσταση τουλάχιστον 2 * Push Radius. Θα αναλυθεί περισσότερο στη §4.2.
- LookAhead Distance : Είναι η μέγιστη απόσταση μεταξύ δύο Agents ώστε αυτός που βρίσκεται πίσω απ'τον άλλο να αρχίσει την διαδικασία προσπέρασης.
- FOV (Field Of View) : Το πεδίο όρασης του Agent σε μοίρες. Διακρίνεται στο στιγμιότυπο της Εικόνας 3.3. Θα αναλυθεί περισσότερο μαζί και με τη LookAhead Distance στη §4.2.
- Max Health : Η μέγιστη τιμή της "υγείας" του Agent. Μειώνεται όταν ο Agent συμμετέχει σε συγκρούσεις με άλλους Agents που θεωρούνται σφοδρές, δηλαδή όταν η σχετική τους ταχύτητα ξεπερνάει κάποιο όριο. Θα αναλυθεί περισσότερο στη §4.2.
- Know Way To Muster : Boolean μεταβλητή. Αν είναι true, τότε ο Agent ξέρει το δρόμο προς το πλησιέστερο Muster Station. Αν είναι false, τότε ο Agent δε ξέρει το δρόμο προς το πλησιέστερο Muster Station οπότε θα κινηθεί προς την κοντινότερη πινακίδα για πάρει περισσότερες πληροφορίες.
- Record Path : Boolean μεταβλητή. Αν είναι true, τότε ο Agent αποθηκεύει τη θέση του σε μια λίστα από Vector3 ανά τακτά χρονικά διαστήματα που ορίζονται απ'την επόμενη παράμετρο. Από αυτή τη λίστα προκύπτει στο τέλος η πορεία που ακολούθησε ο Agent μέχρι να φτάσει στον Muster Station.
- Record Path Frequency : Η συχνότητα (πόσες φορές το δευτερόλεπτο) ο Agent καταγράφει τη θέση του.

Στην περίπτωση που επιλέξουμε να ακολουθήσουμε τις κατευθυντήριες γραμμές της MSC για τον καθορισμό της μέγιστης τιμής της ταχύτητας περπατήματος σε επίπεδο έδαφος και σε σκάλες, ακολουθείται η εξής διαδικασία :

1. Ο κάθε Agent εντάσσεται σε μια ηλικιακή (ορίζεται επίσης και το φύλο) κατηγορία σύμφωνα με τις πιθανότητες του Πίνακα 3.1.
2. Αναλόγως την ηλικιακή κατηγορία, η μέγιστη τιμή ταχύτητας περπατήματος σε επίπεδο έδαφος προκύπτει τυχαία μέσα στο εύρος [min, max] που ορίζονται στον Πίνακα 3.2.
3. Ομοίως για την ταχύτητα περπατήματος σε σκάλες - ανεβαίνοντας και κατεβαίνοντας, με τιμές min και max απ'τον Πίνακα 3.3.

Πληθυσμιακές ομάδες - Επιβάτες	**Ποσοστό επιβατών (%)**
Άνδρας, κάτω από 30 χρονών	7
Άνδρας, 30 – 50 χρονών	7
Άνδρας, πάνω από 50 χρονών	16
Άνδρας, πάνω από 50 χρονών με μειωμένη κινητικότητα-1	10
Άνδρας, πάνω από 50 χρονών με μειωμένη κινητικότητα-2	10
Γυναίκα, κάτω από 30 χρονών	7
Γυναίκα, 30 – 50 χρονών	7
Γυναίκα, πάνω από 50 χρονών	16
Γυναίκα, πάνω από 50 χρονών με μειωμένη κινητικότητα-1	10
Γυναίκα, πάνω από 50 χρονών με μειωμένη κινητικότητα-2	10
Πληθυσμιακές ομάδες - Πλήρωμα	**Ποσοστό πληρώματος (%)**
Άνδρας	50
Γυναίκα	50

Πίνακας 3.1 : Σύνθεση πληθυσμού (ηλικία και φύλο)

Πληθυσμιακές ομάδες - Επιβάτες	Ταχύτητα σε επίπεδο έδαφος	
	Min (m/s)	Max (m/s)
Άνδρας, κάτω από 30 χρονών	1.11	1.85
Άνδρας, 30 – 50 χρονών	0.97	1.62
Άνδρας, πάνω από 50 χρονών	0.84	1.40
Άνδρας, πάνω από 50 χρονών με μειωμένη κινητικότητα-1	0.64	1.06
Άνδρας, πάνω από 50 χρονών με μειωμένη κινητικότητα-2	0.55	0.91
Γυναίκα, κάτω από 30 χρονών	0.93	1.55
Γυναίκα, 30 – 50 χρονών	0.71	1.19
Γυναίκα, πάνω από 50 χρονών	0.56	0.94
Γυναίκα, πάνω από 50 χρονών με μειωμένη κινητικότητα-1	0.43	0.71
Γυναίκα, πάνω από 50 χρονών με μειωμένη κινητικότητα-2	0.37	0.61
Πληθυσμιακές ομάδες - Πλήρωμα	**Ταχύτητα σε επίπεδο έδαφος**	
	Max (m/s)	**Max (m/s)**
Άνδρας	1.11	1.85
Γυναίκα	0.93	1.55

Πίνακας 3.2 : Μέγιστη ταχύτητα σε επίπεδο έδαφος

Πληθυσμιακές ομάδες - Επιβάτες	Ταχύτητα σε σκάλες			
	Κατεβαίνοντας		Ανεβαίνοντας	
	Min (m/s)	Max (m/s)	Min (m/s)	Max (m/s)
Άνδρας, κάτω από 30 χρονών	0.76	1.26	0.50	0.84
Άνδρας, 30 – 50 χρονών	0.64	1.07	0.47	0.79
Άνδρας, πάνω από 50 χρονών	0.50	0.84	0.38	0.64
Άνδρας, πάνω από 50 χρονών με μειωμένη κινητικότητα-1	0.38	0.64	0.29	0.49
Άνδρας, πάνω από 50 χρονών με μειωμένη κινητικότητα-2	0.33	0.55	0.25	0.41
Γυναίκα, κάτω από 30 χρονών	0.56	0.94	0.47	0.79
Γυναίκα, 30 – 50 χρονών	0.49	0.81	0.44	0.74
Γυναίκα, πάνω από 50 χρονών	0.45	0.75	0.37	0.61
Γυναίκα, πάνω από 50 χρονών με μειωμένη κινητικότητα-1	0.34	0.56	0.28	0.46
Γυναίκα, πάνω από 50 χρονών με μειωμένη κινητικότητα-2	0.29	0.49	0.23	0.39
Πληθυσμιακές ομάδες - Πλήρωμα	**Ταχύτητα σε σκάλες**			
	Κατεβαίνοντας		**Ανεβαίνοντας**	
	Min (m/s)	**Max (m/s)**	**Min (m/s)**	**Max (m/s)**
Άνδρας	0.76	1.26	0.50	0.84
Γυναίκα	0.56	0.94	0.47	0.79

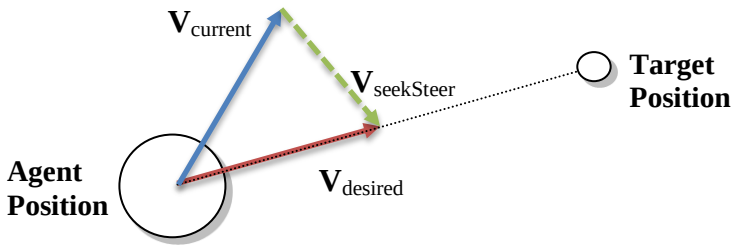
Πίνακας 3.3 : Μέγιστη ταχύτητα σε σκάλες

3.2.2 Συμπεριφορές και κίνηση Agent

Αρχικά θα αναλύσουμε τις συμπεριφορές [8] που μαζί συνθέτουν την κίνηση του Agent και έπειτα θα περιγράψουμε πως υλοποιήθηκε η κίνηση αυτή.

3.2.2.1 Συμπεριφορά Seek

Η Seek είναι η συμπεριφορά που χρησιμοποιείται συχνότερα. Δίνει στον Agent το επιθυμητό διάνυσμα ταχύτητας για να φτάσει τον στόχο του. Έχουμε διαφοροποιήσει τον αλγόριθμο του [9] και [10], από διδιάστατα Vectors σε τριδιάστατα που χρησιμοποιούμε εμείς. Στην παρακάτω εικόνα 3.4 φαίνεται σχηματικά η υλοποίηση (τα σύμβολα σε bold είναι διανύσματα) :



Εικόνα 3.4 : Συμπεριφορά Seek

Αλγόριθμος της Seek :

- Ξέρουμε το διάνυσμα της τωρινής ταχύτητας, $\mathbf{V}_{current}$.
- Ξέρουμε το διάνυσμα θέσης του στόχου μας, **Target Position**.
- Υπολογίζουμε το επιθυμητό διάνυσμα ταχύτητας, $\mathbf{V}_{desired}$, ως εξής :

$$\boldsymbol{V}_{desired} = (normalized\ (\boldsymbol{TargetPosition} - \boldsymbol{AgentPosition})) * maxSpeed$$

- Για να καταλήξει ο Agent στο επιθυμητό διάνυσμα, θα πρέπει να του ασκηθεί το :

$$\boldsymbol{V}_{seekSteer} = \boldsymbol{V}_{desired} - \boldsymbol{V}_{current}$$

- Τέλος, το πολλαπλασιάζουμε με ένα βάρος $w_{seekSteer}$ ώστε να μπορούμε να αλλάζουμε το πόσο πολύ ή λίγο επιδρά στο συνολικό τελικό διάνυσμα κίνησης.

3.2.2.2 Συμπεριφορά Arrive

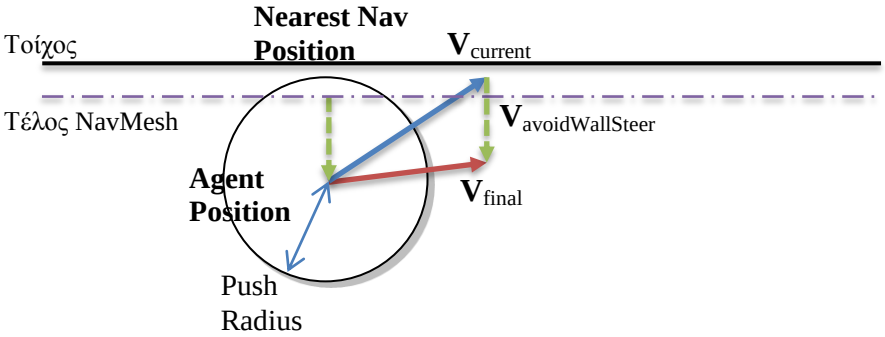
Η Arrive είναι σχεδόν ίδια με την Seek, δηλαδή οδηγεί τον Agent προς κάποιο στόχο αλλά με τη διαφορά ότι από κάποια απόσταση (απ'τον στόχο) ο Agent αρχίζει και επιβραδύνει [9]. Η απόσταση αυτή είναι η μεταβλητή Slowing Distance του Agent. Ο λόγος που γίνεται αυτό είναι για να αποφύγουμε μια φθίνουσα ταλάντωση που γίνεται γύρω απ'το Traget Position όταν ο Agent με την Seek έχει μεγάλη ταχύτητα και προσπερνάει τον στόχο με αποτέλεσμα να γυρνάει προς τα πίσω και η διαδικασία να επαναλαμβάνεται μέχρι να σταματήσει. Ενώ, με την Arrive, ο Agent επιβραδύνει τόσο πιο πολύ όσο η απόσταση απ'το Target Position μειώνεται, και όταν το πλησίασει, έχει ήδη σχεδόν μηδενική ταχύτητα.

Αλγόριθμος της Arrive :

- Ξέρουμε το διάνυσμα θέσης του Agent, **Agent Position**.
- Ξέρουμε το διάνυσμα θέσης του στόχου μας, **Target Position**.
- Υπολογίζουμε το διάνυσμα ανάμεσα στις δύο θέσεις, $\mathbf{V}_{\text{toTarget}}$, ως εξής :
$$\boldsymbol{V}_{toTarget} = (\boldsymbol{TargetPosition} - \boldsymbol{AgentPosition})$$
- Υπολογίζουμε την απόσταση μεταξύ των δύο θέσεων, distance :
$$distance = magnitude(\boldsymbol{V}_{toTarget})$$
- Ελαττώνουμε την ταχύτητα (το μέτρο της) αναλόγως την απόσταση distance και τον παράγοντα decelerationRate που ορίζει πόσο γρήγορα επιβραδύνει ο Agent :
$$speed = \frac{distance}{decelerationRate}$$
- Βεβαιώνουμε ότι η ταχύτητα δε ξεπερνά τη μέγιστη τιμή της :
$$speed = Min(speed, maxSpeed)$$
- Βρίσκουμε κανονικοποιημένο το διάνυσμα της επιθυμητής ταχύτητας διαιρώντας το $\mathbf{V}_{\text{toTarget}}$ με τη distance (έτσι είναι και γρηγορότερο απ'το να κάναμε normalized και μετά επί maxSpeed όπως κάναμε στο Seek) :
$$\boldsymbol{V}_{desired} = \boldsymbol{V}_{toTarget} * \frac{speed}{distance}$$
- Οπότε τελικά το $\mathbf{V}_{\text{arriveSteer}}$ προκύπτει ως :
$$\boldsymbol{V}_{arriveSteer} = \boldsymbol{V}_{desired} - \boldsymbol{V}_{current}$$
- Ομοίως, το πολλαπλασιάζουμε μετά με ένα βάρος $w_{\text{arriveSteer}}$.

3.2.2.3 Συμπεριφορά AvoidWall

Η AvoidWall ενεργοποιείται όταν ο Agent βρεθεί κοντά σε τοίχο. Συγκεκριμένα, όταν η απόσταση του κέντρου του Agent από τη πλησιέστερη NavMeshEdge είναι μικρότερη από την Push Radius του Agent. Τότε θα του ασκηθεί ένα διάνυσμα δύναμης που απομακρύνει τον Agent απ'τον τοίχο. Στην παρακάτω εικόνα 3.5 φαίνεται σχηματικά η υλοποίηση (τα σύμβολα σε bold είναι διανύσματα) όπου έχουμε θεωρήσει ότι δεν υπάρχουν άλλα εφαρμοζόμενα steering διανύσματα :



Εικόνα 3.5 : Συμπεριφορά AvoidWall

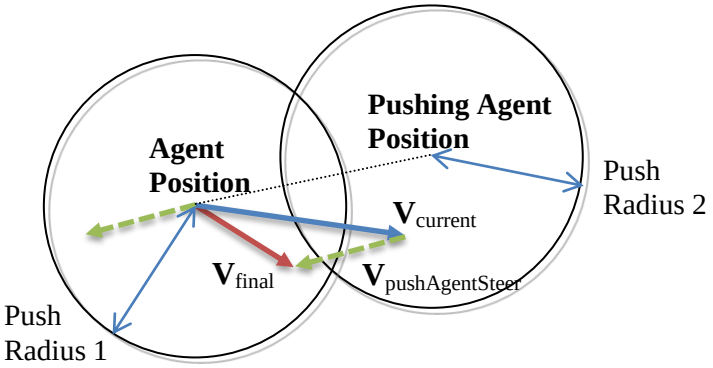
Αλγόριθμος της AvoidWall :

- Ο Agent κάθε frame εκτελεί την function FindClosestEdge που του επιστρέφει το κοντινότερο σημείο της πλησιέστερης NavMeshEdge.
- Βρίσκουμε την απόσταση του Agent απ'αυτό το σημείο :
$$distance = magnitude(\boldsymbol{NearestNavPosition} - \boldsymbol{AgentPosition})$$
- Χρησιμοποιούμε και έναν παράγοντα approachEdgeFactor με τον οποίο διαιρούμε την distance για να βρούμε τη συνολική απόσταση που θα θέλαμε να αρχίζει να ασκεί ο τοίχος δύναμη, finalPushRadius.
- Αν αυτή η distance είναι μικρότερη της finalPushRadius, τότε θα ασκηθεί στον Agent κάθετο διάνυσμα με κατεύθυνση απομάκρυνσης απ'τον τοίχο.
- Το μέτρο του avoidWallSteer διανύσματος προκύπτει ως γραμμική παρεμβολή του normal διανύσματος που προκύπτει απ'τη διαφορά της θέσης του Agent με τη θέση NearestNavPosition ως προς την απόστασή τους. Δηλαδή :
$$if\ (distance < \text{finalPushRadius})$$
$$\boldsymbol{V}_{normal} = normalized(\boldsymbol{AgentPosition} - \boldsymbol{NearestNavPosition})$$
$$\boldsymbol{V}_{avoidWallSteer} = Vector3.Lerp(\boldsymbol{V}_{normal}, \boldsymbol{V}_{zero}, distance/finalPushRadius)$$
- Ομοίως το πολλαπλασιάζουμε μετά με ένα βάρος $w_{avoidWallSteer}$.

3.2.2.4 Συμπεριφορά PushAgent

Η PushAgent ενεργοποιείται όταν δύο Agents βρεθούν ο ένας μέσα στο χώρο του άλλου, δηλαδή στον κύκλο με ακτίνα PushRadius. Η PushAgent υλοποιείται στο ίδιο κομμάτι κώδικα μαζί με την επόμενη συμπεριφορά AvoidNearest καθώς και οι δύο χρειάζονται να ελέγξουν τη θέση του Agent σε σχέση με τις θέσεις των γειτόνων του. Οπότε για να αποφύγουμε τον έλεγχο της λίστας με τους γειτονικούς Agents για δεύτερη φορά, έχουμε γράψει τις δυο συμπεριφορές μαζί και απλά επιστρέφουμε δύο διανύσματα, ένα για την κάθε συμπεριφορά.

