ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Δημιουργία 3D παιχνιδιών σε C# και Unity»

Unity

Του φοιτητή:
Δημοσθένη Γκεμιτζόγλου
Αρ. Μητρώου: 514305

Επιβλέπων
Ονοματεπώνυμο: Παναγιώτης Τζέκης
Βαθμίδα: Καθηγητής

Ημερομηνία: 2/9/25

Τίτλος Δ.Ε.: Δημιουργία 3D παιχνιδιών σε C# και Unity
Κωδικός Δ.Ε.: 23283
Ονοματεπώνυμο φοιτητή: Δημοσθένης Γκεμιτζόγλου
Ονοματεπώνυμο εισηγητή: Παναγιώτης Τζέκης
Ημερομηνία ανάληψης Δ.Ε.: 22/10/23
Ημερομηνία περάτωσης Δ.Ε.: 2/9/25

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή Δημοσθένη Γκεμιτζόγλου που την εκπόνησε. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης, ο συγγραφέας/δημιουργός εκχωρεί στο Διεθνές Πανεπιστήμιο της Ελλάδος άδεια χρήσης του δικαιώματος αναπαραγωγής, δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσης της εργασίας διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος. Η ανοικτή πρόσβαση στο πλήρες κείμενο της εργασίας, δεν σημαίνει καθ' οποιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού, ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, πώληση, εμπορική χρήση, διανομή, έκδοση, μεταφόρτωση (downloading), ανάρτηση (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

Πρόλογος

Το ακαδημαϊκό έτος 2022-2023, εκπονήθηκε η πρακτική άσκηση στην εταιρεία Robotixlab (Κέντρο Εκπαιδευτικής Ρομποτικής), του κυρίου Αντώνη Κάνουρα. Σε αυτήν την εταιρεία μπορεί ένα παιδί να μάθει ρομποτική, ηλεκτρονικά κυκλώματα, προγραμματισμό, τρισδιάστατη εκτύπωση (3D printing) και ανάπτυξη παιχνιδιών (game development). Εκεί έγινε η πρώτη επαφή με την ανάπτυξη παιχνιδιών, μέσω του Microsoft MakeCode Arcade και του Scratch, όπου και τα δύο είναι ιστοσελίδες, μέσω των οποίων μπορεί να αναπτύξει κάποιος ένα παιχνίδι, με την βοήθεια εντολών-μπλοκ (command blocks), δηλαδή με εντολές που είναι σχεδιασμένες σαν ορθογώνια παραλληλόγραμμα, έχουν υποδοχές στην κορυφή τους και προεξοχές στο κάτω μέρος τους και έχουν σχεδιαστεί με τρόπο που τους επιτρέπει να συνδέονται (στοιβάζονται) με άλλες εντολές-μπλοκ. Μετά από κάποιες εβδομάδες ενασχόλησης με αυτό, ξεκίνησε η εκμάθηση περισσότερων πραγμάτων για την ανάπτυξη παιχνιδιών, ώσπου πραγματοποιήθηκε ένα σεμινάριο για ανάπτυξη παιχνιδιών μέσω της γλώσσας προγραμματισμού C# και της μηχανής παιχνιδιών Unity. Όσο περνούσε ο καιρός και το σεμινάριο εμβάθυνε περισσότερο, μαθαίνονταν όλο και πιο πολλά πράγματα για την συγκεκριμένη βιομηχανία. Σκοπός είναι ο εμπλουτισμός των γνώσεων και να ξεκινήσει μια καριέρα πάνω σε αυτό γεμάτη εκπλήξεις, συνεργασία με υπέροχους ανθρώπους και εν τέλει να γίνει η δημιουργία κάτι καινοτόμου και κάτι που θα προσφέρει διασκέδαση σε πολύ κόσμο.

Περίληψη

Η παρούσα διπλωματική εργασία (Δ.Ε.) επικεντρώνεται στην ανάπτυξη τριών τρισδιάστατων (3D) πρωτότυπων (demo) παιχνιδιών με τη χρήση της γλώσσας προγραμματισμού C# και της μηχανής παιχνιδιών Unity.

Στο πρώτο κεφάλαιο, γίνεται η αρχή, κάνοντας μία εισαγωγή στην ανάπτυξη παιχνιδιών. Στην συνέχεια, γίνεται μελέτη των πιο συνηθισμένων εργαλείων και τεχνολογιών που χρησιμοποιούνται στην ανάπτυξη παιχνιδιών και αμέσως μετά γίνεται μία εισαγωγή της Unity και της C#.

Στο δεύτερο κεφάλαιο, αναπτύσσεται το πρώτο παιχνίδι της διπλωματικής εργασίας και μέσα σε αυτό γίνεται λόγος για τον σχεδιασμό παιχνιδιού, για την διαχείριση των εισόδων του παίχτη (inputs) και για την σύγκρουσή του με άλλα αντικείμενα (game objects). Επιπλέον, πραγματοποιείται η προσθήκη πολλών και διαφορετικών ειδών εμποδίων, το καθένα λειτουργώντας με διαφορετικό τρόπο, δημιουργείται μία αντίστροφη μέτρηση μέσω της διεπαφής χρήστη (user interface (UI)) και επιδεικνύεται ο τρόπος με τον οποίο γίνεται ο σχεδιασμός πίστας.

Έπειτα, στο τρίτο κεφάλαιο, με την ανάπτυξη του δευτέρου παιχνιδιού, γίνεται η αρχή κάνοντας ξανά τον σχεδιασμό παιχνιδιού και γίνεται αναφορά στο onion design και στον τρόπο με τον οποίο δημιουργείται. Στην συνέχεια, υλοποιείται ο τρόπος κίνησης του παίχτη, η δυνατότητα σύγκρουσης του παίχτη με το περιβάλλον, γίνεται η προσθήκη ηχητικών και οπτικών εφέ και επιδεικνύεται επίσης η ανάγκη φωτισμού του χώρου. Ακόμα, με βάση τον σχεδιασμό πίστας, εντάσσονται διαφορετικών ειδών εμπόδια, όπως και στο πρώτο παιχνίδι, το καθένα έχοντας ξεχωριστή συμπεριφορά, αλλά γίνεται και η χρήση πολλαπλών και διαφορετικών σκηνών του παιχνιδιού (δημιουργία διαφορετικών επιπέδων δυσκολίας). Πέρα από αυτά, γίνεται και λόγος για την αλλαγή των ρυθμίσεων του έργου, όπως η βαρύτητα, αλλά και για τον τρόπο με τον οποίο ο παίχτης μπορεί να κλείσει το παιχνίδι, όταν δεν θέλει να παίξει άλλο.

Στο τέταρτο κεφάλαιο, με την ανάπτυξη του τρίτου και τελευταίου παιχνιδιού, γίνεται ξανά ο σχεδιασμός παιχνιδιού και χρησιμοποιείται και πάλι το onion design στην αρχή, επιδεικνύονται όλα τα βασικά στοιχεία ενός παιχνιδιού με οπτική πρώτου προσώπου (first-person view), αναφέρεται ο τρόπος δημιουργίας εδάφους (terrain), γίνεται η προσθήκη έτοιμων assets από το Asset Store της Unity για διαφορετικά είδη όπλων και εχθρικών πλασμάτων, όπως και επισημαίνεται ο τρόπος ανίχνευσης και ζημιάς των εχθρικών πλασμάτων με τα όπλα του παίχτη. Επιπροσθέτως, δημιουργείται τεχνητή νοημοσύνη για τα εχθρικά πλάσματα, γίνεται η χρήση του NavMeshAgent για την κίνησή τους, πραγματοποιείται ο τρόπος κίνησης των μοντέλων των εχθρικών πλασμάτων μέσω του animator, δημιουργείται ο τρόπος επίθεσης των εχθρικών πλασμάτων και γίνεται η προσθήκη πόντων ζωής αυτών, αλλά και του παίχτη. Αμέσως μετά, εντάσσεται η διεπαφή χρήστη (user interface (UI)) για να μπορέσει να ξαναπαίξει την ίδια πίστα ο παίχτης ή για να γίνει έξοδος από το παιχνίδι. Η εργασία επίσης εστιάζει στην δημιουργία διαφορετικών ειδών σφαιρών για κάθε όπλο, εξετάζει επίσης κατά την σχεδίαση πίστας την κατασκευή κτηρίων με τη χρήση του εργαλείου ProBuilder και εν συνεχεία γίνεται η προσθήκη οπτικού εφέ βροχής και η μετεπεξεργασία για τη βελτίωση της συνολικής εμπειρίας.

Στο πέμπτο και τελευταίο κεφάλαιο, αναφέρονται όλα τα συμπεράσματα και οι πιθανές προτάσεις βελτίωσης αυτής της διπλωματικής.

Τα παραπάνω οδηγούν στη δημιουργία ολοκληρωμένων τρισδιάστατων πρωτότυπων παιχνιδιών που αναδεικνύουν τις δυνατότητες της C# και της Unity και προσφέρουν μια πλήρη εμπειρία στον σχεδιασμό και στην ανάπτυξη παιχνιδιών.

“3D Game Development in C# with Unity”

Dimosthenis Gkemitzoglou

Abstract

This thesis focuses on the development of three three-dimensional (3D) prototype (demo) games using the C# programming language and the Unity game engine.

In the first chapter, the thesis begins with an introduction to game development. It continues with an overview of the most commonly used tools and technologies in the industry, followed by an introduction to C# and Unity.

In the second chapter, the first game of the thesis is developed. This chapter discusses game design, player input management, and collision with other game objects. It also involves the implementation of various types of obstacles—each operating differently—a countdown timer through the user interface (UI), and demonstrates how level design is implemented.

In the third chapter, with the development of the second game, the process starts again with game design and introduces the onion design pattern and how it is created. Then, the player movement system is built, including collision detection with the environment. Sound and visual effects are added, and the importance of lighting is emphasized. Based on the level design, new types of obstacles are introduced like in the first game, each with unique behaviors, and the game is expanded to include multiple scenes (levels with varying difficulty). This chapter also covers how to change project settings, such as gravity and how to allow the player to exit the game when desired.

In the fourth chapter, through the development of the third and last game, the process starts again with game design and the onion design, and all fundamental elements of a first-person game are demonstrated. The chapter explains how to create terrain, how to add ready-made assets from Unity's Asset Store for different types of weapons and enemies, and how to detect and apply damage to enemy creatures using the player's weapons. Furthermore, artificial intelligence (AI) is implemented for enemy creatures using the NavMeshAgent for movement. Their animations are controlled through the animator, their attack behavior is developed, and both enemies and the player are given health points. A user interface (UI) is also added to allow the player to replay the level or exit the game. The chapter also focuses on creating different types of ammo for each weapon, building structures using the ProBuilder tool during level design, and enhancing the experience with visual effects such as rain and post-processing.

In the fifth and last chapter, all the conclusions and possible proposals for improvement of this thesis are mentioned.

All of the above contribute to the creation of complete three-dimensional prototype games that highlight the capabilities of C# and Unity, offering a comprehensive experience in game design and development.

Ευχαριστίες

Θέλω να ευχαριστήσω, ιδιαίτερα, τον κύριο Αντώνη Κάνουρα, ιδρυτή της Robotixlab, που έκανε την αρχή, καθώς με εισήγαγε στην ανάπτυξη παιχνιδιών (game development), την οικογένειά μου, που με στηρίζει καθ' όλη αυτή την διάρκεια, όσους φίλους με βοήθησαν και με βοηθούν ακόμα, για να κυνηγήσω το όνειρό μου και τον κύριο Παναγιώτη Τζέκη για την καθοριστική βοήθειά του στην παρούσα διπλωματική εργασία.

Περιεχόμενα

Πρόλογος iii
Περίληψη iv
Abstract v
Ευχαριστίες vi
Περιεχόμενα vii
Κατάλογος Σχημάτων x
Κατάλογος Εικόνων xi
Συντομογραφίες xii
Κεφάλαιο 1ο: Ανάπτυξη παιχνιδιών (Game development) 1
1.1 Εισαγωγή 1
1.2 Εργαλεία και τεχνολογίες για την ανάπτυξη παιχνιδιών 2
1.3 Unity και C# 4
1.3.1 Εισαγωγή στην Unity 4
1.3.2 Εισαγωγή στην C# 6
1.4 Επίλογος 7
Κεφάλαιο 2ο: Dodgy 9
2.1 Εισαγωγή 9
2.2 Σχεδιασμός παιχνιδιού του Dodgy (Game design of Dodgy) 9
2.3 Χειρισμός Dodgy 9
2.4 Σύγκρουση Dodgy με το περιβάλλον 12
2.5 OnCollisionEnter() 12
2.6 Μεταβολή αρνητικού score 14
2.7 Προσθήκη εμποδίων 15
2.7.1 Dropper 16
2.7.2 Spinner 18
2.7.3 Roller 20
2.8 Αφετηρία, τερματισμός και αντίστροφη μέτρηση 22
2.9 Σχεδιασμός πίστας (Level design) 25
2.10 Debugging 26
2.11 Επίλογος 27
Κεφάλαιο 3ο: SpacEscape 29
3.1 Εισαγωγή 29

3.2 Σχεδιασμός παιχνιδιού του SpacEscape (Game design of SpacEscape) 29
3.3 Onion design 30
3.4 Κύρια στοιχεία παιχνιδιού 31
3.5 Χειρισμός Starseeker 32
3.6 Αφετηρία, τερματισμός και σύγκρουση 33
3.7 Σχεδιασμός πιστών 39
3.7.1 Σχεδιασμός πίστας της Γης (Earth's level design) 39
3.7.2 Σχεδιασμός πίστας του Ερμή (Mercury's level design) 44
3.7.3 Σχεδιασμός πίστας της Αφροδίτης (Venus' level design) 45
3.7.4 Σχεδιασμός πίστας του Άρη (Mars' level design) 50
3.8 Ρύθμιση βαρύτητας 53
3.9 Έξοδος από το παιχνίδι 55
3.10 Επίλογος 55
Κεφάλαιο 4ο: The Awakening 57
4.1 Εισαγωγή 57
4.2 Σχεδιασμός παιχνιδιού του The Awakening (Game design of The Awakening) 57
4.3 Onion design 57
4.4 Ελεγκτής πρώτου προσώπου (First-person controller) 58
4.5 Έδαφος, δέντρα και χόρτα πίστας 62
4.6 Όπλα παίχτη 62
4.6.1 Sholo 62
4.6.2 Buster 63
4.6.3 Gorlock 64
4.6.4 Plazmius 65
4.6.5 Soulpiercer 66
4.7 Physics.Raycast() 67
4.8 Εναλλαγή όπλων 69
4.9 Εχθρικά πλάσματα 70
4.9.1 Shabumi 70
4.9.2 Arach 71
4.10 Τεχνητή νοημοσύνη εχθρικών πλασμάτων 72
4.11 NavMeshAgent 73
4.12 Animations εχθρικών πλασμάτων 74
4.13 Επίθεση εχθρικών πλασμάτων 75
4.14 Πόντοι ζωής εχθρικών πλασμάτων 76

4.15 Πόντοι ζωής παίχτη 78
4.16 Οπτικά στοιχεία μάχης 79
4.17 Τέλος παιχνιδιού και προσθήκη ουρανού 82
4.18 Ζουμ όπλων 84
4.19 Είδη σφαιρών 85
4.20 Σχεδιασμός πίστας (Level design) 87
4.21 Οπτικό εφέ βροχής 90
4.22 Φακός παίχτη 91
4.23 Ανεφοδιασμός σφαιρών και μπαταριών 92
4.24 Post processing 97
4.25 Τέλος πίστας 97
4.26 Επίλογος 99
Κεφάλαιο 5ο: Συμπεράσματα και προτάσεις βελτίωσης 101
ΒΙΒΛΙΟΓΡΑΦΙΑ 103
ΠΑΡΑΡΤΗΜΑ: ΚΩΔΙΚΑΣ C# 109

Κατάλογος Σχημάτων

Σχήμα 2.1: *Λογική για το script Mover* 10
Σχήμα 2.2: *Λογική για το script ObjectHit* 13
Σχήμα 2.3: *Λογική για το script Scorer* 15
Σχήμα 2.4: *Λογική για το script Dropper* 17
Σχήμα 2.5: *Λογική για το script Spinner* 19
Σχήμα 2.6: *Λογική για το script EndGameCondition* 24
Σχήμα 2.7: *Λογική για το script ThankYouMessage* 24
Σχήμα 3.1: *Λογική για το script Movement* 33
Σχήμα 3.2: *Λογική για το script CollisionHandler* 36
Σχήμα 3.3: *Λογική για το script ThrusterCollisionDetector* 36
Σχήμα 3.4: *Λογική για το script ScreenFader* 37
Σχήμα 3.5: *Λογική για το script ThankYouMessage* 38
Σχήμα 3.6: *Λογική για το script GameState* 39
Σχήμα 3.7: *Λογική για το script ObstacleRotation* 40
Σχήμα 3.8: *Λογική για το script ObstacleOscillation* 42
Σχήμα 3.9: *Λογική για το script SinusoidalWaveMovement* 48
Σχήμα 3.10: *Λογική για το script ObstacleTeleportation* 52
Σχήμα 3.11: *Λογική για το script GravityHandler* 54
Σχήμα 3.12: *Λογική για το script QuitApplication* 55
Σχήμα 4.1: *Λογική για το script FirstPersonController* 59
Σχήμα 4.2: *Λογική για το script BasicRigidBodyPush* 61
Σχήμα 4.3: *Λογική για το script StarterAssetsInputs* 61
Σχήμα 4.4: *Λογική για το script Weapon* 69
Σχήμα 4.5: *Λογική για το script WeaponSwitcher* 70
Σχήμα 4.6: *Λογική για το script EnemyAI* 73
Σχήμα 4.7: *Λογική για το script EnemyAttack* 76
Σχήμα 4.8: *Λογική για το script EnemyHealth* 77
Σχήμα 4.9: *Λογική για το script PlayerHealth* 79
Σχήμα 4.10: *Λογική για το script DisplayDamage* 81
Σχήμα 4.11: *Λογική για το script DeathHandler* 82
Σχήμα 4.12: *Λογική για το script SceneLoader* 84
Σχήμα 4.13: *Λογική για το script WeaponZoom* 85
Σχήμα 4.14: *Λογική για το script Ammo* 86
Σχήμα 4.15: *Λογική για το script AmmoType* 87
Σχήμα 4.16: *Λογική για το script LocalRainFollow* 91
Σχήμα 4.17: *Λογική για το script FlashlightSystem* 92
Σχήμα 4.18: *Λογική για το script AmmoPickup* 93
Σχήμα 4.19: *Λογική για το script EnumExtensions* 94
Σχήμα 4.20: *Λογική για το script BatteryPickup* 96
Σχήμα 4.21: *Λογική για το script EndingMessage* 98
Σχήμα 4.22: *Λογική για το script TriggerScrollingText* 99

