NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF NAVAL ARCHITECTURE AND MARINE ENGINEERING
DIVISION OF SHIP DESIGN & MARITIME TRANSPORT
SHIP DESIGN LABORATORY

INTER-DEPARTMENTAL POSTGRADUATE PROGRAMME:
"MARINE TECHNOLOGY AND SCIENCE"

Crowd Evacuation Simulation of a Passenger Ship in Unity3D

POSTGRADUATE DIPLOMA THESIS

Georgios I. Karafotias

Supervisor: Alexandros I. Ginnis
Assistant Professor, N.T.U.A.

Athens, May 2015

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF NAVAL ARCHITECTURE AND MARINE ENGINEERING
DIVISION OF SHIP DESIGN & MARITIME TRANSPORT
SHIP DESIGN LABORATORY

INTER-DEPARTMENTAL POSTGRADUATE PROGRAMME:
"MARINE TECHNOLOGY AND SCIENCE"

Crowd Evacuation Simulation of a Passenger Ship in Unity3D

POSTGRADUATE DIPLOMA THESIS

Georgios I. Karafotias

Supervisor: Alexandros I. Ginnis
Assistant Professor, N.T.U.A.

Approved by the three-member dissertation committee on May 27th 2015.

……..	……..	……..
A. Ginnis	G. Zarafonitis	K. Kostas
As. Professor N.T.U.A.	As. Professor N.T.U.A.	As. Professor T.E.I.A.

Athens, May 2015

.....
Georgios I. Karafotias
Electrical and Computer Engineer, N.T.U.A.

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 framework for simulating a passenger ship's evacuation procedure after an emergency, usually caused by internal threats. The simulation framwork will be developed using the Unity3D software package, which includes a 3-D rendering engine and an editor environment where the virtual scenes are built. The autonomous characters (agents), which all together represent a human crowd, will have artificial intelligence that is programmed in two parts: the first will control their movement using Unity3D's embedded path navigation system, and the second will define the ways and rules that govern agent interaction. The final outcome will be the formulation of a crowd model with realistic movement and collision control.

This model will be used in the advanced evacuation analysis method that is authored by Maritime Safety Committee (MSC) and further described in the: "Interim guidelines for evacuation analysis for new and existing passenger ships". These guidelines are 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 fatalities accidents involving passenger ships and from recent shipbuilding of new cruise liners capable of carrying several thousand passengers. So, using our model could 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, Evacuation, Passenger Ship, Artificial Intelligence, Autonomous Character, Agent, Crowd Model, Congestion, MSC, IMO, Unity3D.

Contents

Table Contents

Figure Contents

Chapter 1

Introduction

This thesis came up as a natural continuation of the semester project that we undertook for the course "Solid Modelling: Special Topics and Applications in the Virtual Ship" of the Inter-Departmental Postgraduate Programme "Marine Technology and Science". In that venture, we studied the fundamental principles and functions of artificial intelligence in the Unity3D environment. We mainly delved into the following subjects:

- Finite State Machine
- Pathfinding using the algorithm A*
- Pathfinding using the Nav Mesh system
- Pathfollowing
- Obstacle Avoidance
- Flocking

Using these and other additional techniques, such as various agent behaviours that we will analyze later, we ended up constructing a model for analyzing the evacuation process of a passenger-ferry - which is the subject of this report. We will try to prove that this model is consistent with the requirements of the *advanced* evacuation analysis method approved by the Maritime Safety Committee (MSC) of the International Maritime Organization (IMO). Before proceeding to the description of our model, let us first see a short description of the MSC and generally the safety of passenger ships.

1.1 Brief description of the MSC

The Maritime Safety Committee is a subsidiary of the IMO Council [1]. The committee is consisted by representatives from all the member governments, and is the highest technical department of the organization. Its responsibilities include any matter within the jurisdiction of the organization which is associated with:

- Any type of navigation aids
- Ship construction and equipment
- Manning security
- Collision prevention rules
- Management of dangerous and/or harmful cargo
- Procedures and requirements of maritime safety
- Hydrographic information/data
- Logbook and navigation files
- Investigation of marine accidents,
- Salvage and rescue
- Any other issue that affects maritime safety

The commitee also provides mechanisms regarding the execution of any duty assigned by the IMO convention, or any duty - within the issues mentioned above – assigned by an international organization which has already been accepted by the IMO. In addition, it has the responsibility to examine and submit recommendations and guidelines, regarding safety, to be discussed and approved by the IMO.

The MSC [2], having already approved the guidelines for a simplified evacuation analysis method of RO-RO (Roll On-Roll Off) ships as a guide for the implementation of Regulation II-2 / 28-1.3 of SOLAS (Safety Of Life At Sea), asked, in the May of 1999, the subcommittee which is responsible for the fire protection to develop rules for analyzing the evacuation process in passenger ships in general but also in high-speed passenger ships.

In June 2001, following the recommendations of the subcommittee, the MSC approved the guidelines for the simplified evacuation analysis method of high-speed passenger ships. In May 2005, it considered a proposal of the subcommittee and finally approved the new guidelines replacing the previous ones.

Meanwhile, in May 2002, the MSC accepted the guidelines of the subcommittee on evacuation analysis for new and existing passenger ships. It also asked member governments to collect and submit to the subcommittee any data or information that was gained from research, development or testing, and any findings about human behaviour that could be used to improve these guidelines.

The current version of the guidelines was given by the MSC in October 2007. They apply to new and existing passenger ships, including the RO-RO. There are two different methods for the analysis of evacuation:

- The simplified method
- The advanced method

In this report, we will use the advanced method, which we will further analyze it in Chapter 3.

1.2 Safety in passenger ships

In the Greek Merchant Marine Academy – Deck Officers School the following recommendations concerning safety on passenger ships are specified [3]:

1. The IMO Convention for the Safety of Life at Sea requires a sufficient number of trained personnel to be on board for the guidance and assistance of untrained people.
2. The crew which is defined by the division tables to assist passengers in emergency situations, should undertake additional training to ensure that it is able to perform its duties efficiently. The number of the trained crew members must be included in the safety manning document of the ship.
3. Training that takes place in a series of courses on land, should be supplemented by on-board training before undertaking the tasks referred to 2. The training should satisfy the State Flag and should specify a number of means to ensure that the crew members maintain continuous proficiency and efficiency through periodic training courses, exercises or related work experience.
4. The communication skills of designated seafarers should be adequate to assist passengers during an emergency, taking into account the following criteria:
    a. the language(s) needed for communication with all the passengers of different nationalities carried on a particular route,
    b. the ability to use elementary English vocabulary for basic instructions, which provides a way to interact with a passenger in need of assistance whether the passenger and crew member speak a common language or not,
    c. the possible need to communicate by other means (e.g. by demonstration, or gestures, or asking him to pay attention to the instructions, assembly stations, means of rescue or evacuation routes) during an emergency when verbal communication is not feasible,
    d. providing complete and thorough safety instructions to passengers in their native language,
    e. the different languages in which emergency announcements can be transmitted during an emergency or an exercise to carry vital instructions to passengers and to help crew members in assisting passengers.
5. The training provided under recommendation 2. should include, but not necessarily limited to, the following theoretical and practical items:

a. awareness of the plans of rescue equipment and firefighting plans and knowledge of division tables and emergency instructions including:
    i. the general alarms and the procedures to assemble the passengers in designated stations,
    ii. areas of responsibility with emphasis on “designated sectors”.
b. the general arrangement of the ship with special emphasis on the position of the assembly stations where embarkation into the lifeboats is possible, the accesses and escape routes,
c. the location and use of emergency equipment in relation to the tasks of 2. with emphasis on “designated sectors” and escape routes from there,
d. the location of life jackets for adults and children,
e. the location of other evacuation supplies (e.g. blankets), which must be transferred to lifeboats,
f. basic first aid skills and transportation of injured people,
g. communication:
    i. use of interphone systems,
    ii. alerting,
    iii. updating the passengers,
    iv. reporting and notification.
h. evacuation:
    i. use of passengers lists or admesurement,
    ii. alarms,
    iii. assembly – order maintainance and avoiding panic procedures,
    iv. emergency exits,
    v. evacuation equipment,
    vi. passengers control at passageways, staircases and doors,
    vii. maintaining escape routes free and functionable,
    viii. assistance en route to the assembly stations and embarkation into the lifeboats,
    ix. methods available to evacuate people with mobility limitations and/or needing of special assistance,
    x. restrictions on the use of lifts,
    xi. searching within accommodation areas,
    xii. ensuring that passengers are suitably clothed and have properly worn their lifejackets.
i. fire cases:
        i. fire detection and initial restriction,
        ii. alarming,
        iii. smoke inhalation risk,
        iv. respiratory protection.
j. abandoning ship cases:
    i. correct use of personal safety equipment, e.g. lifejackets, immersion suits, lifebuoys, light signals, and fumigants,
    ii. need of assistance in particular cases.

k. familiarity through repeated, organized, guided tours in the ship,
l. repeated participation in fire exercises and embarkation into the lifeboats, including simulated transportation of injured people,
m. repeated exercises in use of equipment, such as wearing lifejackets and appropriate protective clothing,
n. repeated exercises in use of interphone systems,
o. repeated evacuation exercises.

6. Before the ship departure, instructions regarding emergency procedures and evacuation should be given to the passengers.
7. Wherever possible, a video regading safety should be displayed to passengers, right after boarding.
8. Clear emergency signals should be positioned at a suitable height in a major understandable language to assist passengers follow the routes to assembly stations and to the boxes with the lifejackets. For this purpose, international IMO symbols should be used.
9. Embarking into lifeboats exercises should take place according with the SOLAS guidelines. The rest lifesaving equipment should be frequently checked and maintained in good condition. Manufacturers' instructions about maintainance and replacement should always be followed.
10. The whistles and communication systems should be regularly tested and maintained in good working condition.
11. The exercises and the procedures regarding man at sea should be carried out frequently.

Chapter 2

Software

In this section we will briefly describe the three different programs we used to complete the current project. For each program we will mention general information, show its basic functions and explain how we used it to achieve the creation of the simulation model of the evacuation process.

2.1 AutoCAD

2.1.1 General

It is the most known program from the Autodesk company [4]. It is a design software (Computer-Aided Design, CAD) which exists since 1982 and is considered the most widely used CAD program worldwide. It is mainly used by engineers, so there are specialized versions, such as:

- For architects: AutoCAD Architecture.
- For civil engineers: AutoCAD Civil 3D.
- For electrical engineers: AutoCAD Electrical.
- Για mechanical engineers: AutoCAD Mechanical.
- For surveyor engineers: AutoCAD Map 3D (GIS).
- For building construction engineers: AutoCAD MEP.
- For piping construction engineers: AutoCAD P&ID.
- For plant construction engineers: AutoCAD Plant 3D.

2.1.2 Operation Mode

Design in AutoCAD is based on the use of basic shapes such as lines, polygons, circles and the modification of them to achive every desired geometry. This modification is made using transformations, e.g. rotation, displacement (offset), resize (scale and stretch), mirror, cut (trim), compouned curved line (fillet), join objects, split objects (break), reverse

the direction of a line and a lot more.

Also, there are many helpful features that facilitate our work such as creating and editing objects as a group (group), the use of explanatory text (comments), inserting objects in a layer so that we can work only with it without affecting the objects of the other layers, counting accurate dimensions (distance) and programming various additional functions in the languages: VBA, .NET, AutoLISP, Visual LISP, ObjectARX (based on C ++).

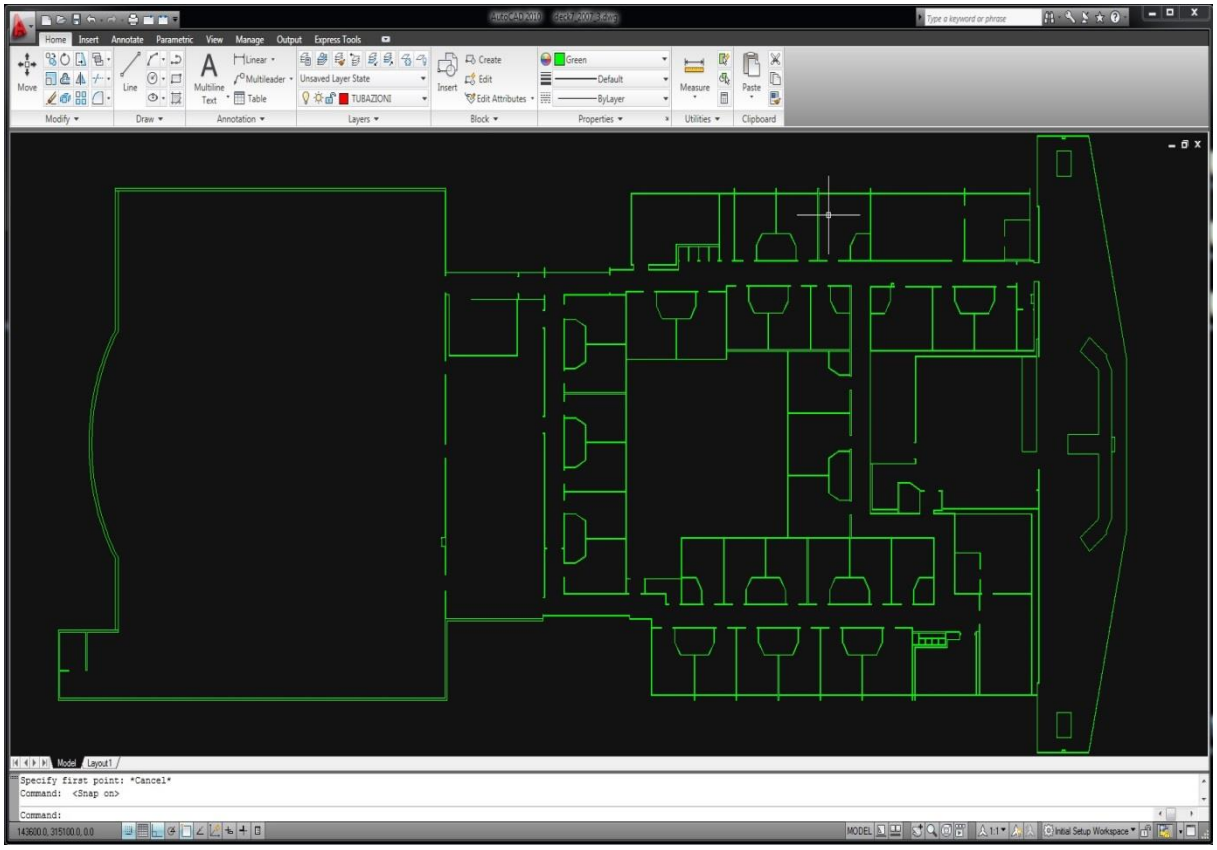

Figure 2.1: Bridge Deck – deck7

In Figure 2.1 we can distinguish the *user interface*. The screen is basically divided into three parts. The top part has all the features available, the middle and larger one is the designing area of our objects and the bottom one is the console where we can see some useful information or errors but also where we can input direct commands (e.g. the command *line* creates a line from two points that we can define), or enter the parameters of a command. Under the console, there is a line of auxiliary functions for the selection and placement of objects and also for the management of our working area (i.e. the central portion), for example the ability to zoom / unzoom.

2.1.3 Usage

We worked with the basic version AutoCAD 2010. We designed three decks of a passenger-ferry which we used for the advanced analysis method of the evacuation process. Figure 2.1 shows the design of the bridge deck (bridge deck or deck7 as we named the reconstructed 3-D object in the Unity3D software). We will explain the conversion process in from a 2-D design to a 3-D object in paragraph §2.2.3.

In the following Figures 2.2 and 2.3 we can see the designs of the passenger (Passenger Deck - deck6) and embarkation or boarding (Embarkation Deck - deck5) deck respectively. The numbers 5, 6, 7 denote the level of each deck in the ship. For example, the bridge deck is at level 7, the highest level in the ship.

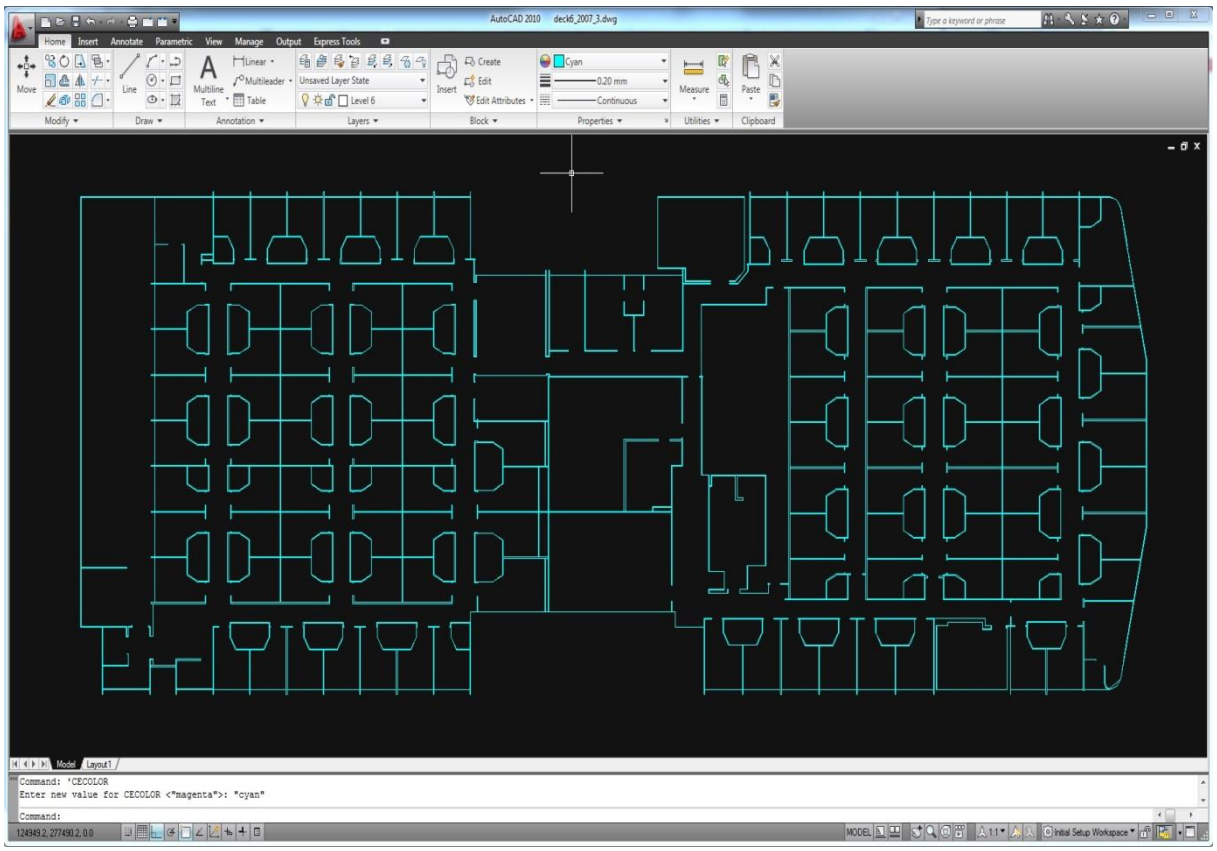

Figure 2.2: Passenger Deck – deck6

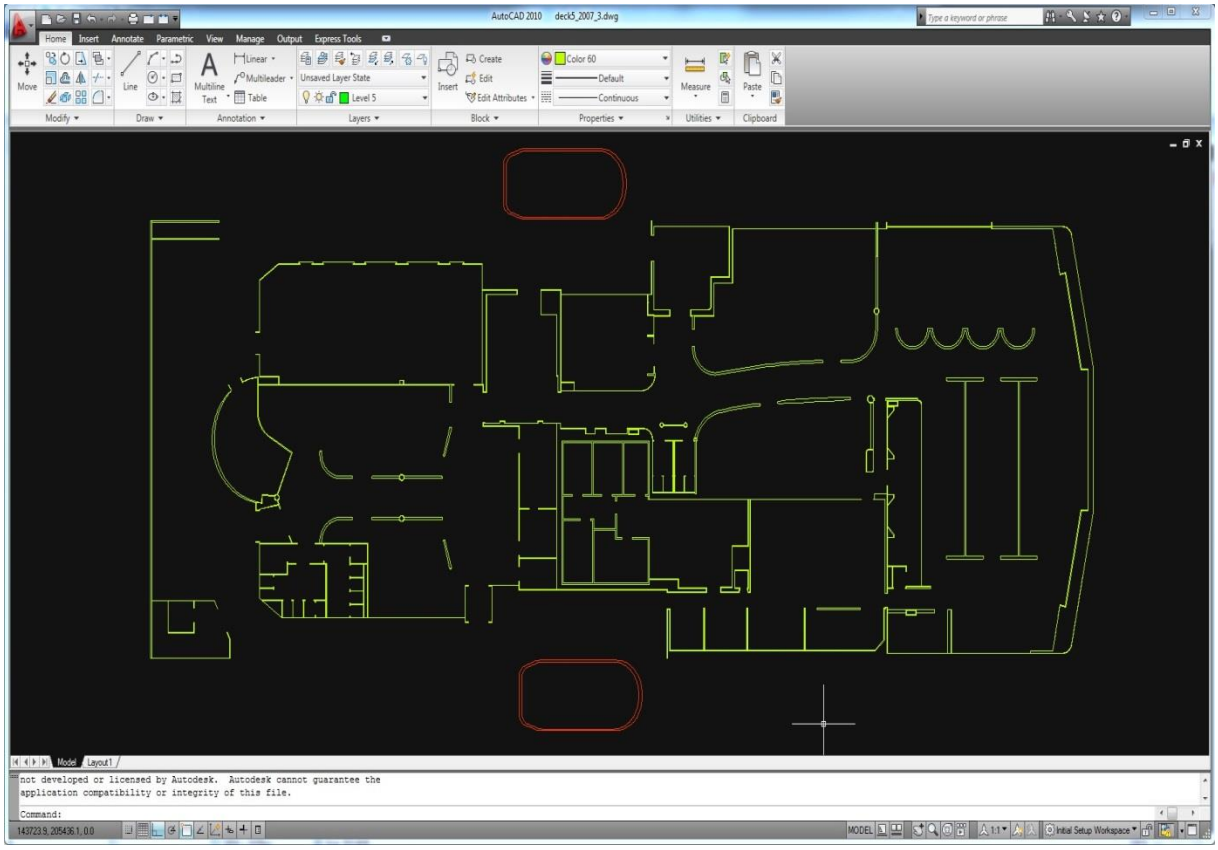

Figure 2.3: Embarkation Deck – deck5

We worked mainly with DWG files, the default file format in AutoCAD. These files will later be imported into the 3D Studio Max software to create 3-D models of the topology of decks but also 2-D models of the deck floors. Upon these models, our Agents will move within the Unity3D software to simulate the evacuation.