Η περίπτωση της συμπεριφοράς PushAgent μοιάζει με τη AvoidWall και την Separate Behaviour του [11]. Ας δούμε σχηματικά την περίπτωση της PushAgent (εικόνα 3.6) όπου έχουμε θεωρήσει ότι δεν υπάρχουν άλλα εφαρμοζόμενα steering διανύσματα :



Εικόνα 3.6 : Συμπεριφορά PushAgent

Αλγόριθμος της PushAgent :

- Ο Agent κάθε frame ελέγχει τη θέση του σε σχέση με τους γειτονικούς του Agents και υπολογίζει το διάνυσμα :

$$\boldsymbol{distanceVec} = \boldsymbol{AgentPosition} - \boldsymbol{PushingAgentPosition}$$

- Έπειτα υπολογίζει την απόσταση μεταξύ των δύο Agents, δηλαδή το μέτρο του προηγούμενου διανύσματος :

$$distance = magnitude(\boldsymbol{distanceVec})$$

- Εάν η απόσταση είναι μικρότερη του αθροίσματος PushRadius1 + PushRadius2, τότε οι δύο Agents έχουν εισέρθει στο χώρο του άλλου και θα πρέπει να τους ασκηθεί ένα steering διάνυσμα που θα τους απομακρύνει. Κανονικοποιούμε το διάνυσμα **distanceVec** και το χρησιμοποιούμε σε γραμμική παρεμβολή όπως στην AvoidWall :

$$if\ (distance < totalPushRadius)$$
$$\boldsymbol{V}_{normal} = normalized(\boldsymbol{distanceVec})$$
$$\boldsymbol{V}_{\text{pushAgentSteer}} = Vector3.Lerp(\boldsymbol{V}_{normal}, \boldsymbol{V}_{zero}, distance\ /\ totalPushRadius)$$

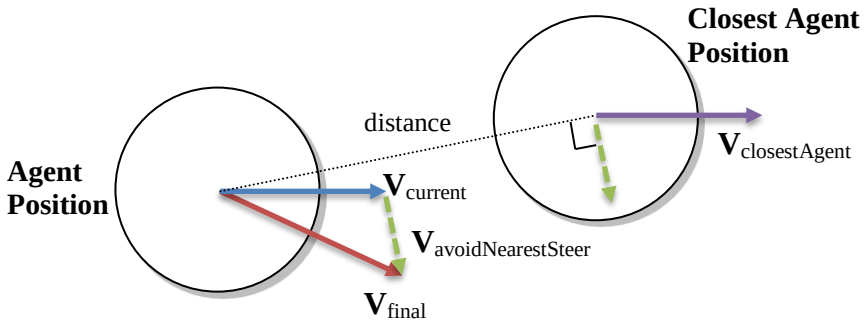
- Το πολλαπλασιάζουμε και με τον αντίστροφο λόγο των μαζών, ώστε ο βαρύτερος Agent να δέχεται μικρότερη δύναμη :

$$\boldsymbol{V}_{\text{pushAgentSteer}} = \boldsymbol{V}_{\text{pushAgentSteer}} * (PushingAgentMass \ / \ AgentMasss)$$

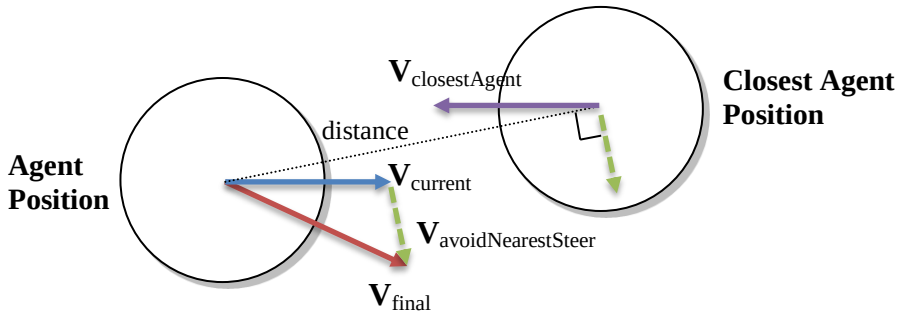
- Ομοίως το πολλαπλασιάζουμε μετά με ένα βάρος $w_{pushAgentSteer}$.

3.2.2.5 Συμπεριφορά AvoidNearest

Η PushAgent [10] ενεργοποιείται για τον πλησιέστερο από όλους τους γειτονικούς Agents που βρίσκονται στο Field Of View του Agent που εξετάζουμε. [12], §5.5.1, page 69. Όπως είπαμε και στην §3.2.2.4, η AvoidNearest γίνεται μαζί με την PushAgent ακριβώς μετά τον έλεγχο της λίστας με τους γειτονικούς Agents. Η λειτουργία είναι παρόμοια είτε ο Agent και ο πλησιέστερός του έχουν ταχύτητες με ίδια κατεύθυνση είτε με αντίθετη. Η πρώτη περίπτωση διακρίνεται στην εικόνα 3.7 και η δεύτερη στην εικόνα 3.8. Και στις δύο πιθανότητες έχουμε θεωρήσει ότι δεν υπάρχουν άλλα εφαρμοζόμενα steering διανύσματα.



Εικόνα 3.7 : Συμπεριφορά AvoidNearest με ταχύτητες ίδιας κατεύθυνσης



Εικόνα 3.8 : Συμπεριφορά AvoidNearest με ταχύτητες αντίθετης κατεύθυνσης

Παρατηρούμε ότι το $\mathbf{V}_{\text{avoidNearestSteer}}$ παραμένει το ίδιο, γεγονός αναμενόμενο καθώς είναι ανεξάρτητο απ'το διάνυσμα της ταχύτητας του πλησιέστερου Agent.

Αλγόριθμος της AvoidNearest :

- Ο Agent κάθε frame ελέγχει τη θέση του σε σχέση με τους γειτονικούς του Agents και βρίσκει τον πλησιέστερο από όσους βρίσκονται μέσα στο Field Of View του [13]. Τότε υπολογίζουμε το πρόσημο του εσωτερικού γινομένου των ταχυτήτων τους :

$$cos\theta = Vector3.Dot(\boldsymbol{V}_{current} - \boldsymbol{V}_{closestAgent})$$

- Αν cosθ < 0 τότε οι Agents κατευθύνονται ο ένας προς τον άλλο και πρέπει να τους ασκηθεί ένα steering διάνυσμα για να αποφευχθεί η σύγκρουση. Επίσης, αν ο Agent και ο πλησιέστερός του έχουν ταχύτητες ίδιας κατεύθυνσης αλλά ο Agent που είναι πίσω έχει μεγαλύτερη maxSpeed, τότε πάλι υπάρχει η πιθανότητα σύγκρουσης. Άρα υπολογίζουμε το διάνυσμα των αποστάσεων τους και το μέτρο του :

$$\boldsymbol{distanceVec} = \boldsymbol{AgentPosition} - \boldsymbol{ClosestAgentPosition}$$
$$distance = magnitude(\boldsymbol{distanceVec})$$

- Για να βρούμε το κάθετο διάνυσμα (στην μεταξύ τους απόσταση) που θα βοηθήσει στην αποφυγή της σύγκρουσης, θα υπολογίσουμε δύο φορές το εξωτερικό γινόμενο της ταχύτητας του Agent με το **distanceVec** :

$$\boldsymbol{V}_{normal} = normalized(Cross(Cross(\boldsymbol{distanceVec}, \boldsymbol{V}_{current}), \boldsymbol{distanceVec}))$$

- Έπειτα το χρησιμοποιούμε σε γραμμική παρεμβολή λαμβάνοντας υπόψη και την μεταβλητή lookAheadDistance που αναπαριστά σε ποια απόσταση αρχίζει την διαδικασία προσπέρασης :

$$\boldsymbol{V}_{\text{avoidNearestSteer}} = Vector3.Lerp(\boldsymbol{V}_{normal}, \boldsymbol{V}_{zero}, distance\ /\ lookAheadDistance)$$

- Τέλος, το πολλαπλασιάζουμε είτε με 1.2 όταν οι ταχύτητες τους έχουν ίδια κατεύθυνση, είτε με 2.4 όταν έχουν αντίθετη κατεύθυνση. Αυτό γίνεται γιατί όταν θα έχουν αντίθετη κατεύθυνση, θα συγκρουσθούν σε μικρότερο χρόνο, οπότε το αποτρεπτικό κάθετο διάνυσμα να έχει μεγαλύτερο μέτρο.
- Ομοίως το πολλαπλασιάζουμε μετά με ένα βάρος $w_{avoidNearestSteer}$.

3.2.2.6 Κίνηση Agent

Όπως είπαμε και στην §3.1, προσπαθήσαμε να ακολουθήσουμε μια μικροσκοπική έκδοση μοντελοποίησης των ανθρώπων (επιβάτες και πλήρωμα). Έτσι, αναπαραστήσαμε τον άνθρωπο με μια οντότητα που υπακούει τους νόμους κίνησης του Νεύτωνα. Οπότε, για την κίνησή του ισχύει ο θεμελιώσης νόμος της δυναμικής (2^{ος} νόμος Νεύτωνα) – οι bold χαρακτήρες συμβολίζουν διανύσματα :

$$\sum_n \boldsymbol{F} = m\boldsymbol{a} \Rightarrow \boldsymbol{\alpha} = \frac{\sum_n \boldsymbol{F}}{m} \tag{3.5}$$

Δηλαδή η επιτάχυνση **a** ενός σώματος μάζας **m** προκύπτει απ΄το άθροισμα των n δυνάμεων $\sum \boldsymbol{F}$ που ασκούνται στο σώμα διά της μάζας του.

Επίσης, ο ορισμός της επιτάχυνσης είναι :

$$\boldsymbol{a} = \frac{d\boldsymbol{V}}{dt} \tag{3.6}$$

Όπου d**V** η στιγμιαία μεταβολή της ταχύτητας σε χρονικό διάστημα dt. Άρα, απ'τις (3.5) και (3.6) προκύπτει :

$$\frac{d\boldsymbol{V}}{dt} = \frac{\sum_n \boldsymbol{F}}{m} \Rightarrow d\boldsymbol{V} = \frac{\sum_n \boldsymbol{F}}{m}\, dt \tag{3.7}$$

Έτσι, και στον κώδικα θεωρώντας ότι έχουμε μια λίστα, SteerList, με αθροισμένα όλα τα steering διανύσματα, δηλαδή τις δυνάμεις που ασκούνται στον Agent από τις διάφορες (ενεργοποιημένες) συμπεριφορές του, θα είναι αντίστοιχα :

*AgentList[i].velocity += SteerList[i] * deltaTime*

Επομένως, το διάνυσμα της ταχύτητας κάθε Agent i μεταβάλλεται απ'το συνολικό steering διάνυσμα i. Το deltaTime είναι ο χρόνος ανάμεσα σε δύο διαδοχικά frames του rendering της εφαρμογής.

Ομοίως, ξεκινώντας απ'τον ορισμό της ταχύτητας, βρίσκουμε ότι για την μετατόπιση **r** του Agent, η φυσική εξίσωση είναι :

$$\boldsymbol{V} = \frac{d\boldsymbol{r}}{dt} \Rightarrow d\boldsymbol{r} = \boldsymbol{V}dt \tag{3.8}$$

Ανάλογος είναι ο κώδικας υπεύθυνος για την αλλαγή της θέσης του Agent i στον τριδιάστατο χώρο :

*AgentList[i]. position += AgentList[i].velocity * deltaTime*

Δηλαδή, η θέση του κάθε Agent μεταβάλλεται απ'το δίανυσμα της ταχύτητας, που έχει προκύψει απ'το άθροισμα των steering διανυσμάτων που του εφαρμόζονται, σε χρόνο deltaTime.

Έτσι, το μοντέλο μας υπακούοντας τους φυσικούς νόμους προσομοιάζει όσο το δυνατόν πιστότερα μια φυσιολογική ανθρώπινη τριδιάστατη κίνηση.

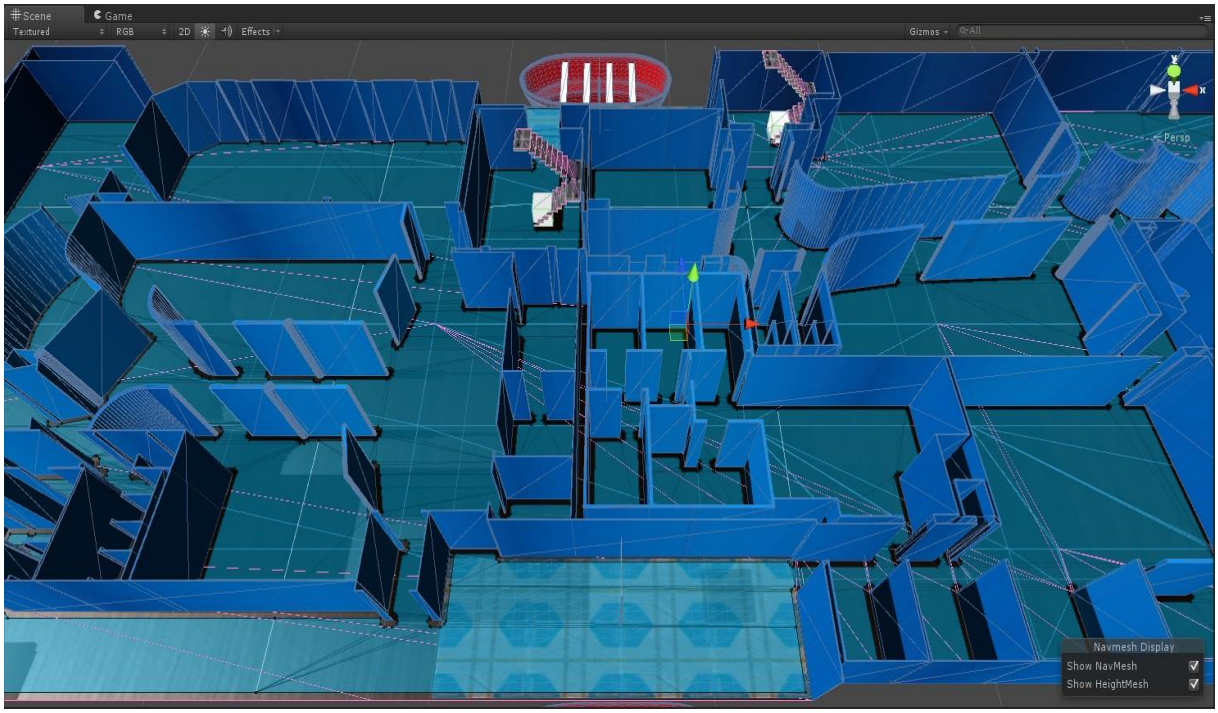
Κεφάλαιο 4

Υλοποίηση προσομοίωσης στο Unity3D

Σε αυτό το κεφάλαιο θα περιγράψουμε πως υλοποιήσαμε την κεντρική σκηνή της προσομοίωσης της διαδικασίας εκκένωσης στο Unity3D. Επίσης, θα αναλύσουμε περισσότερο το μοντέλο του Agent και θα εμβαθύνουμε σε μερικές περιοχές του κώδικα διαφόρων Scripts που έχουν το μεγαλύτερο ενδιαφέρον.

4.1 Κεντρική σκηνή

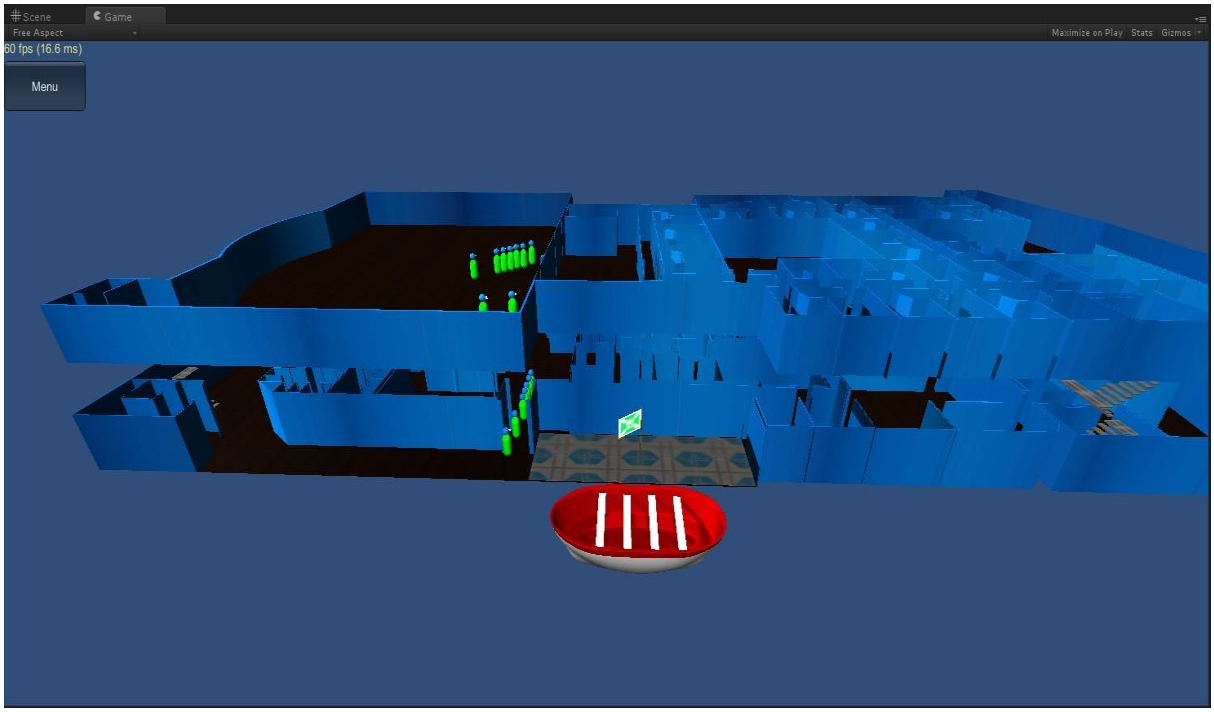
Περιγράψαμε στην §2.2.3 πώς ξεκινώντας από ένα σχέδιο καταστρώματος σε AutoCAD καταλήξαμε σε ένα τριδιάστατο μοντέλο του καταστρώματος στο Unity. Αφού δημιουργήσουμε δύο Directional Lights και την Main Camera στη σκηνή μας, κάνουμε import τα μοντέλα των καταστρωμάτων μαζί με τα πατώματά τους. Τα χαρακτηρίζουμε ως Navigation Static, οπότε μπορούμε να δημιουργήσουμε ένα NavMesh από τη γεωμετρία του κάθε ζευγαριού Deck και Deck_ground. Είναι το γαλάζιο επίπεδο που διακρίνεται ακριβώς πάνω απ’το πάτωμα στην Εικόνα 4.1 :



Εικόνα 4.1 : Δημιουργία NavMesh του Deck5

Το NavMesh φτιάχνεται στο παράθυρο Navigation με το button Bake. Στην εικόνα 4.1 φαίνονται και μωβ γραμμές, κυρίως στις σκάλες, που αποτελούν το HeightMesh. Μέσω αυτού γίνεται η σύνδεση μεταξύ επιπέδων NavMesh διαφορετικού ύψους. Τις σκάλες τις δημιουργήσαμε εμείς στο Unity, αποτελούνται από πολλά κυβοειδή και έχουν το Stairs Layer. Υπάρχουν δύο είδη, το πρώτο με ύψος 3.2m που ενώνει τα Deck5 και Deck6 και το δεύτερο με ύψος 2.7m που ενώνει τα Deck6 και Deck7. Ομοίως, φτιάχνουμε τα NavMesh και για τα άλλα καταστρώματα. Αυτά τα NavMesh χρησιμοποιεί το Navigation σύστημα του Unity για να φτιάχνει το Path προς το στόχο που θέλουν να πάνε οι Agents.