Κατάλογος Εικόνων

Εικόνα 1.1: *H Unity συγκρινόμενη με την Unreal Engine* [26].....5
Εικόνα 1.2: *H Godot Engine συγκρινόμενη με την Unity* [27].....6
Εικόνα 2.1: *Dodgy*.....11
Εικόνα 2.2: *Δάπεδο πίστας*.....11
Εικόνα 2.3: *Τοίχοι πίστας*.....12
Εικόνα 2.4: *Τοίχοι εντός της πίστας*.....16
Εικόνα 2.5: *Dropper*.....18
Εικόνα 2.6: *Spinner*.....20
Εικόνα 2.7: *Κεκλιμένα επίπεδα*.....21
Εικόνα 2.8: *Roller*.....21
Εικόνα 2.9: *Αφετηρία και τερματισμός*.....25
Εικόνα 2.10: *Αντίστροφη μέτρηση*.....25
Εικόνα 2.11: *Σχεδιασμός πίστας (Level design)*.....26
Εικόνα 3.1: *Μοντέλο του onion design*.....30
Εικόνα 3.2: *Έδαφος Γης και πλατφόρμες απογείωσης και προσγείωσης*.....31
Εικόνα 3.3: *Πύραυλος Starseeker*.....32
Εικόνα 3.4: *Rotating Obstacle*.....41
Εικόνα 3.5: *Back And Forth Obstacle*.....43
Εικόνα 3.6: *Πίστα της Γης*.....43
Εικόνα 3.7: *Rotating Oscillator Obstacle*.....44
Εικόνα 3.8: *Πίστα του Ερμή*.....45
Εικόνα 3.9: *Gap Obstacle*.....46
Εικόνα 3.10: *Up And Down Obstacle*.....47
Εικόνα 3.11: *Sine Obstacle*.....49
Εικόνα 3.12: *Πίστα της Αφροδίτης*.....49
Εικόνα 3.13: *Teleporting Obstacle*.....52
Εικόνα 3.14: *Πίστα του Άρη*.....53
Εικόνα 3.15: Οπτική παίχτη στην *πίστα του Άρη*.....53
Εικόνα 4.1: *Χαρακτήρας του παίχτη (Michael Shaw)*.....58
Εικόνα 4.2: *Sholo*.....63
Εικόνα 4.3: *Buster*.....64
Εικόνα 4.4: *Gorlock*.....65
Εικόνα 4.5: *Plazmius*.....66
Εικόνα 4.6: *Soulpiercer*.....67
Εικόνα 4.7: *Πληροφορίες του όπλου Buster*.....69
Εικόνα 4.8: *Shabumi*.....71
Εικόνα 4.9: *Arach*.....72
Εικόνα 4.10: *NavMeshAgent*.....74
Εικόνα 4.11: *Παράθυρο του animator*.....75
Εικόνα 4.12: *Ένδειξη ζημιάς του όπλου Gorlock*.....78
Εικόνα 4.13: *Δείκτης σκόπευσης*.....80
Εικόνα 4.14: *Ένδειξη ζημιάς παίχτη*.....81
Εικόνα 4.15: *Εμφάνιση μενού όταν χάνει ο παίχτης*.....83
Εικόνα 4.16: *Ουρανός πίστας*.....83

Εικόνα 4.17: *Είσοδος ασύλου v1* 88
Εικόνα 4.18: *Είσοδος ασύλου v2* 89
Εικόνα 4.19: *Είσοδος ασύλου v3* 90
Εικόνα 4.20: *Βαλίτσα με σφαίρες για το Soulpiercer* 95
Εικόνα 4.21: *Ένδειξη ποσότητας και είδους σφαιρών που πάρθηκαν* 95
Εικόνα 4.22: *Βαλίτσα με μπαταρίες για τον φακό του παίχτη* 97

Συντομογραφίες

Δ.Ε.	Διπλωματική Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
Π.Ε.	Πτυχιακή Εργασία
UI	User Interface
AR	Augmented Reality
VR	Virtual Reality
IDE	Integrated Development Environment
FPS	Frames Per Second
FPS	First-Person Shooter
AI	Artificial Intelligence

Κεφάλαιο 1ο: Ανάπτυξη παιχνιδιών (Game development)

1.1 Εισαγωγή

Η ανάπτυξη παιχνιδιών (game development, το οποίο συχνά συντομεύεται ως gamedev) είναι η διαδικασία δημιουργίας ενός παιχνιδιού. Πρόκειται για μια πολυεπιστημονική πρακτική που περιλαμβάνει προγραμματισμό, σχεδιασμό, τέχνη, ήχο, διεπαφή χρήστη (user interface (UI)) και συγγραφή. Κάθε μία από αυτές τις κατηγορίες μπορεί να αποτελείται από πιο εξειδικευμένες δεξιότητες. Για παράδειγμα, η τέχνη περιλαμβάνει την δημιουργία τρισδιάστατων μοντέλων αντικειμένων, μοντελοποίηση χαρακτήρων, κινήσεις αντικειμένων (animations), οπτικά εφέ και άλλα. Η ανάπτυξη υποστηρίζεται από την διαχείριση έργων, την παραγωγή και την διασφάλιση ποιότητας. Οι ομάδες μπορεί να αποτελούνται από εκατοντάδες άτομα, μια μικρή ομάδα ή ακόμα και από ένα μόνο άτομο. [1]

Η ανάπτυξη εμπορικών παιχνιδιών χρηματοδοτείται συνήθως από έναν εκδότη και μπορεί να διαρκέσει από δύο έως πέντε χρόνια για να ολοκληρωθεί. Η δημιουργία παιχνιδιών από μικρές, αυτοχρηματοδοτούμενες ομάδες ονομάζεται ανεξάρτητη ανάπτυξη (independent development). Η τεχνολογία που χρησιμοποιείται σε ένα παιχνίδι μπορεί να γραφτεί από την αρχή ή να βασιστεί σε λογισμικό ιδιοκτησίας μιας συγκεκριμένης εταιρείας. Καθώς η ανάπτυξη παιχνιδιών έχει γίνει πιο περίπλοκη, είναι πλέον σύνηθες τόσο για εταιρείες, όσο και για ανεξάρτητους προγραμματιστές να χρησιμοποιούν έτοιμες «μηχανές» όπως η Unity, η Unreal Engine ή η Godot Engine. [2][3]

Η εμπορική ανάπτυξη παιχνιδιών ξεκίνησε την δεκαετία του 1970 με την εμφάνιση των arcade παιχνιδιών, των κονσολών πρώτης γενιάς, όπως το Atari 2600 και των οικιακών υπολογιστών, όπως το Apple II. Στη δεκαετία του 1980, ένας μόνο προγραμματιστής μπορούσε να αναπτύξει ένα πλήρες και ολοκληρωμένο παιχνίδι, όπως το Pitfall!. Ωστόσο, με τις κονσόλες δεύτερης και τρίτης γενιάς στα τέλη της δεκαετίας του 1980, την αυξανόμενη δημοτικότητα των τρισδιάστατων γραφικών στους προσωπικούς υπολογιστές και τις υψηλότερες προσδοκίες για οπτικά εφέ και ποιότητα, έγινε δύσκολο για ένα μόνο άτομο να παράγει ένα παιχνίδι των τάσεων. Το μέσο κόστος παραγωγής ενός πολύ καλού παιχνιδιού (συχνά αναφερόμενου ως AAA) αυξήθηκε αργά από ένα έως τέσσερα εκατομμύρια δολάρια των ΗΠΑ το 2000, σε πάνω από διακόσια εκατομμύρια δολάρια έως το 2023. Ταυτόχρονα, η ανεξάρτητη ανάπτυξη παιχνιδιών έχει ανθίσει. Το πιο επιτυχημένο σε πωλήσεις παιχνίδι όλων των εποχών, το Minecraft, αρχικά γράφτηκε από ένα μόνο άτομο, στη συνέχεια υποστηρίχθηκε από μια μικρή ομάδα, πριν η εταιρεία εξαγοραστεί από τη Microsoft και επεκταθεί σημαντικά. [1]

Τα εμπορικά παιχνίδια των τάσεων αναπτύσσονται γενικά σε φάσεις. Αρχικά, δημιουργείται μια ιδέα, η οποία στη συνέχεια περνά στη φάση της προ-παραγωγής, όπου γράφονται πρωτότυπα και δημιουργείται το σχέδιο για ολόκληρο το παιχνίδι. Ακολουθεί η πλήρους κλίμακας ανάπτυξη ή παραγωγή και σε ορισμένες περιπτώσεις, μια φάση μετά την παραγωγή, όπου το παιχνίδι βελτιώνεται. Έχει γίνει συνηθισμένο για πολλούς προγραμματιστές, ιδιαίτερα μικρότερους, να κυκλοφορούν δημόσια τα παιχνίδια τους σε μορφή πρώιμης πρόσβασης, όπου η ανάπτυξη πραγματοποιείται παράλληλα με τη λήψη ανατροφοδότησης από πραγματικούς παίχτες. [1]

1.2 Εργαλεία και τεχνολογίες για την ανάπτυξη παιχνιδιών

Στην ανάπτυξη παιχνιδιών, χρησιμοποιούνται διάφορα εργαλεία και τεχνολογίες για τη δημιουργία, τον σχεδιασμό και τη βελτιστοποίηση των παιχνιδιών. Αυτά περιλαμβάνουν μηχανές παιχνιδιών, εργαλεία δημιουργίας αντικειμένων/μοντέλων, λογισμικό προγραμματισμού και πλατφόρμες δοκιμών.

Μηχανές παιχνιδιών (Game engines):

α. Unity: Η Unity είναι μια μηχανή ανάπτυξης σε πραγματικό χρόνο με δυνατότητες πολλαπλών πλατφορμών, η οποία χρησιμοποιείται για τη δημιουργία δισδιάστατων και τρισδιάστατων παιχνιδιών, προσομοιώσεων και διαδραστικών εμπειριών. Επιτρέπει στους προγραμματιστές να δημιουργούν μια φορά και να αναπτύσσουν το έργο τους σε περισσότερες από είκοσι πλατφόρμες, όπως κινητά, υπολογιστές, κονσόλες, επαυξημένη πραγματικότητα (augmented reality (AR)) και εικονική πραγματικότητα (virtual reality (VR)). Η Unity χρησιμοποιεί τη γλώσσα C# για scripting και παρέχει ένα ισχυρό λογισμικό επεξεργασίας, μηχανή απεικόνισης γραφικών και ένα εκτενές κατάστημα εργαλείων (Asset Store). [4] Πέρα από τη δημιουργία παιχνιδιών, η Unity χρησιμοποιείται ευρέως σε τομείς όπως η αρχιτεκτονική, ο σχεδιασμός αυτοκινήτων, η εκπαίδευση και η παραγωγή κινηματογραφικών έργων. [5]

β. Unreal Engine: Η Unreal Engine, που αναπτύχθηκε από την Epic Games, είναι ένα ολοκληρωμένο εργαλείο δημιουργίας τρισδιάστατων εφαρμογών σε πραγματικό χρόνο, το οποίο χρησιμοποιείται για την ανάπτυξη υψηλής ποιότητας παιχνιδιών και διαδραστικών εμπειριών σε πλατφόρμες όπως υπολογιστές, κονσόλες, κινητές συσκευές, εικονική πραγματικότητα (virtual reality (VR)) και επαυξημένη πραγματικότητα (augmented reality (AR)). Προσφέρει προηγμένα χαρακτηριστικά όπως φωτορεαλιστική απεικόνιση γραφικών, δυναμικό φωτισμό και ένα ισχυρό σύστημα φυσικής. Η μηχανή υποστηρίζει προγραμματισμό σε C++, καθώς και ένα σύστημα οπτικού προγραμματισμού που ονομάζεται «Blueprints», επιτρέποντας στους προγραμματιστές να δημιουργούν πολύπλοκους μηχανισμούς παιχνιδιού χωρίς εκτεταμένη κωδικοποίηση. Πέρα από τα παιχνίδια, η Unreal Engine χρησιμοποιείται σε βιομηχανίες όπως ο κινηματογράφος, η αρχιτεκτονική και ο σχεδιασμός αυτοκινήτων για προσομοιώσεις και οπτικοποιήσεις. [6]

γ. Godot Engine: Η Godot Engine είναι μια δωρεάν και ανοιχτού κώδικα πλατφόρμα ανάπτυξης παιχνιδιών που υποστηρίζει την δημιουργία τόσο δισδιάστατων, όσο και τρισδιάστατων παιχνιδιών. Προσφέρει ένα ολοκληρωμένο σύνολο εργαλείων, συμπεριλαμβανομένου ενός αποκλειστικού δισδιάστατου μηχανισμού, ενός ισχυρού τρισδιάστατου μηχανισμού, ενός συστήματος κινούμενης εικόνας και μιας ενσωματωμένης γλώσσας προγραμματισμού που ονομάζεται «GDScript», η οποία είναι συντακτικά παρόμοια με την Python. Το σύστημα σκηνών της Godot Engine επιτρέπει στους προγραμματιστές να δημιουργούν στοιχεία παιχνιδιού χρησιμοποιώντας ένα δέντρο κόμβων, διευκολύνοντας την οργανωμένη και επαναχρησιμοποιήσιμη ανάπτυξη. Η μηχανή υποστηρίζει ανάπτυξη σε πολλαπλές πλατφόρμες, συμπεριλαμβανομένων των Windows, macOS, Linux, Android, iOS και φυλλομετρητών ιστού μέσω HTML5. Πέρα από τα παιχνίδια, η Godot Engine χρησιμοποιείται για προσομοιώσεις και διαδραστικές εφαρμογές σε διάφορες βιομηχανίες. [7]

Εργαλεία δημιουργίας γραφικών:

α. Blender: Το Blender είναι μια δωρεάν και ανοιχτού κώδικα σουίτα δημιουργίας τρισδιάστατων γραφικών που υποστηρίζει ολόκληρη τη διαδικασία παραγωγής 3D, συμπεριλαμβανομένων της μοντελοποίησης, του rigging, των animations, της προσομοίωσης, της απεικόνισης γραφικών, της σύνθεσης και της παρακολούθησης κίνησης. Προσφέρει επίσης δυνατότητες επεξεργασίας βίντεο και δημιουργίας παιχνιδιών. Το Blender είναι πολλαπλών πλατφορμών και συντηρείται από μια παγκόσμια κοινότητα προγραμματιστών. [8]

β. Zbrush: Το ZBrush, αναπτυγμένο από την Pixologic (τώρα μέρος της Maxon), είναι ένα επαγγελματικό λογισμικό ψηφιακής γλυπτικής και ζωγραφικής που συνδυάζει μοντελοποίηση

3D/2.5D, υφή και ζωγραφική. Χρησιμοποιεί την ιδιόκτητη τεχνολογία «pixol», αποθηκεύοντας πληροφορίες για το βάθος, τον προσανατολισμό και το υλικό, επιτρέποντας στους καλλιτέχνες να δημιουργούν εξαιρετικά λεπτομερή μοντέλα εκατομμυρίων πολυγώνων. Το ZBrush προσφέρει μια μεγάλη ποικιλία από προσαρμόσιμα πινέλα και εργαλεία, επιτρέποντας διαισθητικές και αποδοτικές ροές εργασίας για την δημιουργία περίπλοκων λεπτομερειών και υφών. Χρησιμοποιείται ευρέως σε βιομηχανίες όπως ο κινηματογράφος, τα παιχνίδια και η εικονογράφηση, καθιστώντας το ένα πρότυπο εργαλείο για τη δημιουργία μοντέλων υψηλής ανάλυσης και σύνθετων σχεδίων. [9]

γ. Autodesk 3ds Max: Το Autodesk 3ds Max είναι ένα επαγγελματικό λογισμικό τρισδιάστατης μοντελοποίησης, απεικόνισης γραφικών και animations που επιτρέπει στους χρήστες να δημιουργούν εκτεταμένους κόσμους και υψηλής ποιότητας σχέδια. Προσφέρει ισχυρά εργαλεία μοντελοποίησης, μια εκτενή αρχιτεκτονική προσθέτων και υποστηρίζει λειτουργίες όπως δυναμική προσομοίωση και συστήματα σωματιδίων. Το 3ds Max χρησιμοποιείται ευρέως στην ανάπτυξη παιχνιδιών, στην παραγωγή ταινιών και στην αρχιτεκτονική απεικόνιση. [10]

δ. Autodesk Maya: Το Autodesk Maya είναι ένα επαγγελματικό λογισμικό 3D που χρησιμοποιείται για τη δημιουργία ρεαλιστικών χαρακτήρων και εντυπωσιακών εφέ. Παρέχει ελκυστικά εργαλεία για animations, διαισθητικές δυνατότητες μοντελοποίησης και υποστηρίζει την δημιουργία πολύπλοκων προσομοιώσεων και οπτικών εφέ. Το Maya χρησιμοποιείται εκτενώς στην παραγωγή ταινιών και παιχνιδιών. [11]

ε. Mixamo: Το Mixamo είναι μια εταιρεία τεχνολογίας τρισδιάστατων γραφικών που προσφέρει διαδικτυακές υπηρεσίες για animations τρισδιάστατων χαρακτήρων. Παρέχει μια εκτενή συλλογή έτοιμων προς χρήση τρισδιάστατων χαρακτήρων και animations, μαζί με μια υπηρεσία αυτόματου rigging που επιτρέπει στους χρήστες να ανεβάζουν προσαρμοσμένα μοντέλα και να τα λαμβάνουν πίσω με πλήρη σκελετικά rigs. Οι υπηρεσίες του Mixamo χρησιμοποιούνται ευρέως στην ανάπτυξη παιχνιδιών, στον κινηματογράφο και σε άλλα διαδραστικά μέσα. [12]