The dimensions of the three decks are:

	Length	**Width**	**Height**
deck5	72.438m	27.147m	3.200m
deck6	71.084m	26.740m	2.700m
deck7	70.316m	32.503m	2.750m

Table 2.1: Decks Dimensions

The sequence of operations, i.e. starting from an AutoCAD drawing without any unnecessary lines, then making a respective 3-D model and then running the simulation in a 3-D rendering engine was inspired from here: [5].

2.2 3D Studio Max

2.2.1 General

For the reconstruction of the 3-D models of the decks and their floors, we will use the 3D Studio Max software from Autodesk Company [6]. The 3D Studio specializes in creating 3-D objects, characters, environments and generally detailed 3-D scenes which can include cameras, light sources and shadows. Furthermore, we can apply textures to the objects and determine how they can move (animation). The most important feature of the 3D Studio is to create complex objects and apply various transformations to them.

2.2.2 Operation Mode

Figure 2.4 shows the interface of 3D Studio Max and particularly the deck7 and deck7_ground models:

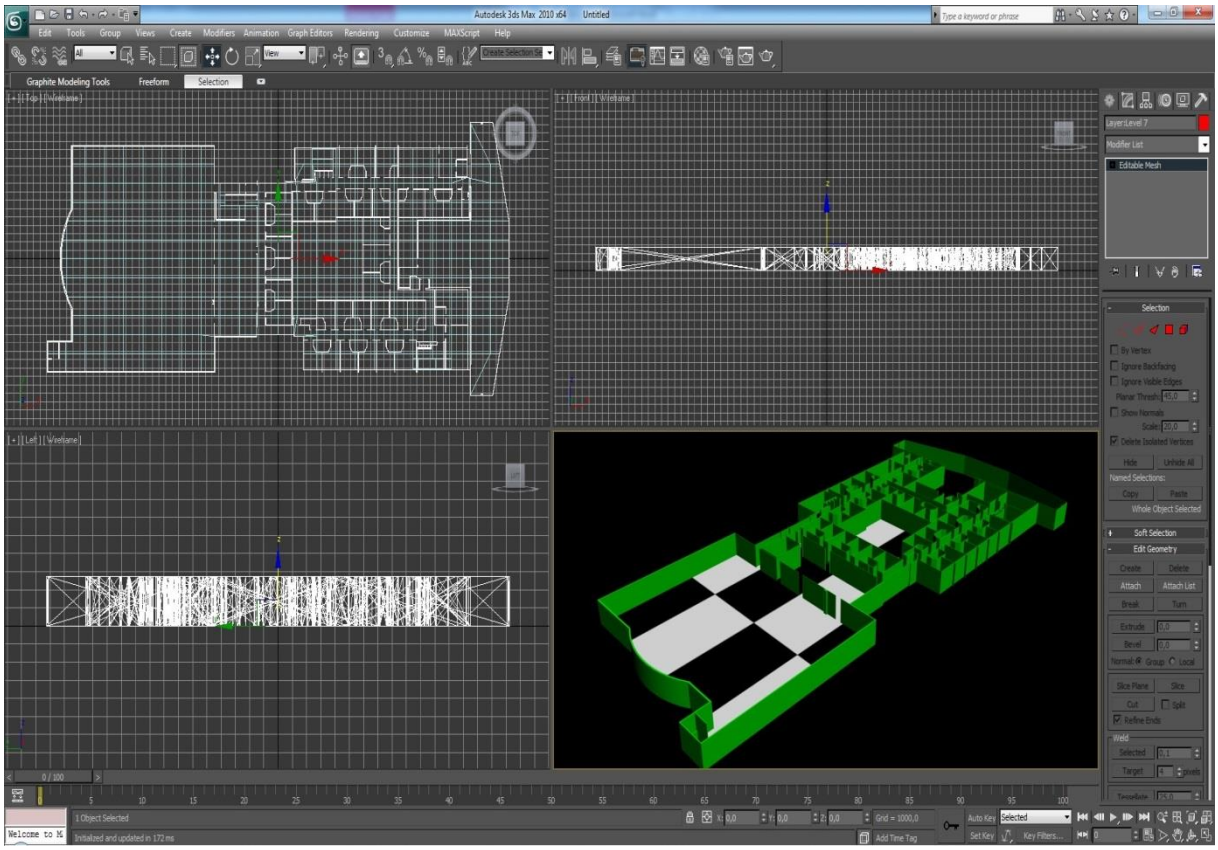

Figure 2.4: deck7 and deck7_ground models

We can see the top, side, front and perspective view of the object under construction, thus having a better 3-D understanding of the scene. At the top, there are the mostly used tools, such as: selecting items or a group of them, position, rotation, size, inversion and reversal transformations, attributes editor, layers, curves and materials. At the bottom there is a range of frames where we change the attributes / characteristics of the object in each frame, thereby giving it the illusion of motion. On the right side there are the most important functions of creating and shaping / transforming objects. For example, we can make several basic 3-D objects (spheres, cubes, cylinders, pyramids, spirals, cones, tubes), extended objects (axles, spindles, cubes and cylinders with rounded edges, prisms, capsules, spirals bonds, polyhedrons) and apply set operations to them (union, intersection, and subtraction A-B or B-A) and various other transformations to create new complex objects. The transformations can apply either to whole objects or only to parts of them and they are of three types:

- Mesh.
- Spline.
- Polygon.

The most important are: bend, bevel, cross section, extrude, flex, melt, mirror, noise, normalize, projection, ripple, shell, skew, skin morph, slice, spherify, squeeze, stretch, subdivide, substitute, sweep, symmetry, taper, tessellate, trim, twist, wave. We can apply a large number of transformations to an object and they are placed in a transformation queue. The queue is a First In First Out (FIFO) collection, so every new transformation will be applied to the resulting object from the previous transformations. An added advantage is that we can disable a transformation so it won't affect the object and that we can also change the transformations order.

Furthermore, on the right side there are the functions of objects color selection, creation of: particle systems to simulate e.g. a fire-smoke objects system, dynamic objects that move because of forces that imitate the physics laws of the real world (e.g. gravity), cameras, light sources and their shadows. It is worth mentioning the ability to construct very detailed two-legged (biped) characters. Initially, we make the mesh which consists of independent parts that correspond to the different body parts and then we apply an underlying invisible skeleton (rigging) that can connect these parts. This independency permits the movement of just one body part if that is needed (e.g. wave only the right hand), and the rigging connection makes the general movement of the whole body to be more natural. Then we can place various materials upon our model (unwrap uvw), which determine how the surfaces reflect the light from the light source(s) of the scene. Usually, the material has a texture applied to it, in order for the character to look "clothed" with it.

The 3D Studio can extract the scene, that we have created, in various file formats, so we can then import and use in other programs like Unity3D. A similarity with Unity3D is the ability to export the scene in an executable file that can run (so we can see the scene) in a computer that does not have the 3D Studio Max program installed. Finally, the 3D Studio, like other programs of Autodesk company, can use and the Autodesk Backburner system. It is

a Distributed Queueing System (DQS) involving several computers of various operating systems (Linux, Windows, Irix) that work collectively on the same network. As the complexity of a scene in the 3D Studio can be very high because of the big number of objects and the high level of detail it may contain, the rendering of the scene from a single computer can take a long time. So, using the DQS, many computers (Render Nodes) in the same network cooperate to achieve a common work, e.g. the rendering, in far less time (comparing to the needed time for a single computer to do the same job). The DQS is composed of:

- A computer (called Render Client) an application of which sends the requested work (rendering) to the Render Nodes.
- Render Nodes are one or more computers that are responsible for the execution of the job.
- A computer (Backburner Manager) who is responsible for the distribution and management of the work of the Render Nodes.
- One or more computers (Backburner Monitor) which monitor the work performed at the Render Nodes.

The four parts communicate with each other using the TCP/IP protocol.

2.2.3 Usage

We will briefly describe the steps in the process of reconstructing the 3-D deck models in 3D Studio Max 2010 from the 2-D designs of AutoCAD:

- Creation of the design in AutoCAD and saving it as a DWG file.
- In 3D Studio we import the DWG file. In the dialog box that automatically pops up, we confirm that all the options in Geometry Options sector (and especially the Cap Closed Spline option) are selected. This is needed, because when we execute the Extrude command later to create the walls, they should be closed in both their upper and lower side.
- We select the 2-D model which is now an Editable Spline, and apply the Extrude transformation so that the lines will be lifted vertically and converted into 3-D walls.
- We then Collapse the Extrude transformation (and therefore the model is converted to an Editable Mesh) and add the Unwrap UVW transformation which prepares the faces (triangles) to accept textures and also indicates the design errors (in green colour). These errors were created because of unclosed polygons (Figure 2.5 top left sub window). We also activate the Backface Cull option in the model's Object Properties (right-click) in the General tab. This should indicate whether which of the faces have wrong direction (normal vector) and consequently they should be flipped.
- We then Collapse the Unwrap UVW transformation and add the UVW Map one. In the parameters, we choose Planar Mapping.

- We Collapse the UVW Map. We confirm that the model is positioned at the coordinate origin (0, 0, 0) and that every part of it is in the same layer and have the same material.
- With the model selected, we choose the Export Selected option and export it as a FBX file.
- We Import the FBX file in the Unity3D software where it will be ready to be used in our scene. We set the Scale Factor in the model's Rig the value 0.001, so that the 3D Studio's mm units will correspond correctly to Unity3D's m units.

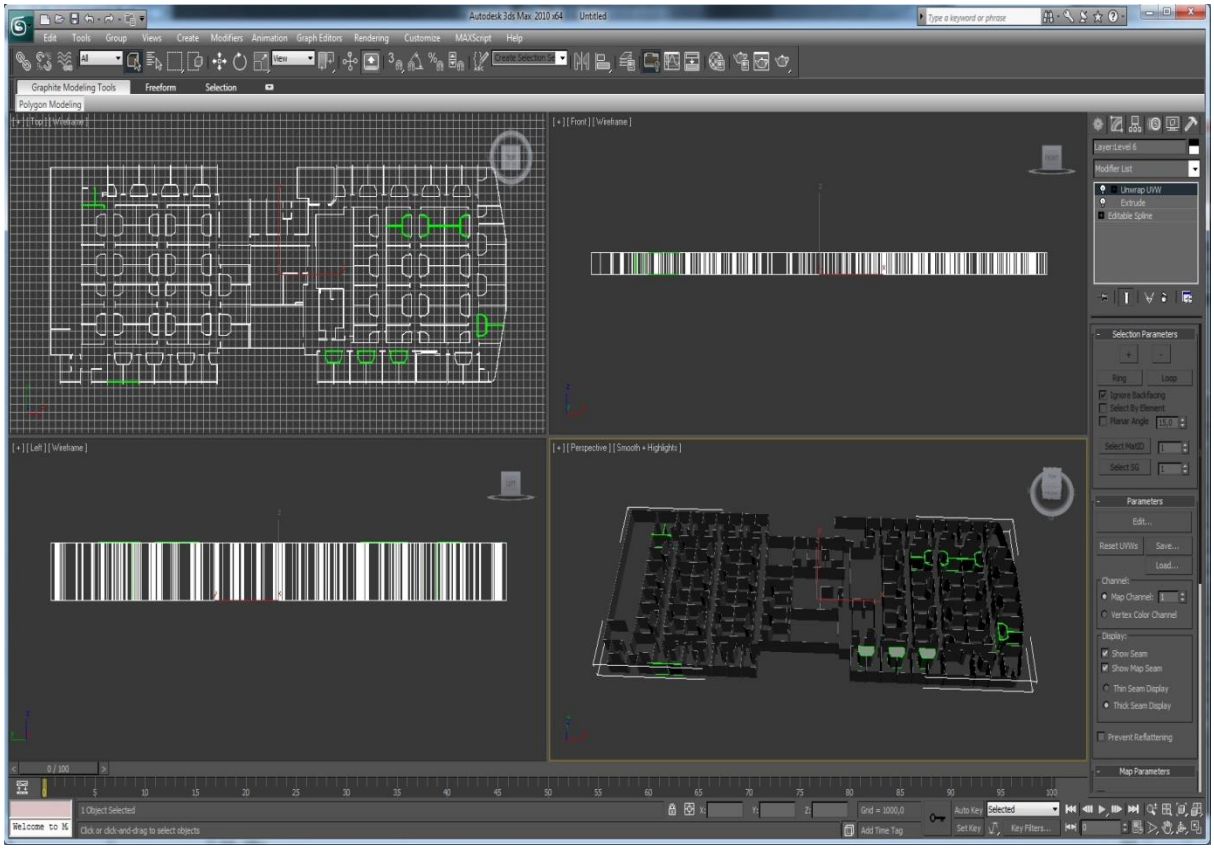

Figure 2.5: Design errors checking with the Unwrap UVW transformation

The height values that we set in the Extrude transformation are: 3200mm for deck5, 2700mm for deck6 and 2750mm for deck7 (Table 1).

For the floor models, we execute the same steps except for the Extrude transformation since they have no height as they are simple planes. The original floors in the AutoCAD were manufactured by first creating a plane and then dividing into several sections so more vertices will be generated. Afterwards, we started moving the vertices to the desired positions so the 2-D floor would take the needed form in all of the decks. Also, we created manually some empty spaces in specific positions, so that the 3-D deck objects in the Unity3D can be connected by stair objects. The stair objects were created in the Unity3D from groups of primitive boxes that were transformed to the sizes of real life stair steps.

For the sake of completion, we present below the Figures of the rest of the decks together with their floors.

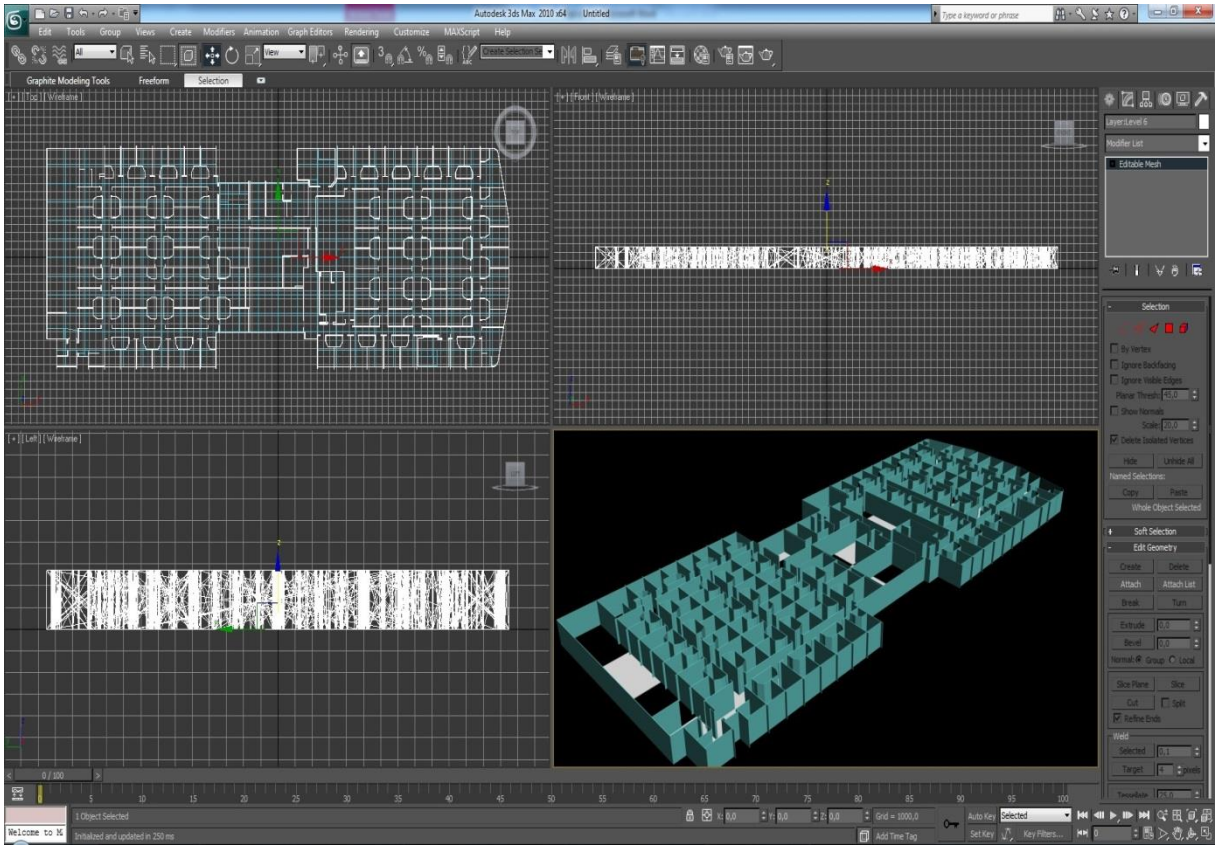

Figure 2.6: deck6 and deck6_ground models

In the floor models we have applied a material with a checkboard texture (Diffuse: Checker in the Material Editor). This is usually done to test if the UVW Mapping is of the wanted type and that there are no other errors. In our case, it is a simple Planar Mapping.

Obviously, afterwards in the Unity3D we can change the checker material with any other combination of material / texture we want to achieve the desired visual effect. For example, we have used a material with a dark brown texture imitating a hardwood floor (see Figure 2.8 or Figure 2.9).

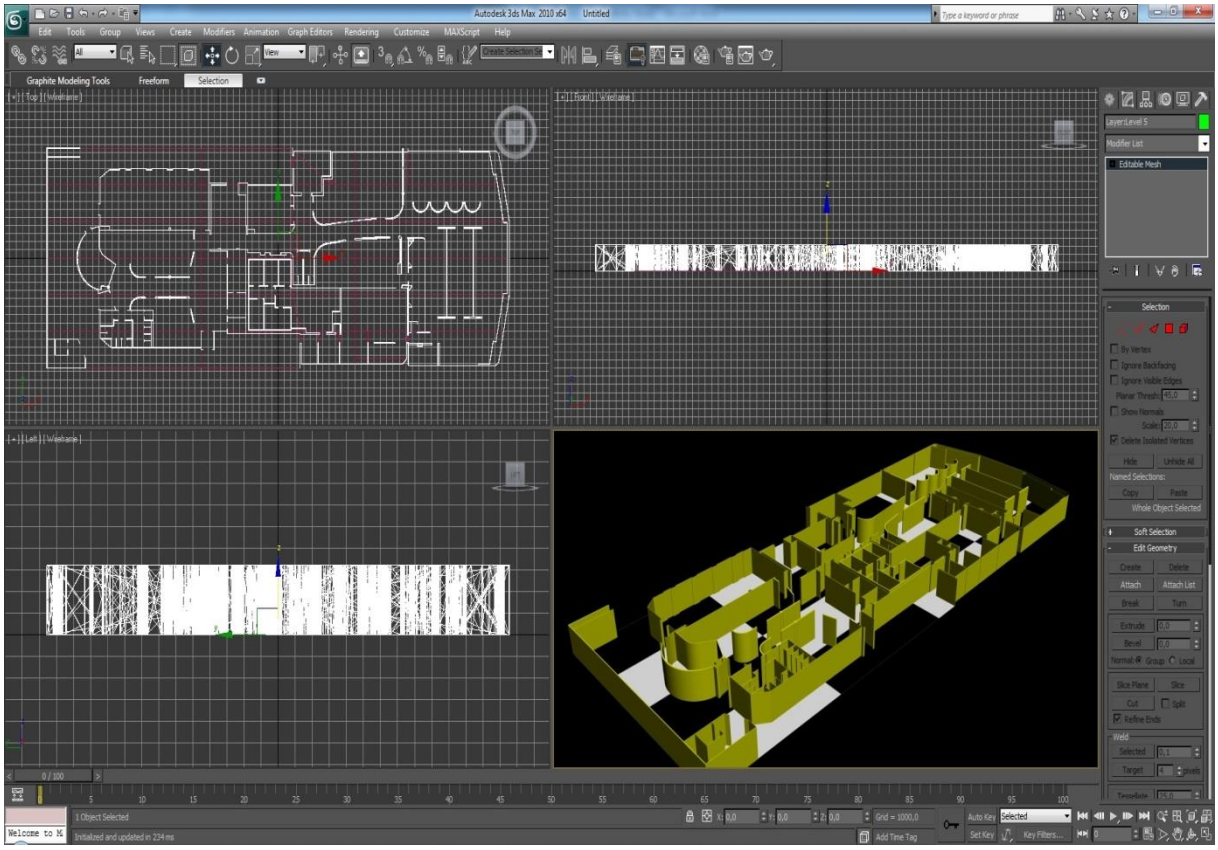

Figure 2.7: deck5 and deck5_ground models

2.3 Unity3D

2.3.1 General

Unity3D software (or Unity for short) has been developed by the Unity Technologies company and includes a complete 3-D rendering engine and an Integrated Development Environment (IDE) where the user can create and evolve 2-D or 3-D applications [7]. To implement new capabilities of an application project, the user can write code and develop scripts for these specific functions. Unity deploys, by default, the (cross-platform) compiler: MonoDevelop, but other compilers can be selected and used as well, such as the Microsoft Visual Studio. The following languages are supported: Boo, C# and JavaScript. The major advantages of Unity are:

- Little time required to learn the basic functions.
- Developing complex and also detailed scenes is quite easy.
- The existence of many integrated systems: Physics System (PhysX), Collision System, Graphical User Interface, Terrain Creator & Editor, Particle System, NavMesh Navigation System, Lighting & Shadows, Animation, etc, which can be smoothly imported into the main application.
- The usage of scripts for the development and customization of applications.
- The ability to export the application in 21 different platforms. The major ones are the following: Pc, Mac, Linux, Web Player, Android, iOS, Windows Phone 8, BlackBerry 10, PS4, Xbox 360, Wii U and Oculus Rift; without the user needing to know the individual IDEs of each platform like the DirectX, OpenGL, OpenGL ES, etc.