Στην κάμερα της σκηνής έχουμε προσθέσει ένα Script (*FreeCamera.cs*) υπεύθυνο για την κίνησή της. Η μετατόπιση και η περιστροφή της λειτουργούν με τον ίδιο χειρισμό όπως στον Editor του Unity. Με το πλήκτρο ‘c’ η κάμερα μεταφέρεται σε μια συγκεκριμένη θέση ακριβώς πάνω απ’τα decks και τα κοιτάει προς τα κάτω, δηλαδή σαν κάτοψη. Στο ίδιο Script, με τα πλήκτρα ‘5’, ‘6’, και ‘7’ “εξαφανίζουμε” και “εμφανίζουμε” τα Deck5, Deck6 και Deck7 αντίστοιχα μαζί με όλους τους Agents που βρίσκονται σε αυτά. Βλέπουμε στην εικόνα 4.2 ένα παράδειγμα όπου είναι εξαφανισμένο το Deck6 και οι Agents που περπατάνε σε αυτό:



Εικόνα 4.2 : Main camera culling του Deck6

Αυτό επιτυγχάνεται με τη χρήση δυαδικών μασκών που αντιστοιχούν σε κάθε Layer. Υπάρχουν τρία Layers για τα τρία Decks και άλλα τρία Layers για τους Agents όταν βρίσκονται σε κάποιο Deck. Στην εικόνα 4.1 φαίνονται μερικοί δύο άσπροι κύβοι στο κάτω μέρος των σκαλών. Η λειτουργία τους είναι ότι όταν κατεβαίνει κάποιος Agent τη σκάλα, να αλλάζει Layer σε αυτόν και στα “παιδιά” Game Objects του (Script : *ChangeLayer.cs*). Αντίστοιχοι τέτοιοι κύβοι υπάρχουν στο πάνω άκρο των σκαλών ώστε να αλλάζουν το Layer των Agents που ανεβαίνουν. Ο MeshRenderer των κύβων εξαφανίζεται όταν ξεκινήσει η εφαρμογή καθώς δε χρειάζεται να τον βλέπουμε.

Ας εξηγήσουμε πως δουλεύουν η culling mask της Main Camera με ένα παράδειγμα. Αρχικά, με τις εντολές :

layerDeck6Mask = LayerMask.NameToLayer("Deck6"); *(έστω = 11_{10})*

layerAgent6Mask = LayerMask.NameToLayer("Agent6"); (έστω = 14_{10})

βρίσκουμε τους ακεραίους αριθμούς που αντιστοιχούν στα Layers : Deck6 και Agent6 (αυτό είναι το Layer για όσους Agents περπατάνε στο Deck6). Όταν ξεκινά η εφαρμογή, η culling mask της Main Camera είναι (32-bit) :

11111111111111111111111111111111

δηλαδή όλα τα bits της είναι 1 (ενεργοποιημένα), οπότε όλα τα Layers είναι ορατά. Όταν πατήσουμε π.χ. το πλήκτρο ‘6’ γίνονται τα εξής :

layerDeckMask = 1 << layerDeck6Mask;

layerAgentMask = 1 << layerAgent6Mask;

οπότε φτιάχνουμε μια καινούργια μάσκα, layerDeckMask, ξεκινώντας απ’το :

00000000000000000000000000000001

και το κάνουμε left shift (<<) τόσες θέσεις όσες είναι ο ακέραιος του layerDeck6Mask = 11, άρα γίνεται :

00000000000000000000100000000000

Ομοίως και το layerAgentMask τελικά γίνεται :

00000000000000000100000000000000

Η επόμενη εντολή είναι :

camera.cullingMask ^= (layerDeckMask | layerAgentMask)

οπότε έχουμε μια συνολική μάσκα που προέκυψε απ’το | (OR) των δύο μασκών :

00000000000000000100100000000000

Και κάνουμε αυτή την μάσκα ^ (XOR) στην cullingMask που τελικά γίνεται :

11111111111111111011011111111111

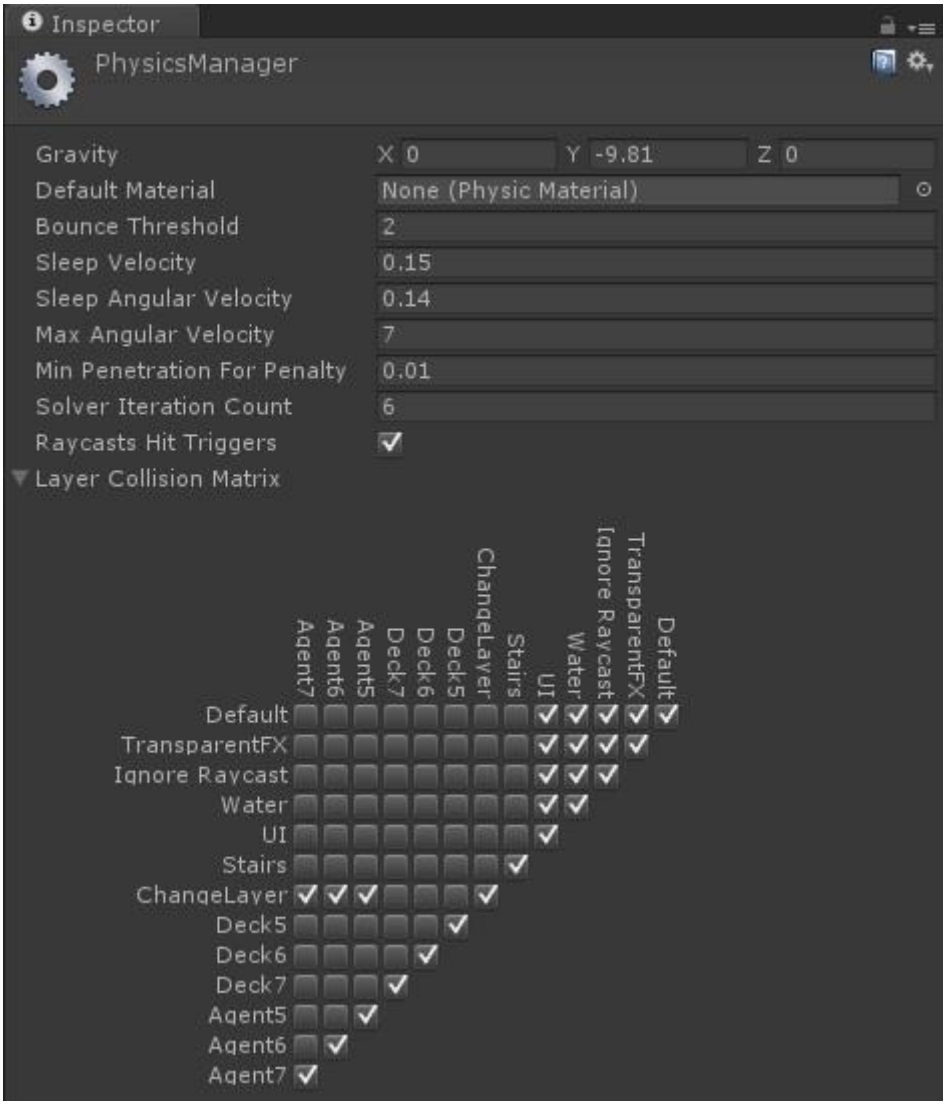
Έτσι τώρα η Main Camera δεν κάνει render τα layers που ανήκουν στα απενεργοποιημένα bit (0), σε αυτήν την περίπτωση τα Layers : Deck6 και Agent6.

Αν ξαναπατήσουμε το πλήκτρο ‘6’ τότε η cullingMask θα ξαναγίνει XOR με την συνολική :

11111111111111111011011111111111
<u>00000000000000000100100000000000</u>
11111111111111111111111111111111

άρα θα γίνουν rendered πάλι όλα τα Layers.

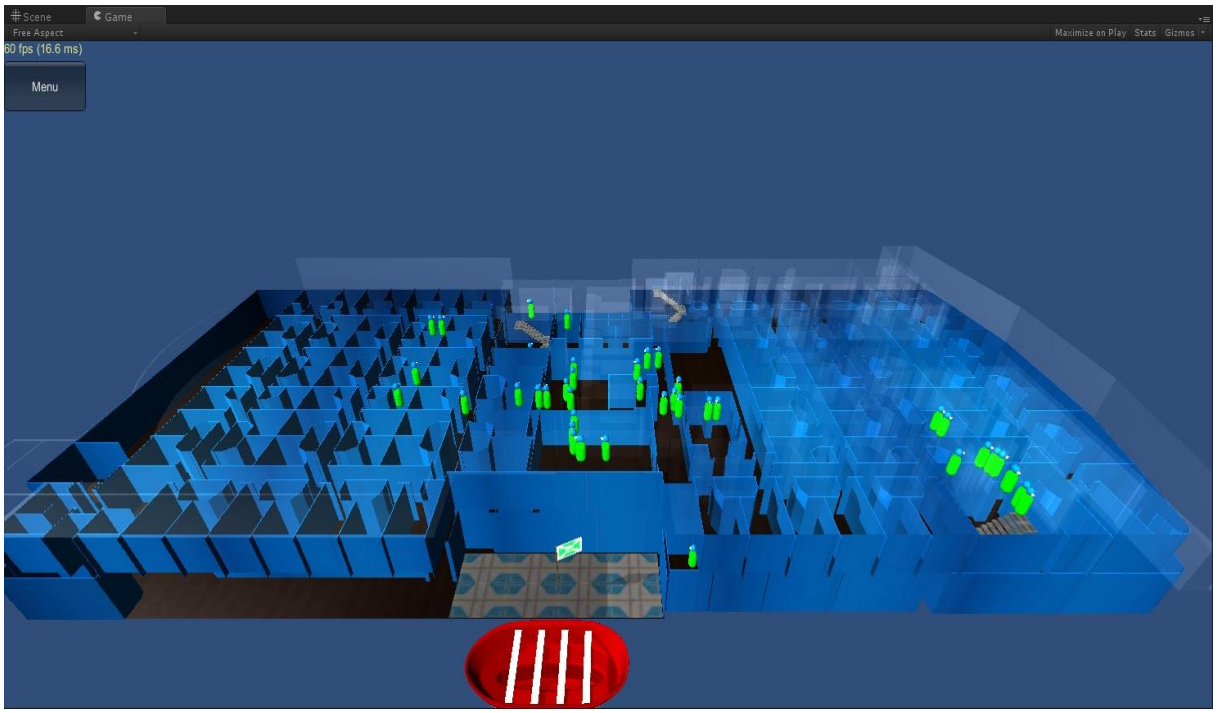
Για να λειτουργήσουν σωστά οι αλλαγές στα Layers που προκαλούν οι άσπροι κύβοι στους Agents όταν αυτοί περνάνε από μέσα τους, θα πρέπει πρώτα να έχουμε ορίσει σωστά στο σύστημα φυσικής ποια Layers προκαλούν συγκρούσεις μεταξύ τους. Αυτό το καθορίζουμε στο Edit => Project Settings => Physics. Είναι το παράθυρο της εικόνας 4.3 :



Εικόνα 4.3 : Μήτρα συγκρούσεων μεταξύ Layers

Μας ενδιαφέρει κυρίως η Layer Collision Matrix, όπου διακρίνονται ποια Layers αλληλεπιδρούν μεταξύ τους και προκαλούν συγκρούσεις. Έτσι, π.χ., οι άσπροι κύβοι με το Layer : ChangeLayer προκαλούν συγκρούσεις και με τα τρία Layers των Agents, τα Agent5, Agent6 και Agent7. Ενώ το ChangeLayer δεν προκαλεί collisions με τα τρία Layers των Decks : Deck5, Deck6 και Deck7. Βέβαια, τόσο στους Agents όσο και στους άσπρους κύβους οι Collisions έχουν ορισθεί ως Trigger, οπότε δεν υπάρχει παρεμπόδιση της κίνησης των Agents, απλά ενεργοποιείται το Event : OnTriggerEnter.

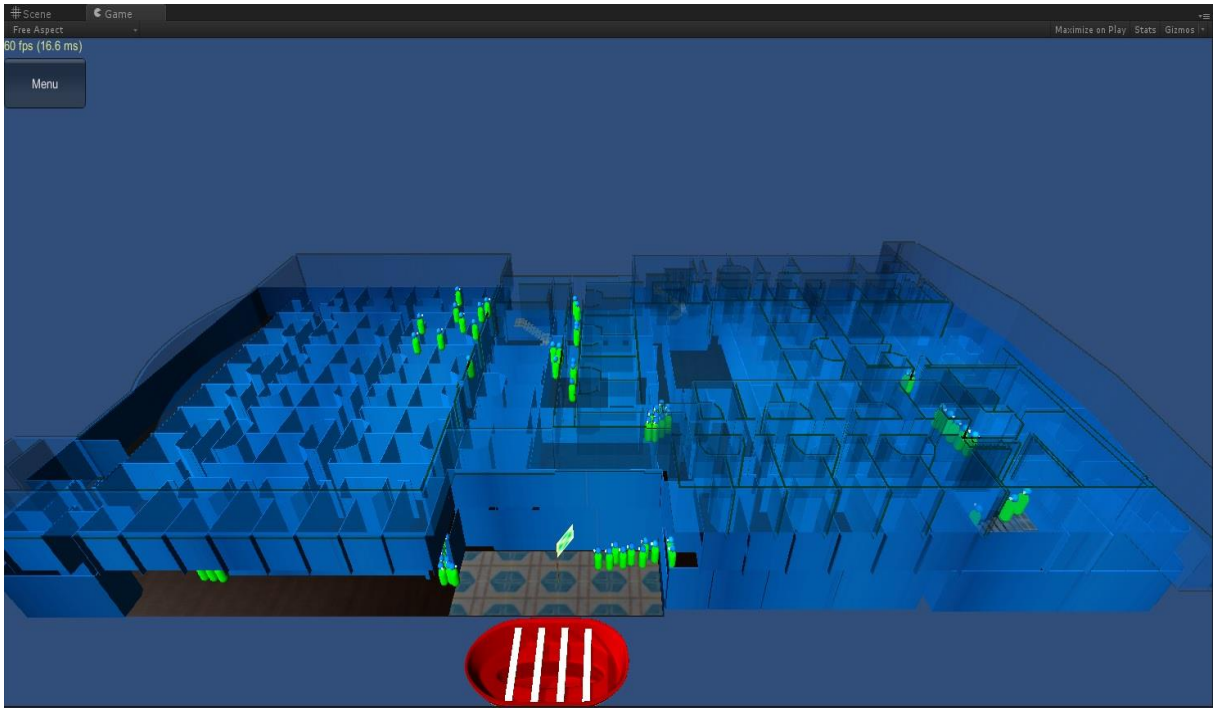
Υπάρχει ένα ακόμα Script (*ShadersControl.cs*) στην Main Camera που αλλάζει τα materials των Decks και των Deck_grounds. Σε κάθε Deck και Deck_ground έχουμε προσθέσει δύο Materials. Έτσι, βάζοντας διαφορετικούς Shaders στα Materials μας, επιτυγχάνουμε διαφορετικούς οπτικούς συνδυασμούς. Στις εικόνες που έχουμε εξετάσει μέχρι τώρα τόσο τα Decks όσο και τα Deck_grounds είναι με τα Materials με τους απλούς Diffuse shaders. Στην παρακάτω εικόνα 4.4, βλέπουμε στα Materilas του Deck7 και του Deck7_ground να υπάρχει ένας custom Shader, Glass, που προσομοιάζει την εμφάνιση του γυαλιού. Το οπτικό αποτέλεσμα είναι ότι διακρίνονται οι Agents πίσω απ'τους τοίχους του Deck7 αλλά και οι Agents κάτω απ'το Deck7_ground δηλαδή όσοι βρίσκονται στο Deck6 :



Εικόνα 4.4 : Deck7 και Deck7_ground με Glass Shader

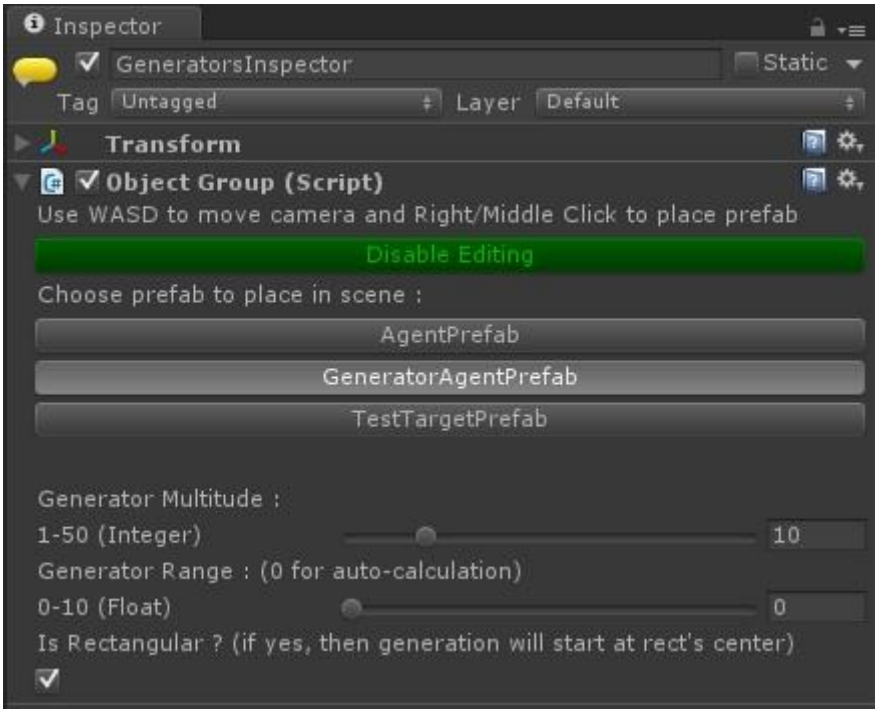
Άλλο ένα παράδειγμα φαίνεται στην εικόνα 4.5 όπου έχουμε χρησιμοποιήσει ένα συνδυασμό δύο Shaders στα Deck7 και Deck7_ground. Ο πρώτος είναι ένας Transparent-Diffuse με τιμή Alpha (η οποία ορίζει το ποσοστό της διαφάνειας) 80 στα 256. Ο δεύτερος είναι ένας ακόμα custom Shader, Outline, που τονίζει με μια μαύρη γραμμή τα περιγράμματα των Meshes στα οποία έχει εφαρμοσθεί.

Οι συνδυασμοί των Shaders εναλλάσσονται πατώντας τα πλήκτρα '1', '2' και '3' για τα Deck5/Deck5_ground, Deck6/Deck6_ground και Deck/Deck7_ground αντίστοιχα.