Λογισμικά προγραμματισμού:

α. Visual Studio: Το Visual Studio είναι ένα ολοκληρωμένο περιβάλλον ανάπτυξης (integrated development environment (IDE)) που αναπτύχθηκε από τη Microsoft. Υποστηρίζει την ανάπτυξη εφαρμογών σε διάφορες πλατφόρμες, συμπεριλαμβανομένων των Windows, macOS, Android, iOS και web. Το Visual Studio προσφέρει λειτουργίες όπως επεξεργασία κώδικα με IntelliSense, αποσφαλμάτωση (debugging), προφίλ και μια σειρά εργαλείων για συνεργασία ομάδων. Υποστηρίζει πολλές γλώσσες προγραμματισμού, όπως C#, C++, Python και JavaScript, επιτρέποντας στους προγραμματιστές να δημιουργούν διάφορες εφαρμογές, από εφαρμογές επιτραπέζιων και κινητών συσκευών έως υπηρεσίες web και cloud. [13]

β. Visual Studio Code: Το Visual Studio Code είναι ένας δωρεάν, ανοιχτού κώδικα επεξεργαστής κώδικα που αναπτύχθηκε από την Microsoft. Είναι πολλαπλών πλατφορμών, λειτουργώντας σε Windows, macOS και Linux. Το Visual Studio Code παρέχει λειτουργίες όπως επισήμανση σύνταξης, συμπλήρωση κώδικα με IntelliSense, υποστήριξη αποσφαλμάτωσης (debugging) και ενσωματωμένο έλεγχο Git. Η επεκτάσιμη αρχιτεκτονική του επιτρέπει στους προγραμματιστές να εγκαθιστούν επεκτάσεις για πρόσθετη λειτουργικότητα, συμπεριλαμβανομένης της υποστήριξης για διάφορες γλώσσες προγραμματισμού και πλαίσια εργασίας. Το Visual Studio Code χρησιμοποιείται ευρέως για την ανάπτυξη εφαρμογών web και cloud λόγω της ελαφριάς φύσης του και του ισχυρού συνόλου λειτουργιών του. [14]

Πλατφόρμες δοκιμών και βελτιστοποίησης:

α. Unity Profiler: Το Unity Profiler είναι ένα εργαλείο ανάλυσης απόδοσης ενσωματωμένο στη μηχανή Unity, που επιτρέπει στους προγραμματιστές να παρακολουθούν και να βελτιστοποιούν τις εφαρμογές τους. Παρέχει λεπτομερείς πληροφορίες για διάφορα μετρικά απόδοσης, όπως η χρήση CPU και GPU, η κατανάλωση μνήμης και τα στατιστικά απόδοσης. Οι προγραμματιστές μπορούν να συνδέσουν το Unity Profiler σε συσκευές στο δίκτυό τους ή απευθείας στον υπολογιστή τους για να δοκιμάσουν την απόδοση της εφαρμογής στις επιθυμητές πλατφόρμες.

Επιπλέον, το Unity Profiler επιτρέπει την προσθήκη προσαρμοσμένων δεικτών ανάλυσης για τη συλλογή συγκεκριμένων δεδομένων απόδοσης, διευκολύνοντας τον εντοπισμό και την επίλυση προβλημάτων. [15]

β. Unreal Insights: Το Unreal Insights είναι ένα σύστημα προφίλ ενσωματωμένο στην Unreal Engine, σχεδιασμένο για τη συλλογή, ανάλυση και οπτικοποίηση δεδομένων που εκπέμπονται από την μηχανή. Βοηθά τους προγραμματιστές να εντοπίζουν σημεία συμφόρησης απόδοσης, παρέχοντας λεπτομερείς πληροφορίες για διάφορες δραστηριότητες της μηχανής, όπως η χρήση CPU και GPU, οι χρόνοι φόρτωσης των assets και η απόδοση του δικτύου. Το Unreal Insights μπορεί να καταγράφει δεδομένα τόσο τοπικά όσο και απομακρυσμένα, επιτρέποντας ευέλικτη ανάλυση απόδοσης κατά τη διάρκεια διαφορετικών σταδίων ανάπτυξης. [16]

γ. Godot Profiler: Το Godot Profiler είναι ένα εργαλείο ανάλυσης απόδοσης ενσωματωμένο στον πίνακα αποσφαλμάτωσης (debugging) της Godot Engine. Επιτρέπει στους προγραμματιστές να παρακολουθούν μετρικές απόδοσης σε πραγματικό χρόνο, όπως ο χρόνος καρέ, ο χρόνος καρέ φυσικής και οι χρόνοι εκτέλεσης των συναρτήσεων των scripts. Ξεκινώντας το Godot Profiler κατά τη διάρκεια μιας συνεδρίας παιχνιδιού, οι προγραμματιστές μπορούν να συλλέγουν δεδομένα που βοηθούν στον εντοπισμό προβλημάτων απόδοσης και στην βελτιστοποίηση του κώδικα ανάλογα. Το Godot Profiler παρουσιάζει αυτά τα δεδομένα σε γραφική μορφή, καθιστώντας ευκολότερη την ερμηνεία και την αντιμετώπιση πιθανών σημείων συμφόρησης. [17]

Η επιλογή των κατάλληλων εργαλείων και τεχνολογιών εξαρτάται από τις ανάγκες του εκάστοτε έργου, τον προϋπολογισμό και την εμπειρία της ομάδας ανάπτυξης.

1.3 Unity και C#

1.3.1 Εισαγωγή στην Unity

Η Unity είναι μια πολυπλατφορμική μηχανή παιχνιδιών που δημιουργήθηκε από την Unity Technologies. Παρουσιάστηκε για πρώτη φορά τον Ιούνιο του 2005 στο Apple Worldwide Developers Conference, ως εργαλείο ανάπτυξης ειδικά για το Mac OS X. Έκτοτε, η μηχανή έχει επεκταθεί για να υποστηρίζει ένα ευρύ φάσμα πλατφορμών, όπως υπολογιστές, κινητές συσκευές, κονσόλες παιχνιδιών, καθώς και εφαρμογές επαυξημένης και εικονικής πραγματικότητας. Είναι ιδιαίτερα δημοφιλής στην ανάπτυξη παιχνιδιών για κινητά iOS και Android. Χάρη στην εύκολη χρήση της, προτιμάται συχνά από αρχάριους προγραμματιστές και αποτελεί μια πολύ δημοφιλή επιλογή για ανεξάρτητους δημιουργούς παιχνιδιών. [18] Η Unity επιτρέπει την δημιουργία τόσο τρισδιάστατων, όσο και δισδιάστατων παιχνιδιών, καθώς και διαδραστικών προσομοιώσεων. [19][20] Πέρα από την βιομηχανία των παιχνιδιών, η Unity χρησιμοποιείται επίσης σε τομείς όπως η κινηματογραφική παραγωγή, ο σχεδιασμός οχημάτων, η αρχιτεκτονική, η μηχανική, οι κατασκευές και ακόμη και από τις Ένοπλες Δυνάμεις των Ηνωμένων Πολιτειών. [21]

Χαρακτηριστικά και δυνατότητες:

α. Η Unity υποστηρίζει την γραφή κώδικα (scripting) στην γλώσσα προγραμματισμού C#. Η C# είναι μια διαχειριζόμενη, αντικειμενοστραφής γλώσσα προγραμματισμού, η οποία αποτελεί μέρος της πλατφόρμας .NET και εκτελείται στο πολυπλατφορμικό περιβάλλον εκτέλεσης .NET (runtime). [22] Επιπλέον, το περιβάλλον scripting της Unity είναι δομημένο γύρω από τη C#, και η μηχανή μεταγλωττίζει (compiles) τον κώδικα C# για ανάπτυξη σε διάφορες πλατφόρμες. [23]

β. Ενσωματώνει ένα φυσικό σύστημα φυσικής (NVIDIA PhysX) [24], το οποίο επιτρέπει την υλοποίηση ρεαλιστικών αλληλεπιδράσεων μεταξύ αντικειμένων.
γ. Η μηχανή προσφέρει το Visual Scripting (Bolt) [25], ένα σύστημα οπτικού προγραμματισμού, που επιτρέπει σε μη προγραμματιστές να δημιουργούν λογική παιχνιδιού χωρίς να γράφουν κώδικα. Παράλληλα, το Unity Profiler [15] βοηθά τους προγραμματιστές να αναλύσουν την απόδοση του παιχνιδιού τους, εντοπίζοντας σημεία που χρειάζονται βελτιστοποίηση.
δ. Ένα από τα μεγαλύτερα πλεονεκτήματα της μηχανής είναι το Unity Asset Store [4], μια αγορά γεμάτη με έτοιμα assets, scripts και plugins, που επιτρέπει στους προγραμματιστές να επιταχύνουν την διαδικασία ανάπτυξης και να προσθέσουν νέες λειτουργίες στα έργα τους.

Σύγκριση με άλλες μηχανές:

Αυτός ο πίνακας συγκρίνει την Unity και την Unreal Engine και δίνει μια σαφέστερη εικόνα για το που υπερτερεί κάθε μηχανή, βοηθώντας κάποιον να επιλέξει αυτήν που ταιριάζει καλύτερα στο παιχνίδι του.

Χαρακτηριστικό	**Unity**	**Unreal Engine**
Ευκολία χρήσης	Εύκολη να την μάθει κανείς, ιδανική για αρχάριους	Πανίσχυρη, αλλά πιο δύσκολη να την τελειοποιήσει κανείς
Ποιότητα γραφικών	Καλή, αλλά όχι κορυφαία	Τα κορυφαία γραφικά της βιομηχανίας
Απόδοση	Χρειάζεται λίγη τροποποίηση	Βελτιστοποιημένη για έργα βαρέων καθηκόντων
Συγγραφή κώδικα	Συγγραφή κώδικα σε C#, ευέλικτη	Οπτική συγγραφή κώδικα με Blueprint
Κατάστημα εργαλείων	Τεράστια ποικιλία εργαλείων	Ισχυρό, αλλά μικρότερες επιλογές
Υποστήριξη πλατφόρμας	Πάνω από 25 πλατφόρμες	Ιδανική για υψηλής ποιότητας πλατφόρμες
Κοινότητα και υποστήριξη	Μεγάλη, χρήσιμη κοινότητα	Μεγάλη και ενεργή υποστήριξη
Κόστος	Δωρεάν επιλογή, πληρωμή για πρόσθετες λειτουργίες	Δωρεάν για να ξεκινήσει κανείς, με δικαιώματα αργότερα

Εικόνα 1.1: *Η Unity συγκρινόμενη με την Unreal Engine* [26]

Αυτός ο πίνακας συγκρίνει την Godot Engine με την Unity και δίνει μια σαφέστερη εικόνα για το που υπερτερεί κάθε μηχανή, βοηθώντας κάποιον να επιλέξει αυτήν που ταιριάζει καλύτερα στο παιχνίδι του.

Χαρακτηριστικό	Godot Engine	Unity
Γλώσσες για την συγγραφή κώδικα	Χρησιμοποιεί GDScript (παρόμοια με την Python)	Χρησιμοποιεί C#
Γραφικά και σκιάσεις	Δυνατά, αλλά λιγότερο προηγμένα όσον αφορά την σκίαση της Godot, σε σχέση με αυτή της Unity	Ανώτερα γραφικά με προηγμένες τεχνικές απόδοσης
Κατάστημα εργαλείων	Μικρότερο, κινούμενο από την κοινότητα	Εκτεταμένο και καλά εδραιωμένο
Κοινότητα και υποστήριξη	Μικρότερη, αναπτυσσόμενη κοινότητα	Μεγάλη, ενεργή κοινότητα με εκτεταμένους πόρους
Απόδοση	Ελαφριά, αποτελεσματική για 2D και μικρότερα 3D έργα	Άριστη σε παιχνίδια 3D υψηλής ποιότητας και σε εφαρμογές VR/AR
Υιοθέτηση στην βιομηχανία	Χρησιμοποιείται σπανιότερα, αλλά αυξάνεται η υιοθέτησή της	Αναγνωρίζεται και χρησιμοποιείται ευρέως
Ευκολία εκμάθησης	Πιο απλή διεπαφή και η GDScript είναι πιο εύκολη να την μάθει κανείς	Πιο ισχυρή μηχανή, αλλά έχει πιο απότομη καμπύλη εκμάθησης

Εικόνα 1.2: *Η Godot Engine συγκρινόμενη με την Unity* [27]

Χρήση της Unity πέρα από τα παιχνίδια:

Από το 2010 και μετά, η Unity Technologies χρησιμοποίησε την μηχανή παιχνιδιών της για να μεταβεί σε άλλες βιομηχανίες μέσω της πλατφόρμας real-time 3D, συμπεριλαμβανομένων του κινηματογράφου και της αυτοκινητοβιομηχανίας. [28][29] Η Unity πειραματίστηκε για πρώτη φορά με τον κινηματογράφο μέσω του Adam, μιας μικρού μήκους ταινίας για ένα ρομπότ που δραπετεύει από τη φυλακή. Αργότερα, η Unity συνεργάστηκε με τον σκηνοθέτη Neill Blomkamp, του οποίου το Oats Studios χρησιμοποίησε τα εργαλεία της μηχανής, συμπεριλαμβανομένου του real-time rendering και του Cinemachine, για να δημιουργήσει δύο ταινίες μικρού μήκους με υπολογιστή, τις Adam: The Mirror και Adam: The Prophet. [28] Στο συνέδριο Unite Europe 2017 στο Άμστερνταμ, η Unity επικεντρώθηκε στη δημιουργία ταινιών με το Cinemachine, εργαλείο της έκδοσης Unity 2017.1. [30] Το 2018, η Disney Television Animation κυκλοφόρησε τρία shorts με τίτλο Baymax Dreams, τα οποία δημιουργήθηκαν με τη μηχανή Unity. [31] Η μηχανή Unity χρησιμοποιήθηκε επίσης από τη Disney για τη δημιουργία των φόντων στην ταινία The Lion King του 2019. [32] Οι αυτοκινητοβιομηχανίες χρησιμοποιούν την τεχνολογία της Unity για να δημιουργούν μοντέλα πλήρους κλίμακας νέων οχημάτων σε εικονική πραγματικότητα, να χτίζουν εικονικές γραμμές συναρμολόγησης και να εκπαιδεύουν εργαζομένους. [29] Η Unity αναπτύσσει επίσης λύσεις στους τομείς της αρχιτεκτονικής, της μηχανικής και των κατασκευών. [33] Επιπλέον, η μηχανή της Unity χρησιμοποιείται από την DeepMind, μια εταιρεία της Alphabet, για την εκπαίδευση τεχνητής νοημοσύνης. [34] Άλλες χρήσεις που διερευνώνται από την Unity Technologies περιλαμβάνουν την αρχιτεκτονική, τη μηχανική και τις κατασκευές. [33]

1.3.2 Εισαγωγή στην C#

Η C# είναι μια γλώσσα προγραμματισμού υψηλού επιπέδου γενικής χρήσης, η οποία υποστηρίζει πολλαπλά παραδείγματα προγραμματισμού. Περιλαμβάνει χαρακτηριστικά όπως στατική τυποποίηση [35]:4, ισχυρή τυποποίηση, λεξιλογική εμβέλεια, προστακτικό και δηλωτικό ύφος, καθώς και λειτουργικό, γενικευμένο [35]:22, αντικειμενοστραφές (με χρήση κλάσεων) και προσανατολισμένο σε συνιστώσες προγραμματισμό. [36]

Η ανάπτυξή της έγινε κυρίως από τους Anders Hejlsberg, Scott Wiltamuth και Peter Golde, εργαζόμενους στη Microsoft. [36] Η πρώτη ευρεία διάθεσή της πραγματοποιήθηκε τον Ιούλιο του 2000 [36], και αργότερα εγκρίθηκε ως διεθνές πρότυπο από την Ecma (ECMA-334) το 2002 και από τον ISO/IEC (ISO/IEC 23270 και 20619) το 2003. Η Microsoft παρουσίασε την C# μαζί με το .NET Framework και το Microsoft Visual Studio, προϊόντα που εκείνη την εποχή ήταν κλειστού κώδικα. Τότε, η Microsoft δεν είχε ακόμη παρουσιάσει λογισμικό ανοιχτού κώδικα. Τέσσερα χρόνια αργότερα, το 2004, ξεκίνησε το δωρεάν και ανοιχτού κώδικα έργο Mono, το οποίο πρόσφερε compiler και runtime περιβάλλον για την C# σε πολλαπλές πλατφόρμες. Μια δεκαετία αργότερα, η Microsoft κυκλοφόρησε το Visual Studio Code (συντάκτης κώδικα), τον compiler Roslyn και την ενιαία πλατφόρμα .NET, τα οποία είναι όλα δωρεάν, ανοιχτού κώδικα και συμβατά με την C#. Το Mono τελικά ενσωματώθηκε στη Microsoft, χωρίς όμως να συγχωνευθεί με το .NET. [37]

Από τον Ιανουάριο του 2025, η πιο πρόσφατη σταθερή έκδοση της γλώσσας προγραμματισμού C# είναι η C# 13.0, η οποία κυκλοφόρησε επίσημα το 2024, μαζί με την πλατφόρμα .NET 9.0. [38][39]

1.4 Επίλογος

Στο πρώτο κεφάλαιο έγινε εισαγωγή με τα βασικότερα πράγματα, που πρέπει να γνωρίζει κάποιος, για την ανάπτυξη παιχνιδιών. Αμέσως μετά, αναφέρθηκαν τα εργαλεία και οι τεχνολογίες που χρησιμοποιούνται για την ανάπτυξη παιχνιδιών. Ύστερα, η διπλωματική εργασία εστίασε στην μηχανή παιχνιδιών Unity, που είναι ένα πασίγνωστό εργαλείο στον χώρο της ανάπτυξης παιχνιδιών, επισήμανε βασικά πράγματα για αυτήν και την συνέκρινε με άλλες μηχανές παιχνιδιών. Στο τέλος, το πρώτο κεφάλαιο έκλεισε, κάνοντας συζήτηση για την C# και τον ρόλο της.