Currently (April 2015), it has reached the version 5.0.1. There are two variants of Unity, the personal which is available without any cost and the professional that has to be purchased but has more features such as application development in the cloud.

2.3.2 Operation Mode

Figure 2.8 illustrates the Unity interface, when it is in edit mode:

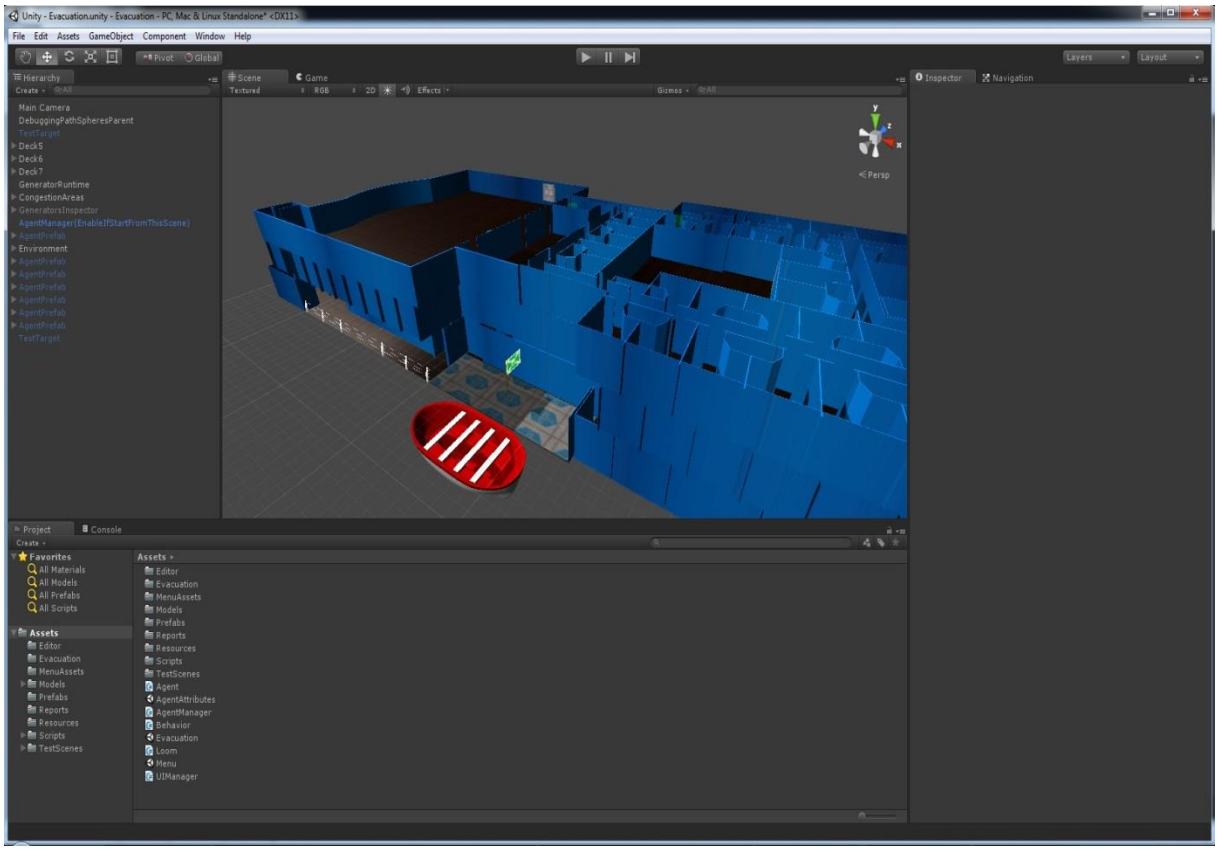

Figure 2.8: Interface of Unity3D (edit mode)

We can see that it is actually divided into four windows. At the bottom (Project) there are all the files (assets) belonging to the application we are developing. Examples of asset files are the 3-D models, images, sounds and music, the scenes that make up the application, the various scripts we have developed, etc.

Out of all the project assets, we can see exactly which ones we are using in a particular scene in the left window, the Hierarchy. Meaning that the Hierarchy is all the objects that make up a particular scene. Those shown in dark blue are objects that have been created from prefabs. The prefabs could be perceived as object molds, from which we obtain similar objects with common properties. Of course, we can create an object from a prefab and later change its properties.

This is done in the right window, Inspector. Having selected the main camera object (Main Camera) in the Hierarchy window, we can see its properties in the Inspector. So here, we can change the values of any object property, e.g. the degrees of the field of view (FOV). Similarly, we can click, select any other object in the Hierarchy and alter its properties values.

Let us emphasize here that in the Inspector, only the public variables of our scripts are visible by default.

In the main window (Scene), our scene is being rendered. Here, all the changes that we make to the transformation properties of our objects will appear, e.g. translating an object or scaling its dimensions. When we press the Play button at the top of the screen, then the Scene window changes focus to the Game window where our application executes and runs. The Pause button temporarily stops the execution until we press again Play, and the rightmost button advances the execution/time by one frame. In Figure 2.9, we have pressed the Play button and Unity is in play mode (play mode). We can see some passengers, which have been created at the beginning of the simulation, walking on the decks.

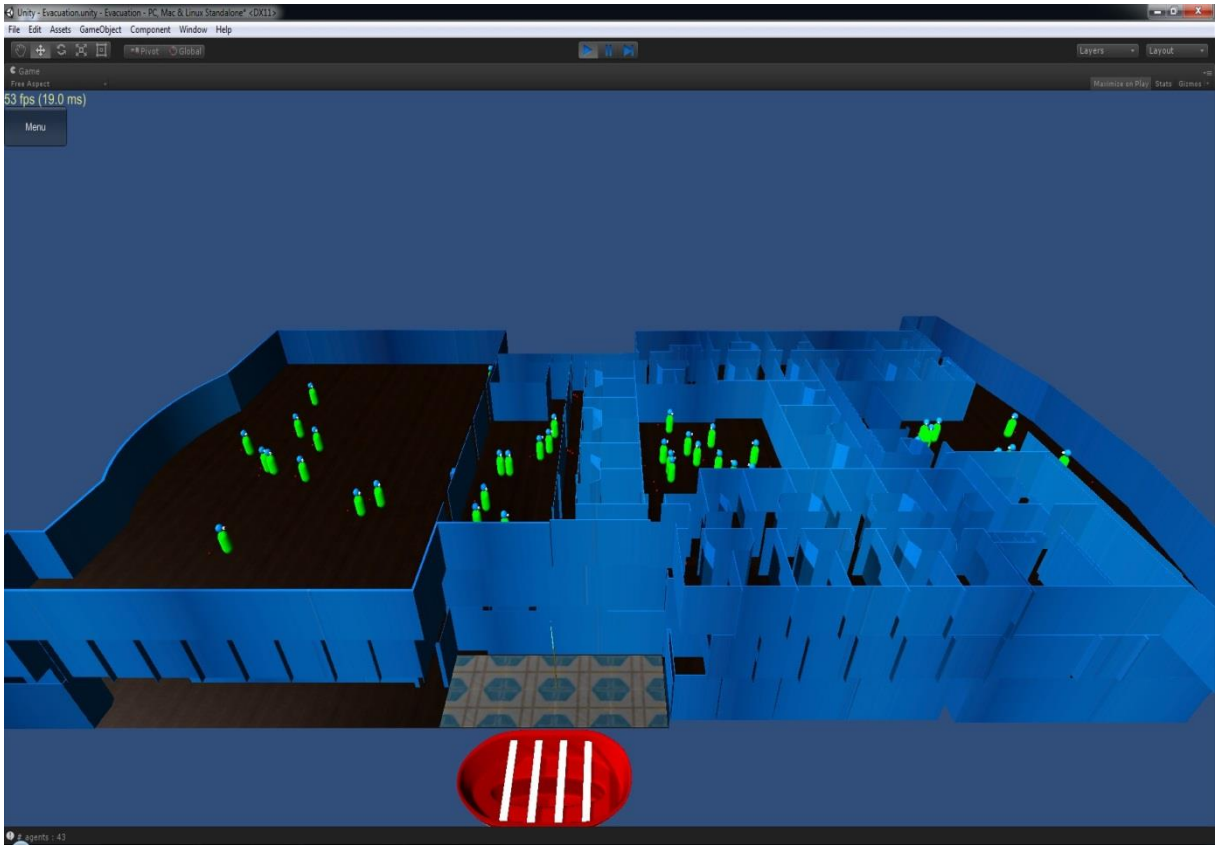

Figure 2.9: Interface of Unity3D (play mode)

In general, the creation of applications in Unity is based on two key concepts: the Game Object and the Component. A Game Object is every object in our scene. Some Game Objects have a visual substance, i.e. a graphical representation (mesh) in the scene so they can be seen (such as the model of a deck or a passenger), while other Game Objects are not drawn/rendered as they serve some function where the visualization is not necessary, e.g. a Game Object which is responsible (through a specialized script) to listen for specific events (such as a proximity event) and forward them to all other Game Objects that represent animated characters (Agents). Also, a Game Object can become a “child” of another Game Object to inherit the translation, rotation and scaling properties of the parent object.

The second basic concept, Component, is applied to Game Objects and gives them the corresponding properties we would. All Game Objects have at least the Transform Component which consists of three 3-D vectors (Vector3): Position, Rotation, and Scale. So, if we want a Game Object to function as a light source in the scene, we add to it a Light Component and then give values to multiple parameters, e.g. the kind of the light source should be Directional and its color should be dark red. Another Component example is: if we would prefer a Game Object to obey the laws of physics and for example to react to collisions with other encountered Game Objects, we could add two Components to it: a Rigidbody and a Collider of some type (e.g. a Capsule Collider).

As we mentioned in §2.2.3, one of the biggest advantages of Unity is also the ability that is granted to the developer to create new features for a Game Object. Meaning, we can program in a language these functions and save them in a script. Then, we can add this script as a Component in any Game Object we want, thus giving it the desired functionalities.

2.3.3 Usage

The process to import the 3-D deck and floor models into Unity were presented in §2.2.3. Their usage inside the Unity environment and the description of how: 1) we modelled a crowd of passengers and 2) we simulated the evacuation procedure in Unity, are the central themes of the next chapter.

Chapter 3

Method Model

In this chapter we will first deal with the advanced evacuation analysis method analyzing its purpose, its affairs and the procedure to calculate the total evacuation time.

Next, we will present how we modelled the ship passengers as autonomous, moving and intelligent entities. For brevity, we will call them Agents.

In the end, we will describe how the physical laws affect the movement of the Agents and how they were implemented in our code.

3.1 Advanced Evacuation Analysis Method

As we saw in §1.1, the Maritime Safety Committee (MSC) of the International Maritime Organization (IMO), in the «Interim guidelines for evacuation analysis for new and existing passenger ships», proposes two different methods for the analysis of the evacuation process. The first (simplified) considers the number of passengers as a fluid motion which is defined by the Navier-Stokes equations. The accuracy of this method decreases as the complexity of the ship increases, i.e. when increasing the number of different types of passengers and accommodation spaces, the number of decks and stairs. For this reason and also because it is easily implemented, it is mainly used only in the initial stages of the design of a new ship to give an approximation of the expected performance of the evacuation.

The second (advanced) method is usually considered as a simulation of evacuation in a computer, but having a more microscopic modelling of people (passengers and crew) [2]. That is, each person is regarded as a separate entity containing information about the topology of the model ship, and the main point of examination is the interactions between these entities as well as between the entities and the modelled ship. We will analyze it more in the next paragraphs.

Let us emphasize here that the requirements of MSC are given in the form of guidelines, which means that they are considered more as recommendations/suggestions than imposed regulations.

3.1.1 Advanced Method Purpose

The purpose of the advanced method is to:

- Identify and eliminate, as much as practically possible, areas of increased congestion which may be created during an evacuation.
- Demonstrate that the escape arrangements are sufficiently flexible to provide alternative solutions if some escape routes, assembly stations, embarkation stations or lifeboats are not available due to an accident.

3.1.2 Advanced Method Assumptions

The following assumptions have been made:

- The passengers and crew are represented as unique entities with specific properties and response times.
- Passengers and crew will use the main escape routes.
- The escape mechanisms are 100% available.
- The crew will be in the right positions to help passengers.
- The passengers follow the instructions of the crew and the emergency signs to reach the assembly stations.
- Smoke, heat and the toxic products of fire do not affect the performance of the crew and passengers.
- There is no family group behaviour.
- The movement and heel of the ship are not considered.

3.1.3 Method of Calculating the Total Evacuation Time

The total evacuation time is calculated by the formula:

$$T_{tot} = 1.25 * T + \frac{2}{3} * (E + L) \quad (3.1)$$

Where **T** is the Travel time, 1.25 is a safety factor, **E** is the Embarkation time and **L** is the Launch time. According to the MSC instructions, the following must apply:

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

Where **n** is the maximum allowable evacuation time and is equal to:

$$n = \begin{cases} 60\ min,\ \ for\ Ro-Ro\ passenger\ ships \\ 60\ min, for\ passenger\ ships\ (except\ Ro-Ro)\ with\ no\ more\ than\ 3\ main\ vertical\ zones \\ 80\ min, for\ passenger\ ships\ (except\ Ro-Ro)\ with\ more\ than\ 3\ vertical\ zones \end{cases} \quad (3.3)$$

This restriction should also apply:

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

The timing requirements of (3.2) and (3.3) schematically:

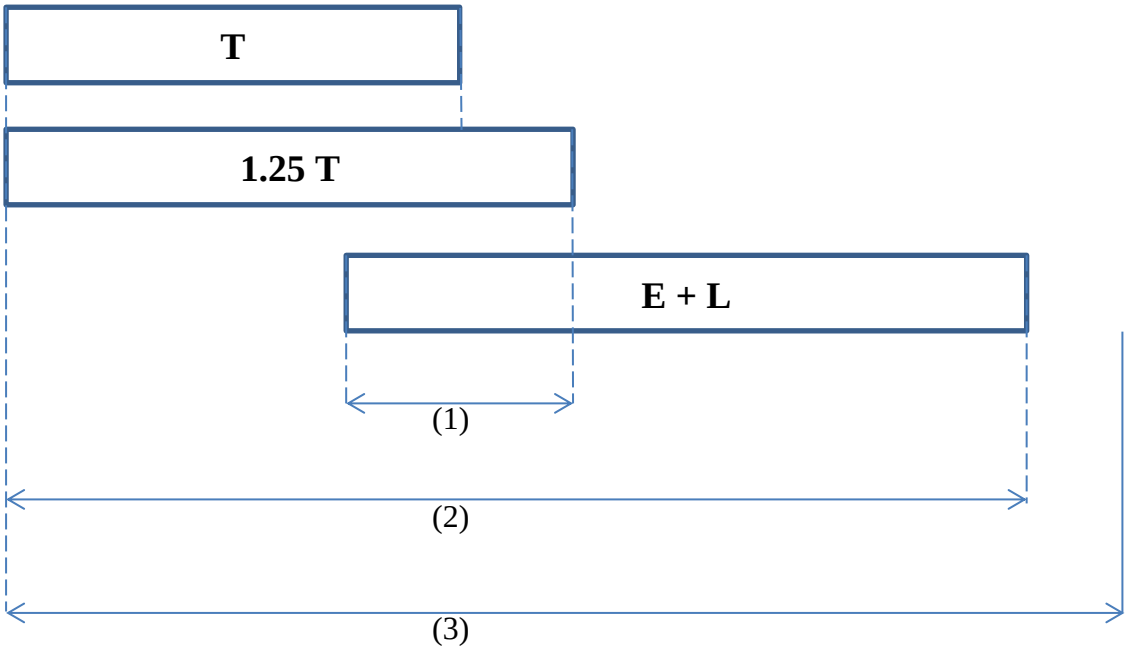

Figure 3.1: Total Evacuation Time

Where:
(1) is the Overlap Time and equals to (E + L) / 3,
(2) is the Calculated Evacuation Time,
(3) is the Maximum Allowable Evacuation Time, n, calculated from (3.3)

3.2 Entity Modelling – Agent

Each individual entity of the simulation that represents persons participating in the evacuation, will be called Agent. Thus, the Agent is the basis of our simulation. It has been given such parameters to convey the human movement as realistically as possible. In addition, it has a set of behaviours that define its general route but also create local small movements, e.g. to avoid another nearby Agent.

According to MSC Circular 1238, Annex 2 (I.M.O. 2007), the Agent model should have at least these characteristics:

- Each person is represented individually (by an Agent).
- The properties of each Agent are determined by parameters, some of which are probabilistic (analyzed in the next paragraph §3.2.1).
- The path each Agent follows is recorded (the user selects the recording frequency, i.e. the number of times per second that the Agent's position is recorded).
- The parameters should differ between individuals that make up the population of the evacuation (according to the Tables 3.1, 3.4 and 3.5 of MSC Circular 1238, Annex 2 (I.M.O. 2007) for the various age groups and the corresponding Agent movement speeds).
- The basic rules for individual decisions and movements are the same for all Agents and are described in a universal algorithm (we have adopted this technique which will be described in paragraph §3.2.2).
- The time difference between two movements of any Agent in the simulation should not exceed one second of simulated time; with the help of multi-threading programming, we succeeded to make all the Agents monitor and adjust their movement every frame. Thus, in a computer that can achieve a high frames per second (FPS) rate, let's say 30 fps, the time difference between the two movements is 1/30 s ≈ 0.0333 s.

3.2.1 Agent Parameters

The Agent parameters are shown in the Figure 3.2. At the top, there is a boolean variable GetValuesFromManager. If this is true (which is its default value), then the Agent does not take into account the values of the parameters that have been given to it through the Inspector of Figure 3.2 (in the Unity Editor), but takes the values from the AgentManager, which is a Game Object that manages and controls all the Agents in the scene.

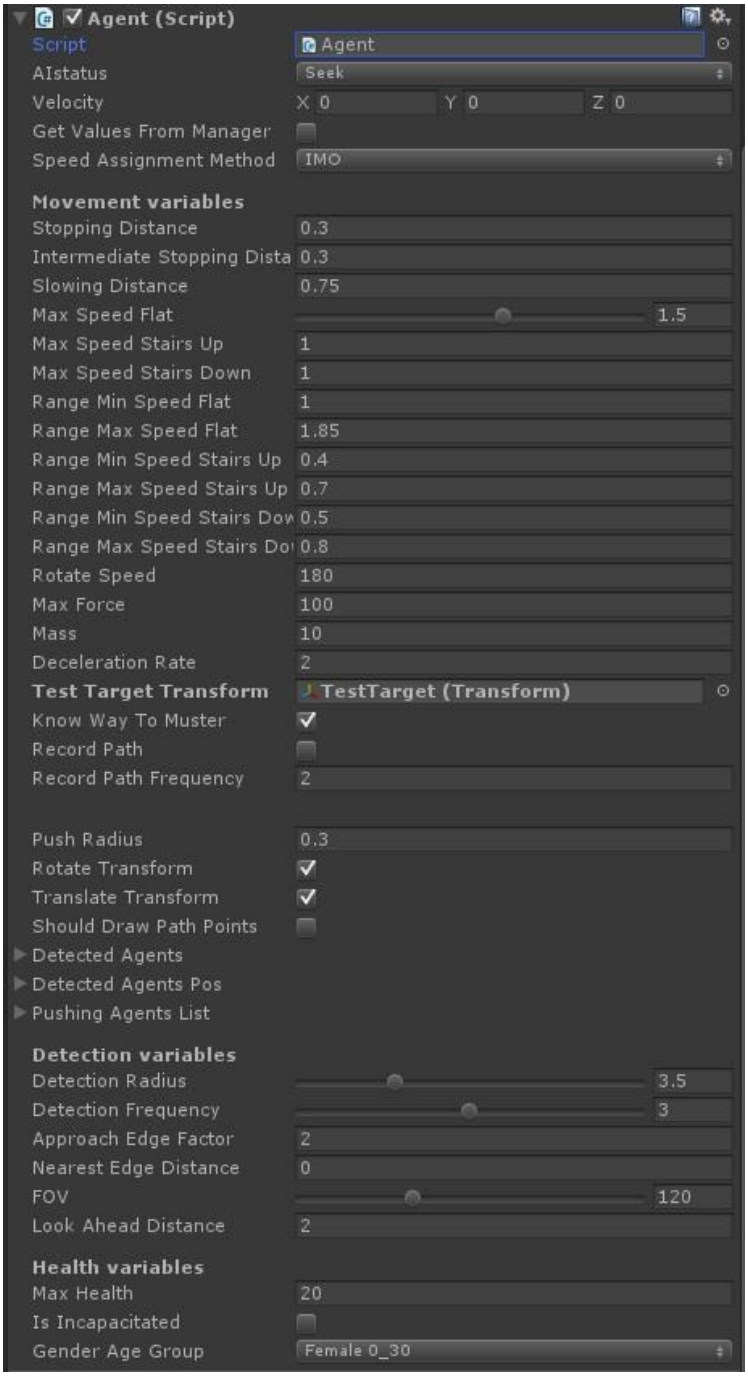

Figure 3.2: Agent Parameters

So, through the AgentManager, the user can easily change the values of the parameters of all the agents. If, however, they wish to give special values to some specific Agents, then they simply give the false value in the GetValuesFromManager variable in the Inspector window, so these Agents will use their own values inputted in the Inspector (Figure 3.2) which can be customized for each one separately.

We have also created a scene in which the user can use a graphical environment, for greater convenience, to give values to the parameters of the AgentManager. This is shown in Figure 3.3. The value changes are appearing in real time. Another alternative for the user is to directly give values to the parameters of the AgentManager in its Inspector.