Εικόνα 4.5 : Deck7 και Deck7_ground με Transparent και Outline Shaders

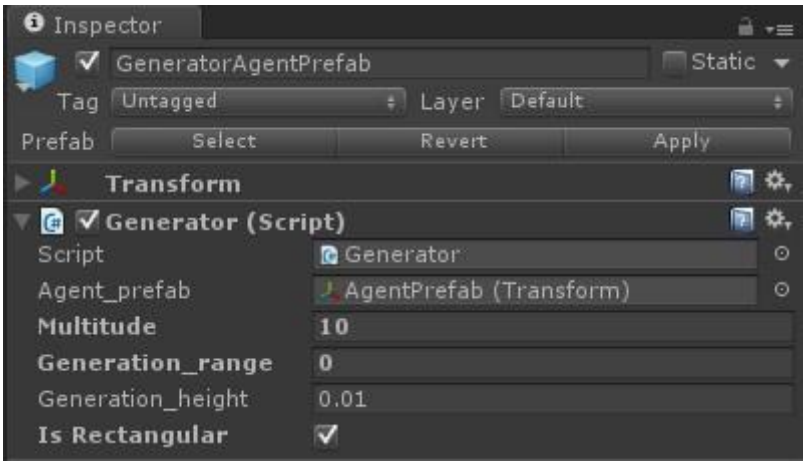
Στην κεντρική σκηνή υπάρχει και ένα αντικείμενο που ονομάζεται GeneratorsInspector. Αυτό έχει ως Component ένα Editor Script (*ObjectGroupInspector.cs*). Τα editors Scripts πρέπει να βρίσκονται αναγκαστικά στον φάκελο Editor και δεν λειτουργούν στο runtime. Η χρησιμότητά τους περιορίζεται μόνο όταν το Unity είναι σε edit mode. Το συγκεκριμένο Script μας βοηθάει να τοποθετούμε γρήγορα Prefabs (δηλαδή έτοιμα Game Objects) στη σκηνή μας. Το χρησιμοποιούμε κυρίως για την τοποθέτηση Prefabs : GeneratorAgent, δηλαδή "γεννητριών" Agents. Ενώ είμαστε σε edit mode στο Unity, ενεργοποιούμε το Editor Script με το πρώτο του button και γίνεται πράσινο. Τότε, μπορούμε να επιλέξουμε ποιο Prefab θέλουμε να τοποθετήσουμε στη σκηνή. Το Script ψάχνει το φάκελο Prefabs και όσα βρει τα παρουσιάζει στη λίστα. Έτσι, επιλέγουμε το Prefab : GeneratorAgent και μεταβάλλουμε τις παραμέτρους που βρίσκονται από κάτω. Τοποθετείται στη θέση του mouse cursor με δεξί click. Το interface του Editor Script μας φαίνεται στην εικόνα 4.6 :



Εικόνα 4.6 : Editor Script για τοποθέτηση Prefab : GeneratorAgent

Η παράμετρος Generator Multitude ορίζει πόσους Agents θα δημιουργήσει ο GeneratorAgent όταν ξεκινήσει η εφαρμογή. Η επόμενη παράμετρος Generator Range ορίζει την ακτίνα του κύκλου (αν η τελευταία παράμετρος IsRectangular είναι false) μέσα στον οποίο θα δημιουργηθούν οι Agents. Αν επιλέξουμε ως Generator Range = 0 και η τελευταία παράμετρος IsRectangular είναι true, τότε ο GeneratorAgent "εκτοξεύει" τέσσερεις ακτίνες, προς βορρά/νότο/ανατολή/δύση και καταμετρά τις αποστάσεις των κοντινότερων τοίχων που σταματάνε τις ακτίνες. Έπειτα μεταφέρει τον εαυτό του στο κέντρο του παραλληλογράμμου που έχει σχηματίσει από αυτές τις απόστασεις και δημιουργεί τους Agents σε τυχαίες θέσεις μέσα σε αυτό το παραλληλόγραμμο.

Ας δούμε και το ανάλογο Script πάνω στο GeneratorAgentPrefab που αφού κληρονομήσει τις παραμέτρους απ'το Editor Script επιτελεί τις προαναφερθείσες ενέργειες για τη δημιουργία των Agents :

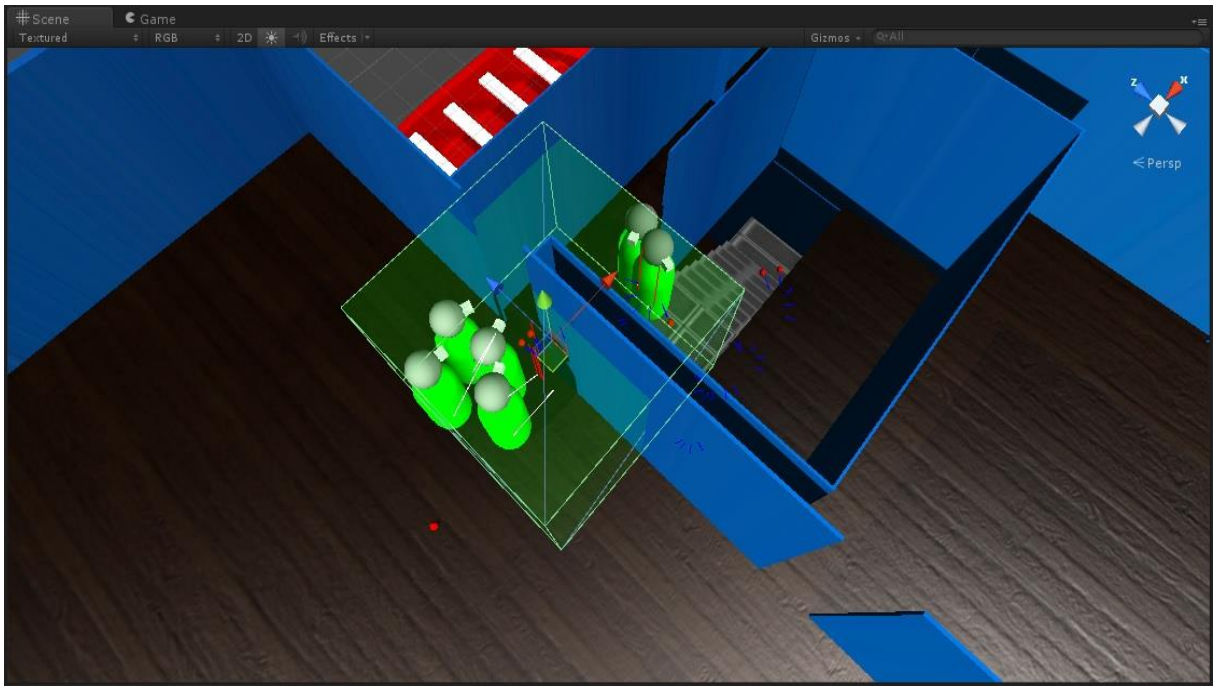


Εικόνα 4.7 : Generator Script για δημιουργία Agents

Προφανώς, μπορούμε να επιλέξουμε το AgentPrefab στο Editor Script και με το δεξί click να τοποθετούμε έναν Agent σε συγκεκριμένη θέση. Οι παράμετροι από κάτω δε λαμβάνονται υπόψη, καθώς το Editor Script ελέγχει αν το επιλεγμένο Prefab έχει πάνω του ένα Generator Script, ή όχι, ώστε να του περάσει τις τιμές των παραμέτρων.

Έτσι, έχουμε ένα γρήγορο τρόπο να δημιουργούμε GeneratorAgent αντικείμενα σε διάφορες θέσεις τα οποία με την σειρά τους θα παράξουν συγκεκριμένα πλήθη Agents.

Στην κεντρική σκηνή υπάρχουν και μερικά Game Objects με όνομα CongestionArea που έχουν τοποθετηθεί σε συγκεκριμένες θέσεις στα καταστρώματα. Επειδή η λειτουργία τους είναι να καταμετρούν την κυκλοφοριακή συμφόρηση, τοποθετούνται σε περιοχές που αναμένουμε να προκύψουν προβλήματα συμφόρησης, όπως σε εισόδους σε περιοχές σκαλών. Πρόκειται για Trigger Colliders που με το Script : *Congestion.cs* μετράνε πόσοι Agents βρίσκονται μέσα στον όγκο τους. Ο μετρητής αυξάνεται όταν υπάρχει ένα Event : *OnTriggerEnter*, δηλαδή είσοδος Agent, και μειώνεται όταν υπάρχει το Event : *OnTriggerExit*, δηλαδή όταν καποιος Agent εξέρχεται. Στο τέλος της προσομοίωσης, μπορεί ο χρήστης να αποθηκεύσει τις τιμές της συμφόρησης καθ'όλη τη διάρκεια της εκκένωσης. Έτσι, οι μετρήσεις της τιμής Congestion καταμετρώνται με συχνότητα checkFrequency και αποθηκεύονται σε μια λίστα CongestionList. Επίσης κατακρατάται και η μέγιστη τιμή που απέκτησε η τιμή Congestion. Η διαδικασία αποθήκευσης αναλύεται περισσότερο στην §4.4. Στην παρακάτω εικόνα διακρίνεται μια Congestion Area :

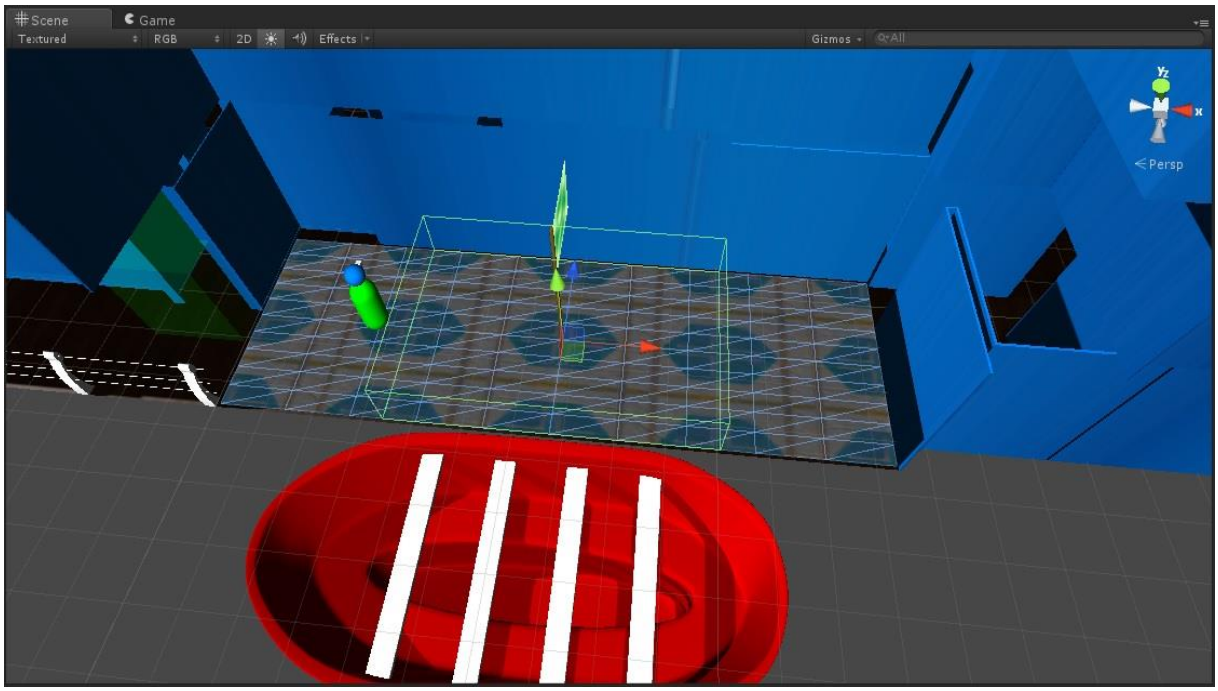


Εικόνα 4.8 : Congestion Area

Τα τελευταία αντικείμενα της κεντρικής σκηνής είναι τα δύο Game Objects : Muster Station. Πρόκειται για τους σταθμούς συγκέντρωσης προς τους οποίους κατεθύνονται όλοι οι Agents. Όπως και οι Congestion Areas, έτσι και οι Muster Station είναι ουσιαστικά μόνο Trigger Colliders (με το Script : *MusterStationCollider.cs*). Η χρησιμότητά τους είναι διπλή (αναλύονται περισσότερο στην §4.4) :

a. Να καταστρέφουν τους Agents όταν αυτοί τους αγγίζουν. Αυτό γίνεται για να μην υπάρχει μεγάλη συγκέντρωση Agents σε μικρό χώρο που συνεπάγεται μείωση της απόδοσης της εφαρμογής.
b. Να ενημερώσει τον AgentManager να ελέγξει αν ο Agent, που μόλις μπήκε στον χώρο του και καταστράφηκε ήταν ο τελευταίος κινούμενος Agent. Αν ναι, τότε πρέπει να μεταφερθούμε στην κατάσταση όπου έχει ολοκληρωθεί η προσομοίωση της εκκένωσης.

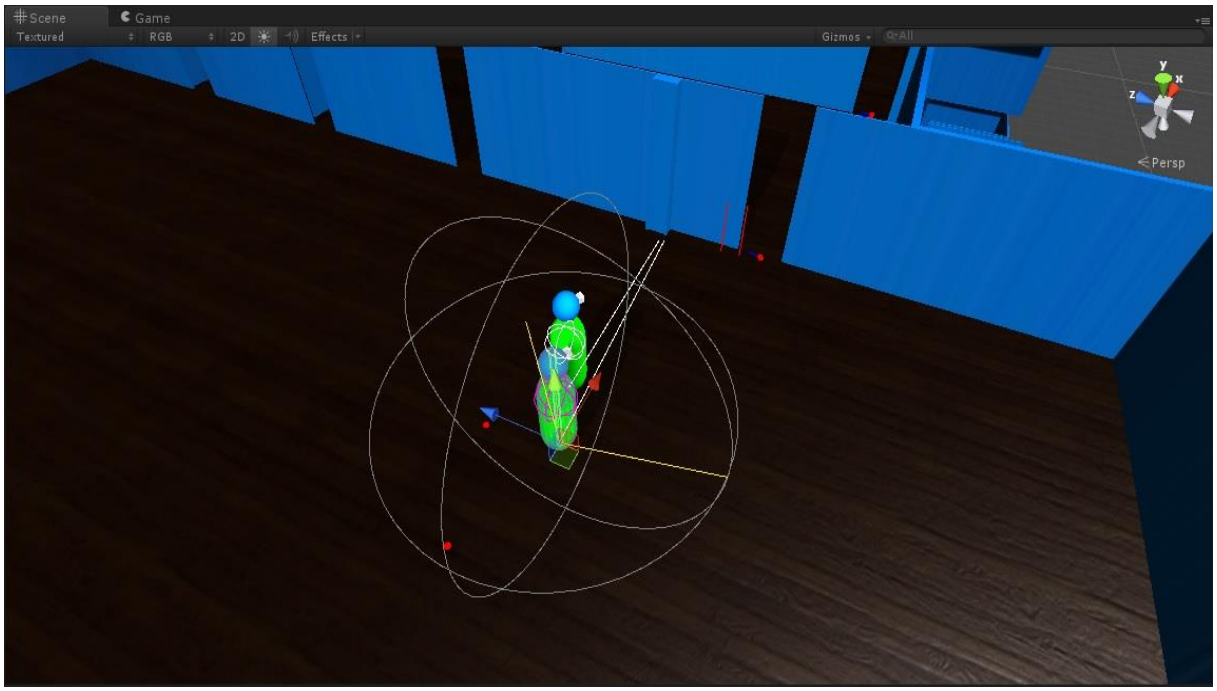
Στην εικόνα 4.9 φαίνεται ο Trigger Collider του Muster Station :



Εικόνα 4.9 : Muster Station

4.2 Ανάλυση Agent

Σε αυτή την ενότητα θα αναλύσουμε σε μεγαλύτερο βάθος τις λειτουργίες που εκτελεί τόσο ο Agent όσο και ο AgentManager που ελέγχει όλους τους Agents στη σκηνή. Στην εικόνα 4.10 έχουμε ξεκινήσει την εφαρμογή με το Play, την έχουμε σταματήσει προσωρινά με το Pause και βρισκόμαστε στο παράθυρο Scene. Στο Script του Agent (*Agent.cs*) έχουμε προσθέσει κατάλληλες Debug εντολές (όπως *Debug.DrawLine*, *Gizmos.DrawWireSphere* και *Gizmos.DrawRay* ώστε να φαίνονται περισσότερες πληροφορίες στο παράθυρο Scene. Οι πληροφορίες αυτές θα μας βοηθήσουν στην καλύτερη κατανοήση των ενεργειών που επιτελούν οι Agents. Τονίζουμε ότι αυτές οι οπτικές πληροφορίες δε φαίνονται στο παράθυρο Game ή όταν τρέχουμε το εκτελέσιμο. Στην εικόνα 4.10 είναι επιλεγμένος ο Agent που είναι πιο χαμηλά στην οθόνη :



Εικόνα 4.10 : Debug πληροφορίες στο παράθυρο Scene

Όταν δημιουργείται ο Agent επιτελεί τις παρακάτω αρχικοποιήσεις :

- Εγγράφεται σε μια λίστα με όλους τους Agents την οποία διατηρεί ο AgentManager.
- Ελέγχει το ύψος που βρίσκεται και το συγκρίνει με τα ύψη (δηλαδή τη συντεταγμένη y) των Decks, ώστε να πάρει το Layer του σωστού Deck.
- Εάν η boolean μεταβλητή του, getValuesFromManager, είναι true τότε θα πάρει τις τιμές των παραμέτρων κίνησης, ανίχνευσης και υγείας απ'τον AgentManager. Αλλιώς, θα χρησιμοποιήσει τις δικές του προσωπικές τιμές.
- Ελέγχει αν έχει κάποια θέση-στόχο. Αν ναι , τότε κατευθύνεται προς τα εκεί. Αν όχι, τότε "κοιτάζει" την boolean μεταβλητή του, knowWayToMuster, και αν είναι true τότε θα φτιάξει ένα path προς τον πλησιέστερό του Muster Station. Αλλιώς, θα

φτιάξει path προς τη κοντινότερή του πινακίδα για να πάρει περαιτέρω πληροφορίες για την πορεία του.