Κεφάλαιο 2ο: Dodgy

2.1 Εισαγωγή

Το παιχνίδι «Dodgy», είναι το πρώτο παιχνίδι αυτής της διπλωματικής εργασίας και είναι ένα παιχνίδι με οπτική τρίτου προσώπου (third-person view), όπου ο παίχτης, χειριζόμενος τον Dodgy, προσπαθεί να αποφύγει διάφορα εμπόδια που εμφανίζονται μπροστά του, καθ' όλη την διάρκεια της πίστας.

2.2 Σχεδιασμός παιχνιδιού του Dodgy (Game design of Dodgy)

Στο συγκεκριμένο παιχνίδι, υπάρχει ένας παίχτης, στον οποίο δεν γίνεται παρέμβαση στην εμφάνισή του και υπάρχει μία κάμερα που τον ακολουθεί καθ' όλη την διάρκεια του παιχνιδιού. Όλα τα game objects αποτελούνται από απλά αντικείμενα που βρίσκει κάποιος στα ήδη έτοιμα game objects της Unity. Οι κανόνες του παιχνιδιού είναι απλοί. Ο παίχτης πρέπει να αποφύγει όσα περισσότερα εμπόδια μπορεί, ξεκινώντας από την αφετηρία (σημείο A) μέχρι να φτάσει στον τερματισμό (σημείο B) και προτού τελειώσει η αντίστροφη μέτρηση. Στο συγκεκριμένο παιχνίδι υπάρχουν διαφόρων ειδών εμπόδια και όταν ο παίχτης ακουμπήσει κάποιο από αυτά, υπάρχει μία είδους αρνητική βαθμολογία που αυξάνεται κάθε φορά που ο παίχτης ακουμπάει ένα από τα εμπόδια. Ο χειρισμός του Dodgy γίνεται μέσω των κουμπιών W, A, S, D, όπου το W είναι το μπροστά, το A είναι το αριστερά, το S είναι το πίσω και το D είναι το δεξιά.

Μέσα στον σχεδιασμό παιχνιδιού, εντάσσεται και η εμπειρία του παίχτη (player's experience), η οποία σχετίζεται με το πως πρέπει το παιχνίδι να κάνει τον παίχτη να αισθανθεί, όσο ο ίδιος το παίζει. Στην συγκεκριμένη περίπτωση, ο παίχτης πρέπει να είναι επιδέξιος και ευκίνητος. Γενικά πρέπει να κινείται όσο πιο γρήγορα γίνεται, διότι υπάρχει η αντίστροφη μέτρηση και όσο περισσότερο καθυστερεί να προχωρήσει στην πίστα, το παιχνίδι γίνεται όλο και πιο δύσκολο γιατί προστίθενται περισσότερα εμπόδια.

Ένα ακόμα βασικό στοιχείο του σχεδιασμού παιχνιδιού είναι ο κύριος μηχανισμός του παιχνιδιού (core mechanic). Στο συγκεκριμένο παιχνίδι, ο κύριος μηχανισμός του παιχνιδιού είναι πολύ απλός και είναι να κινείται ο παίχτης μόνιμα και γρήγορα για να αποφεύγει τα εμπόδια.

Το τελευταίο βασικό στοιχείο του σχεδιασμού παιχνιδιού είναι το ποιος είναι ο σκοπός του παιχνιδιού (game loop). Με άλλα λόγια, τι είναι αυτό που κάνει ο παίχτης συνέχεια σε κάθε πίστα. Αυτό που κάνει είναι να πηγαίνει από την αφετηρία (σημείο A) στον τερματισμό (σημείο B), αποφεύγοντας τα εμπόδια που υπάρχουν στην πίστα.

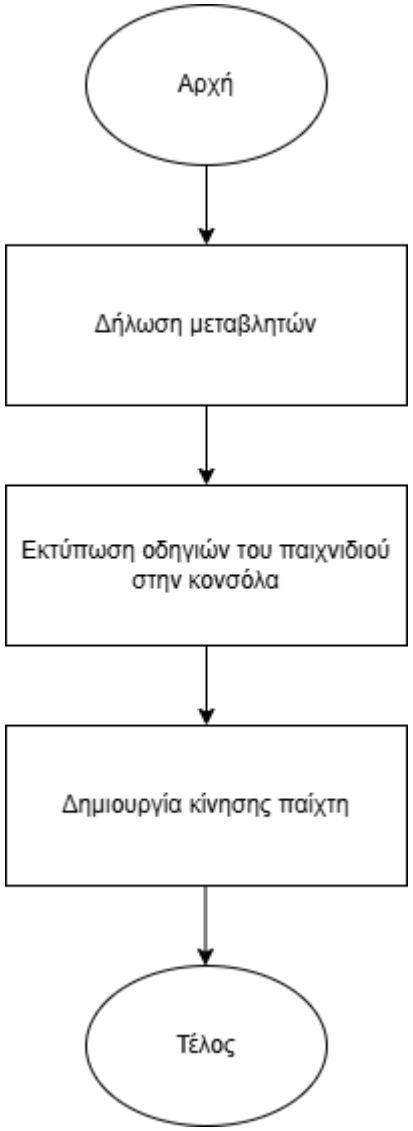
2.3 Χειρισμός Dodgy

Αρχικά, το πρώτο πράγμα που πρέπει να γίνει στην Unity, είναι να δημιουργηθεί ένα game object για το δάπεδο της πίστας και να προστεθεί ένα νέο game object για τον παίχτη, τον Dodgy. Το game object που χρησιμοποιείται για τον Dodgy, είναι ένας κύβος.

Στην συνέχεια, αυτό που έχει σειρά είναι να δημιουργηθεί ένα script και να τοποθετηθεί πάνω στον Dodgy, το οποίο είναι υπεύθυνο για την κίνησή του, αλλά και για την εκτύπωση των οδηγιών του παιχνιδιού στην κονσόλα, όταν το παιχνίδι ξεκινάει. Η κίνηση του Dodgy γίνεται μέσω των κουμπιών

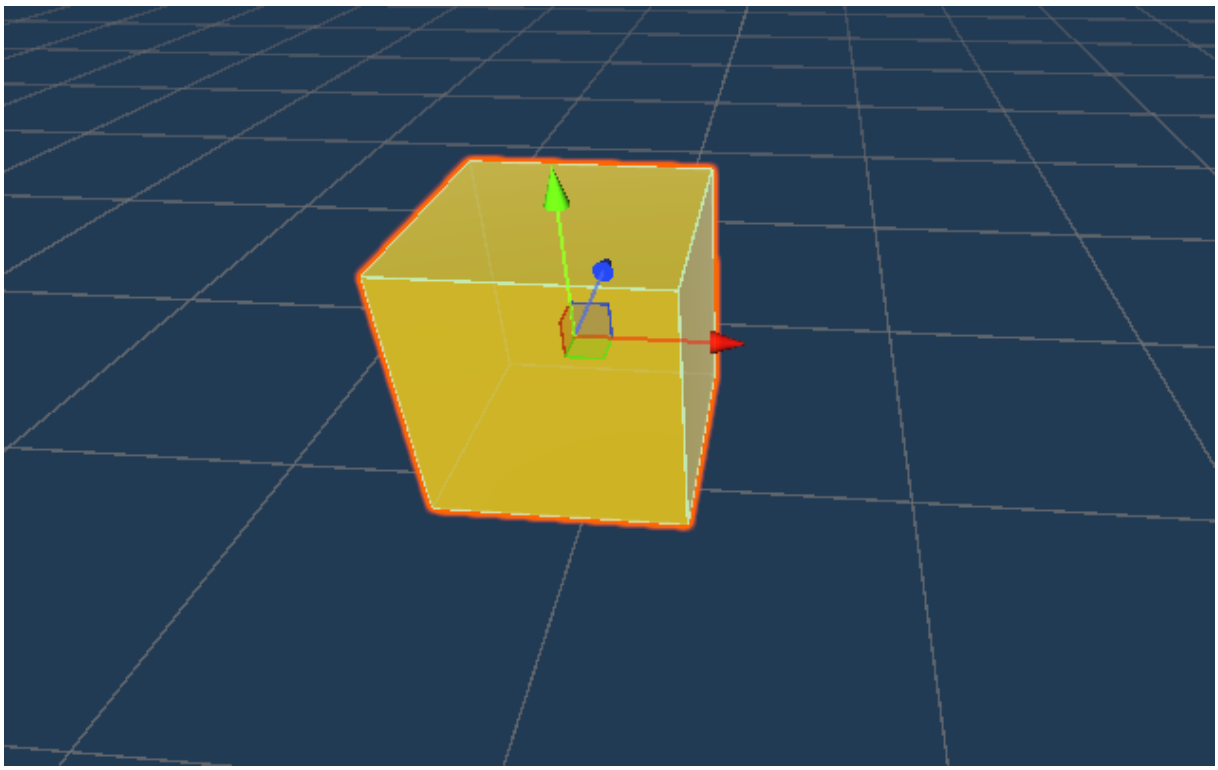
W, A, S, D. Γι' αυτόν τον σκοπό, γίνεται η χρήση της μεθόδου Update. Στην μέθοδο Update, ό,τι υπάρχει μέσα της, εκτελείται μία φορά σε κάθε καρέ (frame). Εφόσον δίνονται τιμές, μέσω των κουμπιών W, A, S, D, σε διάφορες στιγμές του παιχνιδιού και όχι μόνο όταν ξεκινάει το παιχνίδι, χρησιμοποιείται η μέθοδος Update και όχι η Start, διότι η μέθοδος Start εκτελείται μόνο μία φορά, μόλις τρέξει το script.

Επίσης, το παιχνίδι πρέπει να έχει το ίδιο αποτέλεσμα είτε το παίζει κάποιος σε έναν γρήγορο υπολογιστή, είτε σε έναν αργό. Δηλαδή είτε έχει πολλά καρέ ανά δευτερόλεπτο (frames per second (fps)), είτε όχι, το παιχνίδι πρέπει να τρέχει με τον ίδιο τρόπο. Επειδή χρησιμοποιείται η Update για την κίνηση του Dodgy και η Update εξαρτάται από την ταχύτητα του κάθε υπολογιστή (για παράδειγμα μπορεί να τρέχει είκοσι φορές το δευτερόλεπτο σε έναν αργό υπολογιστή και εκατό φορές το δευτερόλεπτο σε έναν γρήγορο), χρειάζεται να χρησιμοποιηθεί η κλάση Time και η στατική ιδιότητα deltaTime, που βρίσκεται μέσα σε αυτή την κλάση. Με αυτόν τον τρόπο η Unity μπορεί να πει πόσος χρόνος χρειάζεται για να εκτελεστεί το κάθε frame. Έτσι το παιχνίδι μπορεί να είναι ανεξάρτητο του ρυθμού καρέ (frame rate independent). Το script αυτό δημιουργείται ακολουθώντας την λογική που φαίνεται στο Σχήμα 2.1.

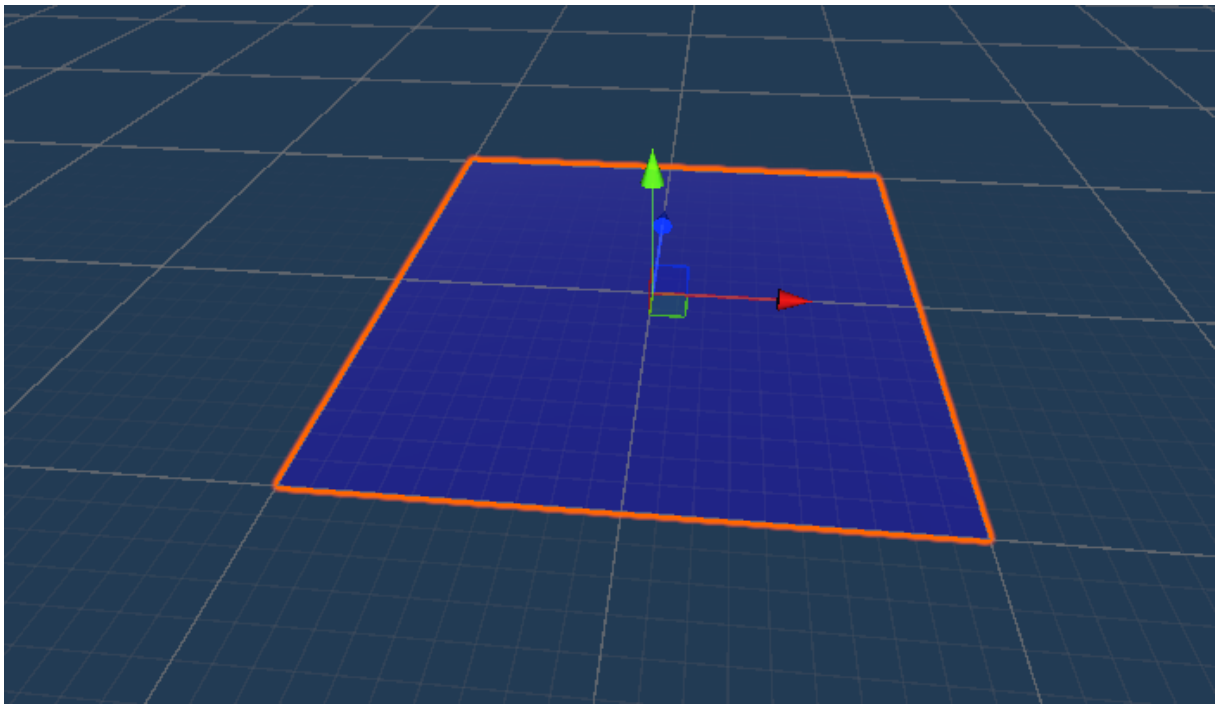


Σχήμα 2.1: *Λογική για το script Mover*

Η αμέσως επόμενη κίνηση είναι να δοθεί χρώμα στον παίχτη και στο δάπεδο, μέσω των materials της Unity. Ο Dodgy έχει κίτρινο χρώμα για να διακρίνεται με ευκολία και το δάπεδο έχει σκούρο μπλε.



Εικόνα 2.1: *Dodgy*



Εικόνα 2.2: *Δάπεδο πίστας*

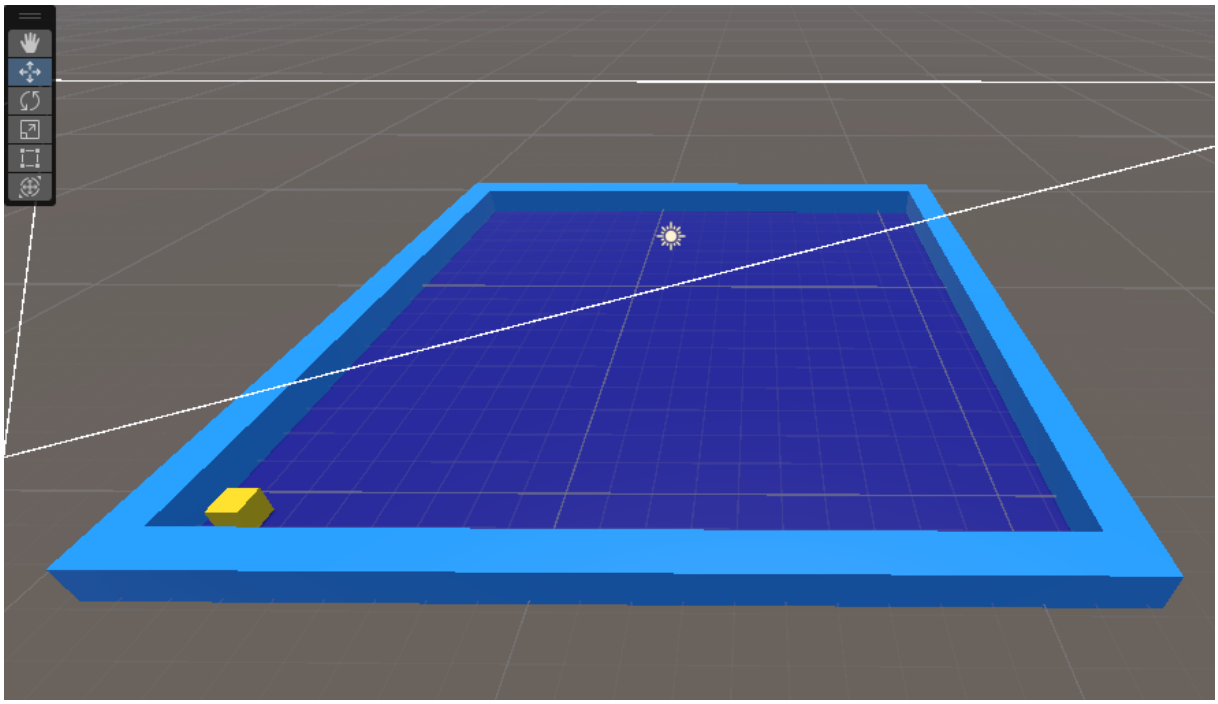
Αυτό που πρέπει να γίνει στην συνέχεια, είναι να προστεθεί μία κάμερα μέσω της Unity, η οποία να κεντράρει και να ακολουθάει μόνιμα τον Dodgy, ακολουθώντας τον ομαλά.

2.4 Σύγκρουση Dodgy με το περιβάλλον

Επόμενη κίνηση, είναι να προστεθούν τοίχοι γύρω από την πίστα, έτσι ώστε να υπάρχουν όρια στην πίστα και να μην μπορεί να βγει ο παίχτης εκτός αυτής. Για να μπορεί όμως ο παίχτης να συγκρουστεί με τους τοίχους της πίστας και να μην τους διαπεράσει, πρέπει να έχει ένα rigidbody στα στοιχεία του (game components). Έτσι το rigidbody, κάνει το κάθε game object να έχει μάζα και να μπορεί να συγκρουστεί με άλλα game objects. Στα στοιχεία του rigidbody, γίνεται uncheck η βαρύτητα, καθώς δεν χρειάζεται, γιατί δεν χρησιμοποιείται ο κάθετος άξονας y στο παιχνίδι.