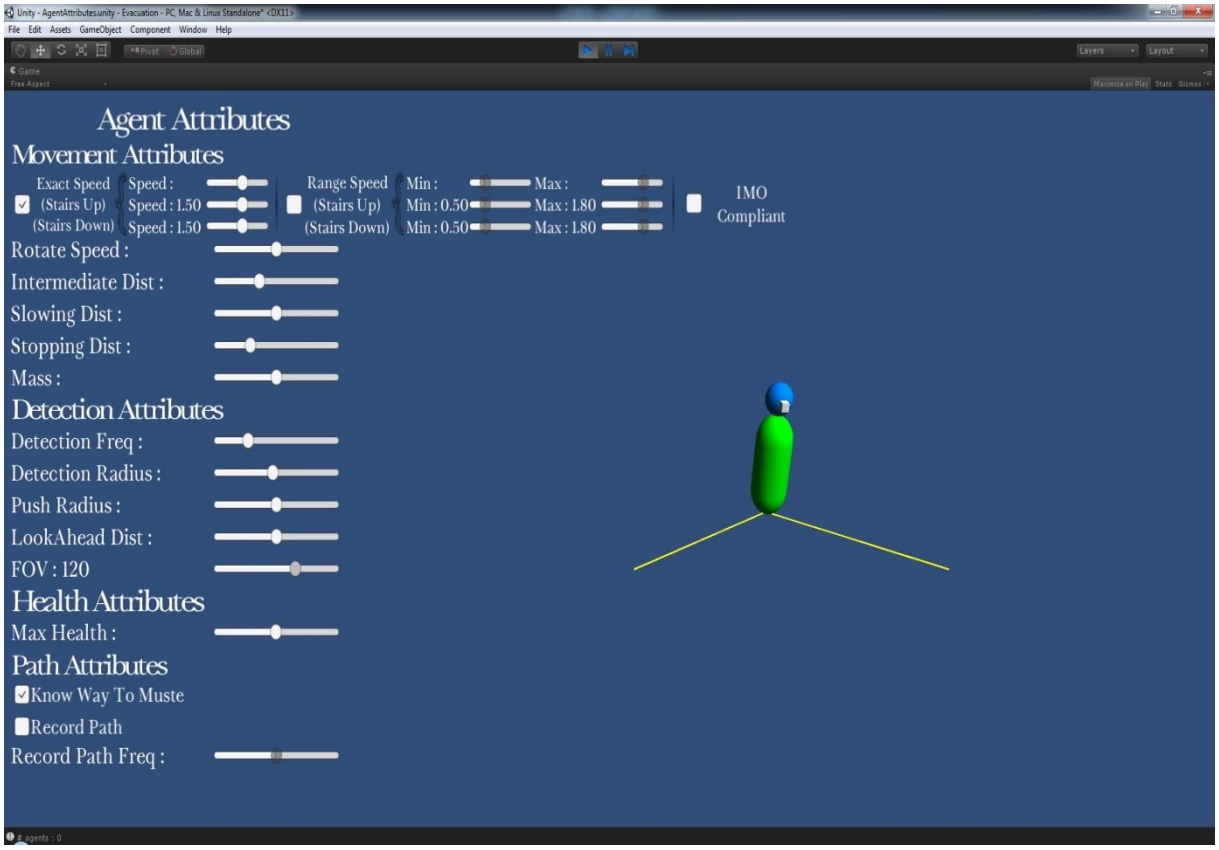

Figure 3.3: Graphical environment for value-assigning to the AgentManager parameters

So in conclusion, there are three different ways for an Agent to get values for its parameters:

- To assign values on each Agent individually in the Inspector and set its parameter GetValuesFromManager = false.
- To set values in the AgentManager Inspector, which all the Agents that have their GetValuesFromManager = true, will "inherit" afterwards.
- To use the graphical environment to assign values to the AgentManager properties and then again all the Agents that have their GetValuesFromManager = true, will "inherit".

In any case, the parameters that characterize and define every Agent are the same, so let's illustrate what each of them represents:

- Initially, the user shoud select the method of determining the maximum speed when the Agent is walking on a horizontal ground and when walking on stairs - ascending and descending. There are three cases:

i The user can precisely set the maximum value for each of the three speeds.

ii The user can set a range [min, max] and the maximum value will randomly take a value within these bounds.

iii The user can choose that the simulation will run according to the MSC guidelines, so the maximum values (shown in Tables 3.1, 3.2 and 3.3) derive from Tables 3.1, 3.4 and 3.5 of Annex 2, respectively.

- Rotate Speed: Angular rotational speed.
- Intermediate Distance: The distance between the Agent and any intermediate node in its path, that is except the last one, to consider that they have "arrived" to this node and should start to move to the next node.
- Slowing Distance: When the distance between the Agent and the last node of its path is equal or smaller than this value, the Agent starts to reduce their speed. Practically, in this distance, the movement behaviour changes from Seek to Arrive. This phenomenon will be further analyzed in §3.2.2.2 and §4.2.
- Stopping Distance: The distance between the Agent and the last node of the path, in which they are considered to have reached their target and completed their path.
- Mass.
- Detection Frequency: The frequency (times per second) the Agent creates an invisible sphere to detect its neighboring Agents. It will be further analyzed in §4.2.
- Detection Radius: The radius of the above sphere.
- Push Radius: It is the radius that defines the bodies of Agents. When two Agents come closer than 2 * Push Radius, then it is considered that their spaces/volumes have been invaded by each other and, consequently, they begin to repel the invading Agent until they are again at a least 2 * Push Radius distance. This will be further analyzed in §4.2.
- LookAhead Distance: This is the maximum distance between two Agents in which the one located behind the other can start a bypassing process.
- FOV (Field Of View): The visual field of the Agent in degrees. It is distinguished in the snapshot of Figure 3.3. This, as well as the LookAhead Distance will be further analyzed in §4.2.
- Max Health: The maximum value of the "health" of the Agent. It is reduced when the Agent is involved in collisions with other Agents that are considered harsh, i.e. when the relative speed exceeds a certain limit. It will be further analyzed in §4.2.
- Know Way To Muster: Boolean variable. If this is true, then the Agent knows the way to the nearest Muster Station. If false, then the Agent does not know the way to the nearest Muster Station so they will move to the nearest sign to get information about where the nearest Muster Station's location.
- Record Path: Boolean variable. If this is true, then the Agent stores its position in a list of Vector3 periodically, defined by the next parameter. From this list, the entire path that the Agent followed until it reached the Muster Station can be extruded.
- Record Path Frequency: The frequency (times per second) the Agent records its position.

If we choose to follow the guidelines of MSC to determine the maximum value of walking speed on a horizontal ground and on stairs, the following procedure is followed:

1. Each Agent is assigned to an age and gender category according to the Table 3.1 percentage values.
2. Depending on the assigned group, the maximum walking speed at horizontal ground get a value randomly in the range [min, max] specified in Table 3.2.
3. Similarly, the walking speed on stairs get a random maximum value for ascending and another value for descending, with the range bounds defined in Table 3.3.

Population Groups - Passengers	**Passenger Percentage (%)**
Male, under 30 years old	7
Male, 30 - 50 years old	7
Male, over 50 years old	16
Male, over 50 years old with mobility limitation-1	10
Male, over 50 years old with mobility limitation-2	10
Female, under 30 years old	7
Female, 30 - 50 years old	7
Female, over 50 years old	16
Female, over 50 years old with mobility limitation-1	10
Female, over 50 years old with mobility limitation-2	10
Population Groups - Crew	**Crew Pecentage (%)**
Male	50
Female	50

Table 3.1: Population composition (age and gender)

Population Groups - Passengers	**Speed at horizontal ground**	
	Min (m/s)	**Max (m/s)**
Male, under 30 years old	1.11	1.85
Male, 30 - 50 years old	0.97	1.62
Male, over 50 years old	0.84	1.40
Male, over 50 years old with mobility limitation-1	0.64	1.06
Male, over 50 years old with mobility limitation-2	0.55	0.91
Female, under 30 years old	0.93	1.55
Female, 30 - 50 years old	0.71	1.19
Female, over 50 years old	0.56	0.94
Female, over 50 years old with mobility limitation-1	0.43	0.71
Female, over 50 years old with mobility limitation-2	0.37	0.61
Population Groups - Crew	**Speed at horizontal ground**	
	Min (m/s)	**Max (m/s)**
Male	1.11	1.85
Female	0.93	1.55

Table 3.2: Maximum speed on horizontal ground

Population Groups - Passengers	Speed on stairs			
	Descending		Ascending	
	Min (m/s)	Max (m/s)	Min (m/s)	Max (m/s)
Male, under 30 years old	0.76	1.26	0.50	0.84
Male, 30 - 50 years old	0.64	1.07	0.47	0.79
Male, over 50 years old	0.50	0.84	0.38	0.64
Male, over 50 years old with mobility limitation-1	0.38	0.64	0.29	0.49
Male, over 50 years old with mobility limitation-2	0.33	0.55	0.25	0.41
Female, under 30 years old	0.56	0.94	0.47	0.79
Female, 30 - 50 years old	0.49	0.81	0.44	0.74
Female, over 50 years old	0.45	0.75	0.37	0.61
Female, over 50 years old with mobility limitation-1	0.34	0.56	0.28	0.46
Female, over 50 years old with mobility limitation-2	0.29	0.49	0.23	0.39
Population Groups - Crew	**Speed on stairs**			
	Descending		**Ascending**	
	Min (m/s)	**Max (m/s)**	**Min (m/s)**	**Max (m/s)**
Male	0.76	1.26	0.50	0.84
Female	0.56	0.94	0.47	0.79

Table 3.3: Maximum speed on stairs

3.2.2 Agent Movement and Behaviour

Initially, we will analyze the behaviours [8] that altogether control the movement of the Agent and then we will describe how this movement is implemented. All the symbols in bold are vectors.

3.2.2.1 Seek Behaviour

The Seek behaviour is the most frequently used. It gives the Agent the desired velocity vector to reach its target. We have changed the algorithms of [9] and [10], from 2-D to 3-D vectors that we are using. Figure 3.4 schematically shows the implementation:

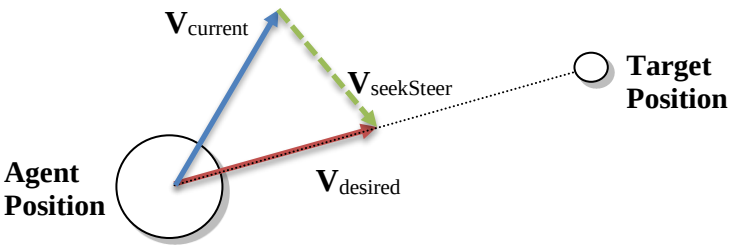

Figure 3.4: Seek Behaviour

Seek Algorithm:

- We know the vector of the current speed, $\mathbf{V}_{current}$.
- We know the position vector of our target, **Target Position**.
- We calculate the desired velocity vector, $\mathbf{V}_{desired}$, as:
  $$\boldsymbol{V}_{desired} = (normalized\ (\boldsymbol{TargetPosition} - \boldsymbol{AgentPosition})) * maxSpeed$$
- In order for the Agent to achieve the desired velocity vector, the following seek vector should be applied to it:
  $$\boldsymbol{V}_{seekSteer} = \boldsymbol{V}_{desired} - \boldsymbol{V}_{current}$$
- Finally, we multiply the above vector by a weight factor $w_{seekSteer}$, so that we are able to control how much the overall final movement vector is affected by the Seek behaviour.

3.2.2.2 Arrive Behaviour

The Arrive behaviour is pretty much like the Seek behaviour, i.e. it leads the Agent to a target, but with the difference that at a certain distance (from the target) the Agent starts to slow down [9]. This distance is the variable Slowing Distance of Agent. The reason for this is to avoid a descending oscillation that is created around the Target Position when the Agent using the Seek Behaviour has high speed and overtakes the target, and then turns back resulting in a repeating back and forth process until they stop. However, using Arrive, the Agent slows down as it approaches the Target Position (and the distance between them is decreasing), and when it is close enough, its speed has decreased to zero. We usually use the Arrive behaviour together with the Slowing Distance parameter of the agent (see §3.2.1), when the agent has almost completed its path and is moving towards its last node.

Arrive Algorithm:

- We know the position vector of Agent, **Agent Position**.
- We know the position vector of our target, **Target Position**.
- We calculate the vector between the two positions, $\mathbf{V}_{\text{toTarget}}$, as:

$$\boldsymbol{V}_{toTarget} = (\boldsymbol{TargetPosition} - \boldsymbol{AgentPosition})$$

- We calculate the distance between the two positions:

$$distance = magnitude(\boldsymbol{V}_{toTarget})$$

- We decrease the speed (the velocity magnitude) depending on the distance and the decelerationRate factor which defines how quickly the Agent is able to slow down:

$$speed = \frac{distance}{decelerationRate}$$

- We make sure that the speed does not exceed its maximum value:

$$speed = Min(speed, maxSpeed)$$

- We calculate the normalized vector of the desired speed by dividing $\mathbf{V}_{\text{toTarget}}$ with the distance (which is faster than normalizing first and then multiplying by maxSpeed as we did in Seek):

$$\boldsymbol{V}_{desired} = \boldsymbol{V}_{toTarget} * \frac{speed}{distance}$$

- Finally $\mathbf{V}_{\text{arriveSteer}}$ equals to:

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

- Similarly, we multiply the above vector with a weight factor $w_{\text{arriveSteer}}$.

3.2.2.3 AvoidWall Behaviour

The AvoidWall behaviour is activated when the Agent approaches a wall. Specifically, it is activated when the distance of the center of the Agent from the nearest NavMeshEdge is less than the Agent's Push Radius parameter. Then a force vector is applied that repels the Agent from the wall. The following Figure 3.5 schematically shows the AvoidWall Behaviour (the symbols in bold are vectors) where we assume that there are no other applicable steering vectors:

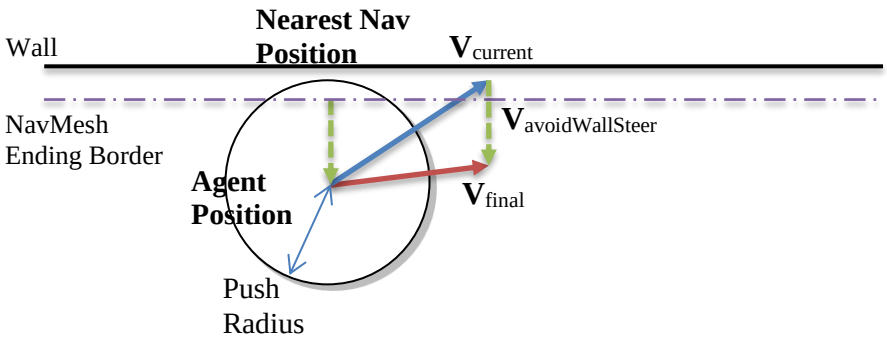

Figure 3.5: AvoidWall Behaviour

AvoidWall Algorithm:

- Every frame, the Agent calls its function FindClosestEdge that returns the nearest point (NearestNavPosition) of the closest NavMeshEdge.
- We calculate the distance between the Agent and this point:

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

- We use a factor, approachEdgeFactor, which we divide the distance by to find the total distance we would like the wall to start applying force.
- If this distance is less than finalPushRadius, then a perpendicular vector with direction away from the wall will be applied on the Agent.
- The magnitude of the avoidWallSteer vector is defined as a linear interpolation of the normal vector obtained from the difference of the Agent's position minus the NearestNavPosition compared to their distance, i.e.:

$$if\ (distance < \text{finalPushRadius})$$

$$\boldsymbol{V}_{normal} = normalized(\boldsymbol{AgentPosition - NearestNavPosition})$$

$$\boldsymbol{V}_{avoidWallSteer} = Vector3.Lerp(\boldsymbol{V}_{normal}, \boldsymbol{V}_{zero}, distance/finalPushRadius)$$

- Similarly, we multiply the above vector with a weight factor $w_{avoidWallSteer}$.

3.2.2.4 PushAgent Behaviour

The PushAgent behaviour is activated when two Agents are found one into the space of the other, where the space of each Agent is defined as a sphere with the Agent in the centre and a radius of PushRadius (see §3.2.1). The PushAgent behaviour is implemented in the same code segment with the next AvoidNearest behaviour, since both need to verify the Agent's position in relation with the positions of its neighbors. So, in order to avoid checking the list of neighboring Agents for the second time, we have written the two behaviours together and we simply return two vectors, one for each behaviour.

The PushAgent Behaviour resembles the AvoidWall and the Separate Behaviour of [11]. Let's see schematically the case of the PushAgent (Figure 3.6), where we assume that there are no other applicable steering vectors:

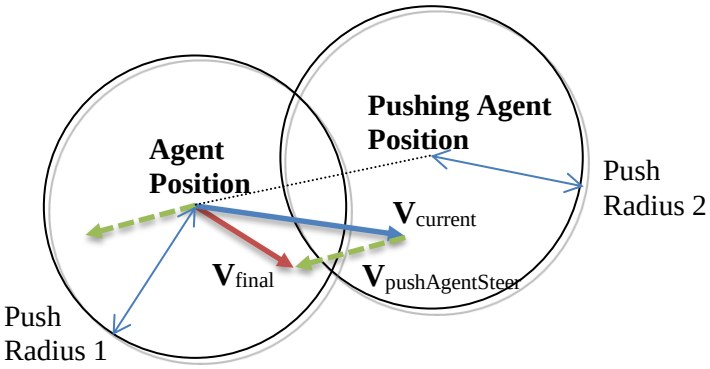

Figure 3.6: PushAgent Behaviour

PushAgent Algorithm:

- Every frame, the Agent checks its position against its neighboring Agents' positions and calculates the vector (for each neighbor):

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

- Then, it calculates the distance between the two Agents, i.e. the magnitude of the previous vector:

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

- If the distance is less than the sum: PushRadius1 + PushRadius2, then the two Agents have entered in each other's space and a steering vector should be applied to push them apart. We normalize the **distanceVec** vector and use a linear interpolation as in AvoidWall:

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

- We multiply with the inverse ratio of the masses, so that the heaviest Agent will receive less force:

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

- Similarly, we multiply the above vector with a weight factor $w_{\text{pushAgentSteer}}$.

3.2.2.5 AvoidNearest Behaviour

The AvoidNearest behaviour [10] is activated just for the nearest Agent existing within the Field Of View of the considered Agent, [12]. As mentioned in §3.2.2.4 as well, the AvoidNearest is executed along with the PushAgent right after the check of the neighboring Agents list. Its function is the same whether the Agent under consideration and the one that is closest to it have velocities with same or opposite direction. The first case is shown in Figure 3.7 and the second one in Figure 3.8. In both cases, we assume that there are no other applicable steering vectors.

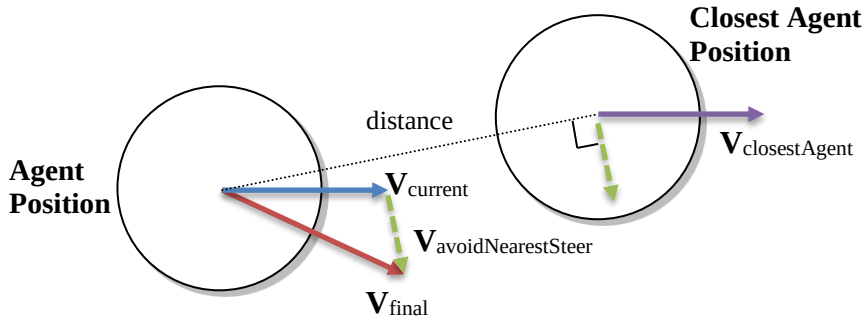

Figure 3.7: AvoidNearest Behaviour – Velocities of same direction

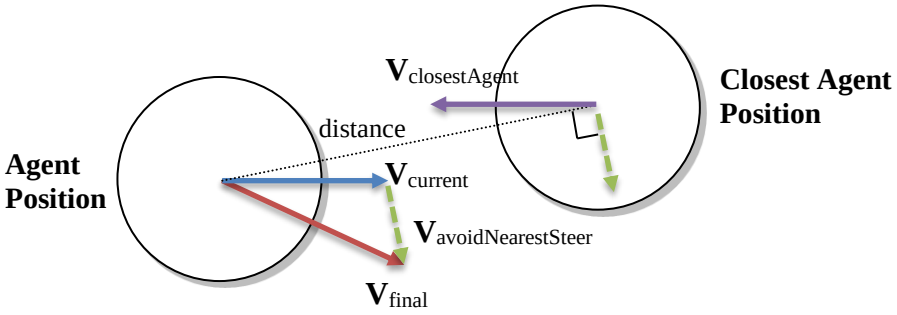

Figure 3.8: AvoidNearest Behaviour – Velocities of opposite direction

We notice that the $\mathbf{V}_{avoidNearestSteer}$ vector remains the same for the two cases, which is expected since it is independent of the velocity vector of the nearest Agent.

AvoidNearest Algorithm:

- In every frame, the Agent inspects its position compared to its neighbors' positions and finds the nearest of those who exists inside its Field Of View [13]. Then, we calculate the sign of the dot product of their velocity vectors:

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

- If cosθ < 0 then the two Agents move towards one another, so an appropriate steering vector must be applied in order to avoid collision. Furthermore, if the two Agents share common velocity directions, but the one behind has greater maxSpeed than the one in the front, then a collision between those two is possible. Therefore, we calculate the vector of their distance and its magnitude:

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

- In order to find the perpendicular vector (to the vector of their distance) which should assist to avoid collision, we will calculate the cross product of the velocity vector of the Agent and **distanceVec** twice:

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

- Afterwards, we use the normalized vector into a linear interpolation, also considering the lookAheadDistance variable (see §3.2.1), which determines the distance where an overtake process should start:

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

- Finally, we multiply it with either 1.2 when the velocities share common direction, or with 2.4 when the velocities have opposite directions. This is because when the Agents have opposite velocity directions, they are going to collide in a smaller time space (than when they have similar directions), and consequently the avoiding perpendicular vector should have a larger magnitude.

- Similarly, we multiply the above vector with a weight factor $w_{\text{avoidNearestSteer}}$.

3.2.2.6 Agent Movement

As we said in §3.1, we tried to create a microscopic modelling of people (passengers and crew). So, we represented each person as an entity that obeys Newton's laws of motion. Thus, the fundamental law of dynamics (Newton's second law) applies to their movement - the bold characters represent vectors:

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

That is, the acceleration **α** of a body of mass **m** results from the sum of the n forces $\sum \boldsymbol{F}$ applied on the body divided by its mass.

Also, the definition of the acceleration is:

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

Where dV is the instantaneous velocity variation in time dt. Therefore, from (3.5) and (3.6) follows:

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

Thus, considering we have a list in our code, the SteerList, with all the steering vectors summed, i.e. the forces applied to each Agent by the various (enabled) behaviours, it will be respectively (where AgentList is a list with all the Agents in the scene):

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

Therefore, the velocity vector of Agent i is modified (every frame) from the total steering vector i. The deltaTime is the time between two successive frames of the rendering of the application.