Επίσης, ξεκινά την περιοδική κλήση των functions (θα τις εξηγήσουμε αμέσως παρακάτω) :

- *DetectNeighbors* : καλείται με συχνότητα detectionFrequency.
- *CheckIfStuck* : καλείται με συχνότητα checkIfStuckFrequency.
- *RecordPath* : καλείται με συχνότητα recordPathFrequency.
- *CheckIfOnStairs* : καλείται με συχνότητα 2 φορές το δευτερόλεπτο

Η *DetectNeighbors* εκτελεί την function του ενσωματωμένου στο Unity συστήματος φυσικής :

Physics.OverlapSphere()

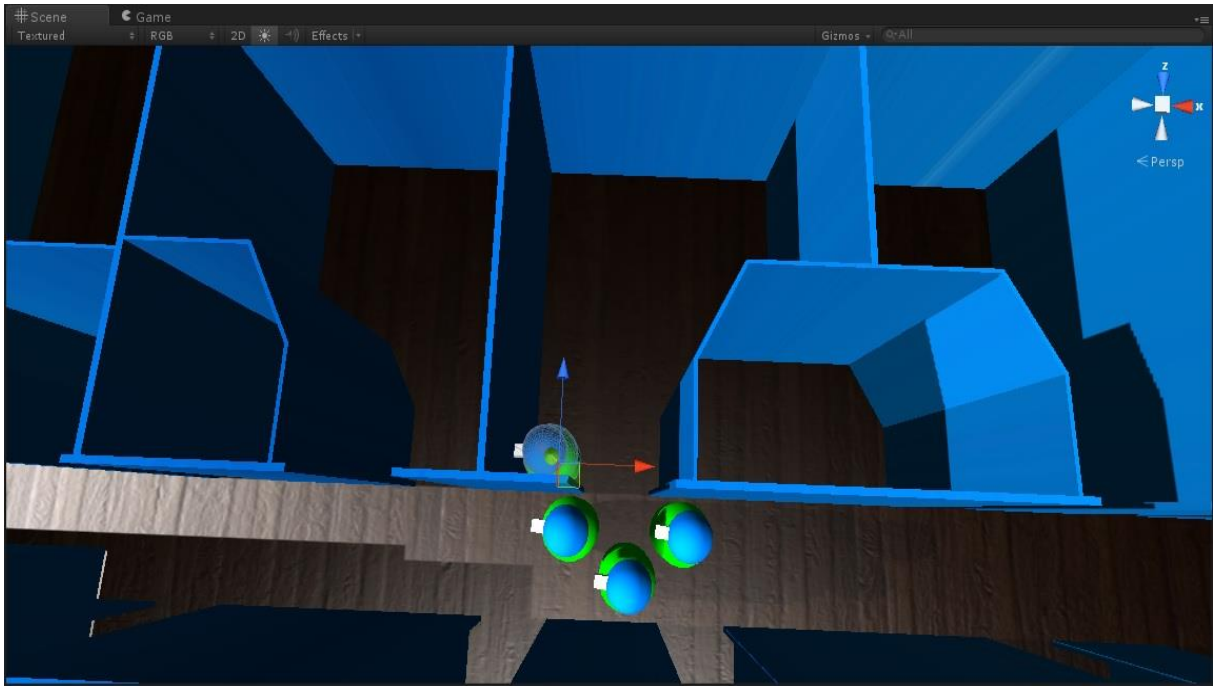
Αυτή δημιουργεί μια σφαίρα με κέντρο τη θέση του Agent και ακτίνα *detectionRadius* και μας επιστρέφει όσους Colliders υπάρχουν μέσα της. Η σφαίρα απεικονίζεται με γκρι χρώμα στην εικόνα 4.10. Χρησιμοποιώντας τα σωστά Layers και μάσκες, επιτυγχάνουμε η *OverlapSphere* να επιστρέφει μόνο Agents και όχι άλλα αντικείμενα, όπως π.χ. τοίχοι/πατώματα, και συγκεκριμένα Agents που είναι στο ίδιο Deck. Για να εξυπηρετηθεί αυτή η function, οι Agents μας έχουν τα δύο Components φυσικής : Rigidbody και Collider (Capsule). Ας επισημάνουμε εδώ ότι το Rigidbody έχει χαρακτηρισθεί ως Kinematic και ο Collider ως Trigger, με αποτέλεσμα τα σώματα να μην απωθούνται μεταξύ τους απ’το σύστημα φυσικής. Την απώθηση την προσομοιώνουμε εμείς με τη συμπεριφορά (§3.2.2.4) PushAgent όταν βρεθούν πολύ κοντά. Η ακτίνα PushRadius, απ’την οποία εξαρτάται η PushAgent εμφανίζεται ως μια μωβ σφαίρα ανάλογης ακτίνας στο κέντρο του Agent.

Αφού η *OverlapSphere* μας επιστρέψει όλους τους Colliders που βρει μέσα της, η *DetectNeighbors* αντιστοιχίζει αυτούς τους Colliders με τους Agents στους οποίους ανήκουν, και στη συνέχεια αποθηκεύει τους Agents σε μια λίστα, detectedAgents. Δηλαδή, με αυτή τη διαδικασία ο κάθε Agent καταλήγει να έχει μια λίστα με τους γειτονικούς του Agents. Αυτή τη λίστα θα επεξεργασθεί μετά ο AgentManager στις συμπεριφορές PushAgent και AvoidNearest. Η AvoidNearest αποφεύγει τον πλησιέστερο Agent μέσα στο FOV. Το FOV απεικονίζεται με τις δύο κίτρινες γραμμές.

Η *CheckIfStuck* εκτελείται κάθε ένα δευτερόλεπτο (μια τιμή της checkIfStuckFrequency που θεωρήθηκε πρακτική) και καταχωρεί σε μια μεταβλητή την απόσταση του Agent απ΄το αμέσως επόμενο σημείο του path του. Έπειτα ελέγχει αν η τωρινή απόσταση και η απόσταση που είχε καταχωρηθεί στην προηγούμενη κλήση της function διαφέρουν κατά έναν μικρό αριθμό. Αν διαφέρουν, τότε ο Agent προχωράει κανονικά κατά μήκος του path του. Αν, όμως, δε διαφέρουν, αυτό σημαίνει ότι ο Agent στο χρονικό διάστημα ενός δευτερολέπτου έχει μετακινηθεί πολύ λίγο, οπότε ίσως έχει “κολλήσει” σε κάποια γωνία. Αυτό συμβαίνει

μερικές φορές λόγω των απωθητικών δυνάμεων που ασκούνται μεταξύ των Agents. Σε αυτή την περίπτωση, ο Agent φτιάχνει νέο path προς τον στόχο του και έτσι “ξεκολλάει”.

Βλέπουμε ένα παράδειγμα στην εικόνα 4.11, όπου ο επιλεγμένος Agent έχει σπρωχθεί και “κολλήσει” στην γωνία :



Εικόνα 4.11 : Παράδειγμα “stuck” Agent

Η *RecordPath* εκτελείται μόνο εάν η recordPath μεταβλητή του Agent είναι true. Τότε, με συχνότητα recordPathFrequency ο Agent καταγράφει τη θέση του σε μια λίστα, recordedPath. Στο τέλος της προσομοίωσης, μπορούμε με το Component : Line Renderer του Agent να ενώσουμε όλα τα σημεία της λίστας με ευθεία γραμμικά τμήματα. Έτσι, ουσιαστικά δημιουργείται μια συνεχόμενη γραμμή που απεικονίζει την πορεία που διένυσε ο Agent απ’την αρχική του θέση μέχρι τον Muster Station.

Η *CheckIfOnStairs* χρησιμοποιεί την *SamplePathPosition* του NavMeshAgent Component της :

navAgent.SamplePathPosition (-1, 0.0f, out stairsHit)

και ελέγχει τη θέση του Agent μέσα στο NavMesh. Στη παράμετρο stairsHit αποθηκεύεται το αποτέλεσμα του ελέγχου και επειδή είναι out, δηλαδή λειτουργεί ως pass by reference, αν η function *SamplePathPosition* της αλλάξει τιμή, η καινούργια τιμή θα παραμείνει και μετά το τέλος της κλήσης της function. Έτσι, κάνοντας μετά binary AND (&) το stairsHit.mask με τη μάσκα του Layer της σκάλας μπορούμε να βρούμε αν ο Agent περπατά πάνω σε σκάλες ή σε επίπεδο έδαφος.

Αν περπατά σε επίπεδο έδαφος, τότε θέτουμε τη μεταβλητή :

maxSpeed = maxSpeedFlat

Αν περπατά σε σκάλα, τότε θέτουμε τη μεταβλητή :

maxSpeed = maxSpeedStairsUp
ή
maxSpeed = maxSpeedStairsDown

αναλόγως αν ανεβαίνει ή κατεβαίνει τη σκάλα αντίστοιχα. Η αναγνώριση αν ο Agent ανεβαίνει ή κατεβαίνει γίνεται συγκρίνοντας το ύψος κάποιου επόμενου σημείου του path με το ύψος της συγκεκριμένης του θέσης.

Τη μεταβλητή maxSpeed χρησιμοποιεί έπειτα ο AgentManager για να κινήσει τον κάθε Agent. Εδώ τελειώνει η αρχικοποίηση των περιοδικών συναρτήσεων που καλεί ο Agent.

Έπειτα, ο Agent ελέγχει την τιμή της boolean μεταβλητής knowWayToMuster. Αν αυτή είναι true, τότε θεωρείται ότι ο Agent ξέρει που βρίσκονται όλοι οι Muster Stations, οπότε ψάχνει τις αποστάσεις απ’τον καθένα και τελικά φτιάχνει path προς τον πλησιέστερό του με την function :

navAgent.CalculatePath()

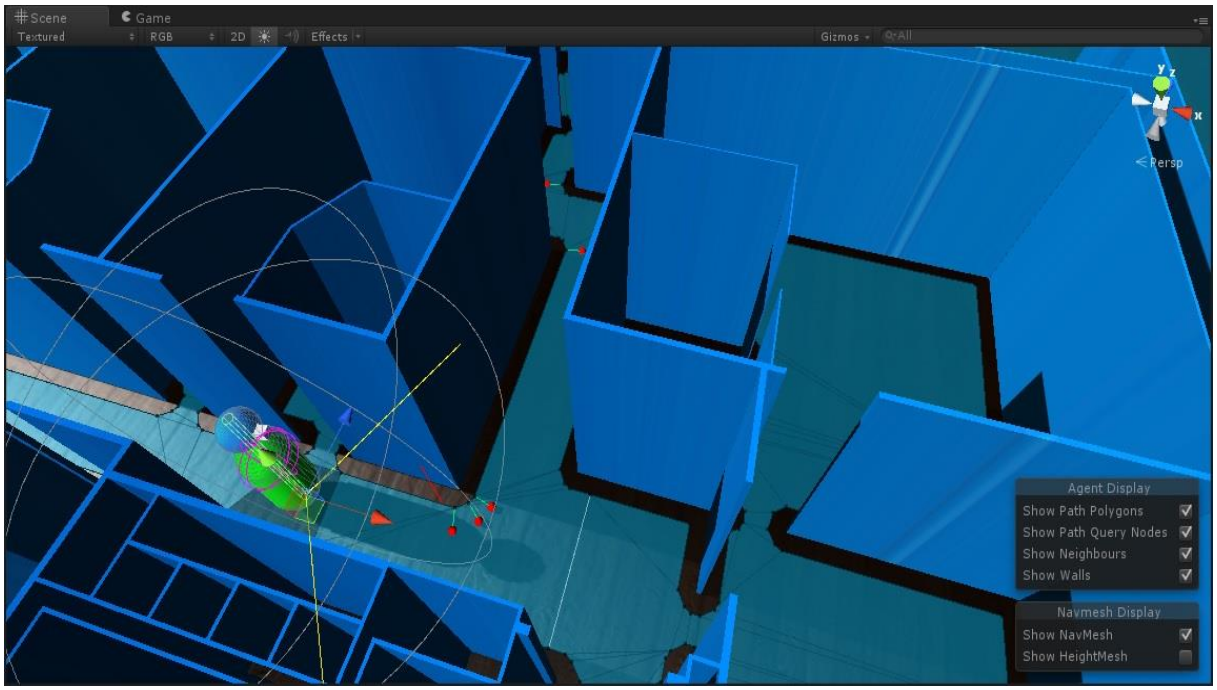
Αν είναι false, τότε θεωρείται ότι δε ξέρει που βρίσκονται οι Muster Stations αλλά ξέρει που βρίσκονται οι πινακίδες κυκλοφορίας-οδηγιών και ομοίως φτιάχνει path προς την κοντινότερη πινακίδα όπου θα πάρει πληροφορίες για να φτιάξει νέο path είτε προς Muster Station είτε προς επόμενη πινακίδα. Για να μπορέσει ο Agent να χρησιμοποεί την function *CalculatePath*, πρέπει να έχει το Component : *NavMeshAgent*.

Όταν εκτελεσθεί η *CalculatePath*, μας επιστρέφεται ένα path, τύπου NavMeshPath, προς τον στόχο που θέλει να πάει ο Agent. Αυτό το path το επεξεργαζόμαστε λίγο με την function :

OffsetCorners()

Βλέπουμε ένα στιγμιότυπο στην εικόνα 4.12 όπου έχουμε και ενεργοποιημένο το παράθυρο Navigation, ώστε να βλέπουμε το δημιουργημένο (Baked) NavMesh. Είναι το γαλάζιο επίπεδο πάνω απ’το πάτωμα.

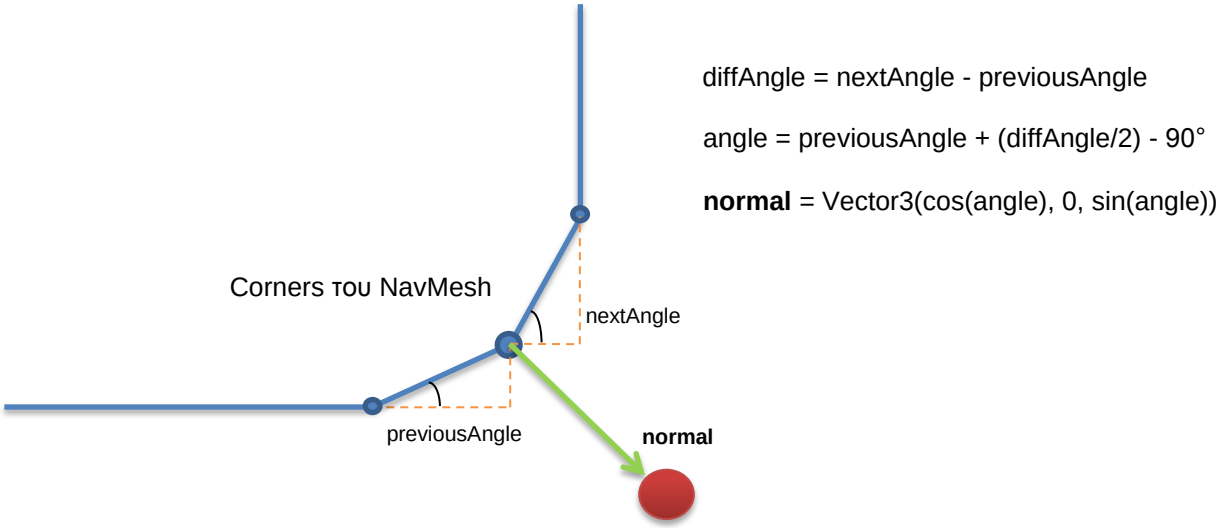
Θα εξηγήσουμε τι κάνει η *OffsetCorners* με τη βοήθεια και της εικόνας 4.13 :



Εικόνα 4.12 : Function : *OffsetCorners*

Έχουμε χρησιμοποιήσει και μια βοηθητική function, *DrawPathSpheres*, που δημιουργεί μικρές κόκκινες σφαίρες στα σημεία που αποτελούν το τελικό path του Agent. Βέβαια, όλα αυτά φαίνονται μόνο στο edit mode του Unity. Στο κανονικό runtime, διακρίνεται μόνο ο Agent να περπατά στο Deck.

Η *CalculatePath* επιστρέφει ένα path όπου τα σημεία του, corners, είναι ακριβώς πάνω στα άκρα του NavMesh, δηλαδή στην εικόνα 4.12 στα βόρειο άκρα των πράσινων ευθύγραμμων τμημάτων. Όμως, έτσι, η κίνηση των Agents δεν φαίνεται πολύ φυσική γιατί κινούνται συνέχεια στα άκρα του NavMesh. Επίσης, όλοι οι Agents όταν έχουν τον ίδιο προορισμό, ακολουθούν ακριβώς τα ίδια σημεία – corners. Η *OffsetCorners* για το κάθε εσωτερικό (δηλαδή όχι το πρώτο και όχι το τελευταίο) σημείο του path υπολογίζει τις γωνίες που σχηματίζει με το προηγούμενο και το επόμενό του. Έπειτα βρίσκει τη διαφορά τους και τελικά την μεσοκάθετο που σχηματίζουν τα δύο ευθύγραμμα τμήματα (εικόνα 4.13). Με βάση τη μεσοκάθετο γίνεται ένας έλεγχος, function *NavMesh.Raycast*, αν σε μια μικρή απόσταση απ’το εξεταζόμενο σημείο υπάρχει ή όχι επιτρεπόμενη περιοχή περπατήματος (walking area) του NavMesh. Αν δεν υπάρχει, τότε απλά αντιστρέφουμε το μοναδαίο διάνυσμα **normal** που σχηματίζεται απ΄τη μεσοκάθετο.



Εικόνα 4.13 : Επεξήγηση Function : *OffsetCorners*

Έτσι, σε κάθε περίπτωση, έχουμε ένα διάνυσμα που δείχνει προς τη σωστή μεριά του NavMesh όπου μπορούν να περπατάνε οι Agents. Στο άκρο του **normal** δημιουργείται το τελικό σημείο απ'το οποίο θα περάσει ο Agent. Όλα τα τελικά σημεία που προκύπτουν απ'την *OffsetCorners* τα αποθηκεύουμε σε μια νέα λίστα, activePath, που είναι και το πραγματικό path που ακολουθούν οι Agents. Φτιάχνουμε και τις κόκκινες σφαίρες στα τελικά σημεία για την δική μας οπτική βοήθεια.

Επομένως, το αποτέλεσμα είναι ότι η κίνηση των Agents φαίνεται πιο φυσιολογική καθώς περπατούν περίπου στο μέσο του διαδρόμου και όχι στις άκρες. Επίσης, πολλαπλασιάζοντας το **normal** με κάποιο τυχαίο αριθμό μέσα σε κάποιο range, επιτυγχάνουμε οι πορείες των Agents να διαφέρουν κατά ένα μικρό ποσοστό και η κίνηση τους να μην είναι ακριβώς ίδια.

Μετά τις αρχικοποιήσεις και τη δημιουργία του activePath του, ο Agent εισέρχεται στη κεντρική loop που "τρέχει" συνέχεια και εκτελεί τα βήματά της μια φορά το frame. Αυτά είναι :

1. Αλλαγή της τιμής της Finite State Machine (FSM) αναλόγως που βρίσκεται.
2. Έλεγχος πόσο κοντά βρίσκεται στο αμέσως επόμενο σημείο του activePath.
3. Έλεγχος αν μπορεί να κατευθυνθεί στο δεύτερο επόμενο σημείο του activePath.
4. Έλεγχος για την πλησιέστερη NavMeshEdge.

Ας τις αναλύσουμε μία μία περισσότερο :

1. Ο Agent ελέγχει αν έχει activePath, δηλαδή κάποια πορεία που πρέπει να ακολουθήσει.

Αν δεν έχει, τότε θέτει την FSM, η οποία χρησιμοποιεί ένα enumeration *AIStatus{ Seek = 0, Arrive = 1, Idle = 2}* και απλά καταγράφει σε ποια κατάσταση βρίσκεται ο Agent, στην τιμή Idle, οπότε ο Agent δεν κινείται.

Αν έχει activePath και αν βρίσκεται σε απόσταση μικρότερη από slowingDistance απ'το τελευταίο σημείο του activePath, τότε η FSM παίρνει την τιμή Arrive και ο Agent θα χρησιμοποιήσει την συμπεριφορά Arrive (§3.2.2.2) για να κινηθεί.

Αν έχει activePath και δεν βρίσκεται σε απόσταση μικρότερη από slowingDistance απ'το τελευταίο σημείο του activePath, τότε η FSM παίρνει την τιμή Seek και ο Agent θα χρησιμοποιήσει την συμπεριφορά Seek (§3.2.2.1) για να κινηθεί.