Ένα πρόβλημα που παρατηρείται είναι ότι αν ο Dodgy συγκρουστεί με κάποιον τοίχο, μπορεί να πεταχτεί στον αέρα. Γι' αυτόν τον λόγο, στα στοιχεία του rigidbody, μπαίνουν περιορισμοί στον Dodgy, για να μην μπορεί να κινηθεί στον άξονα y και να μην μπορεί να περιστραφεί καθόλου στους άξονες x, y και z.

Επιπλέον, προστίθεται στα components των τοίχων, ένα material με το γαλάζιο χρώμα για αυτούς.



Εικόνα 2.3: *Τοίχοι πίστας*

2.5 OnCollisionEnter()

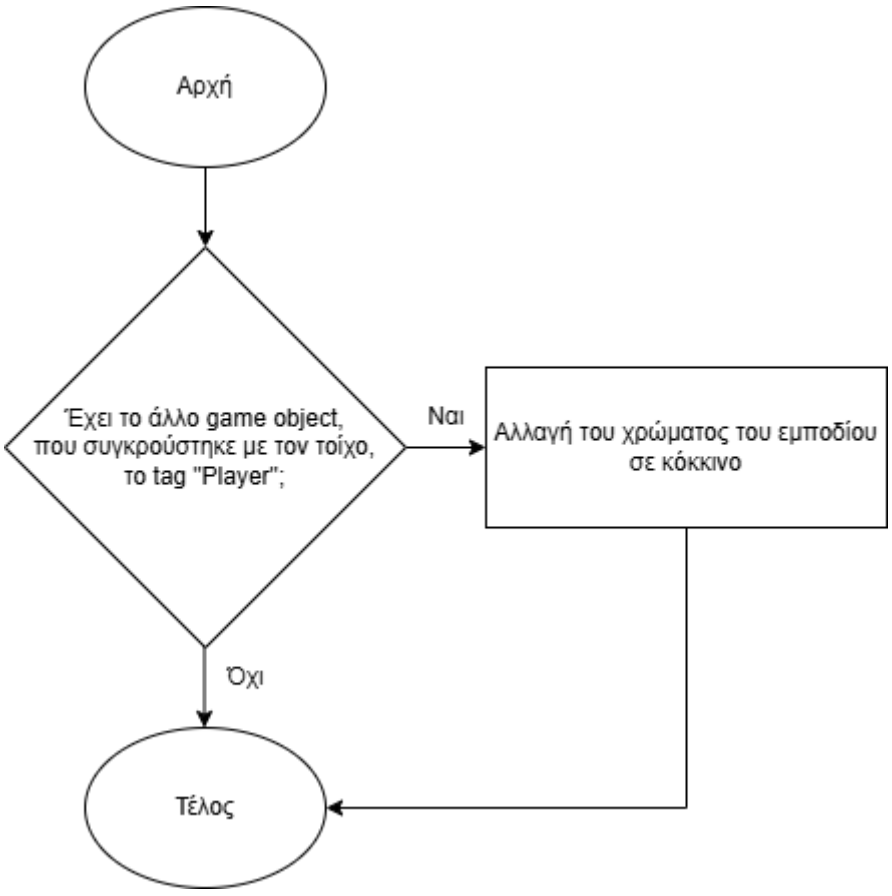
Η επόμενη προσθήκη που πρέπει να γίνει, είναι να δημιουργηθεί ένα script το οποίο τυπώνει στην κονσόλα αν ο παίχτης συγκρούεται με κάποιο άλλο game object. Για να γίνει αυτό, πρέπει αρχικά να προστεθεί το συγκεκριμένο script και στους τέσσερις τοίχους που υπάρχουν γύρω από την πίστα. Ουσιαστικά το script τοποθετείται στους τοίχους και καταλαβαίνει αν κάτι άλλο έρχεται σε επαφή με

αυτούς. Με άλλα λόγια, ειδοποιεί ο κάθε τοίχος ότι χτυπήθηκε από κάτι. Για να επιτευχθεί αυτό χρησιμοποιείται η μέθοδος OnCollisionEnter(). Όταν ένα box collider, αγγίζει ένα άλλο box collider, σηματοδοτείται κάτι που ορίζεται μέσα στο script. Η Unity καταλαβαίνει πότε ακουμπάνε μεταξύ τους τα box colliders και εκτελεί τις εντολές που υπάρχουν μέσα σε αυτή την μέθοδο. Μόλις ακουμπήσουν μεταξύ τους τα box colliders, πρέπει να αλλάζει χρώμα το εμπόδιο με το οποίο συγκρούεται ο παίχτης και να γίνεται κόκκινο. Έτσι υποδηλώνεται ότι το συγκεκριμένο εμπόδιο είναι υπεύθυνο για την σύγκρουση με τον παίχτη. Η μέθοδος OnCollisionEnter() συντάσσεται με τον παρακάτω τρόπο.

```
private void OnCollisionEnter(Collision collision)
{
    Εντολές μέσα στην μέθοδο
}
```

Οι λέξεις «Collision» και «collision» δείχνουν ότι αυτομάτως έχει δημιουργηθεί η παράμετρος, η οποία λέγεται «collision» και Collision είναι ο τύπος της μεταβλητής αυτής.

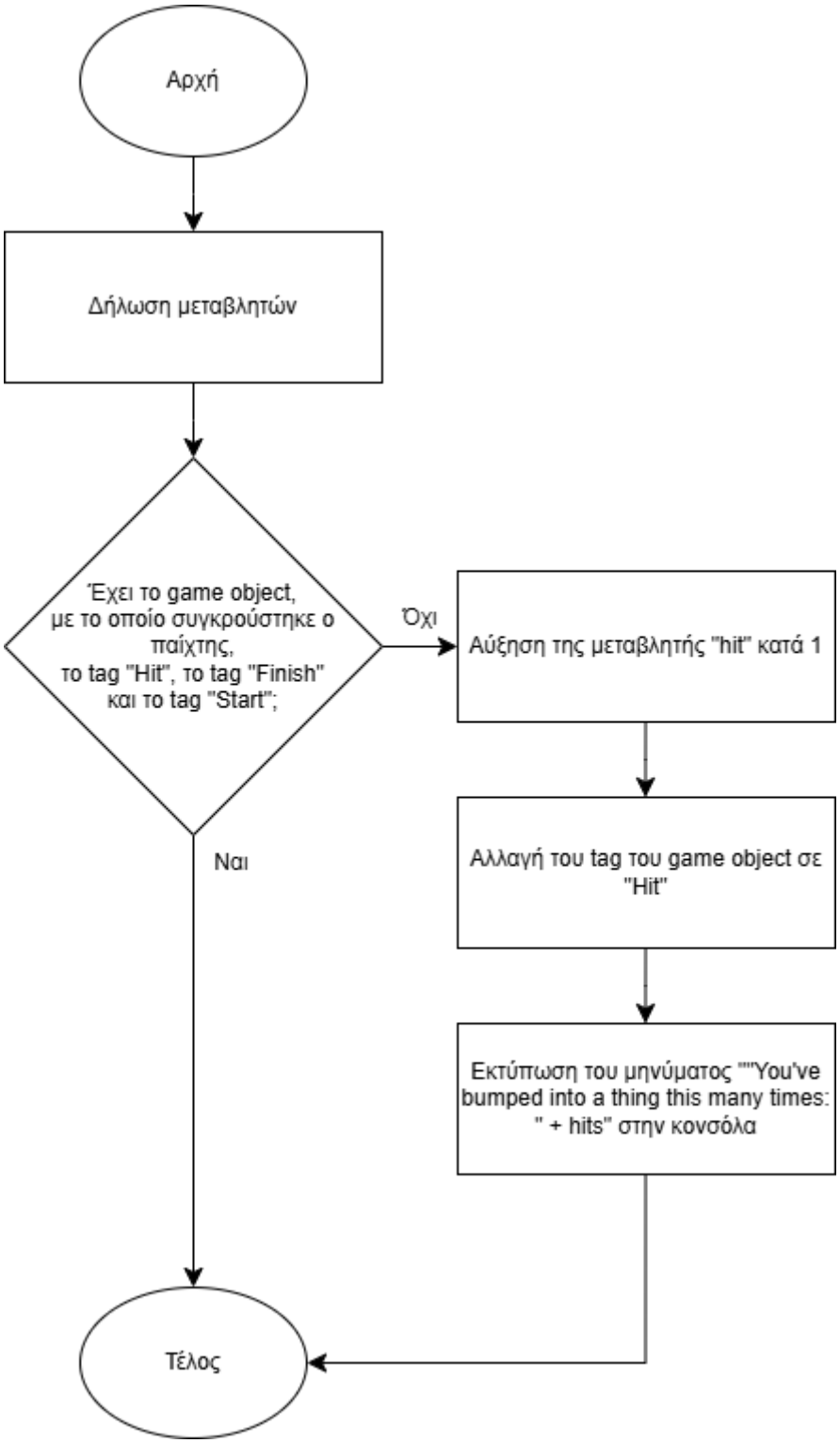
Όταν ο παίχτης χτυπάει στον τοίχο, καλείται η OnCollisionEnter() και την στιγμή που καλείται η μέθοδος αυτή, ζητείται πληροφορία για το ποιος συγκρούστηκε με τον τοίχο. Αυτό ακριβώς είναι η μεταβλητή collision. Δηλαδή, τι ακριβώς είναι το άλλο αντικείμενο που συγκρούεται με το αντικείμενο που έχει πάνω του το script με την μέθοδο OnCollisionEnter(). Το script για να δημιουργηθεί ακολουθεί την λογική που υπάρχει στο Σχήμα 2.2.



Σχήμα 2.2: *Λογική για το script ObjectHit*

2.6 Μεταβολή αρνητικού score

Για να μεταβληθεί το αρνητικό score του παιχνιδιού, δηλαδή για να φανεί πόσες φορές συγκρούεται ο παίχτης με κάποιο εμπόδιο, δημιουργείται ένα καινούριο script και χρησιμοποιείται πάλι η μέθοδος OnCollisionEnter(). Το συγκεκριμένο script εντάσσεται στα components του Dodgy, καθώς αυτός είναι που μεταβάλλει το score, εφόσον αυτός μπορεί να συγκρουστεί με κάποιο άλλο game object. Είναι πιο λογικό να προστεθεί το script μόνο στον Dodgy και όχι σε κάθε ένα game object ξεχωριστά. Κάθε φορά που ο Dodgy συγκρούεται με κάποιο game object που υπάρχει στην σκηνή, αυξάνεται κατά μία μονάδα το αρνητικό score. Το script που είναι υπεύθυνο για την μεταβολή του αρνητικού score, δημιουργείται βάσει της λογικής που παρουσιάζεται στο Σχήμα 2.3.

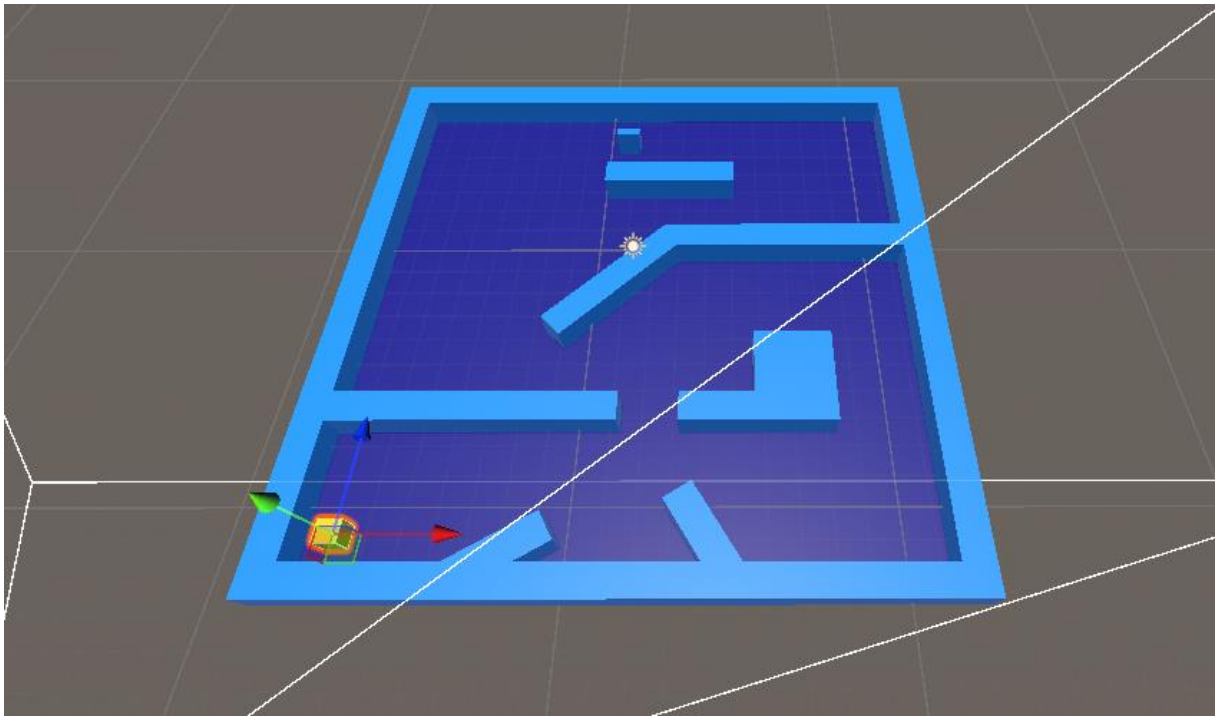


Σχήμα 2.3: *Λογική για το script Scorer*

2.7 Προσθήκη εμποδίων

Αφού γίνουν οι προηγούμενες απαραίτητες προσθήκες, τώρα είναι η κατάλληλη στιγμή για να προστεθούν ακίνητα εμπόδια μέσα στην πίστα. Δημιουργούνται game objects διαφορετικών μεγεθών, αποτελώντας τους τοίχους, που βρίσκονται σε όλη την πίστα, δίνοντας την αίσθηση ενός λαβυρίνθου.

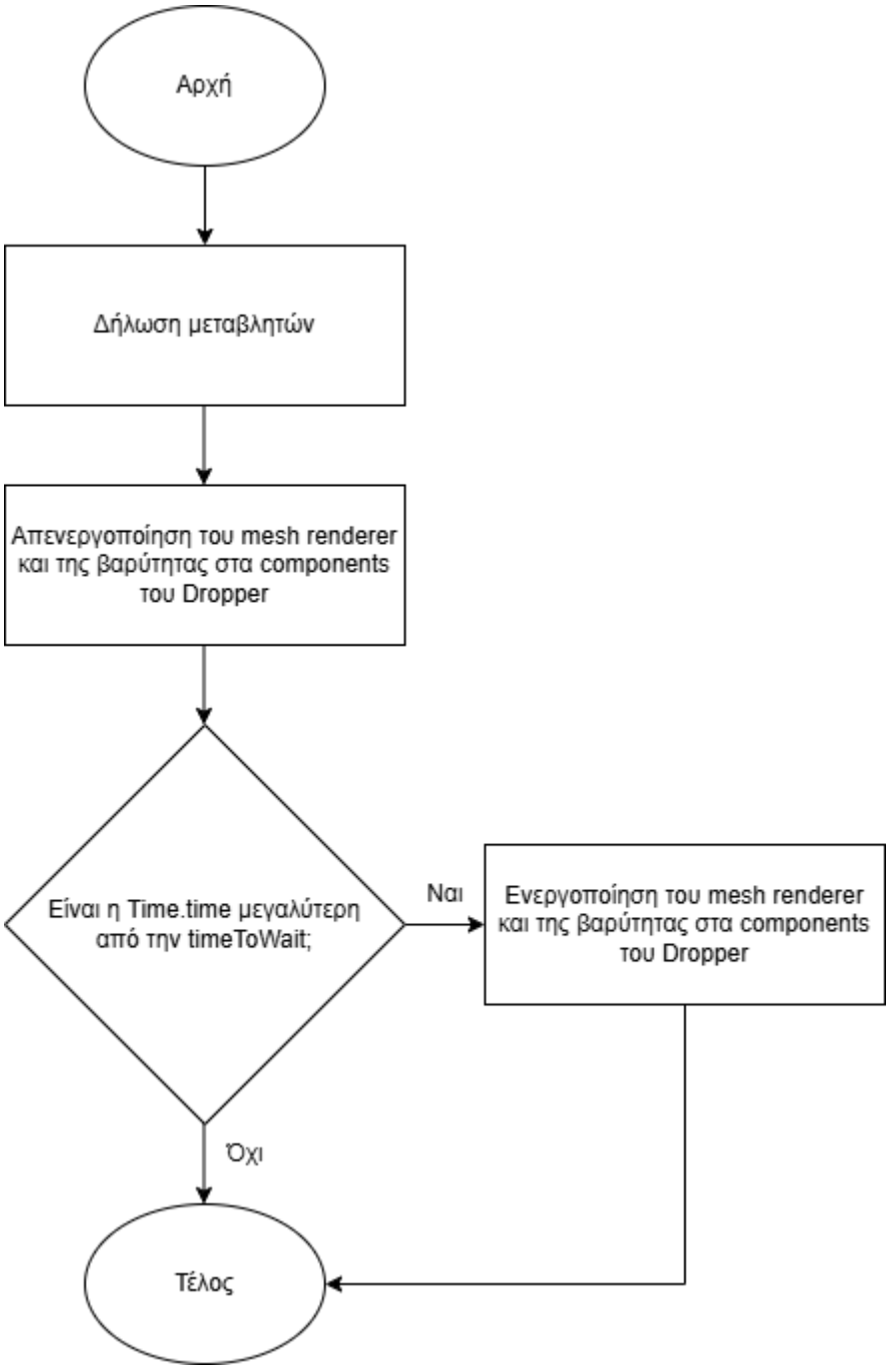
Οι συγκεκριμένοι τοίχοι έχουν το ίδιο χρώμα με τους τέσσερις τοίχους που δημιουργούν τα όρια της πίστας, οπότε χρησιμοποιείται το ίδιο material και γι' αυτούς.