Similarly, starting from the definition of speed, we find that for the displacement r of an Agent, the physical equation is:

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

Accordingly, the code responsible for changing the position vector of Agent i in the 3-D space is:

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

That is, the position of each Agent is modified by the velocity vector, which is derived from the sum of all steering vectors that are applied, for deltaTime time.

Thus, our model, reproducing and applying the natural physics laws, simulates as close as possible a normal 3-D human movement.

Chapter 4

Simulation Implementation in Unity3D

In this chapter we will describe how we implemented the main scene of the simulation of the evacuation process in Unity3D. Also, we will further analyze the Agent model and will delve in the most interesting parts of the code of the various Scripts we have developed.

4.1 Main Scene

We described in §2.2.3 how starting from a deck design in AutoCAD we ended up in a 3-D model of the deck in Unity. After creating two Directional Lights and the Main Camera in our scene, we imported the deck models with their floors. We define them both as Navigation Static, so we can create a NavMesh from the geometry of every Deck and Deck_ground pair. It is the blue plane that can be distinguished over the floor in Figure 4.1:

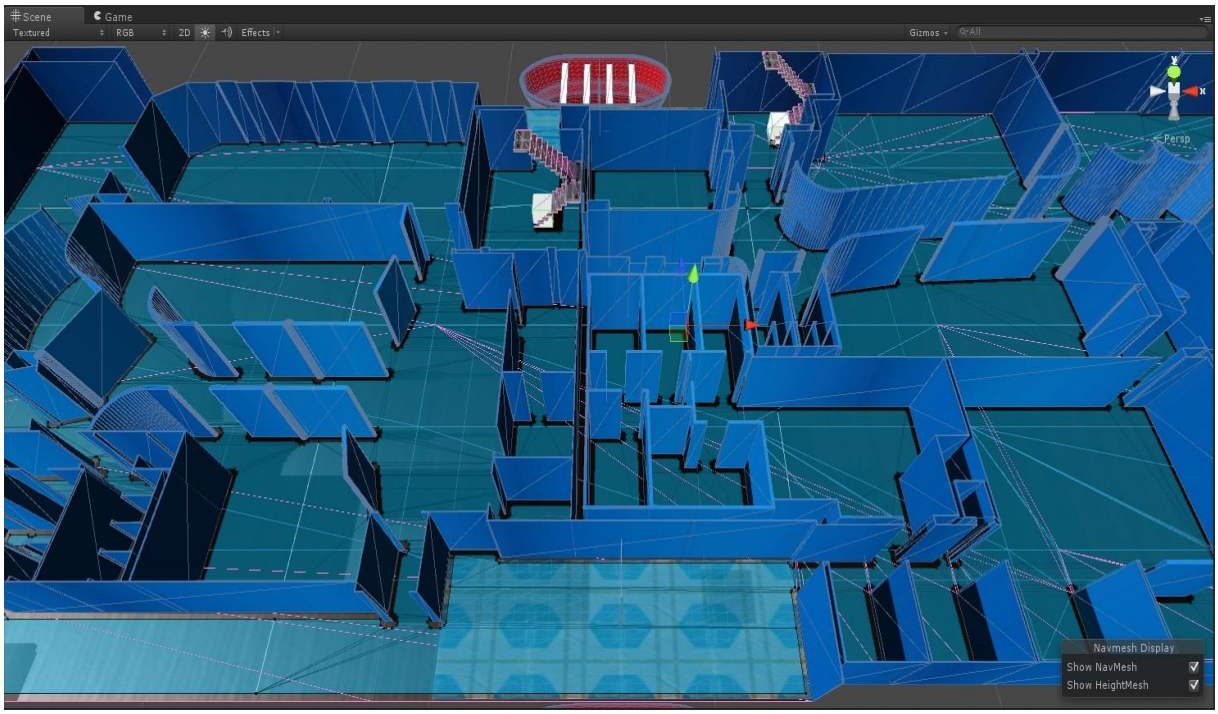

Figure 4.1: Creating NavMesh of Deck5

The NavMesh is created in the Navigation window with the Bake button. In Figure 4.1 there are also shown some purple lines, especially on the stairs, which make up the HeightMesh. Through this, NavMesh levels of different heights are connected. We constructed the stairs in Unity as a composition of many cuboids; they also belong to the Stairs Layer. There are two types of stairs, the first one has 3.2 m height and joins Deck5 and Deck6 and the second has 2.7 m height and joins Deck6 and Deck7. Similarly, we make NavMesh(es) for the rest of the decks. Unity's Navigation system uses these NavMesh(es) in order to create the path to Agents' desired target.

In the scene camera we have added a Script *(FreeCamera.cs*) which manages its movement. Its translation and rotation operate in the same way as in the Editor of Unity. Pressing the 'c' button the camera is moved to a specific position directly above the ship and is facing down, i.e. as a top view. In the same script, with the buttons '5', '6' and '7' we can make the Deck5, Deck6 and Deck7 "visible" and "invisible" along with all the Agents that stand on them. In Figure 4.2 we can see an example where the Deck6 and the Agents that walk on it have "disappeared":

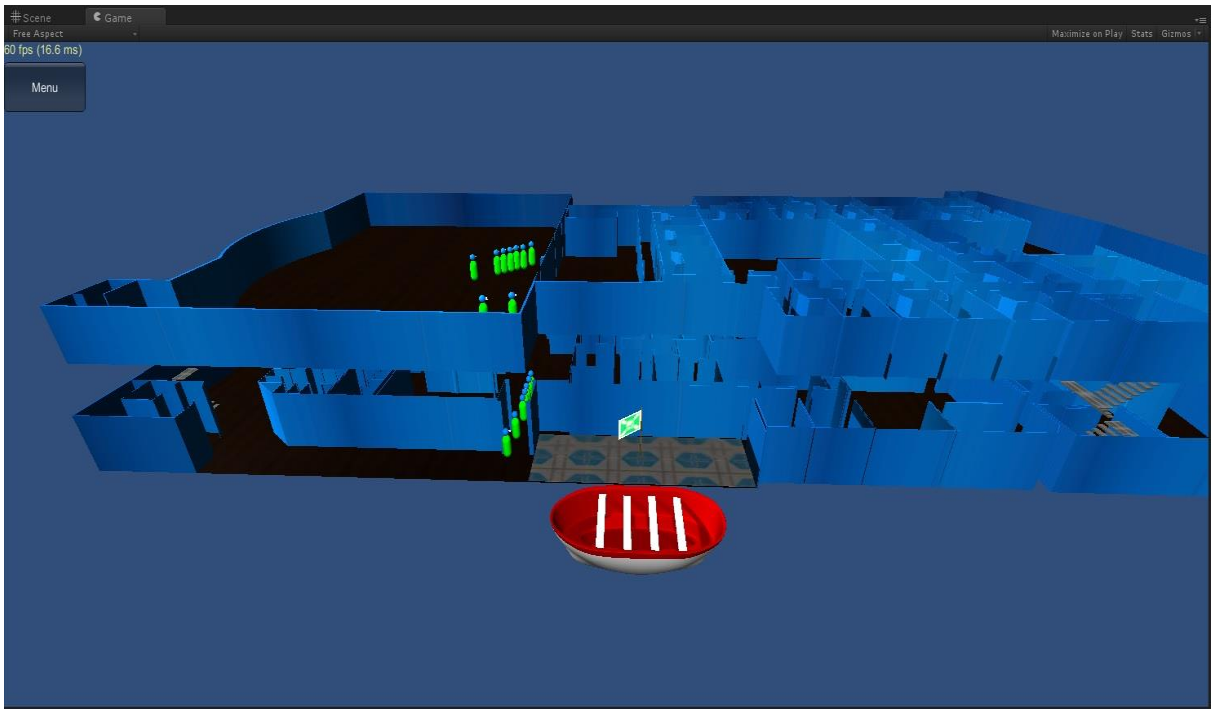

Figure 4.2: Main camera culling of Deck6

This is accomplished by using binary masks corresponding to each Layer. There are three Layers for the three Decks and three Layers for the Agents when they are standing on a Deck. Figure 4.1 shows some (two) white cubes in the bottom of the stairs. Their function is that when an Agent goes down the stairway, it has to go through this cube, so a Trigger collision is called which changes the Agent's Layer as well as all of their "children" Game Objects (script: *ChangeLayer.cs*). Such cubes can be found at the top end of the ladder to change the Layer of Agents going up. The script also disables the MeshRenderer of the blocks so when the application starts, they won't be visible since it is not necessary.

Let us explain how the culling mask of the Main Camera works with an example.

Initially, with the commands:

layerDeck6Mask = LayerMask.NameToLayer("Deck6"); (let = 11_{10})
layerAgent6Mask = LayerMask.NameToLayer("Agent6"); (let = 14_{10})

we find the integer numbers corresponding to Layers: Deck6 and Agent6 (this is the Layer for those Agents walking on Deck6). When the application starts, the culling mask of the Main Camera is (32-bit):

11111111111111111111111111111111

i.e. all the bits are 1 (activated), so all the Layers are visible. If we press for example the '6' button, the following will take place:

layerDeckMask = 1 << layerDeck6Mask;
layerAgentMask = 1 << layerAgent6Mask;

so a new mask is made, layerDeckMask, starting from the value:

00000000000000000000000000000001

and we left shift (<<) it as many positions as needed to reach the integer value of layerDeck6Mask = 11, thus becomes:

00000000000000000000100000000000

Similarly layerAgentMask becomes:

00000000000000001000000000000000

The next command is:

camera.cullingMask ^= (layerDeckMask | layerAgentMask)

So we have a total mask resulting from | (OR) of the two masks:

00000000000000001001000000000000

And we make that mask ^ (XOR) to get the cullingMask which finally becomes:

11111111111111111011011111111111

So now the Main Camera does not render the layers belonging to the disabled bits (0), in this case the Layers: Deck6 and Agent6.

If we press again the '6' button then the cullingMask will be turned back to XOR with the total mask:

11111111111111111011011111111111
00000000000000001001000000000000
11111111111111111111111111111111

so all the Layers will be rendered again.

In order for the white cubes to correctly change the Agents' Layers, when they pass through them, we must have first defined appropriately in the Unity's physics system which Layers collide with which other Layers. This is defined in: Edit => Project Settings => Physics. It is the window shown in Figure 4.3:

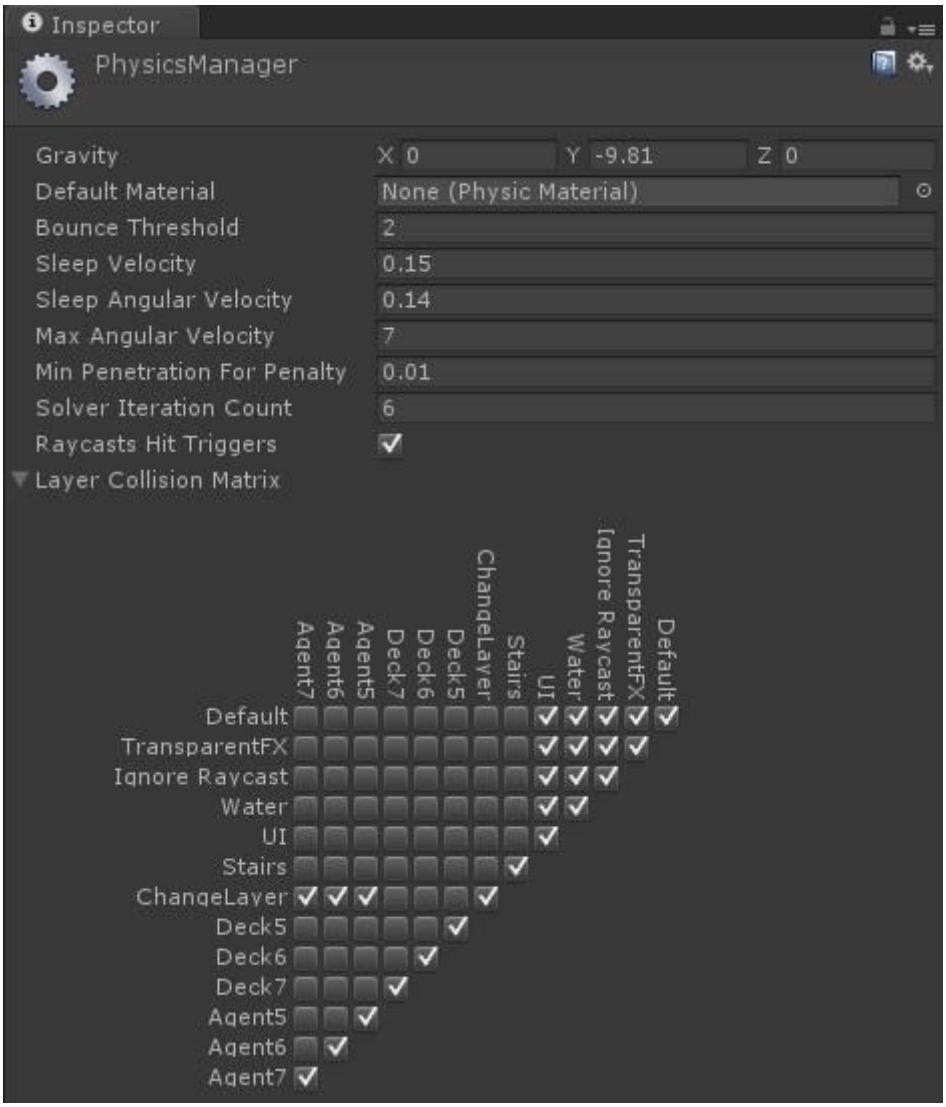

Figure 4.3: Layer Collision Matrix

We are mainly interested in the Layer Collision Matrix, where we can distinguish which Layers interact with each other and cause collisions. Thus, for example, the white cubes with the Layer: ChangeLayer cause collisions with all three Layers of the Agents, Agent5, Agent6, and Agent7. At the same time, ChangeLayer does not cause collisions with any of the three Layers of Decks: Deck5, Deck6, and Deck7. Of course, in both the Agents and white cubes,

the Collisions have been designated as Trigger, so there is no impeding of the Agents' movement; only the event: *OnTriggerEnter* is simply activated.

There is another script (*ShadersControl.cs*) in the Main Camera which changes the materials of the Decks and Deck_grounds Game Objects. In each Deck and Deck_ground we have added two Materials. So, by using different Shaders to those materials, we achieve different visual combinations. In the Figures we have examined so far, both the Decks and the Deck_grounds have Materials with the ordinary Diffuse Shaders. In the following Figure 4.4, we can see in the Materials of Deck7 and Deck7_ground that there is a custom Shader, Glass, resembling the appearance of glass. The visual effect is that we can see the Agents behind the walls of Deck7 and under Deck7_ground (i.e. those on Deck6):

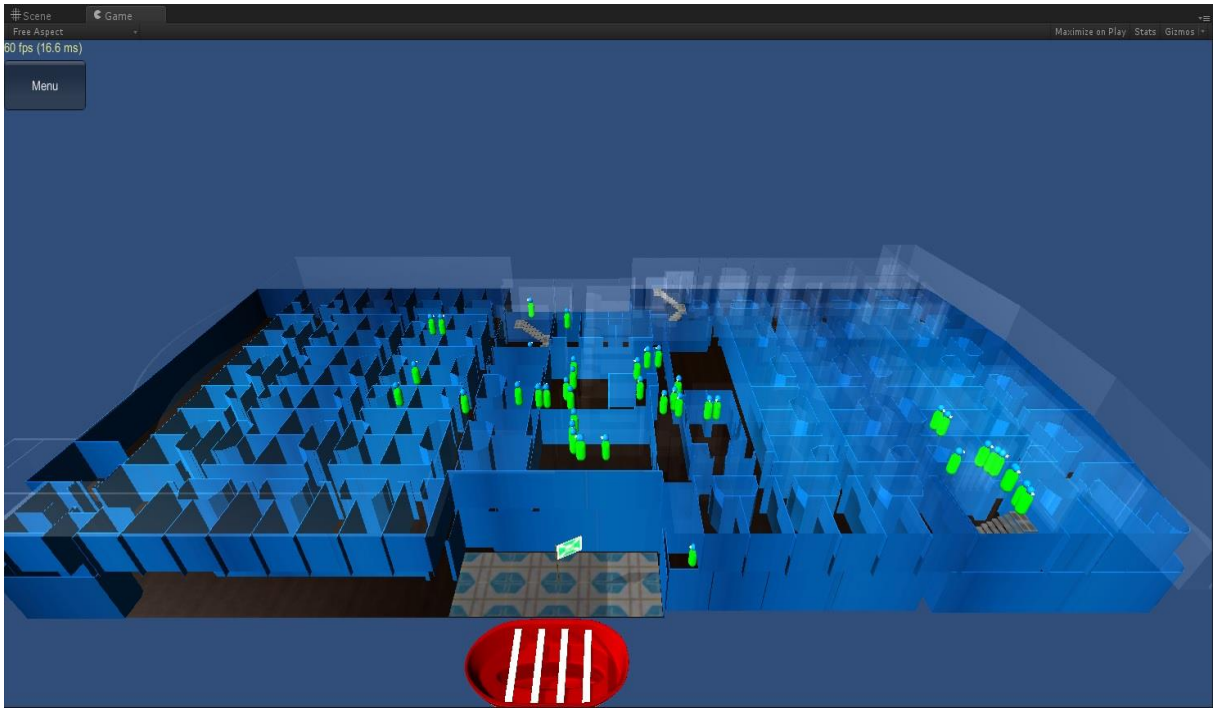

Figure 4.4: Deck7 and Deck7_ground with Glass Shader

Another example is shown in Figure 4.5, where we have used a combination of two Shaders in Deck7 and Deck7_ground. The first one is a predefined Transparent-Diffuse with an Alpha value (which defines the percentage of transparency) of 80 out of 255 (including 0). The second one is another custom Shader, Outline, highlighting with a black line the contours of the Meshes that it have been applied to.

The combinations of the Shaders alternate by pressing the '1' , '2', and '3' buttons for Deck5 / Deck5_ground, Deck6 / Deck6_ground, and Deck / Deck7_ground respectively.

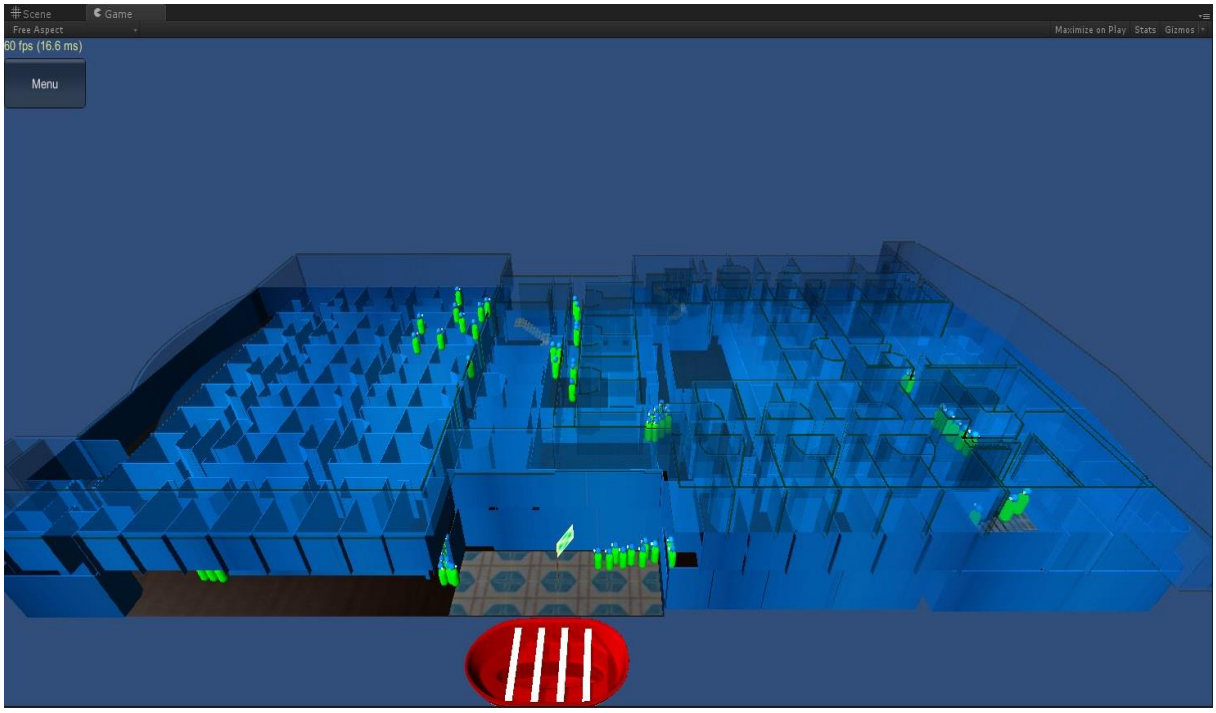

Figure 4.5: Deck7 and Deck7_ground with Transparent and Outline Shaders

In the main scene, there is also an object called GeneratorsInspector. This Game Object has a component which is an Editor Script (*ObjectGroupInspector.cs*). The editors Scripts must necessarily exist in the Assets\Editor folder and do not work at runtime. Their use is limited only when Unity is in edit mode. This particular Script helps us to quickly place Prefabs (i.e. pre-made Game Objects with all appropriate Components) in our scene. We mainly use it to place the Prefabs: GeneratorAgent, i.e. “generators” of Agents. While in edit mode in Unity, we activate the Editor Script with its first button which then turns green. Afterwards, we can choose which Prefab we wish to put inside our scene. The Script searches the Prefabs folder and presents all those found in a list. So, we can select the Prefab: GeneratorAgent and alter the parameters underneath. It is placed exactly at the mouse cursor position when the right mouse button is clicked. The interface of our Editor Script is shown in Figure 4.6:

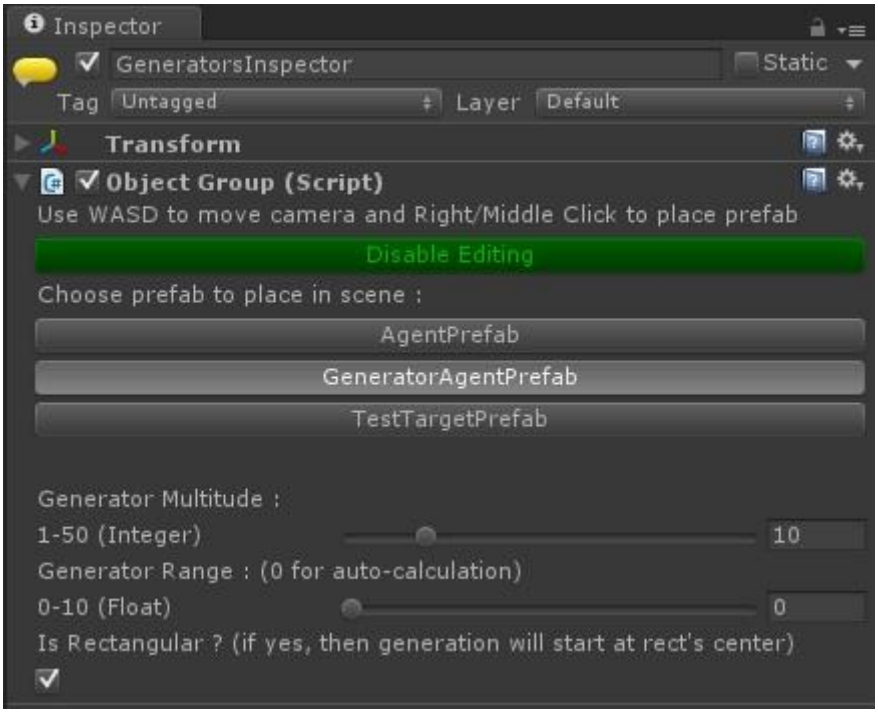

Figure 4.6: Editor Script for placing the Prefab: GeneratorAgent in the scene

The Generator Multitude parameter defines how many Agents will be created by the GeneratorAgent when the application starts. The next parameter Generator Range sets the radius of the circle (if the last IsRectangular parameter is false) within which the Agents will be created. If we choose as Generator Range = 0 and the last IsRectangular parameter is true, then the GeneratorAgent "shoots" four rays, north / south / east / west and counts the distances of the nearest walls which stop the rays. Then it transports itself in the center of the rectangle that is formed by these distances and generates the Agents in random positions inside this rectangle.