2. Ο Agent ελέγχει την απόστασή του απ'το αμέσως επόμενο σημείο του activePath, που είναι και πάντα το πρώτο σημείο στη λίστα (List<Vector3>) που αντιστοιχεί στο activePath.

Αν αυτό είναι ενδιάμεσο, τότε χρησιμοποιεί την τιμή της μεταβλητής intermediateStoppingDistance και αν το έχει πλησιάσει σε τόση απόσταση θεωρεί ότι το "έφτασε" και το αφαιρεί απ'το activePath. Μόλις αφαιρεθεί το πρώτο σημείο της List, αμέσως αυτή αυτόματα μειώνει τις θέσεις όλων των υπολοίπων σημείων και το προηγούμενο δεύτερο σημείο γίνεται τώρα πρώτο και ο Agent αρχίζει και κατευθύνεται προς αυτό.

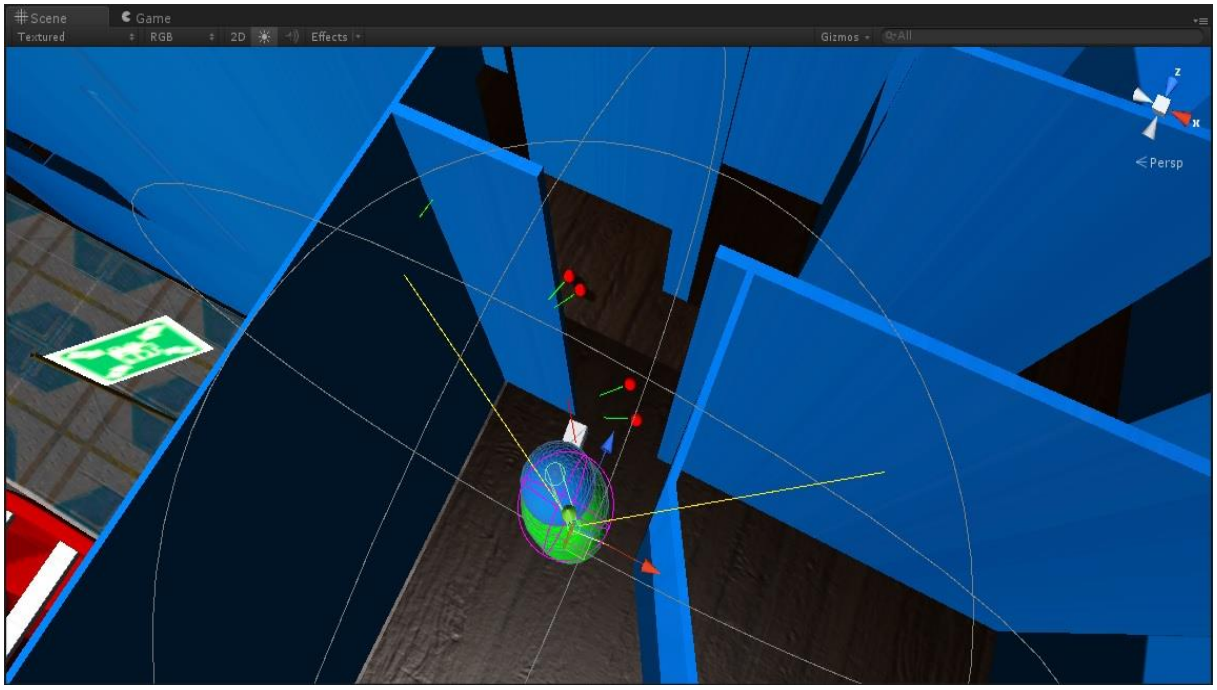
Αν το σημείο είναι το τελευταίο του activePath, τότε χρησιμοποιεί την τιμή της μεταβλητής stoppingDistance και αν το έχει πλησιάσει σε τόση απόσταση θεωρεί ότι "έφτασε" στον τελικό του προορισμό και καλεί την function *OnArrived*. Σε αυτήν μπορούμε να βάλουμε ότι εντολές θέλουμε να εκτελεί ο Agent όταν τελειώνει την πορεία του.

3. Ο Agent "στέλνει" μια αόρατη ακτίνα :

NavMesh.Raycast(position, activePath[1], out hit, Int32.MaxValue);

Η ακτίνα σημαδεύει το δεύτερο σημείο του activePath (η μέτρηση στις Lists και Arrays ξεκινάει από μηδέν). Αν δεν υπάρχει κάποιο εμπόδιο ανάμεσα στον Agent και το σημείο, τότε ο Agent μπορεί να αφαιρέσει το πρώτο σημείο του activePath και να αρχίσει να κινείται κατευθείαν προς το δεύτερο (το οποίο μετά την αφαίρεση θα έχει γίνει πρώτο πλέον).

Αυτό το τέχνασμα βοηθά τους Agents να περνούν κλειστές γωνίες της τοπολογίας, όπως π.χ. στους στενούς διαδρόμους. Στην εικόνα 4.14 φαίνεται ότι ο Agent έχει στείλει μια ακτίνα προς το πρώτο σημείο του activePath και δεν έχει βρει εμπόδιο, επομένως κατευθύνεται προς το δεύτερο. Το επόμενο frame στέλνει ακτίνα προς το επόμενο σημείο, όμως αυτή τη φορά υπάρχει ο τοίχος ανάμεσα, οπότε σχηματίζεται μια κατακόρυφη κόκκινη γραμμή στο σημείο που συγκρούστηκε η ακτίνα με τον τοίχο και επίσης ο Agent δεν αλλάζει κατεύθυνση.



Εικόνα 4.14 : Ο Agent ελέγχει αν μπορεί να συντομεύσει την πορεία του

4. Ο Agent εκτελεί την function :

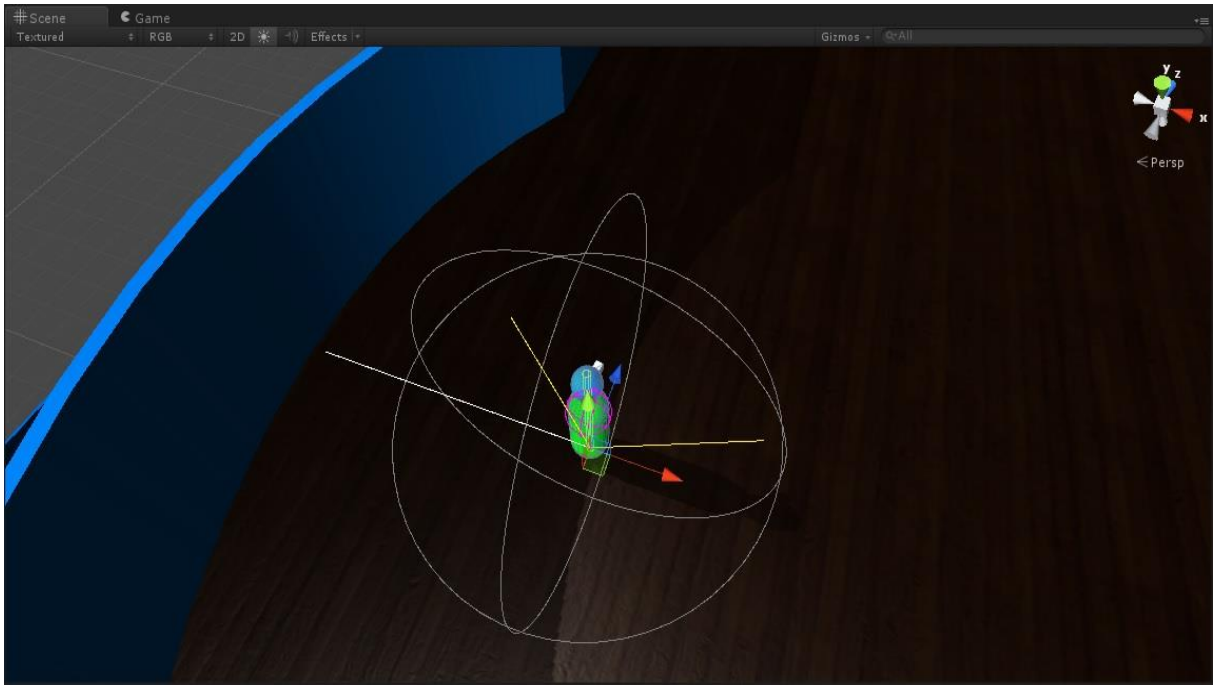
navAgent.FindClosestEdge(out closestNavEdge)

και βρίσκει την πλησιέστερή του άκρη (edge) του NavMesh και την αποθηκεύει στην out παράμετρο closestNavEdge. Η χρήση παραμέτρου out εξηγήθηκε στην ανάλυση της συνάρτησης *CheckIfOnStairs*. Στην closestNavEdge περιέχεται και η πληροφορία της απόστασης της άκρης απ'τον Agent και την αποθηκεύουμε στην μεταβλητή nearestEdgeDistance :

nearestEdgeDistance = closestNavEdge.distance;

Αυτή η απόσταση είναι η άσπρη γραμμή στην εικόνα 4.15.

Ο AgentManager χρησιμοποιεί την μεταβλητή nearestEdgeDistance για να ελέγχει πόσο κοντά στον τοίχο βρίσκεται ο κάθε Agent. Αν η nearestEdgeDistance γίνει μικρότερη της ακτίνας PushRadius του Agent, τότε ο AgentManager, με την συμπεριφορά AvoidWall, προσδίδει ένα steering διάνυσμα που απομακρύνει τον Agent απ'τον τοίχο.



Εικόνα 4.15 : Απόσταση Agent από πλησιέστερη άκρη του NavMesh

Να επισημάνουμε ότι ο Agent έχει δημιουργηθεί ως ένα Empty Game Object στο οποίο έχουμε προσθέσει το Script : Agent.cs, μαζί με όλα τα υπόλοιπα Components, που εκτελεί τις λειτουργίες που μόλις αναφέραμε. Οι γραφικές αναπαραστάσεις του σώματος (ένα primitive Game Objcet : Capsule), του κεφαλιού του (ένα primitive Game Objcet : Sphere) και της μύτης (ένα primitive Game Objcet : Cube) έχουν υλοποιηθεί ως "παιδιά" Game Objects του, οπότε μπορούν να αντικατασταθούν από πολύπλοκα, λεπτομερή, textured meshes κανονικών ανθρώπων χωρίς να επηρεαστεί κατά τ'άλλα η λειτουργία του Agent.

Το χρώμα του κεφαλιού εξαρτάται απ'την ηλικιακή κατηγορία που ανήκει σύμφωνα με τον Πίνακα 3.1. Αν δεν χρησιμοποιήσουμε τον Πίνακα με τις τιμές του ΙΜΟ αλλά θέσουμε εμείς απευθείας είτε με Exact τιμή είτε με Range (εύρος τιμών), τότε το κεφάλι χρωματίζεται γκρι. Επίσης, το πλάτος του σώματος προέκυψαν από εδώ [14].

4.3 Ανάλυση AgentManager

Ο AgentManager δημιουργήθηκε κυρίως απ'την ανάγκη ελέγχου μεγάλου αριθμού Agents.

Στον υπολογιστή που εργαζόμασθαν, η απόδοση της εφαρμογής (μετρούμενη σε Frame Per Second – FPS) άρχιζε να πέφτει όταν στη σκηνή υπήρχαν περισσότεροι από περίπου τετρακόσιους Agents. Τότε, μειωνόταν το FPS με αποτέλεσμα η κίνηση των Agents να φαινόταν μη ομαλή και "σπαστή" καθώς τα frames διαφέραν κατά σημαντικά χρονικά διαστήματα μεταξύ τους.Έτσι, έπρεπε να βρεθεί ένας τρόπος ώστε η εφαρμογή να μπορεί να χειρισθεί έναν μεγάλο αριθμό Agents (π.χ. χίλιους) σε ικανοποιητικό FPS.

Η πρώτη προσπάθεια έγινε χρησιμοποιώντας Coroutines της γλώσσας C#. Η υλοποίησή τους στο Unity επιτρέπει την εκκίνηση μιας function, την παύση της σε κάποιο σημείο και την συνέχισή της απ'το σημείο που είχε σταματήσει σε αργότερο χρόνο. Εμείς το αξιοποιήσαμε ως εξής :

- Ο AgentManager χώριζε τους Agents σε ομάδες (π.χ. των εκατό) και στο 1° frame υπολόγιζε τα Steering διανύσματα για τους πρώτους εκατό. Η Coroutine έκανε παύση σε αυτό το σημείο.
- Στο 2° frame, Η Coroutine θυμόταν που είχε σταματήσει και συνέχιζε υπολογίζοντας για τους επόμενους εκατό.
- Ομοίως, μέχρι να είχαν υπολογισθεί τα Steering διανύσματα για όλους τους Agents, οπότε στο επόμενο frame θα ξανάρχιζε η διαδικασία με τους πρώτους εκατό.

Το μειονέκτημα αυτής της μεθόδου είναι ότι όταν είχαμε μεγάλο αριθμό Agents (π.χ. χίλιους), η πρώτη εκατοντάδα θα ενημερωνόταν για τα Steering διανύσματα της στο 1° frame και η αμέσως επόμενη ενημέρωση θα γινόταν μετά από δέκα frames, στο 11° frame. Άρα θα μεσολαβούσαν δέκα frames στα οποία οι Agents θα κινούνταν με κάποιο Steering διάνυσμα που μπορεί να μην ίσχυε πλέον. Οπότε, οι Agents μπορεί να βρισκόντουσαν μέσα στο χώρο άλλων ή να βρισκόντουσαν πολύ κοντά στους τοίχους χωρίς να έχουν προστεθεί τα διανύσματα που θα τα απέτρεπαν αυτά.

Τελικά, ο τρόπος που χρησιμοποιήσαμε είναι ο παράλληλος προγραμματισμός [15] (Multithreading Programming). Τα νήματα (Threads) που είναι ελαφριές διεργασίες (Processes), δηλαδή μια μικρή ακολουθία εντολών, μπορούν να εκτελούνται σε ξεχωριστούς επεξεργαστικούς πυρήνες (Cores) των σύγχρονων επεξεργαστών (Processors). Έτσι, επιτυγχάνεται η ταυτόχρονη εκτέλεση των νημάτων.

Για την επίτευξη του Multithreading Programming μέσα στο Unity χρησιμοποιήσαμε την free έκδοση της βιβλιοθήκης Loom. Η αρχή της υλοποίησης μοιάζει με αυτήν των Coroutines. Θα την εξηγήσουμε αμέσως παρακάτω :

- Βρίσκουμε τον αριθμό των Cores : *SystemInfo.processorCount* (έστω τέσσερεις).
- Το ιδανικό είναι να φτιάχνουμε ένα Thread ανά Core. Οπότε διαιρούμε το πλήθος των Agents (έστω χίλιους) με τον αριθμό των Cores και ο AgentManager, σε κάθε frame, θα δημιουργεί τέσσερα Threads με το καθένα να "αναλαμβάνει" 1000/4 = 250 Agents.
- Τα τέσσερα Threads θα "τρέχουν" ταυτόχρονα και μόλις ένα από αυτά τελειώσει με τους υπολογισμούς των Steering διανυσμάτων των Agents του θα αυξάνει έναν μετρητή κατά ένα.
- Όταν αυτός ο μετρητής φτάσει το πλήθος των Cores (= πλήθος των Threads) τότε ξέρουμε ότι έχουν τελειώσει οι υπολογισμοί όλων των Agents και μπορούμε στην *Update()* του AgentManager να εφαρμόσουμε τα Steering διανύσματα στις κινήσεις των Agents.
- Μηδενίζουμε τον μετρητή ώστε να είναι επόμενος για το επόμενο frame.
- Τα παραπάνω τέσσερα βήματα επαναλαμβάνονται στην *Update()* του AgentManager μέχρι το τέλος της εφαρμογής.

Το Multithreading ξεκινά σε κάθε frame στην *Update()* όπου καλείται η function :

RunAsync()

Αυτή διαιρεί το πλήθος των Agents με το πλήθος των Cores και δημιουργεί τόσα Threads όσα είναι τα διαθέσιμα cores. Κάθε δημιουργούμενο Thread "αναλαμβάνει" τόσους Agents όσους προέκυψαν απ'την διαίρεση. Έτσι, π.χ. το πρώτο Thread θα έχει τους 1-250 Agents, το δεύτερο τους 251-500, κτλ.

Η δημιουργία των Threads γίνεται με την function :

StartThread(from, to)

Όπου **from** είναι το index του πρώτου Agent και **to** είναι το index του τελευταίου Agents της ομάδας αυτού του Thread. Η *StartThread* με τη σειρά της καλεί την function *RunAsync* της βιβλιοθήκης Loom :

Loom.RunAsync(() => { ... })

Μέσα στη παράμετρο της *RunAsync()* βλέπουμε το σύμβολο => της Lambda expression. Οπότε ο κώδικας που βρίσκεται ανάμεσα στις αγκύλες μετατρέπεται σε Action το οποίο απαιτεί ως παράμετρο η *RunAsync()* της *Loom*. Το Action είναι ένα Delegate που δεν επιστρέφει κάποια τιμή, δηλαδή σαν void. Το Delegate της C# είναι ο αντίστοιχος Function Pointer της C++, δηλαδή μια μεταβλητή που αποθηκεύει τη διεύθυνση μιας function στην μνήμη. Έτσι, όταν χρησιμοποιούμε τον Function Pointer, καλείται η function στην οποία "δείχνει". Συνήθως χρησιμοποιούνται ως παράμετροι σε functions, π.χ. τύπου Callbacks / Listeners.

Η *RunAsync()* της *Loom* προσθέτει την παράμετρο Action στην ουρά του ThreadPool του λειτουργικού συστήματος. Μόλις βρεθεί ένα ελεύθερο Thread του ThreadPool θα εκτελέσει την Action :

ThreadPool.QueueUserWorkItem(RunAction, a);

Όπου **a** είναι η Action μας και *RunAction* μια βοηθητική function.

Η Action, δηλαδή ο κώδικας ανάμεσα στις αγκύλες, είναι υπεύθυνη για την πρόσθεση όλων των Steering διανυσμάτων που προκύπτουν απ'τις συμπεριφορές μας (§3.2.2.1 μέχρι και §3.2.2.5) και την αποθήκευση του τελικού αθροίσματος στη λίστα SteerList που είναι μια List<Vector3>.

Έτσι, οι υπολογισμοί για να προκύψουν τα Steering διανύσματα γίνονται σε Threads, με αποτέλεσμα να μην έχουμε πτώση της απόδοσης της εφαρμογής μας και χαμηλό FPS.