Εικόνα 2.4: *Τοίχοι εντός της πίστας*

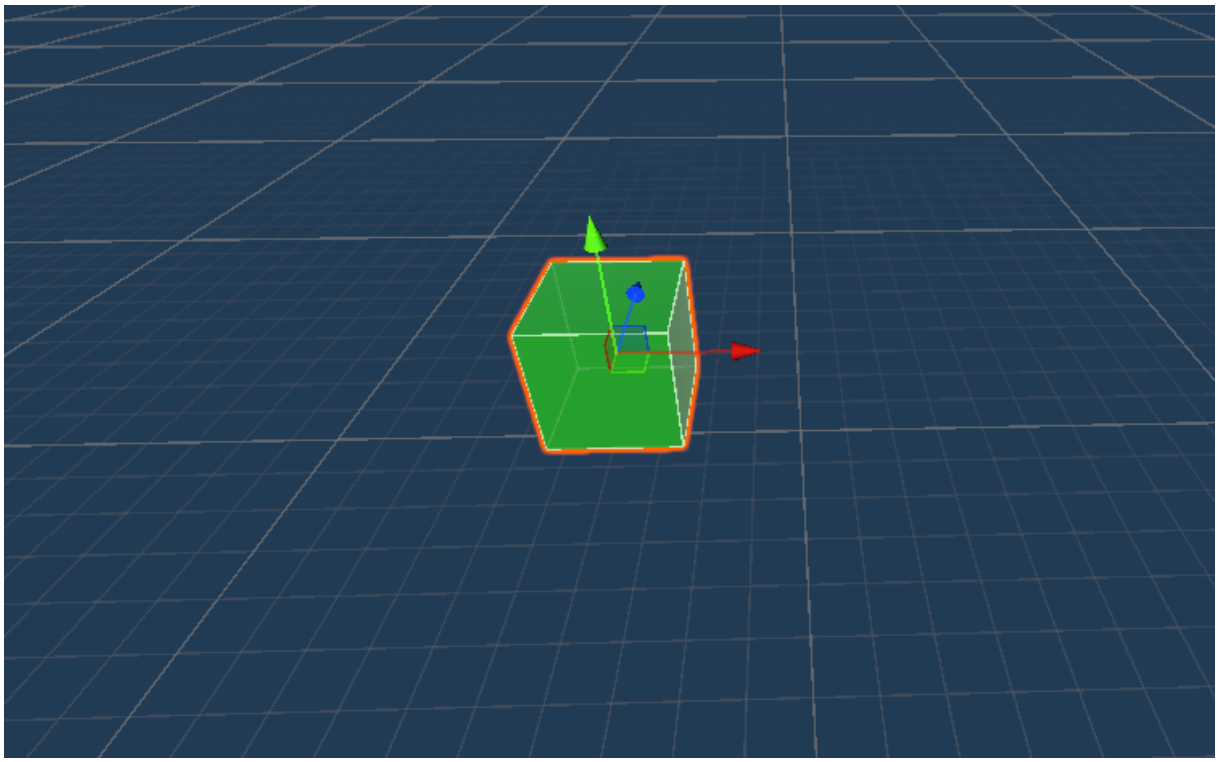
2.7.1 Dropper

Πέρα από τους τοίχους που υπάρχουν μέσα στην πίστα, προστίθενται και άλλα εμπόδια, τα οποία είναι κινούμενα και αναλύεται το καθένα ξεχωριστά στην συνέχεια. Το πρώτο εμπόδιο που δημιουργείται είναι το Dropper. Είναι ένα game object που μετά από έναν καθορισμένο χρόνο εμφανίζεται αφού πρώτα είναι αόρατο και πέφτει από ψηλά, πάνω στο δάπεδο. Το εμπόδιο Dropper είναι ένας κύβος και γι' αυτό χρησιμοποιείται το κατάλληλο game object. Το Dropper έχει χρώμα πράσινο, οπότε δημιουργείται και το κατάλληλο material. Αυτό που γίνεται αμέσως μετά, είναι να τεθεί ο φωτισμός της πίστας σε κατάλληλη γωνία, έτσι ώστε η σκιά του Dropper, να υποδεικνύει περίπου που πέφτει. Εν συνεχεία, δημιουργείται ένα script που είναι υπεύθυνο για την λειτουργία του Dropper και προστίθεται πάνω του. Για να επιτευχθεί η συμπεριφορά του Dropper, πρέπει να χρησιμοποιηθεί ένα χρονόμετρο. Γι' αυτόν τον λόγο χρησιμοποιείται η κλάση Time και η στατική ιδιότητα time, που βρίσκεται μέσα σε αυτή την κλάση. Επιπλέον, για να μπορεί το Dropper να πέφτει λόγω της βαρύτητας, προστίθεται πάνω του ένα rigidbody και ενεργοποιεί ουσιαστικά την βαρύτητα, αφού περάσει ο απαιτούμενος χρόνος για να πέσει στο δάπεδο. Γίνεται ακριβώς το ίδιο για το mesh renderer. Ουσιαστικά, αφού περάσει ο απαιτούμενος χρόνος, το εμπόδιο αυτό, εμφανίζεται, όντας πρώτα αόρατο (απενεργοποίηση του mesh renderer) και πέφτει λόγω της βαρύτητας (ενεργοποίηση της βαρύτητας). Η λογική πίσω από την δημιουργία του script που είναι υπεύθυνο για την συμπεριφορά του Dropper, παρουσιάζεται στο Σχήμα 2.4.



Σχήμα 2.4: *Λογική για το script Dropper*

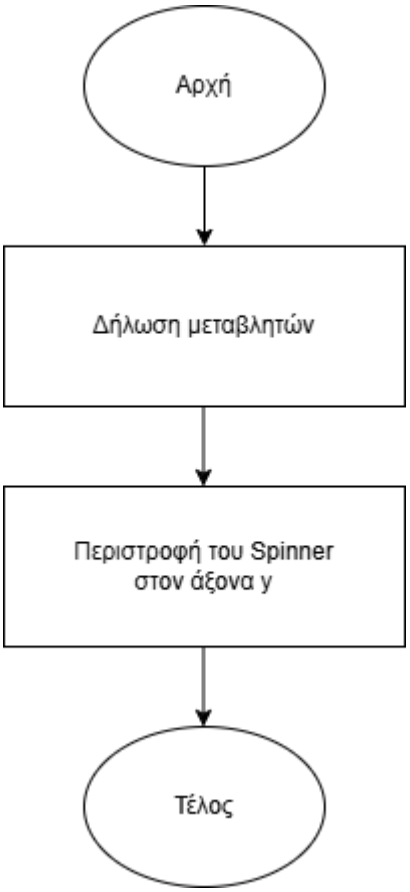
Στο Dropper προστίθεται και το script που είναι υπεύθυνο για την αλλαγή του χρώματος του κάθε εμποδίου σε κόκκινο, όταν ο παίχτης χτυπήσει πάνω του.



Εικόνα 2.5: *Dropper*

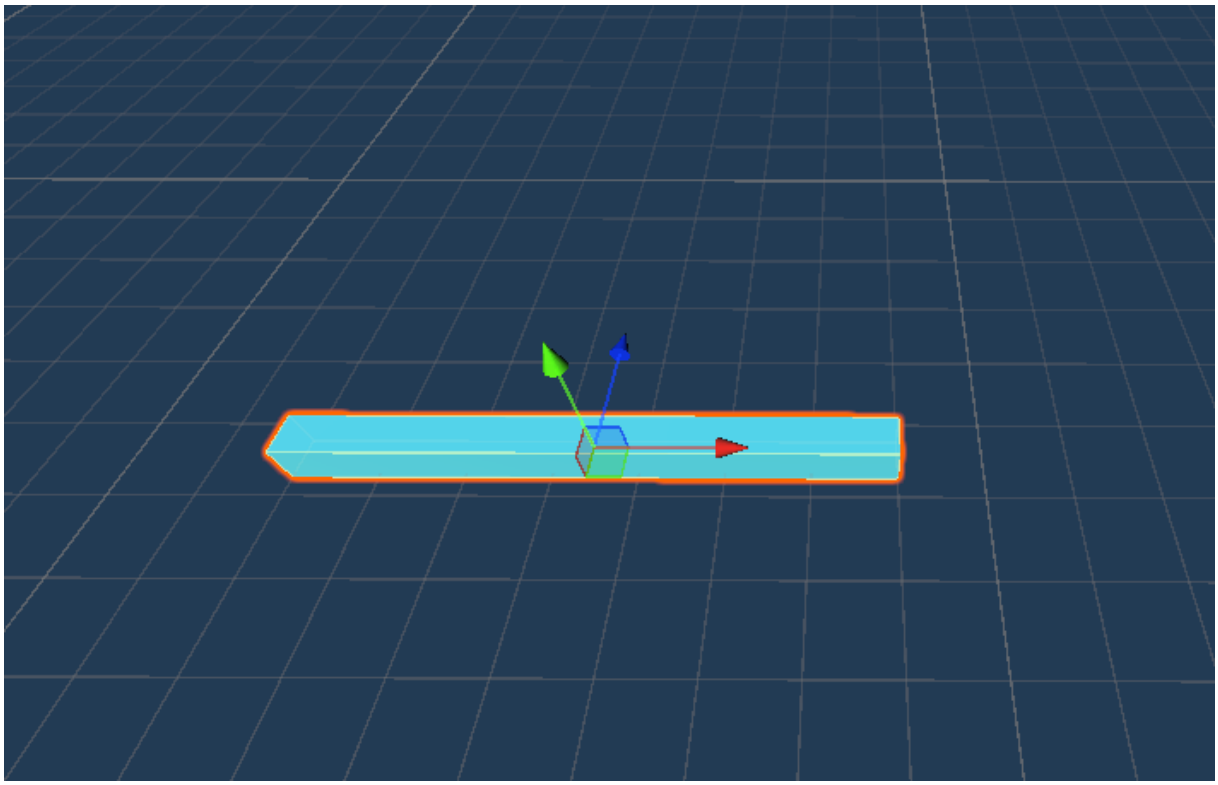
2.7.2 Spinner

Τώρα πρέπει να δημιουργηθεί ένα νέο είδος εμποδίου, το οποίο έχοντας έναν σταθερό άξονα, περιστρέφεται γύρω από αυτόν. Αυτό το νέο είδος εμποδίου ονομάζεται «Spinner». Το συγκεκριμένο εμπόδιο είναι ένα ορθογώνιο παραλληλεπίπεδο, οπότε δημιουργείται ένα game object ενός κύβου και γίνεται η κατάλληλη επεξεργασία για να γίνει ορθογώνιο παραλληλεπίπεδο. Δημιουργείται το κατάλληλο material για να πάρει το Spinner το χρώμα τιρκουάζ. Για να επιτευχθεί η συμπεριφορά του Spinner, προστίθεται στα components του, το script που το κάνει να περιστρέφεται γύρω από τον άξονά του. Η λογική που ακολουθείται για την δημιουργία αυτού του script, φαίνεται στο Σχήμα 2.5.



Σχήμα 2.5: *Λογική για το script Spinner*

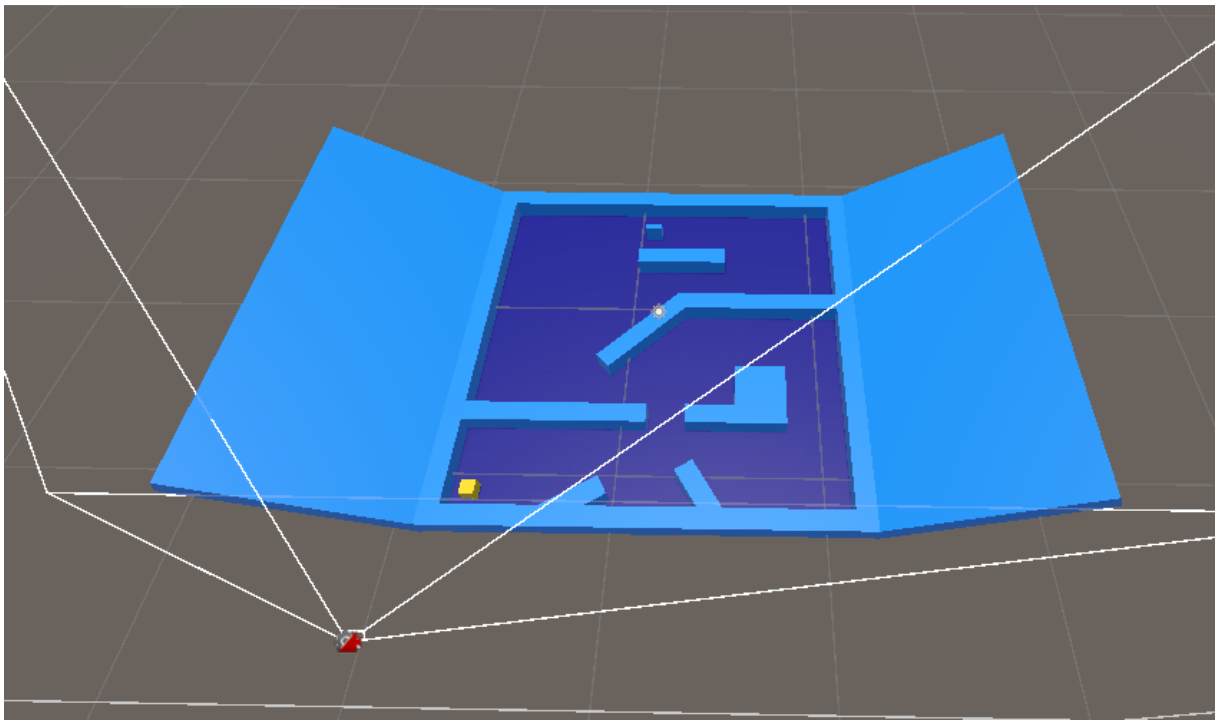
Στο Spinner προστίθεται και το script που είναι υπεύθυνο για την αλλαγή του χρώματος του κάθε εμποδίου σε κόκκινο, όταν ο παίχτης χτυπήσει πάνω του.



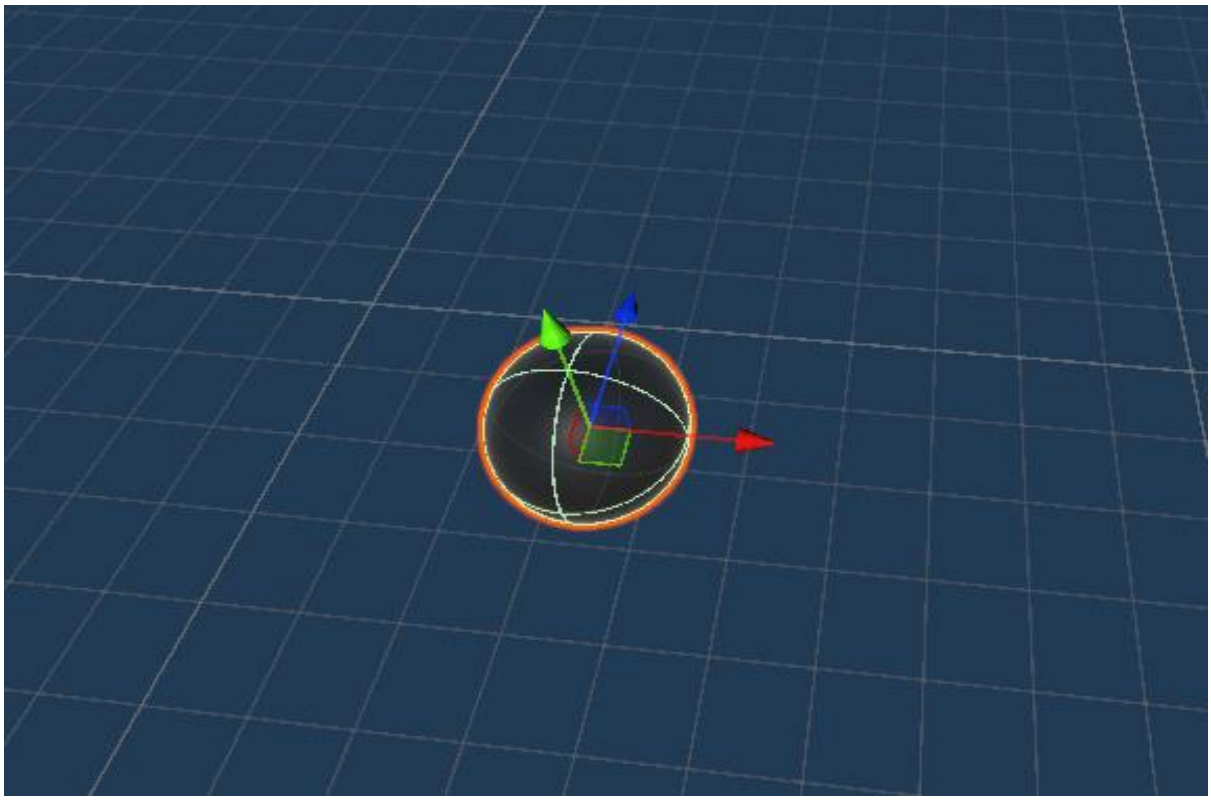
Εικόνα 2.6: *Spinner*

2.7.3 Roller

Το επόμενο και τελευταίο εμπόδιο που προστίθεται στην πίστα, είναι το Roller, το οποίο είναι μια σφαίρα. Το Roller, είναι ένα εμπόδιο, το οποίο λόγω της βαρύτητας πέφτει και κατρακυλάει σε ένα κεκλιμένο επίπεδο και εισέρχεται στην πίστα, χωρίς να έχει κάποια άλλη συγκεκριμένη κίνηση. Για την συμπεριφορά του Roller, δημιουργούνται δεξιά και αριστερά της πίστας, δύο μεγάλα κεκλιμένα επίπεδα, στα οποία πέφτει από ψηλά το roller και ξεκινάει να κατρακυλάει. Η συμπεριφορά του Roller είναι να πέσει από ψηλά, κάθετα, πάνω στο κεκλιμένο επίπεδο, οπότε σηκώνεται ψηλά στον αέρα, η βαρύτητα το κάνει να πέσει στο κεκλιμένο επίπεδο και εν συνεχεία να εισέλθει στην πίστα. Για να μπορέσει όμως να λειτουργήσει η βαρύτητα πάνω στο Roller, του προστίθεται ένα rigidbody. Χρειάζεται επίσης να του προστεθεί το script που είναι υπεύθυνο για την αλλαγή του χρώματος ενός εμποδίου, όταν ο παίχτης συγκρούεται με αυτό. Αμέσως μετά, του δίνεται χρώμα και γι' αυτόν ακριβώς τον λόγο, δημιουργείται ένα material με μαύρο χρώμα και προστίθεται σε αυτό.