Let's take a look on the corresponding Script on the GeneratorAgentPrefab that after "inheriting" the parameters from the Script Editor, it will perform the aforementioned steps for the creation of the Agents (Figure 4.7):

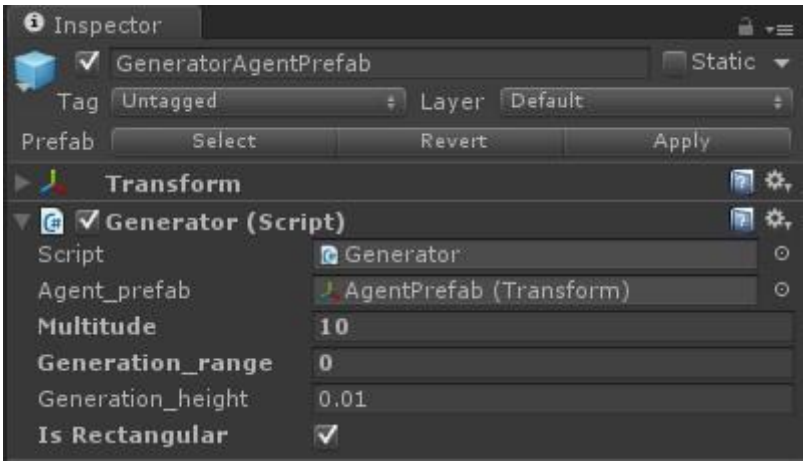

Figure 4.7: Generator Script for the creation of the Agents

Obviously, we can select the AgentPrefab in the Editor Script and right-click to place one Agent in a specific position. The underneath parameters are not considered, as the Editor Script checks if the selected Prefab has a Generator Script on it or not, in order to pass the parameters values to it.

Thus, we have got a fast way to create GeneratorAgent objects in various positions which in turn will produce specific Agents crowds.

In the main scene there are some Game Objects named CongestionArea placed in specific locations on the decks. Since their function is to measure the traffic congestion, they are placed in areas that we expect congestion problems to arise, such as entrances to staircases. They are Trigger Colliders that with the Script: *Congestion.cs* count how many Agents are located within their volume. The counter is incremented when there is an Event: *OnTriggerEnter*, i.e. when an Agent enters them, and decreases when there is the Event: *OnTriggerExit*, i.e. when an Agent exits. At the end of the simulation, the user can store the congestion values throughout the evacuation. The Congestion values are measured with a user-defined frequency (checkFrequency) and are stored in a list (CongestionList). Also, the maximum Congestion value acquired is recorded. The storage process will be further discussed in §4.4. In the picture below we can see a CongestionArea:

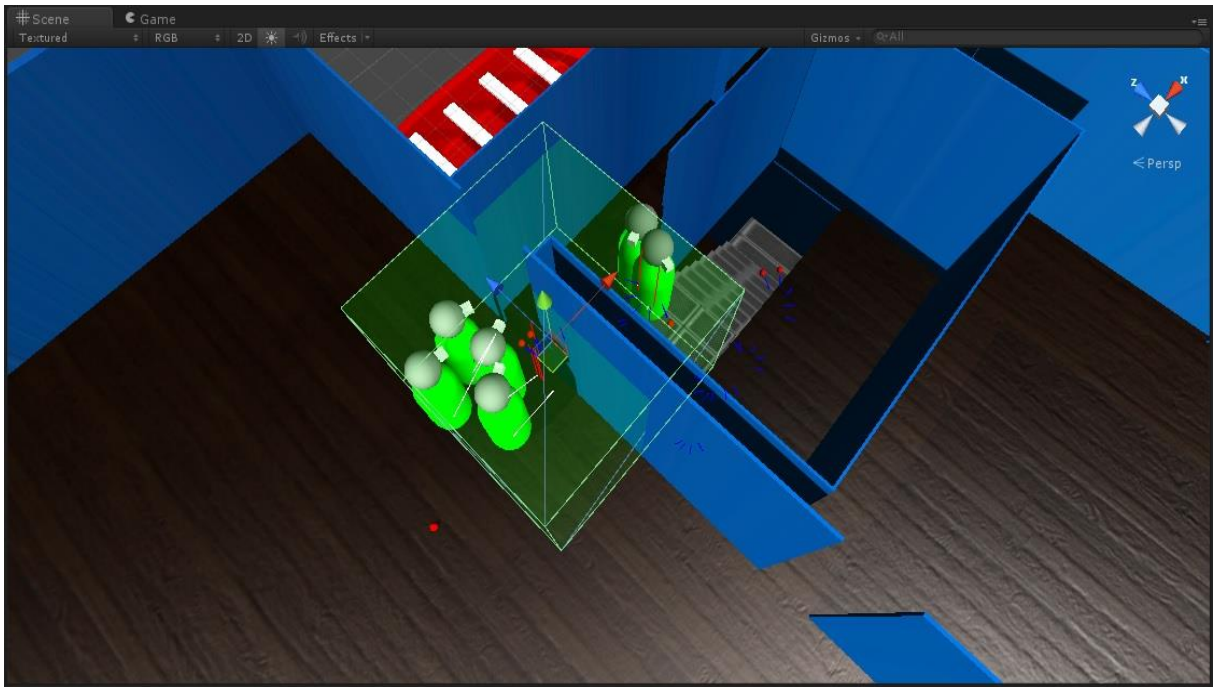

Figure 4.8: CongestionArea Trigger Collider

The last objects of the main scene are the two Game Objects: MusterStation. These are the muster stations which all Agents are trying to reach to. Like the CongestionArea, the MusterStation is basically just a Trigger Collider (with the Script: *MusterStationCollider.cs*). Their use is two-fold (further analyzed in §4.4):

a. To destroy the Agents that touch the MusterStations. This is done to avoid big concentration of the Agents in a small area which results in decreased application performance.
b. To inform the AgentManager to check if the specific Agent, that has just entered the area and got destroyed, was the last moving Agent. If so, then the simulation of the evacuation is complete.

Figure 4.9 shows the Trigger Collider of the MusterStation:

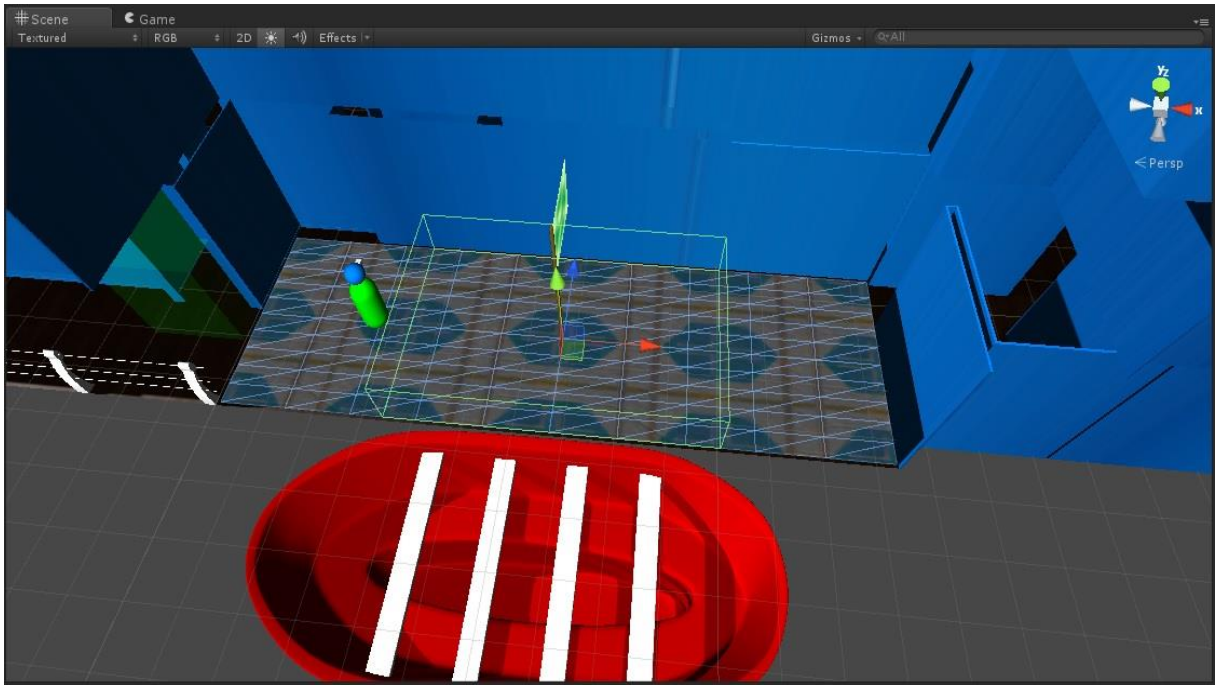

Figure 4.9: MusterStation Trigger Collider

4.2 Agent Analysis

In this section we will analyze in depth the functions of both the Agent and the AgentManager (who is controlling all the Agents in the scene). In Figure 4.10 we have started the application using the Play button, and then we have temporarily stopped it with the Pause button, so we are back in the Scene window. In the Agent's Script (*Agent.cs*) we have added appropriate Debug commands (such Debug.DrawLine, Gizmos.DrawWireSphere and Gizmos.DrawRay) in order to see more visual information in the Scene window. This information will help us to better understand the actions performed by the Agents. We emphasize that this visual information is not shown in the Game window or when we run the executable. In Figure 4.10 the Agent that is lower in the screen is selected:

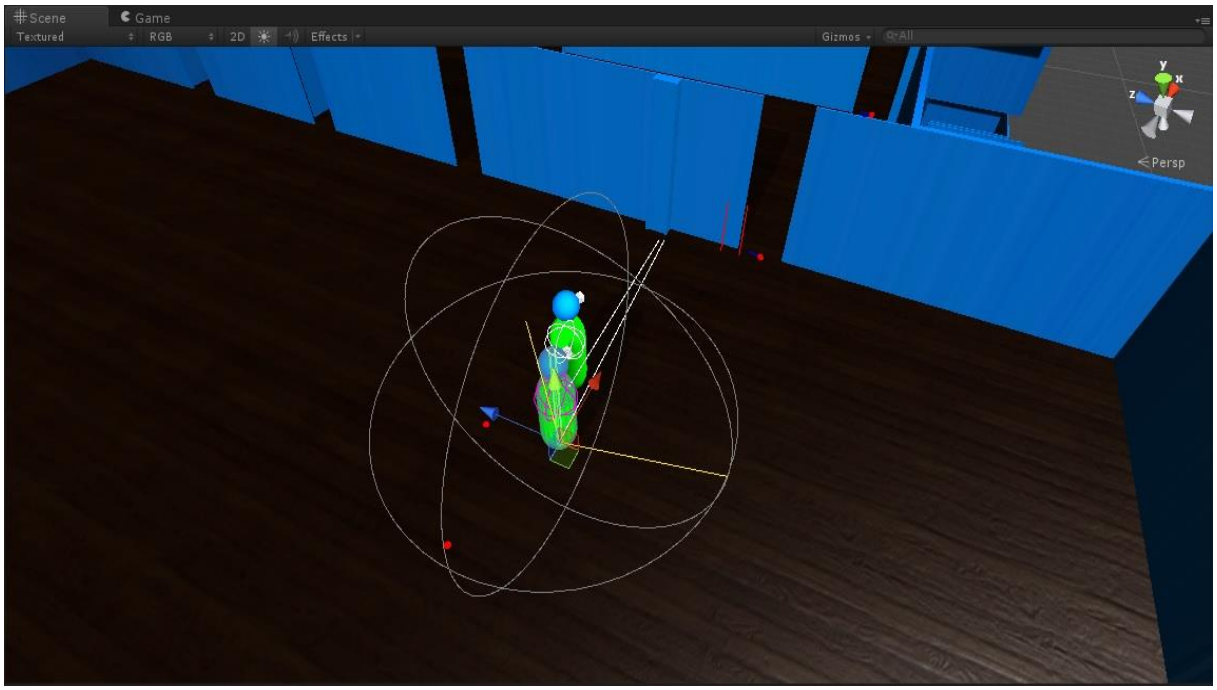

Figure 4.10: Debug information in the Scene window

When the Agent is created, the following initializations are made:

- It becomes part of a list of all the Agents which the AgentManager maintains.
- It checks the height they are located and compares it to the heights (i.e., the coordinate y) of the Decks, to get the Layer of the correct Deck.
- If the boolean variable, getValuesFromManager, is true, then it will get all the values of movement parameters, detection and health by the AgentManager. Otherwise, it will use its own personal values.
- It checks if it has a target position. If so, it starts moving towards it. If not, then it checks its boolean variable, knowWayToMuster, and if that is true then it will create a path to the nearest (known) Muster Station. Otherwise, it will build a path to the nearest sign to get further information.

Also, it starts periodically calling functions (we will explain further right below):

- *DetectNeighbors*: called with detectionFrequency frequency.
- *CheckIfStuck*: called with checkIfStuckFrequency frequency.
- *RecordPath*: called with recordPathFrequency frequency.
- *CheckIfOnStairs*: called with a frequency of two times per second.

DetectNeighbors performs the function of the built-in Unity Physics system:

Physics.OverlapSphere()

This creates a sphere, centered in the position of the Agent and with a radius equal to detectionRadius, and returns all the Colliders that are inside. The sphere is illustrated with the gray lines in Figure 4.10. Using the right Layers and masks, we achieve that the *OverlapSphere* will return only Agents and no other objects, such as walls / floors, and spefically Agents that are in the same Deck. In order for this function to work, our Agents must have two Physics Components: Rigidbody and Collider (Capsule). Let us note here that the Rigidbody has been designated as Kinematic and the Collider as Trigger, so the bodies of the Agents will not repel each other from the Physics system. We simulate repelling using the behaviour (§3.2.2.4) PushAgent when the Agents are very close. The radius PushRadius, which PushAgent depends on, appears as a purple sphere of an equivalent radius in the center of the Agent.

After *OverlapSphere* has returned all the Colliders found within, the *DetectNeighbors* finds the Agents that these Colliders belong to, and then stores those Agents in a list, detectedAgents. That is, with this process, each Agent comes to have a list of all its neighboring Agents. This list will be processed afterwards by the AgentManager in PushAgent and AvoidNearest behaviours. The AvoidNearest avoids the nearest Agent within the FOV. The FOV is illustrated with the two yellow lines.

The *CheckIfStuck* is executed every one second (a value of checkIfStuckFrequency empirically and practically set) and assigns into a variable the distance of the Agent from the very next point of its path. Then it checks if the current distance and the distance that was registered in the previous call of the function differ by a small number. If the difference is greater than this small number, then the Agent proceeds normally along its path. If, however, it is not, this means that the Agent has moved very little during one second, so it is probably "stuck" in a corner. That happens sometimes due to the repulsive forces acting between the Agents as they are walking closeby to each other. It may happen that an Agent gets pushed out of his correct path and gets "stuck" in a small space. In this case, the Agent creates a new path to its target position and thus gets "unstuck".

In Figure 4.11, we can see an example where the selected Agent is pushed by the other Agents out of the corridor and gets “stuck” in the corner:

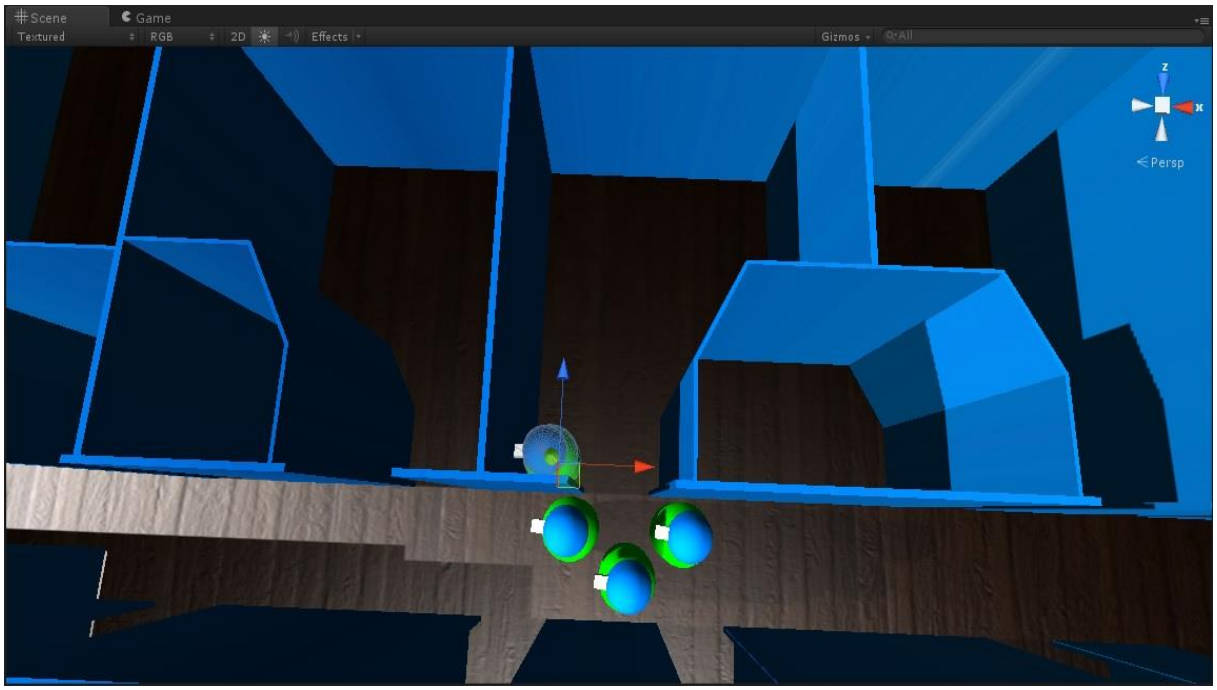

Figure 4.11: “Stuck” Agent Example

RecordPath is executed only if the Agent’s recordPath variable is true. Then, the Agent registers its position in a list, recordedPath, with recordPathFrequency frequency. At the end of the simulation, we can unite all the positions of the list with straight line segments using the Component: Line Renderer of the Agent. So, basically, a continuous line is created which shows the path travelled by the Agent from its starting position up to the Muster Station.

The *CheckIfOnStairs* function uses the *SamplePathPosition* function of its NavMeshAgent Component:

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

and checks the position of the Agent in the NavMesh. The check result is stored in the stairsHit parameter and because this is an out parameter, i.e. it acts as pass by reference, if the function *SamplePathPosition* changes its value, the new value will remain even after the function is returned. So, afterwards, by making a binary AND (&) the stairsHit.mask with the mask Layer of the stairs we can find if the Agent is walking on stairs or on deck ground.

If the Agent walks on the deck, then we set the variable:

maxSpeed = maxSpeedFlat

If the Agent walks on stairs, then we set the variable:

maxSpeed = maxSpeedStairsUp

or

maxSpeed = maxSpeedStairsDown

depending on whether ascending or descending the staircase respectively. The recognition if the Agent goes up or down is done by comparing the height of the second to next point of the path to the height of the current position.

Then, the AgentManager uses this maxSpeed variable to control (and clamp) the movement of every Agent. At this point, the initialization of the Agent's periodic functions is done.

Aftewards, the Agent checks the value of the boolean variable knowWayToMuster. If this is true, then it is considered that the Agent knows where all Muster Stations are, so it will calculate the distances from every one of them and will then finally make a path to the nearest, using the function:

navAgent.CalculatePath()

If it is false, then it is considered that the Agent does not know where the Muster Stations are located, but it always knows where the traffic-instruction signs are, so, similarly, it will create a path to the nearest sign where it will get information to make a new path to a Muster Station or to another sign.

In both cases, in order for the Agent to be able to use the function *CalculatePath*, it must have the Component: NavMeshAgent.

When *CalculatePath* is executed, it returns a path of NavMeshPath type, towards the target the Agent wants to move to. We process this path, using the function:

OffsetCorners()

In Figure 4.12 we can see a screenshot where the Navigation window is also activated, enabling us to see the generated (Baked) NavMesh. It is the light blue plane above the floor and it defines the "walkable" area in the decks, meaning all the places that an Agent can set a target and transverse to.