Να τονίσουμε εδώ ότι λόγω του τρόπου που έχει κατασκευασθεί το Unity, υπάρχει ένας βασικός περιορισμός :

Μόνο το Main Thread του Unity έχει πρόσβαση στις functions του Unity API (Application Programming Interface) και σε Classes. Αυτό σημαίνει ότι τα Threads που φτιάχνουμε εμείς δεν έχουν πρόσβαση. Μπορούν να χρησιμοποιήσουν μόνο τους βασικούς τύπους των μεταβλητών (π.χ. int, float, bool, κτλ) και δομές Struct. Άρα, στα δικά μας Threads δε μπορούμε να εκτελέσουμε πολλές λειτουργίες του Unity όπως η :

Physics.OverlapSphere()

για τον προσδιορισμό των γειτονικών Agents.

Γι' αυτό, τις λειτουργίες απ'το Unity API τις εκτελούμε στον κάθε Agent στην *Update* του (§4.2). Ομοίως, δε μπορούμε απ'τα δικά μας Threads να αλλάξουμε τη θέση κάποιου Agent γιατί δεν έχουμε πρόσβαση στον Transform του Agent. Σε αυτή την περίπτωση, χρησιμοποιήσαμε μόνο το Main Thread όπου στην *Update* του κάνουμε :

```
for (int i=0; i < AgentList.Count; i++)
{
        AgentList[i].position += AgentList[i].velocity * deltaTime;
        AgentList[i].localEulerAngles = new Vector3 (AgentList[i].localEulerAngles.x, newY,
                                                     AgentList[i].localEulerAngles.z);
}
```

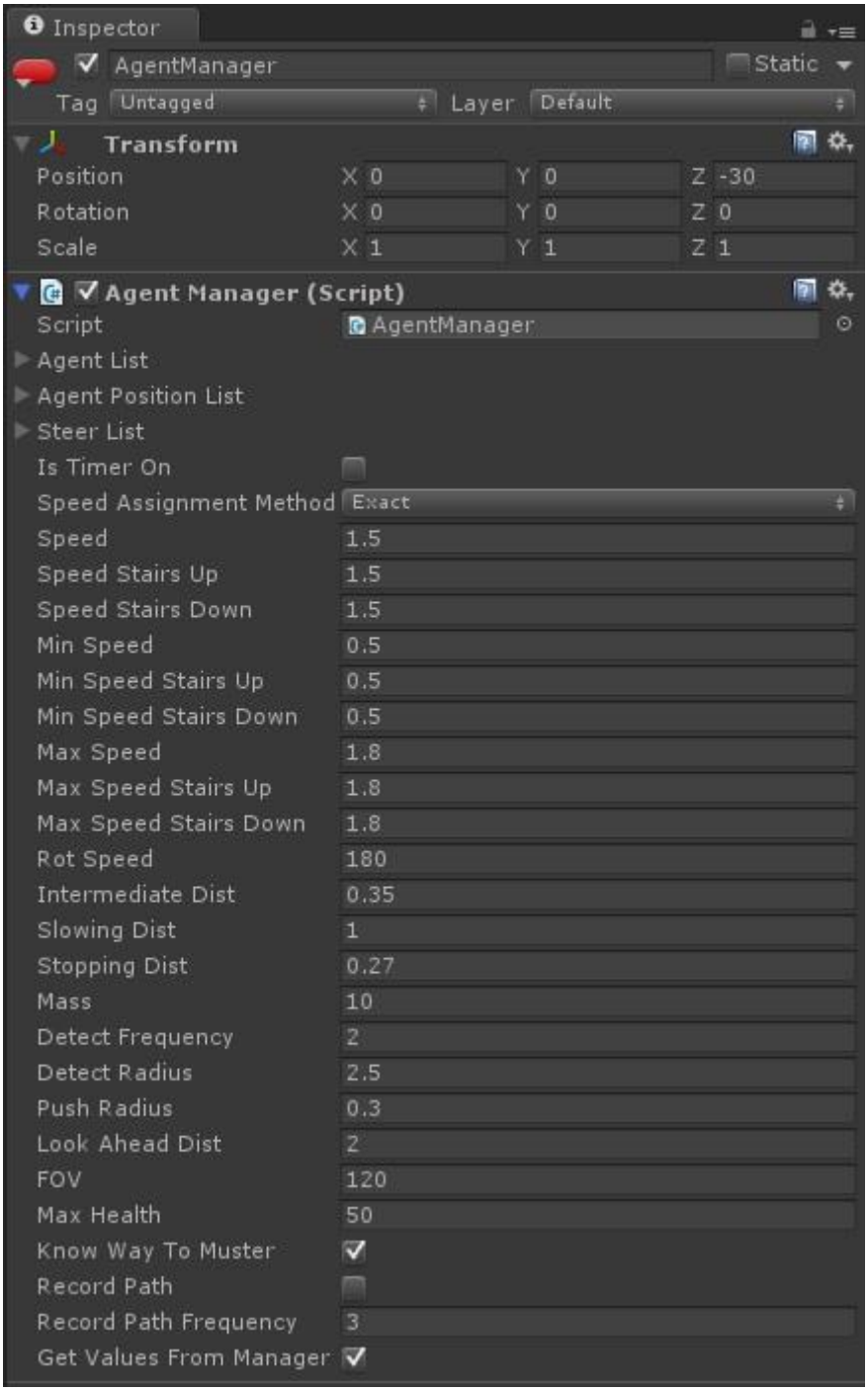
δηλαδή μεταβάλλουμε τη θέση και την περιστροφή όλων των Agents αναγκαστικά στο Main Thread.

Όμως, εκμεταλλευτήκαμε το γεγονός ότι το Vector3 είναι Struct, οπότε μπορούμε να το χρησιμοποιούμε στα δικά μας Threads. Άρα, υλοποιήσαμε όλες τις συμπεριφορές των Agents στον AgentManager με τέτοιο τρόπο ώστε να έχουν εισόδους τους βασικούς τύπους ή/και Vector3 και όμοιου τύπου εξόδους. Έτσι, τα Threads μπορούν και καλούν για τον κάθε Agent (υπενθυμίζουμε ότι ο AgentManager διατηρεί μια λίστα με όλους τους Agents της σκηνής, την AgentList) τις functions που υλοποιούν τις συμπεριφορές μας. Μέσα σε αυτές τις functions γίνονται οι αναγκαίοι υπολογισμοί και το συνολικό Steering διάνυσμα, που προκύπτει ως άθροισμα όλων των Steering διανσυμάτων των συμπεριφορών, αποθηκεύεται σε μια δεύτερη λίστα του AgentManager, την SteerList η οποία και αυτή είναι List<Vector3>. Προφανώς, το στοιχείο AgentList[i] αναφέρεται στον ίδιο Agent με το στοιχείο SteerList[i], δηλαδή οι δύο λίστες έχουν τα ίδια indexes για τους Agents. Οι λίστες αυτές φαίνονται και στην εικόνα 4.16 που απεικονίζει τον Inspector του AgentManager (AgentManager.cs) :

Ο AgentManager έχει και μια ακόμα λίστα, AgentCollidedList, που είναι private όμως και δε φαίνεται στην εικόνα 4.16, η οποία διατηρεί ποιοι Agents συγκρούσθηκαν αρκετά σφοδρά ώστε να χάσουν Health [16] και να αλλάξει χρώμα το σώμα τους. Για την αλλαγή του χρώματος πρέπει να έχουμε πρόσβαση στο material του Agent, άρα μπορούμε να το αλλάξουμε μόνο απ’το Main Thread. Έτσι, στην συμπεριφορά PushAgent (που “τρέχει” στα δικά μας Threads) όταν δύο Agents βρεθούν μέσα στο PushRadius του άλλου, ελέγχουμε τη σχετική τους ταχύτητα και αν αυτή υπερβαίνει κάποιο όριο, τότε η σύγκρουση θεωρείται σφοδρή και οι Agents αποθηκεύονται στη λίστα AgentCollidedList. Στην *Update* του Main Thread, κάθε Agent σε αυτή τη λίστα καλεί την function του :

AgentCollidedList[j].Collided()

που είναι υπεύθυνη για την μείωση του Health και την αλλαγή του χρώματος του material του σώματός του.



Εικόνα 4.16 : Agent Manager

Στην εικόνα 4.16 φαίνεται και η δεύτερη χρήση του AgentManager που είναι να δώσει τιμές στις παραμέτρους των Agents. Όπως είπαμε και στην §3.2.1, αν η μεταβλητή GetValuesFromManager κάποιου Agent είναι true, τότε αυτός ο Agent δε θα χρησιμοποιήσει τις τιμές που έχει στον Inspector του, αλλά στην αρχή της σκηνής θα πάρει τις τιμές απ'τον AgentManager. Με αυτό τον τρόπο, είναι εύκολο και γρήγορο για εμάς να ορίζουμε τις τιμές των παραμέτρων σε μεγάλο αριθμό Agents χωρίς να χρειάζεται να τις θέτουμε στον καθένα ξεχωριστά.

Τέλος να επισημάνουμε ότι ο AgentManager έχει υλοποιηθεί ως Singleton [17]. Πρόκειται για ένα πρότυπο σχεδιασμού στον προγραμματισμό, όπου επιβάλλει την ύπαρξη ενός μόνο αντικειμένου μιας Class. Γενικά το Singleton χρησιμοποιείται όταν χρειαζόμαστε έναν γενικό ελεγκτή κάποιου resource που έχει πρόσβαση ολόκληρο το σύστημά μας. Έτσι, συνηθίζεται να υλοποιούνται ως Singleton όλων των ειδών οι Manager, όπως AudioManager, LoggingManager, SerializationManager, AgentManager, κτλ.

Έτσι, ο AgentManager μας έχει την ευθύνη :

a. δημιουργίας και χρήσης των Threads που "τρέχουν" ταυτόχρονα τις συμπεριφορές όλων των Agents,
b. τον έλεγχο της κίνησης και περιστροφής όλων των Agents.

Επίσης, επειδή λειτουργεί σε global scope, είναι πολύ εύκολο για τα υπόλοιπα Scripts να έχουν πρόσβαση στις public μεταβλητές και functions του. Ένα απλό παράδειγμα :

AgentManager.Instance.AgentList.Add(this);
AgentManager.Instance.SteerList.Add(Vector3.zero);

Δηλαδή, μόλις δημιουργηθεί κάποιος Agent, αμέσως εγγράφει τον εαυτό του (*this*) στη λίστα του AgentManager με όλους τους Agents. Προσθέτει και ένα αρχικό μηδενικό διάνυσμα στην αντίστοιχη λίστα με τα Steering διανύσματα των Agents.

Εμείς υλοποιήσαμε την έκδοση Persistent Singleton, στην οποία υπάρχει η εντολή :

DontDestroyOnLoad();

που λέει στο Unity ότι αυτό το αντικείμενο, το Singleton AgentManager, δε θα καταστρέφεται όταν αλλάζουν οι σκηνές (όπως συμβαίνει σε όλα τα υπόλοιπα αντικείμενα). Έτσι, όταν βρισκόμαστε στη σκηνή που αλλάζουμε σε γραφικό περιβάλλον τις τιμές των παραμέτρων του AgentManager (§3.2.1) και στη συνέχεια μεταφερθούμε στην κυρίως σκηνή της προσομοίωσης, ο AgentManager έχει παραμείνει το ίδιο αντικείμενο και με τις ίδιες τιμές που του δώσαμε.

Ένα μειονέκτημα του Singleton είναι ότι πρέπει να δημιουργείται απ'την πρώτη αρχική σκηνή του κεντρικού μενού και να μην καταστρέφεται όταν μεταφερόμαστε στην σκηνή της προσομοίωσης. Όμως, αν θέλουμε να εργασθούμε στην σκηνή της προσομοίωσης και τη φορτώσουμε στον Editor, τότε δε θα υπάρχει AgentManager καθώς δε θα έχουμε περάσει απ'την πρώτη σκηνή ώστε να δημιουργηθεί. Έτσι, φτιάχνουμε ένα αντίγραφο του AgentManager στην σκηνή που θέλουμε να δουλέψουμε για να μπορούν οι Agents να πάρουν τιμές. Όταν μετά τρέξουμε την εφαρμογή απ'την αρχή, ο κανονικός AgentManager θα σβήσει το αντίγραφο και η προσομοίωση θα λειτουργήσει σωστά. Για να είμαστε σίγουροι, βάζουμε πρώτο το AgentManager.cs στην Script Execution Order (Edit => Project Settings => Script Execution Order). [Δημιούργησε bug λόγω του multi-threading στην έκδοση 4.6.4 οπότε δε χρησιμοποιήσαμε τελικά την Script Execution Order]

4.4 Ολοκλήρωση προσομοίωσης

Όταν κάποιος Agent φτάνει στον Muster Station, τότε τον καταστρέφουμε ώστε να μην υπάρχει συγκεντρωμένος πολύ μεγάλος αριθμός Agents σε έναν μικρό χώρο και προκαλείται μείωση της απόδοσης της εφαρμογής απ'τους αυξημένους υπολογισμούς που γίνονται λόγω των πολλαπλάσιων γειτόνων του κάθε Agent. Επίσης, στην λίστα AgentList του AgentManager, όπου είναι καταχωρημένοι όλοι οι Agents στην σκηνή, θέτουμε την αντίστοιχη καταχώρηση του κατεστραμμένου Agent ως *null*.

Πριν, όμως, τον καταστρέψουμε, ελέγχουμε αν αυτός ο Agent είχε την μεταβλητή recordPath ίση με true, οπότε αντιγράφουμε την λίστα με το path του στον AgentManager (συνάρτηση *AddToAgentsPathsList*) ώστε να μη διαγραφεί με την καταστροφή του Agent.

Ακόμα, όταν συγκρούεται ο Agent με τον Collider του Muster Station, ελέγχουμε αν αυτός είναι ο τελευταίος ενεργός Agent (*CheckIfEvacuationComplete*). Ως ενεργό, θεωρούμε οποιονδήποτε Agent που δεν έχει τεθεί ως *null* στην AgentList (δηλαδή δεν έχει φτάσει ακόμα σε Muster Station) ή δεν είναι Incapacitated. Άρα, η προσομοίωση θα ολοκληρώνεται όταν δεν υπάρχει ούτε ένας ενεργός Agent. Σε κώδικα η λογική είναι :

if (!AgentList.Exists(element => element != null && element.isIncapacitated == false))

όπου *element* είναι η παράμετρος της lambda expression.

Όταν ολοκληρωθεί η προσομοίωση, τότε σταματάμε τον timer, αποθηκεύουμε στο αρχείο Times στον φάκελο Reports την τιμή του και θέτουμε την μεταβλητή *isEvacuationComplete* του AgentManager σε true. Τότε, το Script : GUIDisplay.cs μας δείχνει τέσσερα επιπλέον buttons, δύο για την κυκλοφοριακή συμφόρηση : Show Max Congestion, Save Congestion Information και άλλα δύο για τα paths των Agents : Show Path Routes, Save Path Routes.

Το πρώτο δείχνει σε κάθε Congestion Area αντικείμενο τη μέγιστη τιμή που παρατηρήθηκε (function *ShowMaxCongestion*). Το δεύτερο αποθηκεύει στον φάκελο Reports ένα αρχείο με filename το όνομα του αντικειμένου και την ημερομηνία και ώρα που δημιουργήθηκε. Παράδειγμα : CongestionAreaDeck7_01_230520151754. Σε αυτό το αρχείο καταγράφεται σε κάθε χρονική στιγμή (με περίοδο *checkPeriod* στο Script : Congestion.cs) η τιμή της συμφόρησης σε αυτή την περιοχή (Πίνακας 4.1). Η τιμή προκύπτει απ'το πόσοι Agents βρίσκονται μέσα στον Collider της Congestion Area. Για την προγραμματιστική υλοποίηση της έννοια της Congestion δημιουργήσαμε την class : TimeCongestionValue και μια λίστα CongestionList με τιμές αυτής της class. Η αποθήκευση γίνεται με :

writer.WriteLine(CongestionList[i].Time + "," + CongestionList[i].Value);

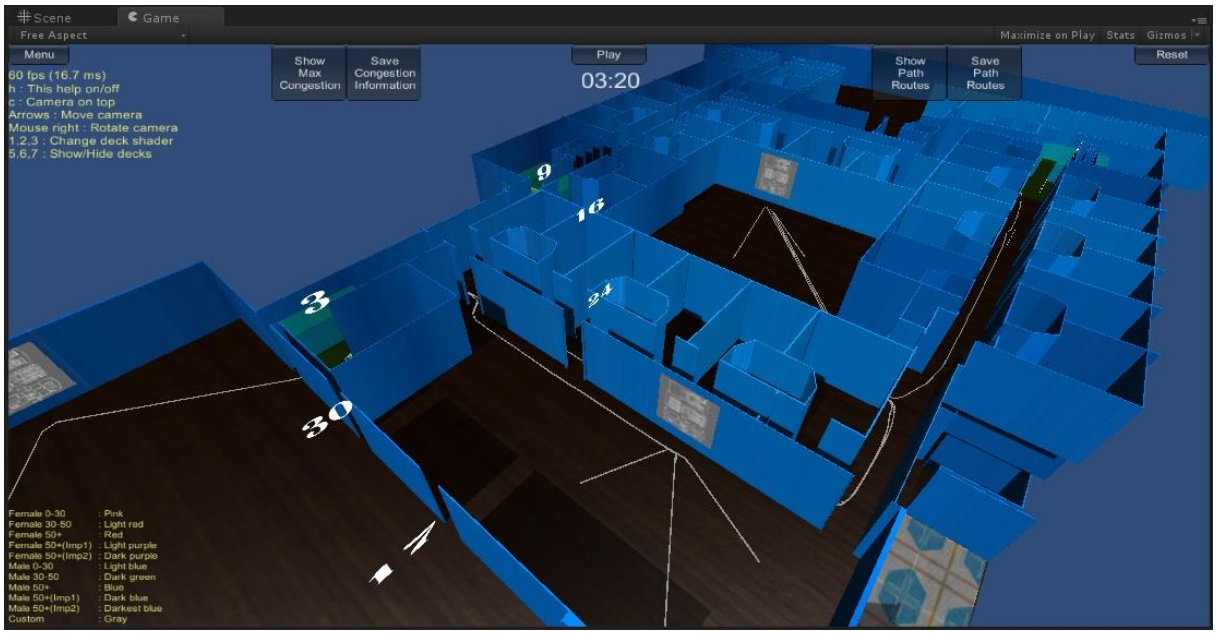
Το button Show Path Routes δείχνει τις πορείες όσων Agents είχαν recordPath = true. Φαίνονται ως γκρι γραμμές [18] και [19] που δημιουργήθηκαν από Line Renderers (function *ShowAllPathRoutes* στον AgentManager). Για κάθε i σημείο του path κάνουμε :

lineRenderer.SetPosition(i++, pathPoint.Position);

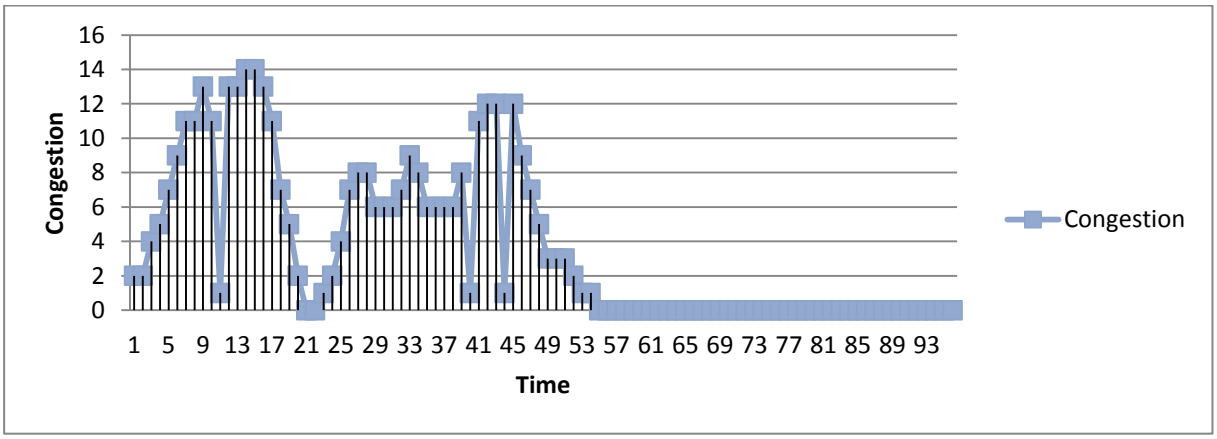
Έτσι, ο lineRenderer ενώνει όλα τα σημεία του path μέχρι την σύγκρουση με τον Collider του Muster Station όπου ο Agent καταστρέφεται.