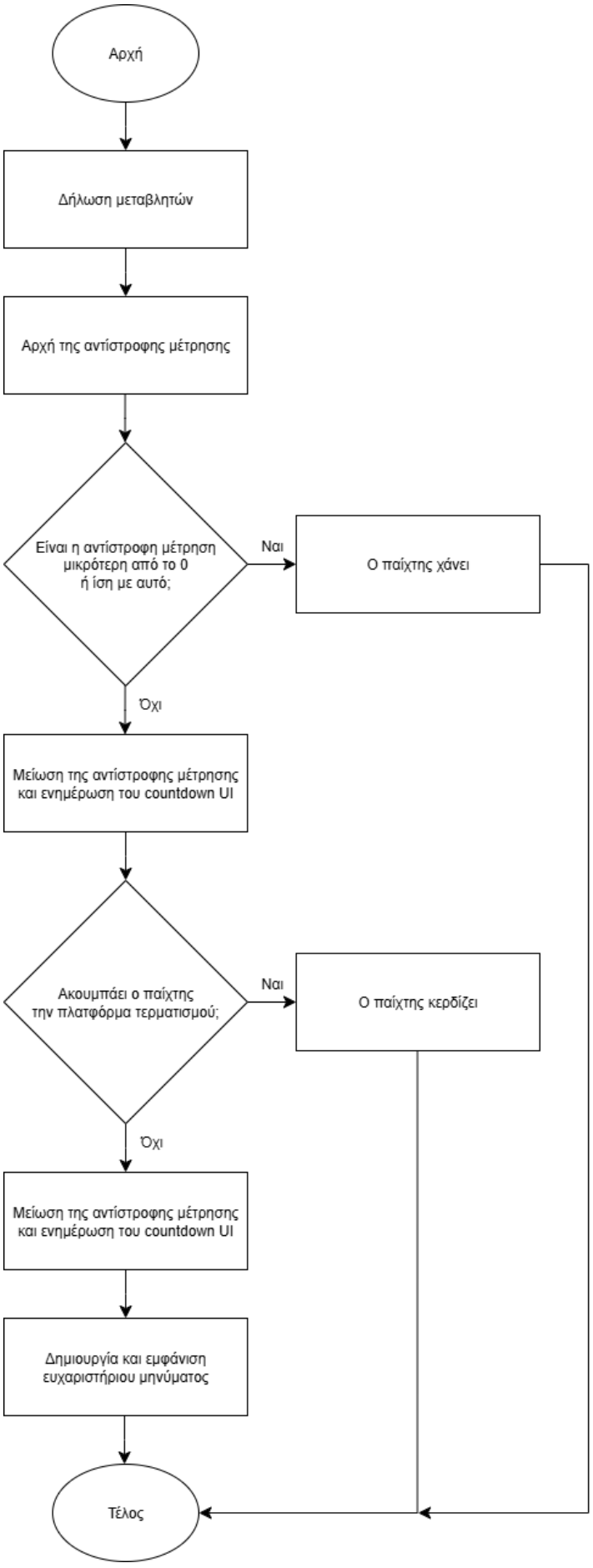
Εικόνα 2.7: *Κεκλιμένα επίπεδα*



Εικόνα 2.8: *Roller*

2.8 Αφετηρία, τερματισμός και αντίστροφη μέτρηση

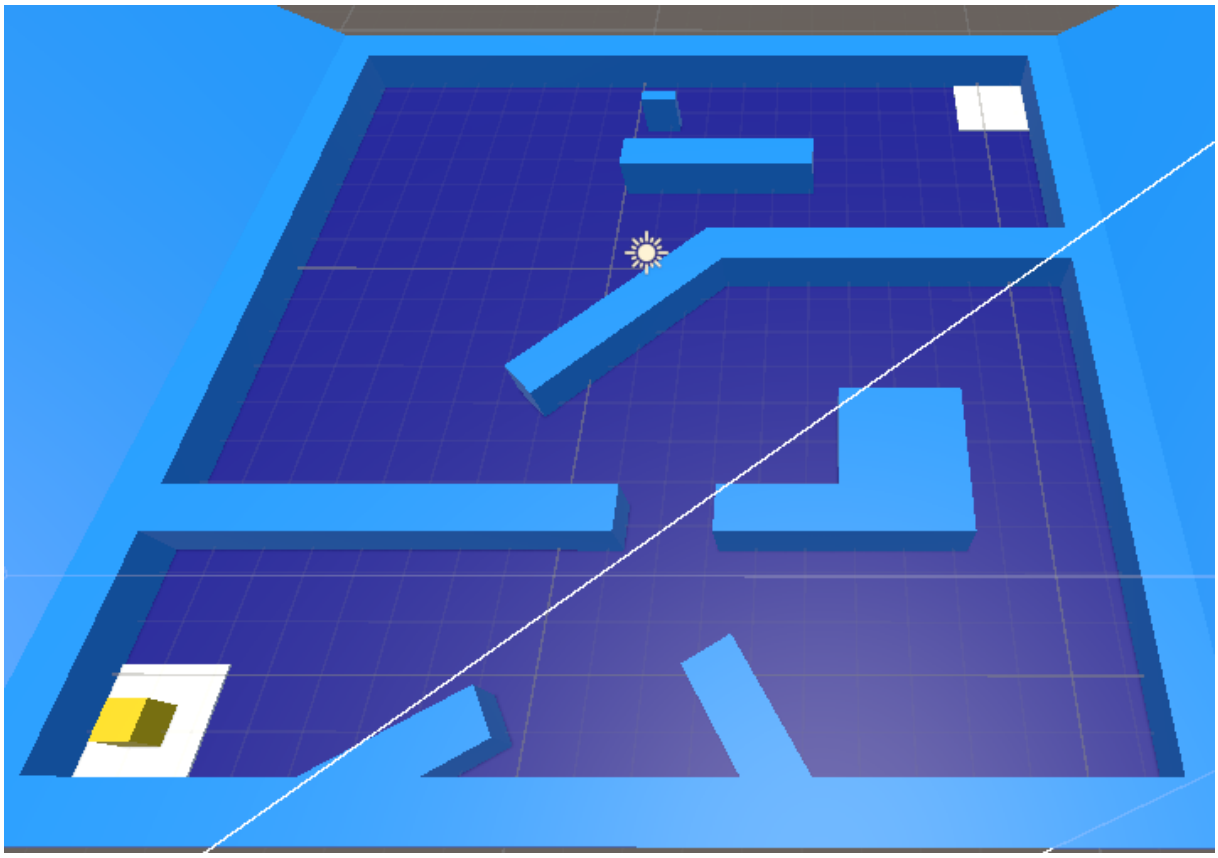
Εφόσον δημιουργηθούν όλα τα είδη εμποδίων, ήρθε η ώρα να δημιουργηθούν η αφετηρία και ο τερματισμός της πίστας. Είναι δύο πλατφόρμες που υποδηλώνουν την αρχή και το τέλος της πίστας. Συγκεκριμένα, όταν ο παίχτης ακουμπάει στην πλατφόρμα του τερματισμού, σταματάει η αντίστροφη μέτρηση που υπάρχει κατά την διάρκεια της πίστας και ο παίχτης κερδίζει. Στις πλατφόρμες δίνεται άσπρο χρώμα και γι' αυτόν τον λόγο τους προστίθεται το κατάλληλο material. Η αφετηρία τοποθετείται στο κάτω αριστερό μέρος της πίστας, ενώ ο τερματισμός στο πάνω δεξιά μέρος της πίστας. Μαζί με τις πλατφόρμες της αφετηρίας και του τερματισμού, δημιουργείται και η αντίστροφη μέτρηση που υπάρχει κατά την διάρκεια του παιχνιδιού. Η αντίστροφη μέτρηση πραγματοποιείται με ένα κείμενο, μέσω της διεπαφής του χρήστη (user interface (UI)) και βρίσκεται ψηλά και κεντρικά στην οθόνη, για να είναι ευδιάκριτη. Για να επιτευχθεί η συμπεριφορά που πρέπει να έχει η πλατφόρμα του τερματισμού και να λειτουργήσει σωστά η αντίστροφη μέτρηση, δημιουργείται ένα script και εντάσσεται στα components του Dodgy. Το script αυτό δημιουργείται με βάση την λογική που παρουσιάζεται στο Σχήμα 2.6.


Αρχή
Δήλωση μεταβλητών
Αρχή της αντίστροφης μέτρησης
Είναι η αντίστροφη μέτρηση μικρότερη από το 0 ή ίση με αυτό;
Ναι
Ο παίχτης χάνει
Όχι
Μείωση της αντίστροφης μέτρησης και ενημέρωση του countdown UI
Ακουμπάει ο παίχτης την πλατφόρμα τερματισμού;
Ναι
Ο παίχτης κερδίζει
Όχι
Μείωση της αντίστροφης μέτρησης και ενημέρωση του countdown UI
Δημιουργία και εμφάνιση ευχαριστήριου μηνύματος
Τέλος

Σχήμα 2.6: *Λογική για το script EndGameCondition*

Επιπλέον, δημιουργείται ένα νέο script και προστίθεται σε ένα panel με το όνομα «Thank You Message Panel». Το script αυτό είναι υπεύθυνο για να δείχνει ένα ευχαριστήριο μήνυμα, αμέσως μετά το τέλος αυτού του demo, είτε ο παίχτης κερδίσει, είτε χάσει. Το script υλοποιείται με βάση την λογική που εμφανίζεται στο Σχήμα 2.7.

Σχήμα 2.7: *Λογική για το script ThankYouMessage*



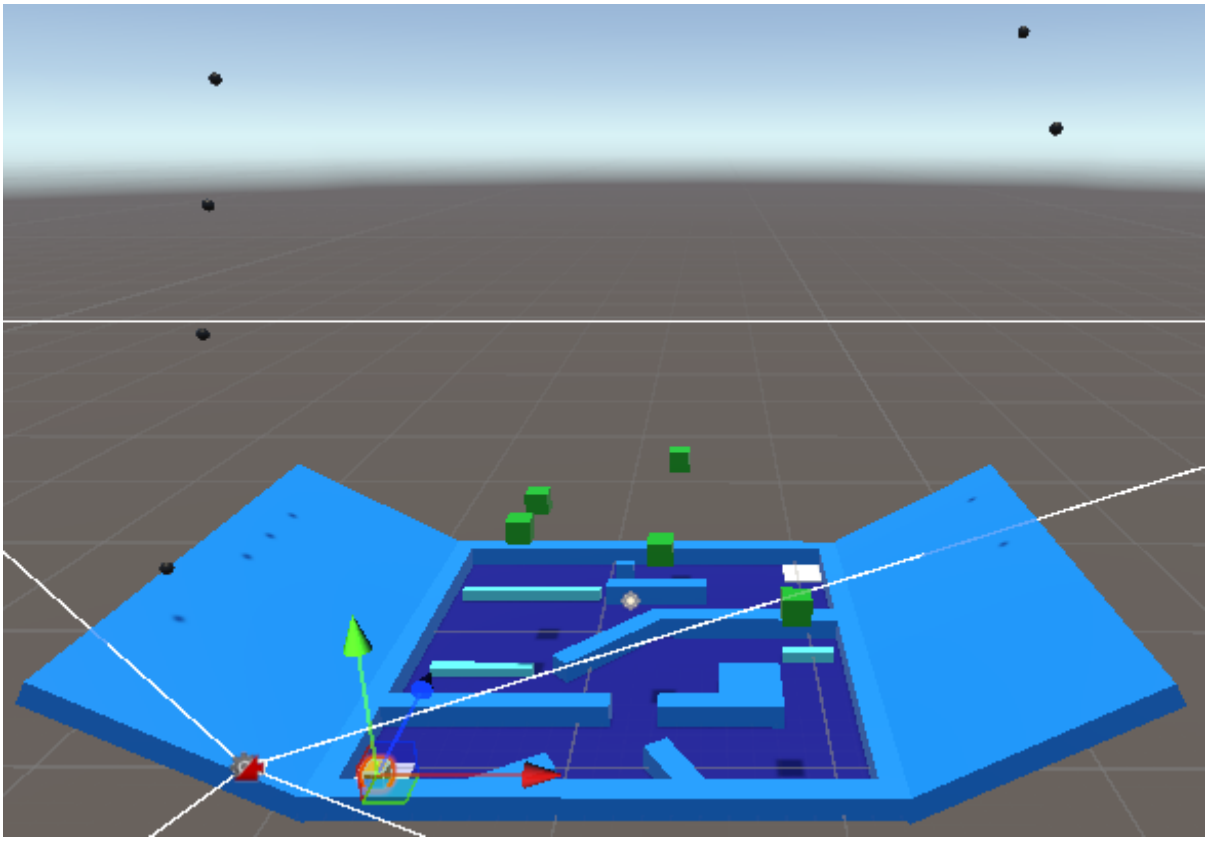
Εικόνα 2.9: *Αφετηρία και τερματισμός*

Εικόνα 2.10: *Αντίστροφη μέτρηση*

2.9 Σχεδιασμός πίστας (Level design)

Αφού γίνουν όλες οι απαραίτητες ενέργειες από άποψη κώδικα και βασικών στοιχείων του παιχνιδιού, ακολουθεί το κομμάτι του σχεδιασμού της πίστας (level design), δηλαδή να σχεδιαστεί η παρούσα πίστα ή το παρόν επίπεδο (level).

Καθώς το παιχνίδι πρέπει να είναι απαιτητικό, αλλά παράλληλα διασκεδαστικό και προσιτό προς όλους και όχι ακατόρθωτο, πρέπει το κάθε εμπόδιο να τοποθετηθεί σε συγκεκριμένο σημείο, που θεωρείται ότι είναι καίριο για τον παίχτη. Εφόσον μπουν τα προαναφερθέντα εμπόδια στα σημεία που πρέπει, γίνεται η προσθήκη των ίδιων εμποδίων και σε άλλα διαφορετικά σημεία της πίστας. Για να έχει περισσότερο ενδιαφέρον η πίστα, το κάθε είδος εμποδίου μπορεί να είναι εν μέρει ίδιο οπτικά, αλλά μπορεί κάλλιστα να έχει διαφορετικές προδιαγραφές και να συμπεριφέρεται διαφορετικά από τα όμοια με αυτό εμπόδια. Γι' αυτόν τον λόγο λοιπόν, το κάθε Dropper πέφτει μετά από διαφορετικό χρόνο, το κάθε Spinner γυρνάει με διαφορετική ταχύτητα και το κάθε Roller πέφτει από διαφορετικό ύψος, πάνω στο κάθε κεκλιμένο επίπεδο, για να έχει λίγο διαφορετική ταχύτητα με την οποία εισέρχεται στην πίστα.



Εικόνα 2.11: *Σχεδιασμός πίστας (Level design)*

2.10 Debugging

Ενώ το παιχνίδι λειτουργεί κανονικά, χωρίς κάποιο bug που να δημιουργεί πρόβλημα, παρατηρήθηκε ότι σε κάποια σημεία της πίστας, αν ο Dodgy αφεθεί ελεύθερος, κυλάει προς μία κατεύθυνση σαν να βρίσκεται πάνω σε πάγο. Ελέγχθηκε αν η πίστα έχει κάποια κλίση στον άξονα z, αλλά δεν υπάρχει κάποιο τέτοιο πρόβλημα. Αυτή η συμπεριφορά του Dodgy παρατηρήθηκε μόνο σε συγκεκριμένα σημεία της πίστας, από την μέση της πίστας και μετά. Αυτό που χρειάστηκε για να λυθεί το πρόβλημα, ήταν η προσθήκη ενός physics material πάνω στο box collider του Dodgy.

2.11 Επίλογος

Καθώς το συγκεκριμένο κεφάλαιο έφτασε στο τέλος του, διακρίνεται ότι για το παιχνίδι Dodgy, χρησιμοποιήθηκαν αρκετά βασικά στοιχεία της Unity και της C#. Πιο συγκεκριμένα, παρόλο που δεν υπάρχουν τρισδιάστατα μοντέλα και αντικείμενα, έγινε χρήση διαφόρων βασικών game objects και λειτουργιών της Unity, όπως και η χρήση αρκετών μεθόδων και εντολών της C# που χρησιμοποιούνται κατά κόρον και στην συνέχεια της παρούσας διπλωματικής εργασίας. Χρησιμοποιήθηκαν βασικά στοιχεία των συγκρούσεων μεταξύ αντικειμένων και έγινε προσθήκη αρκετών εμποδίων σε συγκεκριμένα σημεία που είναι καθοριστικά και για την κίνηση του Dodgy, αλλά και για τον τερματισμό της πίστας.