We will explain what exactly the function *OffsetCorners* does and how it processes the NavMeshPath path, which is returned by *CalculatePath*, with also the help of Figure 4.13.

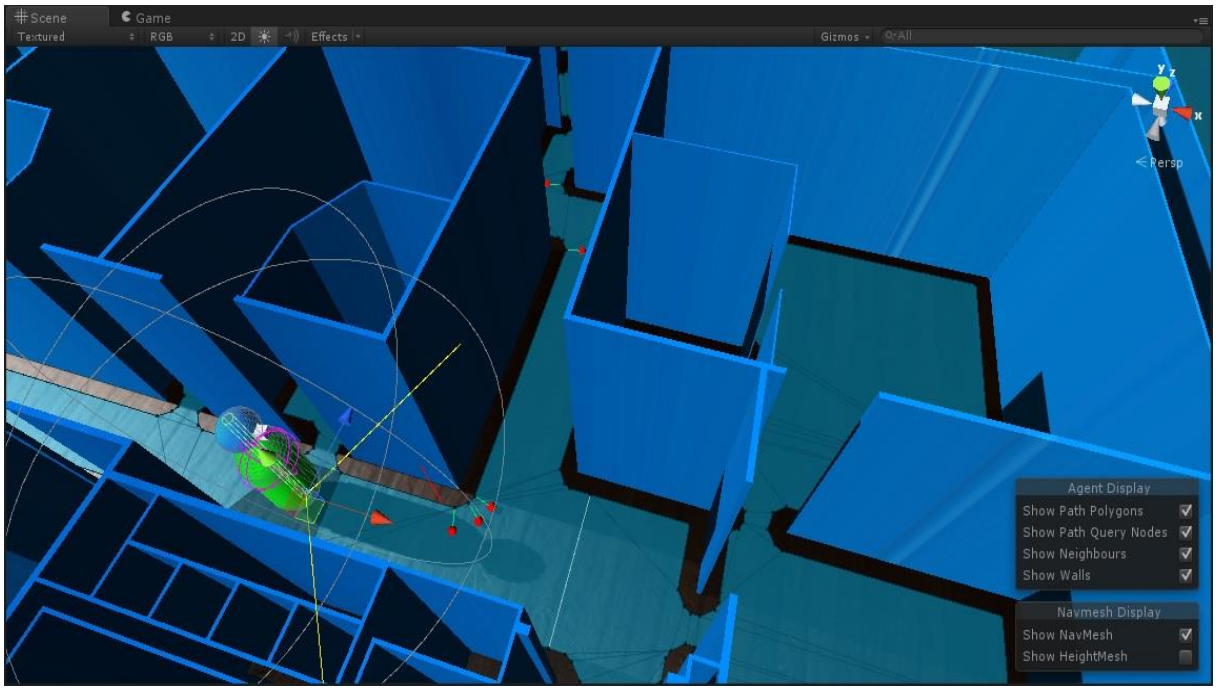

Figure 4.12: Function *OffsetCorners*

For debugging purposes, we have used a helper function, *DrawPathSpheres*, which creates small red balls at the points that comprise the final path of the Agent. Of course, all these visual cues appear only in Unity's edit mode. In the usual play mode, we can see and distinguish only the Agent walking on a deck.

The *CalculatePath* function returns a path which is comprised of several points, called corners, which are right on top of the NavMesh's edges, i.e. in Figure 4.12 at the northern ends of the green line segments where these lines intersect with the NavMesh's edges. However, using these points - corners, the movement of the Agents does not seem very natural, because they continually move across the edges of the NavMesh. Moreover, when all the Agents have the same destination, they follow exactly the same path and corners. The *OffsetCorners* function calculates for every interior point of the path (i.e. not the first and not the last) the angles formed between this point and its previous one, as well as between this and its next one. Then it calculates the difference between these two angles and finally finds the perpendicular bisector that the two segments form (Figure 4.13). Based on the perpendicular bisector a check is performed (function *NavMesh.Raycast*) to find out if there is available "walking area" of the NavMesh within a small distance from the considered point or not. If not, then we simply reverse the normalized vector **normal** which is formed by the perpendicular bisector.

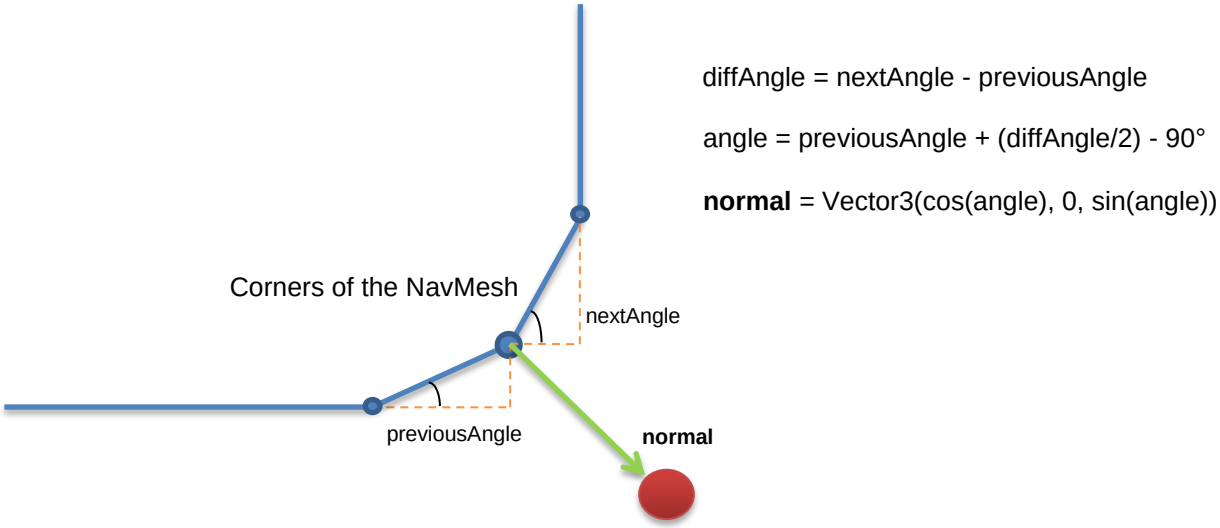

Figure 4.13: Explanation of Function *OffsetCorners*

Thus, in any case, a vector that points to the correct side of the NavMesh is calculated, where the Agents can walk on. The end of the **normal** vector is the point (of the new path) that the Agent will attempt to walk by. All these endpoints, resulting from the *OffsetCorners*, are saved in a new list, activePath, which is the actual path the Agents follow. We also create the red balls in these points for our visual convenience, as explained before.

The result is that the movement of the Agents appears to be more human-like, since they will walk near the center of the passageways and not on their edges. Also, by multiplying the normal by a random number within a defined range, we achieve that the paths, that the Agents follow, differ by some amount, and therefore their movement is not exactly the same.

After all the initializations are done and the activePath is created, the Agent's *Update* function (Unity's main loop) will continuously be "running" where the following steps will be performed each frame (for every Agent):

1. Change the value of the Finite State Machine (FSM) depending on its position.
2. Calculation of the distance to the next point of the activePath.
3. Checking if movement to the second next point of the activePath is feasible.
4. Calculation of the nearest NavMeshEdge.

Let us further analyze each of the above steps:

1. The Agent checks if it has an activePath, i.e. a path to be followed.

If not, then it sets its FSM, which uses an enumeration *AIStatus {Seek = 0, Arrive = 1, Idle = 2}* and simply records the current state/status of the Agent, in the Idle value, resulting that the Agent does not move.

If it has an activePath and is in less than slowingDistance from the last point of activePath, then the FSM gets the Arrive value and the Agent will use the Arrive behaviour (§3.2.2.2) to move.

If it has an activePath and is not in less than slowingDistance from the last point of activePath, then the FSM is assigned the Seek value and the Agent will use the Seek behaviour (§3.2.2.1) to move.

2. The Agent calculates the distance from itself to the next point of the activePath, which is always the first element in the list (List <Vector3>) associated to the activePath.

If this is not the last point, then the Agent will use the intermediateStoppingDistance variable's value and if the calculated distance is less than this value, then it is considered that the Agent has "reached" this point so it will be then removed from the activePath. Once this (the first) point of the list is removed, the list will automatically shift all the other points towards its beginning. So, the point that was previously second, will now become the first one and the Agent will start moving towards it.

If this point is the last one of the activePath, then the Agent will use the value of the stoppingDistance variable and if the calculated distance is less than this value, then it is considered that it has "reached" its final destination and will call the *OnArrived* function. In this function we can put whatever command we would like the Agent to execute upon finishing their path, for example stop recording its path.

3. The Agent "casts" an invisible ray:

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

The ray targets the second point of the activePath (the index in Lists and Arrays starts from zero). If there is not any obstacle between the Agent and the point, then the Agent can remove the first point of the activePath and begin to move directly to the second one (which will be first now after the list element shifting).

This trick helps the Agents to bypass corners with steep values, such as in narrow passageways. Figure 4.14 shows that the Agent has casted a ray to the first point of its activePath and has not found an obstacle, and therefore has started to target the second one. At the next frame, it will send a ray to the next point, but in this case there is a wall in the way, thereby a vertical red line is formed at the point where the ray collided with the wall (in the Figure 4.14 it's the vertical line close to the Agent's "nose"). Also the Agent will not change direction in this case.

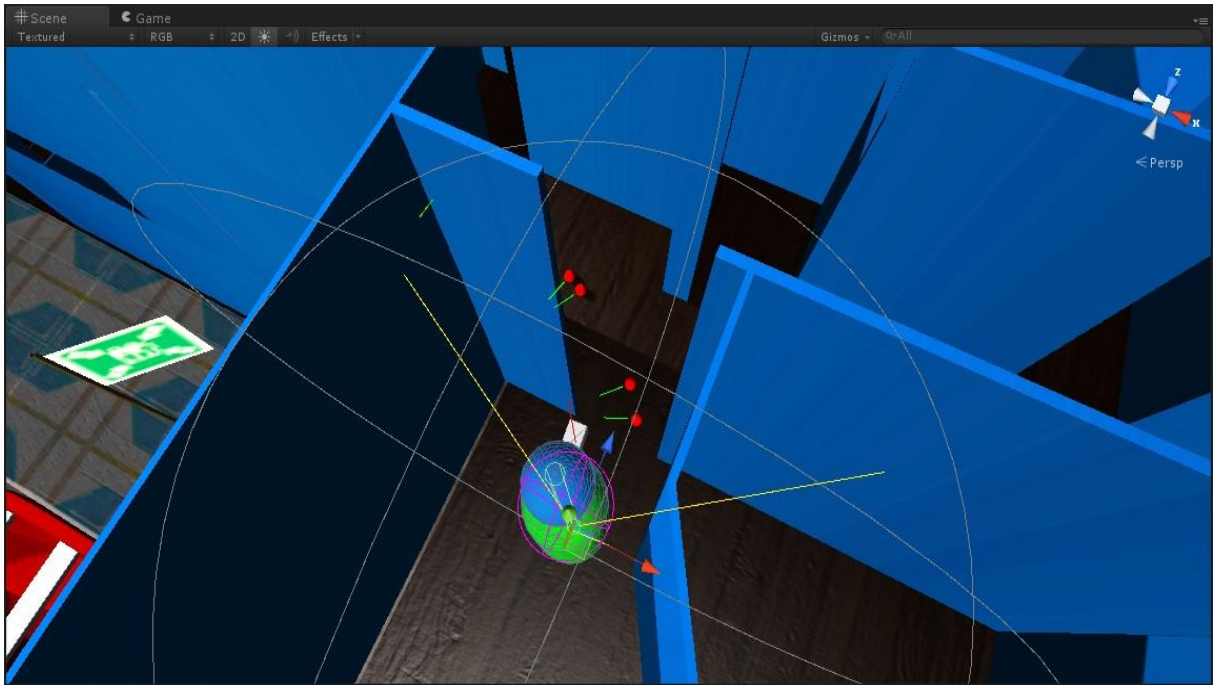

Figure 4.14: The Agent checks if it can skip a path point, shortening its path

4. The Agent calls the function:

navAgent.FindClosestEdge(out closestNavEdge)

and finds the closest (to itself) edge of the current deck's NavMesh and stores it in the out closestNavEdge parameter. The use of an out parameter was explained in the analysis of *CheckIfOnStairs* function (§4.2). Inside the closestNavEdge parameter, the information of the distance between the edge and the Agent is also contained, which we save in the nearestEdgeDistance variable:

nearestEdgeDistance = closestNavEdge.distance;

This distance is the white line shown in Figure 4.15.

The AgentManager uses the value of the nearestEdgeDistance variable to check how close every Agent is to its closest wall. If the nearestEdgeDistance becomes smaller than the PushRadius of the Agent, then the AgentManager will apply a steering vector, through the AvoidWall behaviour (§3.2.2.3), that will push the Agent away from the wall.

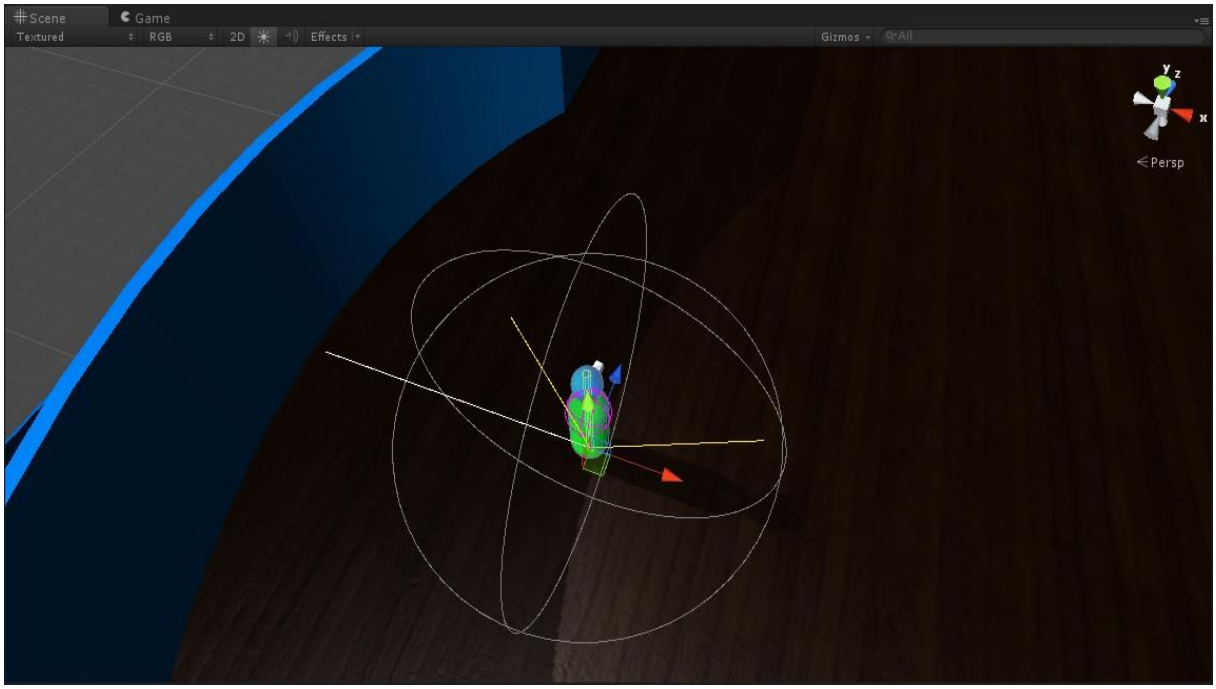

Figure 4.15: The white line is the distance between the Agent and the nearest edge of the NavMesh

Please note that the Agent has been created as an Empty Game Object to which we have added, among other Components, the Script: *Agent.cs* which performs all the functionalities that were just mentioned. The graphical representations of the body (a primitive Game Object: Capsule), the head (a primitive Game Object: Sphere) and the nose (a primitive Game Object: Cube) are implemented as "children" Game Objects of the Empty Game Object, so they can be easily replaced by more complex, detailed, textured models/meshes of people without affecting the Agent's functionality/behaviour.

The head's color depends on the age group the the Agent belongs to according to Table 3.1. If we do not use this Table with the IMO values, but instead set its age directly either with an Exact value or with a random value inside a Range, then the head will be colored gray. The body's color is green. If the injury system is used, then the Agent has a health score which decreases when a severe collision (the relative velocity of two colliding Agents exceeds a determined value) happens. At the same time, the color of the body will get darker, until its health reaches the value 0 and the color will be completely black. Then, the Agent is considered incapacitated, will not move, and act as an obstacle to the other Agents navigating through the deck's NavMesh. The Agent's body dimensions followed these guidelines: [14].

4.3 AgentManager Analysis

The AgentManager was created mainly because of the need to have a central control of a large number of Agents.

On the computer we have been working, the application performance (measured in Frames per Second - FPS) started to deteriorate when there were more than about four hundred Agents in the scene. Then, because of the the reduced FPS, the movement of the Agents seemed abnormal and "jerky/stuttered" as the frames were updated by significant time intervals. So, a way had to be found to enable the application to handle a large number of Agents (e.g. one thousand) in a satisfactory FPS.

The first attempt was made with using the coroutines of the C# language. Their implementation in Unity allows for a function to begin, pause at some point and then continue from where it stopped at a later time. We used the coroutines as follows:

- The AgentManager initially separated the Agents into groups (e.g. per hundred), and in the first frame it calculated the Steering vectors for the first group. The coroutine paused at this point.
- In the second frame, the coroutine remembered where it had stopped and continued the Steering vector calculations for the next group.
- Similarly, the Steering vectors for all the rest Agent groups were calculated in the subsequent frames, so afterwards, the process will start again with the first group.

The disadvantage of this method was that when we had a large number of Agents (e.g. one thousand – 10 groups of 100), the first hundred will be updated with their Steering vectors in the first frame, but their next update will be done after ten frames, in the 11th frame. The result is that the Agents moved for ten frames by a Steering vector that might no longer be correct. So, the Agents could be located within the space of others or inside a walled area without having acquired correctional vectors (from the corresponding behaviours) to avoid these situations.

In the end, we implemented multithreading programming [15]. It supports threads, i.e. a short sequence of programming commands that can be executed concurrently on multiple separate cores of modern processors.

To achieve multithreading programming in Unity we used the free version of Loom library. The principle of this implementation resembles the one from coroutines. We will further explain right next:

- We find the number of cores: *SystemInfo.processorCount* (e.g. four).
- The ideal would be to create a thread per core. Therefore, we divide the multitude of the Agents in the scene (e.g. a thousand) by the number of cores. The AgentManager will then create four threads where in each frame, they will "take over" 1000/4 = 250 Agents.
- The four threads will "run" at the same time and when one of them is finished with the calculations of the Steering vectors of the Agents, a counter will be increased by one.
- When this counter reaches the number of cores (= number of threads) then we know that the calculations of all Agents are finished and we can now apply the Steering vectors in the Agents' movements in the AgentManager's *Update* function.
- We reset the counter for the next frame.
- The above four steps are repeated in the AgentManager's *Update* function until the application ends.

Multithreading starts in each frame in the *Update* function when the function is called:

RunAsync()

This divides the number of Agents by the number of cores and creates as many threads as the available cores. Each generated thread "takes over" as many Agents as the result from the division: total Agents / number of groups. Thus, e.g. the first thread has the 1-250 Agents, the second has the 251 to 500, etc.

The threads creation is done using the function:

StartThread(from, to)

Where **from** is the index of the first Agent and **to** is the index of the last Agent in the group of this thread. The StartThread calls the function RunAsync of the Loom library:

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

Within the parameters of the RunAsync function we can see the lambda expression symbol =>. Therefore, the code that is located between the brackets turns into an action, which Loom's *RunAsync* requires as a parameter. The Action is a delegate that does not return a value, i.e. as a void. C#'s delegate corresponds to C++'s function pointer, i.e. a variable that stores the address of a function in memory. So, when we use the function pointer, the function, in which the pointer is pointing at, is called. Usually, they are used as parameters to functions, e.g. callbacks / listeners types.

Loom's *RunAsync* function adds the action parameter in the queue of the operating system's ThreadPool. Once a free thread is found, the ThreadPool will perform the action:

ThreadPool.QueueUserWorkItem(RunAction, a);

where **a** is our action and *RunAction* an auxiliary function.

The action, i.e. the code between the brackets, is responsible for adding all the Steering vectors, for every Agent, derived from the different behaviours (§3.2.2.1 up to and including §3.2.2.5) and for the storage of the final sum to the SteerList list which is a List <Vector3>.

Thus, the calculations to obtain the Steering vectors are done in threads, helping us to avoid a drop in performance and low FPS of our application.

We should now emphasize that because of the way the Unity is constructed, there is a major limitation:

Only Unity's main thread has access to the classes and functions of the Unity API (Application Programming Interface). This means that the threads we create, do not have access. They can only use the basic types of variables (e.g. int, float, bool, etc.) and structures. So, in our threads we cannot perform many Unity's functions, such as:

Physics.OverlapSphere()

to find neighboring Agents.

Therefore, Unity API's functions are performed in each Agent's *Update* function (§4.2). Similarly, we cannot change an Agent's position from our Threads because we do not have access to the Agent's Transform component. In this case, we used only Unity's main thread (and AgentManager's Update function) where we do:

```
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);
}
```

i.e., we alter the position and rotation/orientation of all the Agents necessarily in the main thread.

However, we took advantage of the fact that the Vector3 is a Struct, so we can use/manipulate it in our own threads. Therefore, we have implemented all the Agent behaviours in the AgentManager in such way as to have as inputs the major types and / or Vector3 and similar type outputs. Thus, the threads can call the functions that implement the different behaviours for each Agent (recall that the AgentManager maintains a list containing all the Agents in the scene, the AgentList). Within these functions the necessary calculations are made and the total Steering vector, resulting from the sum of all the Steering vectors of the behaviours, is stored in a second list of the AgentManager, the SteerList, which is also a List <Vector3>. Obviously, the element AgentList[i] refers to the same Agent as the element SteerList[i], i.e. the two lists have the same indexes for the Agents. These lists are shown in Figure 4.16 which shows the AgentManager's Inspector and its script (*AgentManager.cs*).