Το τελευταίο button Save Path Routes αποθηκεύει στον φάκελο Reports το αρχείο με filename : Paths και την ημερομηνία συν την ώρα δημιουργίας, π.χ. : Paths230520151754. Χρησιμοποιήσαμε την class : TimePositionValue που φτιάξαμε στο Script : Agent.cs. Στο αρχείο περιέχονται οι χρονικές στιγμές και οι θέσεις του κάθε σημείου του path, pathPoint. Για κάθε θέση κάνουμε (εικόνα 4.17) :

writer.WriteLine(pathPoint.Time + "," + pathPoint.Position.ToString());



Εικόνα 4.17 : Ολοκλήρωση προσομοίωσης



Πίνακας 4.1 : Παράδειγμα μετρήσεων τιμών Congestion προς χρόνο

4.5 Τεστ σκηνές

Στην §3.1 είδαμε την προηγμένη μέθοδο ανάλυσης εκκένωσης της MSC σύμφωνα με το [2]. Εκεί, επίσης, η MSC προτείνει μερικά Τεστ για την μέτρηση της πιστότητας του προγραμματιστικού μοντέλου και λογισμικού. Αυτά είναι :

1. Διατήρηση σταθερής ταχύτητας σε διάδρομο.
2. Διατήρηση σταθερής ταχύτητας ανεβαίνοντας σκάλα.
3. Διατήρηση σταθερής ταχύτητας κατεβαίνοντας σκάλα.
4. Ο ρυθμός ροής από μια έξοδο να μην υπερβαίνει : 1.33 άτομα/s.
5. Οι Agents να έχουν ξεχωριστούς προσωπικούς χρόνους απόκρισης.
6. Περνώντας γωνίες χωρίς να διεισδύουν στα τοιχώματα.
7. Απόδοση στους Agents διαφορετικών ταχυτήτων χρησιμοποιώντας τον Πίνακα 3.2.
8. Διαπίστωση ότι ο συνολικός χρόνος εκκένωσης αυξάνεται όταν υπάρχει αντίστροφη ροή.
9. Διαπίστωση ότι ο συνολικός χρόνος εκκένωσης μειώνεται όταν υπάρχουν περισσότεροι έξοδοι στον ίδιο χώρο.
10. Οι Agents να επιλέγουν διαφορετικές εξόδους αναλόγως τις αποστάσεις τους.
11. Επίδειξη περιοχής συμφόρησης σε διάδρομο που οδηγεί σε σκάλα.

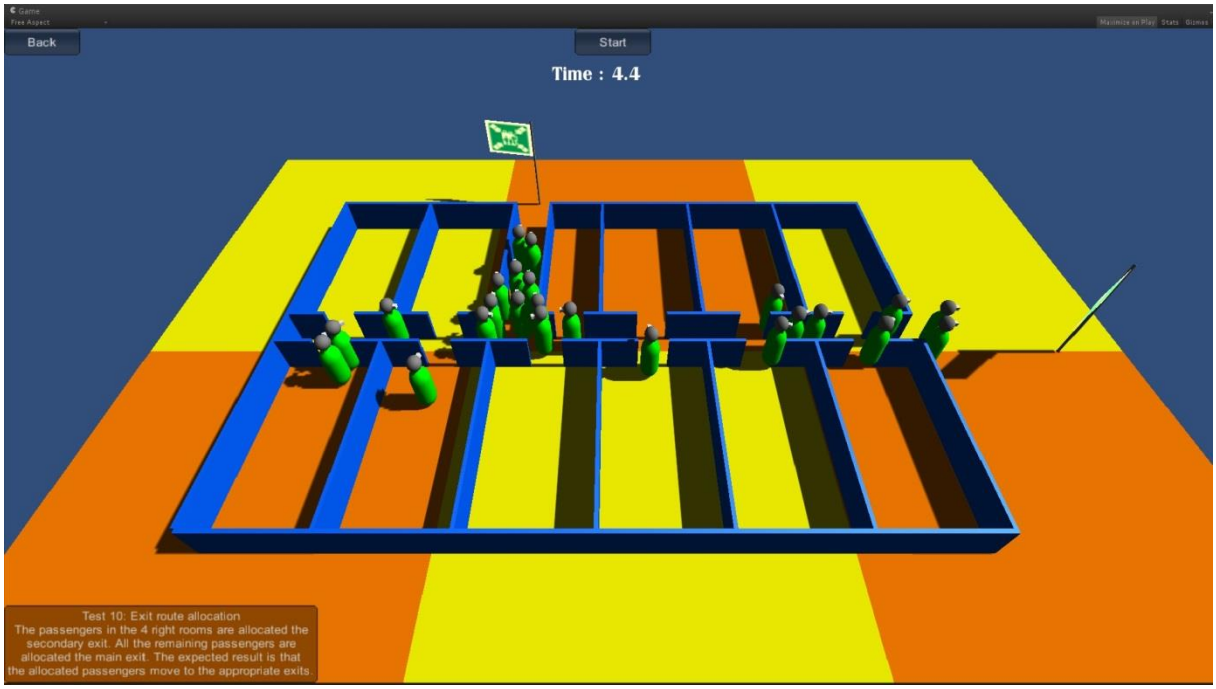
Υλοποίηθηκαν όλα τα Τεστ εκτός απ’το Τεστ5, γιατί στην υλοποίησή μας έχουμε χρησιμοποιήσει τον AgentManager που είναι υπεύθυνος για την κίνηση όλων των Agents. Έτσι, όλοι οι Agents αρχίζουν και κινούνται όλοι μαζί όταν το επιλέξει ο χρήστης. Αν ο κάθε Agent έλεγχε την κίνησή του θα ήταν πολύ εύκολο να είχε και δικό του χρόνο απόκρισης.

Μπορούμε να δούμε τα υπόλοιπα Τεστ στις ομώνυμες Test Scenes που έχουμε δημιουργήσει. Η πρόσβαση γίνεται απ’το κεντρικό μενού επιλέγοντας το button : Test Scenes, όπου μας μεταφέρει σε ένα δεύτερο μενού με όλες τις σκηνές μας.

Στα Τεστ με πολλούς Agents διαπιστώσαμε μια αδυναμία του προγράμματός μας να χειρισθεί μεγάλο αριθμό Agents σε μικρό χώρο. Παράδειγμα είναι το Τεστ4 όπου 100 Agents είναι σε ένα δωμάτιο με μία μόνο έξοδο όπου συμπίεζονται πάρα πολύ απ’τις απωθητικές δυνάμεις όλων των άλλων Agents. Έτσι, δεν διατηρούν τον σωστό τους όγκο και το αποτέλεσμα είναι ο ρυθμός ροής απ’την έξοδο να είναι μεγαλύτερος απ’τον επιτρεπτό.

Επίσης, στο Τεστ9 που έχουμε 1000 Agents σε ένα μεγάλο χώρο, παρατηρήσαμε πτώση της απόδοσης του προγράμματος, δηλαδή μείωση του αριθμού FPS αλλά και απώλεια ελέγχου της κίνησης των Agents.

.



Εικόνα 4.18 : Παράδειγμα Test Scene

Στην παραπάνω εικόνα βλέπουμε ένα στιγμιότυπο απ'το Τεστ10, όπου διακρίνονται οι Agents των τεσσάρων δεξιότερων δωματίων να κατευθύνονται προς τη δεξιά έξοδο, ενώ οι Agents που ξεκινήσαν στα υπόλοιπα δωμάτια να κατευθύνονται στην κεντρική έξοδο

Να τονίσουμε εδώ τον εύχρηστο τρόπο εισαγωγής της περιγραφής της κάθε Test Scene που διακρίνεται κάτω αριστερά στην εικόνα 4.18. Χρησιμοποιήσαμε ένα αρχείο XML [20], στο οποίο είχαμε τις περιγραφές όλων των σκηνών και τις διαστάσεις του παραθύρου που θα απεικονίζονται. Παραδείγματος χάρη για το Τεστ10 :

```
<sceneRecord>
        <name>Test10</name>
        <description>Test 10: Exit route allocation
The passengers in the 4 right rooms are allocated the
secondary exit. All the remaining passengers are
allocated the main exit. The expected result is that
the allocated passengers move to the appropriate exits.</description>
        <length>330</length>
        <height>85</height>
</sceneRecord>
```

Στη συνέχεια χρησιμοποιώντας μέσα στον κώδικά μας τα : *XmlDocument*, *XmlNodeList* και *XmlNode* παίρναμε τα δεδομένα που αντιστοιχούν στην σωστή σκηνή και τα εμφανίζαμε στο παράθυρο με τις σωστές διαστάσεις.

Κεφάλαιο 5

Συμπεράσματα

Δουλεύοντας στο Unity, διαπιστώσαμε τις ευκολίες που προσφέρει ως μια τριδιάστατη rendering μηχανή. Στον editor της υπάρχουν πολλά έτοιμα Components, όπως Colliders, NavMeshAgent, TextMesh, κτλ, αλλά και ολόκληρα συστήματα, όπως το σύστημα φωτισμού και σκιάσεων ή το σύστημα φυσικής. Ενσωματώνονται γρήγορα στην εφαρμογή που θέλουμε να αναπτύξουμε μειώνοντας τον χρόνο που θα χρειαζόταν αν έπρεπε να τα αναπτύξουμε εμείς. Επίσης, η γλώσσα προγραμματισμού C#, που χρησιμοποιείται στο Unity, μαζί με το development framework Mono μας παρέχουν πολλές βιβλιοθήκες συμβατές με το .NET development framework. Έτσι, ελαχιστοποιούμε ακόμα περισσότερο τον όγκο κώδικα που θα πρέπει να γράψουμε.

Στη συγκεκριμένη εφαρμογή, όμως, προέκυψαν διάφορα προβλήματα :

1. Το navigation σύστημα NavMesh που χρησιμοποιήσαμε για να μπορούν οι Agents να έχουν path-finding παρουσιάζει δυσκολίες ταυτόχρονης εξυπηρέτησης μεγάλου αριθμού Agents. Παραδείγματος χάρη, όταν στην σκηνή υπάρχουν περίπου 500 Agents, τότε ο έλεγχος τους είναι πάρα πολύ καλός. Αλλά αν στη σκηνή βρεθούν περίπου 1000 Agents, τότε η κίνηση τους αρχίζει και γίνεται ελαττωματική. Το φαινόμενο είναι ιδιαίτερα ορατό σε περιπτώσεις συγκεντρωμένων πολλών Agents σε μικρό χώρο. Τότε, υπάρχει πιθανότητα οι Agents να βρεθούν εκτός του NavMesh και να τεθούν τελείως εκτός ελέγχου.

   Προσπαθήσαμε να φτιάξουμε το δικό μας σύστημα path-finding με τον αλγόριθμο A*, αλλά ήταν πολύ αργός για τέτοιο πλήθος Agents.

2. Δεν φτιάξαμε σωστά, απ’την αρχή, την βάση της εφαρμογής. Αυτό οφείλεται κυρίως στις αλλαγές που γινόντουσαν κατά τη διάρκεια της ανάπτυξης. Σαν παράδειγμα, αναφέρουμε το Script : Agent.cs στο οποίο αρχικά γινόντουσαν όλοι οι έλεγχοι και οι λειτουργίες του Agent. Δηλαδή κάθε Agent ήταν υπεύθυνος για τον εαυτό του και μόνο. Όμως, όταν στην πορεία εμφανίσθηκε το πρόβλημα με το μεγάλος πλήθος Agents, αλλάξαμε την εφαρμογή έτσι ώστε ο Agent να πραγματοποιεί μόνο τους αναγκαίους ελέγχους (π.χ. το πόσο κοντά βρίσκεται στο επόμενο σημείο του path του ή ποιος είναι ο κοντινότερος τοίχος). Η κίνηση και οι διάφορες συμπεριφορές που εκμεταλλεύονται αυτούς τους ελέγχους μεταφέρθηκαν στον AgentManager που είναι υπεύθυνος για όλους τους Agents. Έτσι, έχοντας κεντρική επίβλεψη, μπορέσαμε και χρησιμοποιήσαμε multi-threading προγραμματισμό.

   Ακόμα και με αυτήν την διάταξη υπάρχει περιθώριο για βελτίωση, παραδείγματος χάρη χωρίζοντας το Script : Agent.cs σε δύο classes. Τότε θα είχαμε μια class με τις

παραμέτρους – χαρακτηριστικά του Agent και μια δεύτερη με τους ελέγχους που πραγματοποιεί περιοδικά ο Agent. Η πρώτη δε θα χρειαζόταν να κληρονομεί απ’την MonoBehaviour, άρα θα μπορούσαμε να είχαμε και την δυνατότητα για null Agents, κάτι που μας δυσκόλεψε αρκετά.

3. Όπως είπαμε και προηγουμένως, χρησιμοποιήσαμε multi-threading προγραμματισμό ώστε να αξιοποιήσουμε τους πολλαπλούς cores των σημερινών επεξεργαστών. Αναθέσαμε ομάδες από Agents σε threads τα οποία εξυπηρετούνταν από τους πυρήνες. Όμως υπάρχει ένας βασικός περιορισμός : οι functions του Unity API μπορούν να καλούνται μόνο απ’το main thread. Γι’ αυτό το λόγο αναγκαστήκαμε και αφήσαμε τους Agents να εκτελούν τους ελέγχους τους με μη multi-threading τρόπο. Αν, για παράδειγμα, θέλει ένας Agent να βρει τους γειτονικούς του Agents, θα πρέπει να το κάνει ο ίδιος στο main thread. Δε μπορούμε να χρησιμοποιήσουμε τον AgentManager για να καλέσει αυτός την function : Physics.OverlapSphere() μέσα από κάποιο thread του. Άρα, όλοι οι Agents την εκτελούν μαζί στο main thread. Έστω ότι ο κάθε Agent βρει 10 Agents μέσα στην περιοχή του OverlapSPhere. Στη συνέχεια, θα πρέπει να κάνει για τον καθένα : GetComponent<Agent>(), που είναι πολύ αργή function, οπότε διαπιστώνουμε πόσο πολύ επιβαρύνεται η εφαρμογή με αποτέλεσμα την μείωση της απόδοσής της. Δυστυχώς για αυτόν τον περιορισμό δεν μπορούμε να κάνουμε κάτι.

   Για την εύρεση των γειτόνων ίσως θα ήταν καλύτερο να χρησιμοποιηθούν KD-trees ή Quadtrees (2 διαστάσεις) / Octrees (3 διαστάσεις) αλλά θα έπρεπε να αναπτυχθούν εξ’ ολοκλήρου.

Βιβλιογραφία

[1] Ιστοσελίδα Ακτοφυλακής Ηνωμένων Πολιτειών Αμερικής : http://www.uscg.mil/imo/msc/

[2] IMO (2007). "*Guidelines for evacuation analysis for new and existing passenger ships*", MSC.1/Circ.1238.

[3] Γεώργιος Μ. Δούναβης (2014). "*Πρόληψη ατυχημάτων επί του πλοίου «εν πλω» και «εν όρμω»*", Βιβλιοθήκη του Ναυτικού, Κεφ. 24.8, σελίδα : 146.

[4] Ιστοσελίδα AutoCAD : http://www.autodesk.com/products/autocad/overview

[5] Ιστοσελίδα 3D Studio Max : http://www.autodesk.com/products/3ds-max/overview

[6] Hubert Klupfel. "*Evacuation of Ships and Buildings based on a CA model*", page : 48, www.traffgo-ht.com

[7] Ιστοσελίδα Unity3D : https://unity3d.com/

[8] Kluge B., Prassler E. (2007), "*Reflective navigation: Individual behaviors and group behavior*", Proc. of International Conf. on Robotics and Automation, pages : 4172-4177.

[9] Mat Buckland (2005). "*Programming Game Ai by Example*", Wordware Publishing, Inc., pages : 91-93.

[10] Daniel Phillips, Dan Ruthrauff. "*Sports Games AI*", CSE 348, pages : 28-31

[11] K.V. Kostas, A.-A.I. Ginnis, C.G. Politis, P.D. Kaklis (2011). "*Crowd simulation for ship-evacuation analysis*", Computer Animation and Virtual Worlds, page : 45.

[12] Nuria Pelechano, Jan Allbeck, Norman Badler (2008), "*Virtual Crowds : Methods, Simulation, and Control*", Synthesis Lectures on Computer Graphics and Animation, Morgan & Claypool, page : 69.

[13] Emil Johansen, "*AI In Unity*", Unity Technologies, page : 11.

[14] IMO (2004). "*Measures to prevent accidents with lifeboats*", Sub-committee on ship design and equipment, DE 48/INF.5, page : 14, Table 2.2.

[15] Γ.Κ. Παπακωνσταντίνου, Ν.Α. Μπιλάλης, Π.Δ. Τσανάκας (1999). "*Λειτουργικά Συστήματα, Μέρος Ι : Αρχές Λειτουργίας*", Κεφ. 3 : Ταυτόχρονες Διεργασίες.

[16] Colin M. Henein, Tony White. "*Agent-Based Modelling of Forces in Crowds*".

[17] Ιστοσελίδα Unity Singletons : http://unitypatterns.com/singletons/

[18] K.V. Kostas, A.-A.I. Ginnis, C.G. Politis, P.D. Kaklis (2010). "*Velos : A VR platform for ship-evacuation analysis*", Computer-Aided Design, page : 1050.

[19] Xiaoshan Pan, Charles S. Han, Ken Dauber, Kincho H. Law. "*A Multi-agent Based Framework for the Simulation of Human and Social Behaviors during Emergency Evacuations*", page : 14.

[20] Matt Smith, Chici Queiroz (2013), "*Unity 4.x Cookbook*", Packt Publishing, Chapter 8, pages : 245-249.