The AgentManager has another list, AgentCollidedList, which is private and therefore not shown in Figure 4.16, that records which of the Agents have collided too severly so they will lose health [16] and change their body's color. To change the color we must have access to the Agent's material, so we can only change it from Unity's main thread. Thus, in the behaviour PushAgent (which "runs" in our threads), when two Agents are found inside each other's PushRadius, we check their relative speed and if it exceeds a threshold, then the collision is considered severe and these two Agents are stored in the AgentCollidedList. In the *Update* of the main thread, each Agent in this list calls its function:

AgentCollidedList[j].Collided()

which is responsible for the reduction of its health and the color change of the material of the body.

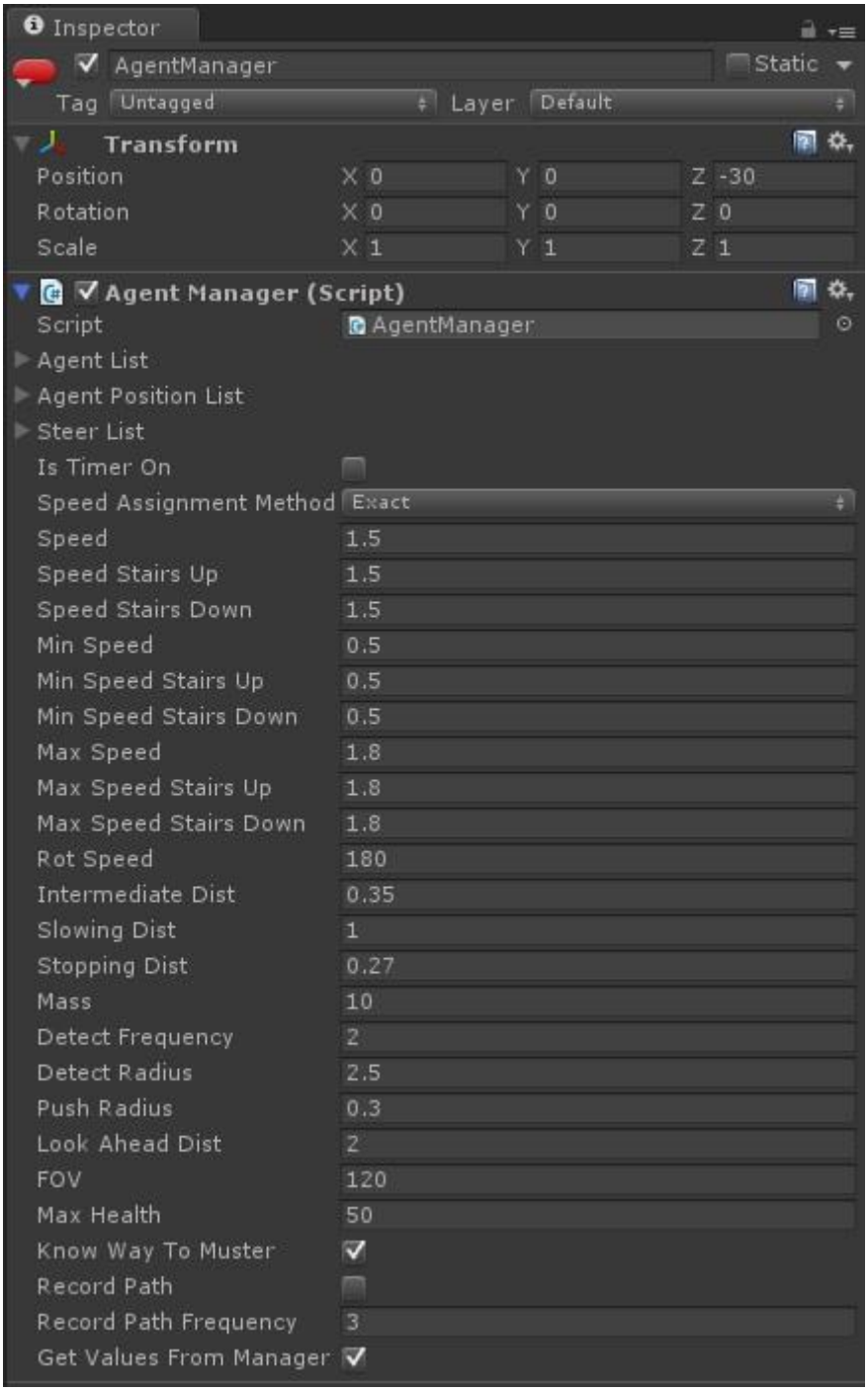

Figure 4.16: AgentManager and its main script of the same name

Figure 4.16 also shows the second usage of the AgentManager which is to assign values to the parameters of the Agents. As we said in §3.2.1, if the GetValuesFromManager variable of an Agent is true, then this Agent will not use the values set in its Inspector, but it will get its values from the AgentManager at the beginning of the application. This is an easy and quick way for us to define the parameter values of a large number of Agents without inserting them in individually.

Finally, we note that the AgentManager has been implemented as a Singleton [17]. It is a programming design pattern, which enforces the existence of just one object of a Class. Generally, Singletons are used when we need a central control point of our system's internal or/and external resources. Thus, all kinds of managers, like the AudioManager, LoggingManager, SerializationManager, AgentManager, etc, are usually implemented as Singletons.

So, our AgentManager is responsible for:

a. creating and using the threads that simultaneously handle the behaviours of all the Agents,
b. controlling the movement and rotation of all the Agents.

Also, since it operates in a global scope, it is very easy for the other Scripts to access its public variables and functions. A simple example:

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

That is, when an Agent is created, they immediately add themselves (this) in the AgentManager's AgentList. They also add an initial zero vector to the corresponding Steering vectors list, SteerList.

We implemented the version Persistent Singleton, where there is this command:

DontDestroyOnLoad();

which informs Unity that this object, the Singleton AgentManager, will not be destroyed when a new scene is loaded (as happens to all other game objects). So, when we are in the scene where we change the AgentManager's parameter values in a graphical environment (§3.2.1) and then we load the main stage of the simulation, the AgentManager won't be destroyed and retain all its values.

A disadvantage of the Singleton is that it should be created from the very first initial scene (main menu) and then it will not be destroyed when we move to the fundamental simulation scene. But, if we want to work on the simulation scene and load it in the Editor, then there will be not an AgentManager as a game object, because we haven't loaded the first scene where it is created. To solve this, we have to make a copy of the AgentManager game object in the scene that we want to work on, which will later give parameter values to the Agents. Then, if we run the application from the beginning, the standard AgentManager will erase this copy and the simulation will work correctly. To be sure, we set the *AgentManager.cs* first in the Script Execution Order (Edit => Project Settings => Script Execution Order). [This created a bug in the 4.6.4 version due to multi-threading, so we did not use the Script Execution Order in the end].

4.4 Simulation Finalization

When an Agent arrives at a Muster Station, we destroy it, in order to avoid having a very large number of Agents in a small space which would cause a reduced application performance due to the increased calculations needed because of the fact that every Agent has many neighbors. Moreover, we set the corresponding entry of the destroyed Agent as *null* in the AgentManager's AgentList where all the Agents in the scene are registered.

However, before destroying the Agent, we check if their recodPath variable is equal to true. If so, we copy its path list to the AgentManager (*AddToAgentPathsList* function), so this path will not be lost by the destruction of this the Agent.

Moreover, when an Agent collides with the Muster Station's Collider, we check if this Agent is the last active (*CheckIfEvacuationComplete*) one. As active, we consider any Agent which has not been set as null in the AgentList (i.e. has not reached yet the Muster Station) or is not Incapacitated. Therefore, the simulation will be completed when there are not any active Agents. In code, the concept is the following:

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

where *element* is the parameter of the lambda expression.

When the simulation is competed, we stop the timer, store its value in the Times file in the Reports folder and set the AgentManager's isEvacuationComplete variable equal to true. Then, the Script: *GUIDisplay.cs* will show four additional buttons, two for congestion: Show Max Congestion, Save Congestion Information and two for the Agents paths: Show Path Routes, Save Path Routes (Figure 4.17 top).

The first shows the maximum value observed (function *ShowMaxCongestion*) in every Congestion Area object. The second window stores, in the folder Reports, a file with its filename comprising of the object name and the date/time that it was created. Example: CongestionAreaDeck7_01_230520151754. In this file, the value of the congestion in this area is recorded with a period specified by the checkPeriod variable in the Script: *Congestion.cs* (Table 4.1). This recorded value shows how many Agents are located inside the Congestion Area's Collider. For the programming implementation of the congestion concept, we created the class: *TimeCongestionValue* and a list (CongestionList) of values for this class. The recording is accomplished by:

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

The button Show Path Routes shows the paths of those Agents having recordPath = true. They are shown as gray lines, [18] and [19], created by Line Renderers (AgentManager's function *ShowAllPathRoutes*). For each point i of path, we do:

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

So, the lineRenderer connects all the points of the path up to the collision with the Muster Station's Collider where the Agent was destroyed.

The last button Save Path Routes saves, in the Reports folder, the file named: Paths along with the date and time of creation, e.g. : Paths230520151754. We used the class: *TimePositionValue* that we have created in the Script: *Agent.cs*. The file contains the time stamps and the locations of every point in the pathPoint list. For every position, we do (Figure 4.17):

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

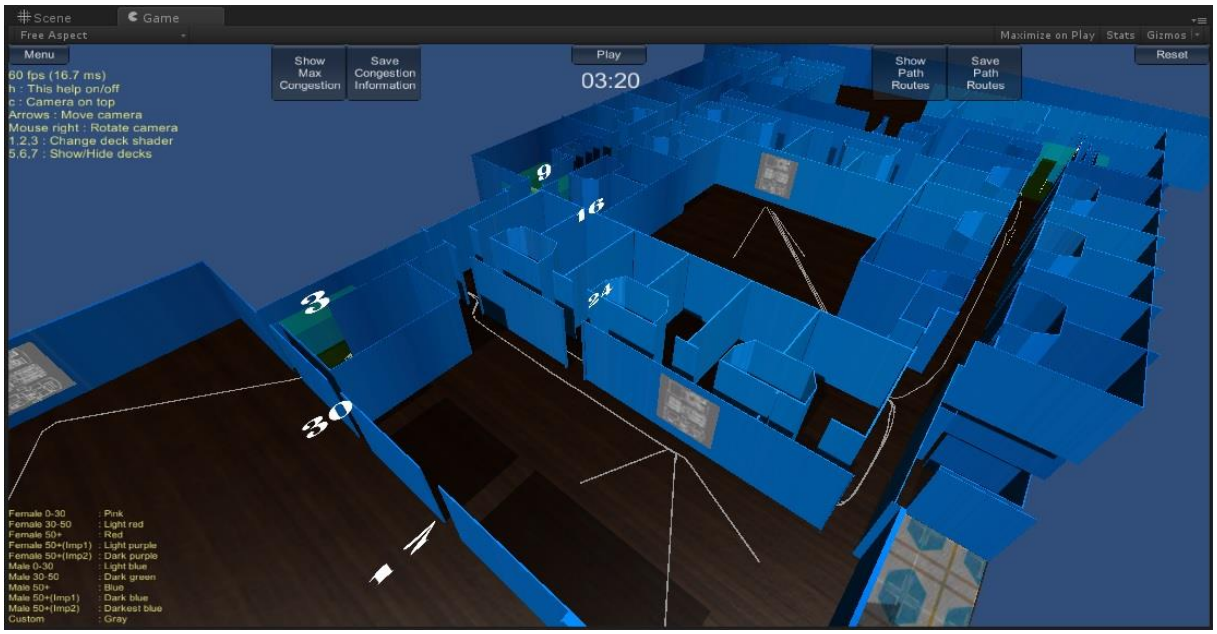

Figure 4.17: Simulation Finalization

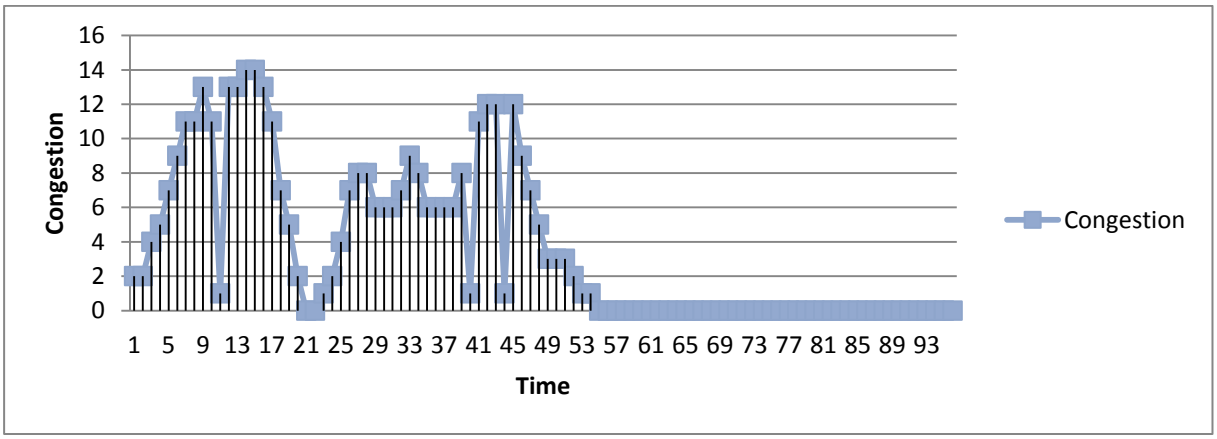

Table 4.1: Example of congestion values versus time (sec)

4.5 Test Scenes

In §3.1 we analyzed MSC's advanced evacuation analysis method according to [2]. There, MSC also suggests some tests to measure the conformity of the programmed model and software. These are:

1. Constant speed maintenance in a passageway.
2. Constant speed maintenance ascending stairs.
3. Constant speed maintenance descending stairs.
4. The flow rate in an exit should not exceed 1.33 persons/s.
5. Agents should have different personal response times.
6. The ability to pass through passageway/corridor angles without penetrating the walls.
7. Agents should have different speeds according to Table 3.2.
8. Proving that the total evacuation time is increased when there is reverse passenger flow.
9. Proving that the total evacuation time is reduced when there are more exits in the same space.
10. Agents should be able to choose different exits depending on their distances.
11. Demonstration of congestion area in a passageway that leads to a staircase.

All of the above tests have been implemented except Test5, because in our implementation, we have used the AgentManager as the entity responsible for moving all the Agents. So, when the user presses the Start button for the evacuation simulation to begin, all the Agents will start moving simultaneously. If each Agent was controlling its own movement, then it would be very easy to have and use a different response time as well.

We can check the rest of the tests at the homonymous test scenes that we've created. We access these scenes through the main menu by selecting the button: Test Scenes, which opens a second menu with all our scenes.

When a test scene included a large number of Agents in a small area, our application could not handle the simulation and the performance dropped. An example is Test4, where 100 Agents are in a room with one single exit. They were hugely compressed by the repulsive forces of the other Agents. Thus, they couldn't maintain their proper volume and the flow rate through the exit was greater than the allowable.

Also, in Test9 where there are 1000 Agents in a large area, we observed decreased program performance, i.e. a reduction in the number FPS and loss of control of the Agents' movement.

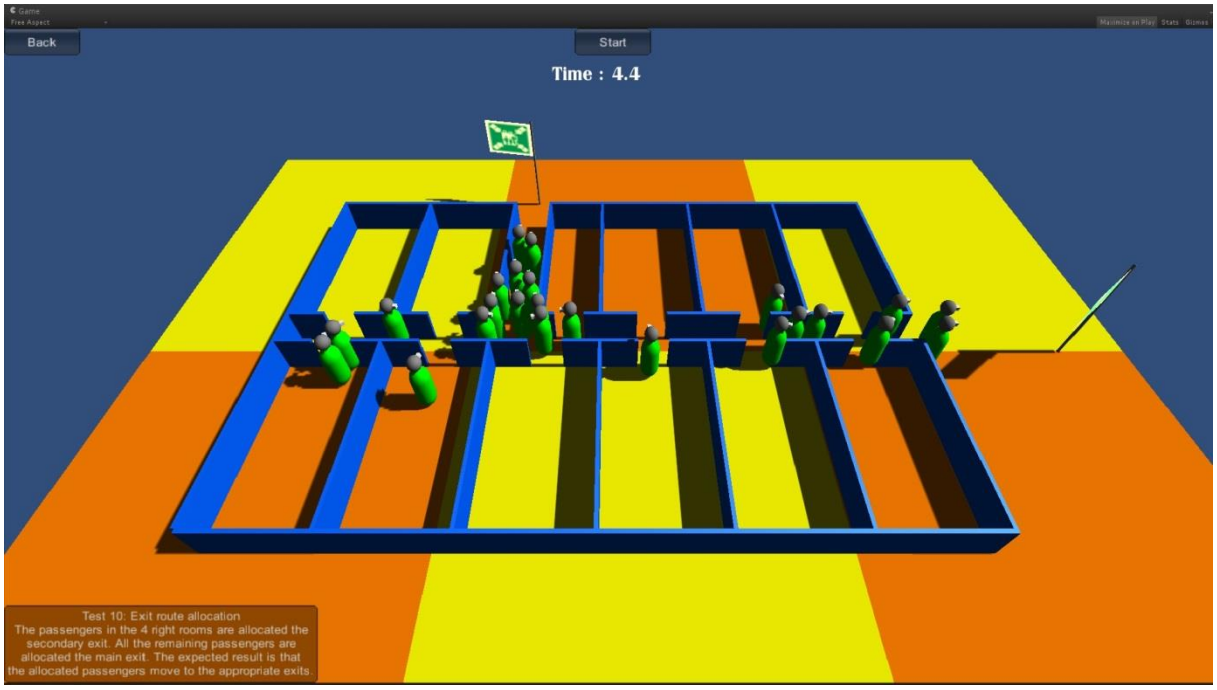

Figure 4.18: Test Scene Example

In Figure 4.18 we can see a snapshot from Test10 where the Agents of the four rightmost rooms are moving to the right exit, while the Agents which started from the other rooms are directing to the main top exit.

We should now note the user-friendly way of how we created every Test Scene's description which is placed in the bottom left corner (see Figure 4.18). We used an XML file [20] in which we had the all the scenes descriptions and the dimensions of the windows that would display them. For example for Test10:

```
<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>
```

Then, using the: XmlDocument, XmlNodeList, and XmlNode in our code, we get the data that correspond to the current scene and display them in a window with the mentioned dimensions.

Chapter 5

Conclusions

Working in Unity, we confirmed the benefits it offers as a 3-D rendering engine. In its editor there are many premade Components, such as: Collider, NavMeshAgent, TextMesh, etc., but also whole Systems, such as the lighting and shadows system or the physical system. These can be fast incorporated into the application that we want to create, saving us the required time to develop everything from the beginning ourselves. Moreover, the programming language C#, that is mainly used in Unity, along with the Mono development framework provide us many libraries which are compatible with the .NET development framework. So, the volume of code to be written is even more lessened.

In this application, however, several problems arose:

1. Unity's navigation system, which we used to provide the NavMeshAgents with a path-finding ability, has difficulties of simultaneously servicing a large number of Agents. For example in our current computer, when there are about 500 Agents in the scene, their control is very good. But if about 1000 Agents are located in the scene, then their movement starts to become defective. This phenomenon is clearly visible in many cases where Agents are concentrated in a small space (Test4 and Test9). Then, there is also the possibility that the Agents will walk out of the NavMesh's edges and be completely out of control.

   We tried to make our own path-finding system with the A* algorithm, but it proved to be too slow for so many Agents.

2. The basis of the application was not built correctly. This happened mainly because of the changes that were made during the development cycle. As an example, the Script: *Agent.cs* initially had all the controls and functions of the Agent. That is, every Agent was responsible for only themself. However, when the problem of managing many Agents appeared, we changed the application so that the Agent carries out only the necessary checks (e.g. how close they are to the next point on the path, or which is the nearest wall-edge). The movement and the different behaviours that need those checks were transferred to the AgentManager which is responsible for all the Agents. Thus, having a global supervision, we were able to use multi-threading programming.

   Even with this arrangement, improvements can be made, for example by separating the Script: *Agent.cs* into two classes. Then we would have a class with all the parameters - characteristics of the Agent and a second one with the all the checks

performed by the Agent periodically. The first one would not have to inherit anything from the MonoBehaviour class, so we could have the ability to implement null Agents, something that proved a big complication and problem.

3. As we said earlier, we used multi-threading programming to take advantage of modern processors' multiple cores. We assigned groups of Agents to threads which were served by the cores. But there is one major limitation: Unity API's functions can be called only by the main thread. For this reason, we had to let the Agents perform their checks in a non multi-threaded way. If, for example, an Agent wants to find its neighboring Agents, they do it in the main thread. We cannot use the AgentManager to call the function: *Physics.OverlapSphere()* through one of its threads. So, all the Agents run their checks together in the main thread. Suppose that each Agent finds 10 Agents within the OverlapSphere area. Then they have to do: *GetComponent<Agent>()* for every one of the 10 Agents, which is a very slow function. This shows how much the application is burdened, thus reducing its performance. Unfortunately, for this limitation we cannot do something, since it's a Unity API's limitation.

   To find neighboring Agents, a better alternative could be to use KD-trees or Quadtrees (2 dimensions) / Octrees (3 dimensions), but they should be developed entirely from the beginning instead of using Unity's Physics system.

Bibliography

[1] USA Coastguard web site: http://www.uscg.mil/imo/msc/

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

[3] Georgio M. Dounavis (2014), "Ship Accident Prevention ''on port'' and ''travelling'', Navy Library, Ch. 24.8, page: 146.

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

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

[6] 3D Studio Max web site: http://www.autodesk.com/products/3ds-max/overview

[7] Unity3D web site: https://unity3d.com/

[8] Kluge B., Prassler E. (2007), "*Reflective navigation: Individual behaviours and group behaviour*", 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] G.K. Papakonstantinou, N.A. Mpilalis, P.D. Tsanakas (1999). "*Software, Part I: Operating Principles*", Ch. 3: Simultaneous Processes.

[16] Colin M. Henein, Tony White (2004), "*Agent-Based Modelling of Forces in Crowds*", Multi-Agent and Multi-Agent-Based Simulation, pages: 173-184.

[17] Unity Singletons web site: 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 (2007), "*A Multi-agent Based Framework for the Simulation of Human and Social Behaviours during Emergency Evacuations*", AI & Society, page: 14.

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