Κεφάλαιο 3ο: SpacEscape

3.1 Εισαγωγή

Το παιχνίδι «SpacEscape», είναι το δεύτερο παιχνίδι αυτής της διπλωματικής εργασίας και είναι ένα 3D platformer παιχνίδι, όπου ο παίχτης, χειριζόμενος τον πύραυλο Starseeker, προσπαθεί να εξερευνήσει τους τρεις πλανήτες τους ηλιακού μας συστήματος, πέρα από την Γη, που έχουν έδαφος και μπορεί ο πύραυλος να προσεδαφιστεί. Ο πύραυλος Starseeker, αρχικά κάνει δοκιμή στην Γη, για να ελεγχθεί η ομαλή κίνησή του και αντιμετωπίζει διάφορες προκλήσεις, προτού ξεκινήσει το ταξίδι του. Σε κάθε πλανήτη, κατά την διάρκεια της εξερεύνησής του, ο πύραυλος βρίσκει μπροστά του αμέτρητα εμπόδια, είτε ακίνητα, είτε κινούμενα, που υπάρχουν στο περιβάλλον και προσπαθεί να τα αποφύγει, έτσι ώστε να μπορέσει να πάει και στους υπόλοιπους πλανήτες, χωρίς να συντριβεί. Παρόλο που ο παίχτης μπορεί να κινηθεί μόνο στους άξονες x και y, το παιχνίδι είναι τρισδιάστατο, καθώς υπάρχει βάθος σε κάθε πίστα και υπάρχουν εμπόδια, τα οποία μπορούν να μεταβάλουν την τιμή τους στον άξονα z, είτε είναι μέσω κάποιας κίνησης στον άξονα z, είτε είναι μέσω περιστροφής σε αυτόν.

3.2 Σχεδιασμός παιχνιδιού του SpacEscape (Game design of SpacEscape)

Στο συγκεκριμένο παιχνίδι, ο πύραυλος και όλο το περιβάλλον μαζί με τα εμπόδια αποτελούνται από απλά αντικείμενα που βρίσκει κάποιος στα ήδη έτοιμα game objects της Unity, γιατί το παιχνίδι εμβαθύνει σε πιο σημαντικά στοιχεία. Καθ' όλη την διάρκεια της πίστας υπάρχει μία κάμερα που ακολουθεί τον πύραυλο, ακριβώς με τον ίδιο τρόπο, όπως και στο παιχνίδι Dodgy. Ο σκοπός του παίχτη είναι να φτάσει από την αφετηρία (σημείο A) στον τερματισμό (σημείο B), χωρίς να συγκρουστεί με κάποιο εμπόδιο. Σε περίπτωση που συγκρουστεί ο παίχτης με κάποιο εμπόδιο, ο πύραυλος εκρήγνυται, ο παίχτης χάνει και ξαναεμφανίζεται στην αφετηρία της ίδιας πίστας, μέχρι να μπορέσει να την περάσει. Αν φτάσει στον τερματισμό, τότε προχωράει στην επόμενη πίστα. Ο χειρισμός του πυραύλου γίνεται κρατώντας πατημένο το κουμπί Space, για ώθηση προς τα πάνω, δηλαδή για ώθηση ως προς τον κάθετο άξονα y και μέσω των κουμπιών A και D, για περιστροφή στον άξονα z. Πιο συγκεκριμένα, όσον αφορά τα κουμπιά A και D, το κουμπί A περιστρέφει τον πύραυλο με την αντίθετη φορά του ρολογιού, ενώ το κουμπί D, τον περιστρέφει με την φορά του ρολογιού. Για να κατέβει προς τα κάτω, ο παίχτης απλά σταματάει να πατάει το κουμπί Space και αφήνει τον πύραυλο να πέσει λόγω της βαρύτητας.

Όσον αφορά την εμπειρία του παίχτη (player's experience), ο παίχτης πρέπει να είναι επιδέξιος και ακριβής στις κινήσεις του, καθώς κάθε λάθος μπορεί να τον φέρει πάλι στην αφετηρία, ξεκινώντας την πίστα πάλι από την αρχή.

Στο συγκεκριμένο παιχνίδι, ο κύριος μηχανισμός του παιχνιδιού (core mechanic) είναι αρκετά απλός, καθώς ο παίχτης πρέπει να είναι επιδέξιος για να έχει μόνιμα τον πύραυλο σε κίνηση και να αποφεύγει τα εμπόδια που υπάρχουν στην διαδρομή μέχρι τον τερματισμό.

Ο σκοπός του παιχνιδιού (game loop), δηλαδή τι είναι αυτό που κάνει ο παίχτης συνέχεια σε κάθε πίστα, είναι να πηγαίνει από την αφετηρία (σημείο A) στον τερματισμό (σημείο B), αποφεύγοντας όλα τα εμπόδια που υπάρχουν στην πίστα, με σκοπό να προχωρήσει στην επόμενη πίστα. Σε περίπτωση που ο παίχτης επιτύχει σε όλες τις πίστες, τότε από την τελευταία πίστα, προχωράει ξανά στην πρώτη πίστα.

3.3 Onion design

Το onion design είναι ένας βασικός τρόπος για να καταλάβει κάποιος τι πρέπει να εντάξει μέσα στο παιχνίδι του. Το όνομά του το πήρε από το πως είναι σχεδιασμένο το κρεμμύδι, το οποίο έχει διάφορες στρώσεις και αλλιώς είναι εσωτερικά, αλλιώς είναι λίγο πιο έξω και αλλιώς είναι στο τελείως εξωτερικό του κομμάτι. Με άλλα λόγια ουσιαστικά για το κάθε παιχνίδι, πρέπει ο σχεδιαστής του να σκεφτεί ποιο είναι το πιο σημαντικό χαρακτηριστικό του παιχνιδιού, που χωρίς αυτό, δεν μπορεί να υπάρξει το παιχνίδι και να το βάλει στο κέντρο του «κρεμμυδιού». Το αμέσως επόμενο στρώμα στο onion design, πρέπει να είναι το δεύτερο πιο σημαντικό χαρακτηριστικό του παιχνιδιού. Αμέσως μετά, στο τρίτο στρώμα, εντάσσει το τρίτο πιο σημαντικό χαρακτηριστικό του παιχνιδιού και αυτό συνεχίζεται με όσα στρώματα θέλει, με το κάθε στρώμα να περικλείει το αμέσως προηγούμενο στρώμα, ακριβώς όπως φαίνεται στο παρακάτω σχήμα.

Εικόνα 3.1: *Μοντέλο του onion design*

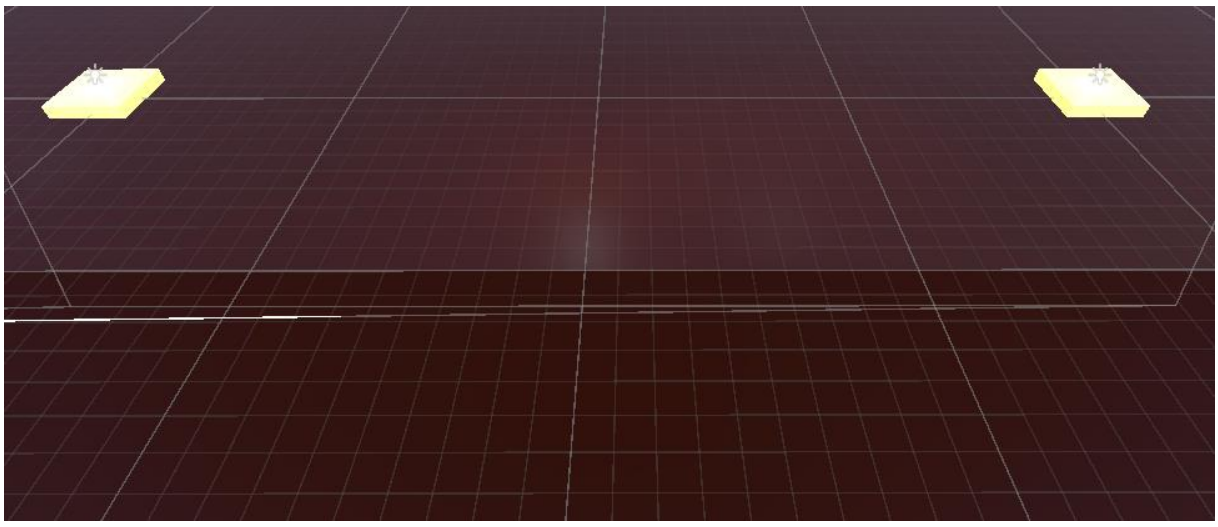
Κάθε στρώση πρέπει να τροφοδοτεί το πιο σημαντικό χαρακτηριστικό του παιχνιδιού, για να το κάνει καλύτερο. Μπορούν να μπουν πολλά χαρακτηριστικά, αλλά εκεί είναι που πρέπει ο σχεδιαστής να σκεφτεί αν όντως μπορεί αυτό το χαρακτηριστικό να κάνει καλύτερο το πιο σημαντικό χαρακτηριστικό που βρίσκεται στο κέντρο του onion design και αν εν τέλει όντως το χρειάζεται, αυτό που θέλει να εντάξει στο παιχνίδι του.

Στο συγκεκριμένο παιχνίδι, το πιο σημαντικό χαρακτηριστικό του παιχνιδιού είναι ο τρόπος κίνησης του πυραύλου, δηλαδή να πετάει. Το δεύτερο πιο σημαντικό χαρακτηριστικό είναι η σύγκρουση με το περιβάλλον, δηλαδή με τα κινούμενα ή ακίνητα εμπόδια που υπάρχουν στην πίστα και κατά συνέπεια η συντριβή του πυραύλου. Το τρίτο πιο σημαντικό και τελευταίο χαρακτηριστικό, είναι να μπορεί ο παίχτης να προχωρήσει στην επόμενη πίστα, μόλις φτάσει στο τέλος της προηγούμενης.

3.4 Κύρια στοιχεία παιχνιδιού

Η πρώτη ενέργεια είναι να δημιουργηθεί το έδαφος της πίστας και να του δοθεί ένα χρώμα. Καθώς στην αρχή ο παίχτης βρίσκεται στην Γη, επειδή κάνει δοκιμή του πυραύλου, το έδαφος της πίστας έχει χρώμα καφέ, οπότε δημιουργείται και το αντίστοιχο material.

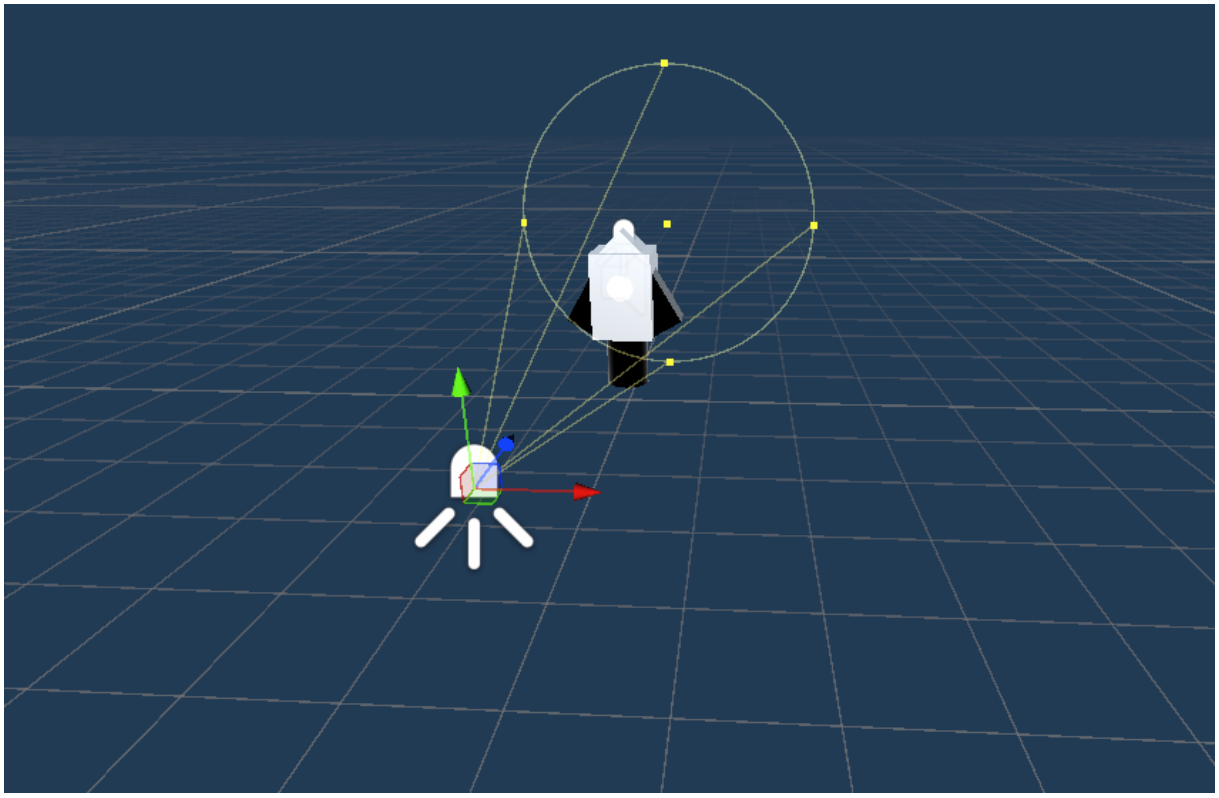
Επόμενη κίνηση, είναι να δημιουργηθούν δύο πλατφόρμες, μία για την απογείωση του πυραύλου και μία για την προσεδάφισή του, όπου η πλατφόρμα της προσεδάφισης χρησιμοποιείται για να υποδηλώσει τον τερματισμό. Πρέπει δηλαδή ο παίχτης να προσγειώσει τον πύραυλό του σε αυτή την πλατφόρμα για να μπορέσει να προχωρήσει στην επόμενη πίστα. Οι πλατφόρμες αυτές έχουν ένα material με άσπρο χρώμα και με εκπεμπόμενο κίτρινο χρώμα και προστίθεται ένα point light πάνω στην καθεμία για να διακρίνονται ακόμα πιο εύκολα.



Εικόνα 3.2: *Έδαφος Γης και πλατφόρμες απογείωσης και προσγείωσης*

Τελευταίο κύριο στοιχείο του παιχνιδιού, είναι ο πύραυλος Starseeker. Για τον πύραυλο χρησιμοποιείται ένας κύβος που του δίνονται οι κατάλληλες διαστάσεις για να γίνει ένα ορθογώνιο παραλληλεπίπεδο και αυτό είναι το σώμα του πυραύλου. Στην συνέχεια, προστίθεται ένας κύβος στραμμένος στις σαρανταπέντε μοίρες στον άξονα z και τοποθετείται πάνω από το σώμα του πυραύλου, ως η κορυφή του. Τώρα προστίθεται ένας μικρός κύλινδρος, που είναι ο κύριος προωθητήρας του πυραύλου και τοποθετείται κάτω από το ορθογώνιο παραλληλεπίπεδο που συμβολίζει το σώμα του.

Επόμενη κίνηση, είναι να δημιουργηθούν δύο ακόμα κύβοι, να μετατραπούν σε ορθογώνια παραλληλεπίπεδα και να τοποθετηθούν το ένα στην αριστερή πλευρά του πυραύλου και το άλλο στην δεξιά, στραμμένα στις μείον τριάντα και στις τριάντα μοίρες στον άξονα z, αντίστοιχα και αυτοί είναι οι πλάγιοι προωθητήρες που χρησιμοποιεί ο πύραυλος για να περιστραφεί στον άξονα z. Το τελευταίο κομμάτι, όσον αφορά το σώμα του πυραύλου, είναι τα δύο παράθυρα που προστίθενται στις δύο πλευρές που δεν υπάρχουν προωθητήρες. Το χρώμα για την κορυφή και το σώμα του πυραύλου είναι άσπρο μεταλλικό, ο κύριος προωθητήρας, όπως και οι πλάγιοι, έχουν χρώμα μαύρο και τα δύο παράθυρα, έχουν μία μίξη λευκού και κίτρινου χρώματος. Τέλος, προστίθεται ένα spot light που φωτίζει από την κορυφή του πυραύλου και ένα ακόμα spot light που φωτίζει πάνω στον πύραυλο, για να ξεχωρίζει αρκετά σε σχέση με τα υπόλοιπα game objects της πίστας.



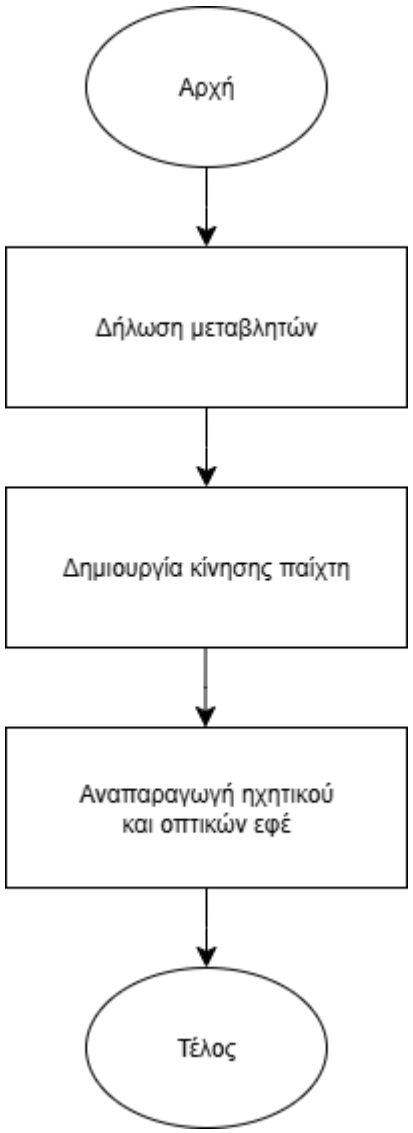
Εικόνα 3.3: *Πύραυλος Starseeker*

3.5 Χειρισμός Starseeker

Για τον χειρισμό του πυραύλου δημιουργείται ένα script και εντάσσεται στα components του πυραύλου. Όπως αναφέρθηκε και προηγουμένως, ο παίχτης χρησιμοποιεί το κουμπί Space για ώθηση προς τα πάνω και τα κουμπιά A και D, για να μπορεί να περιστρέψει τον πύραυλο, προς τα αριστερά και προς τα δεξιά, αντίστοιχα. Για να πετύχει η κίνηση αυτή, χρειάζεται η φυσική που υπάρχει μέσα στην Unity και η βαρύτητα, έτσι ώστε να μπορεί να πέφτει προς τα κάτω ο πύραυλος. Για την βαρύτητα σίγουρα χρειάζεται ένα rigidbody, καθώς αυτό είναι υπεύθυνο για την φυσική των game objects και γι’ αυτόν τον λόγο προστίθεται στον πύραυλο.

Για να μπορέσει ο πύραυλος να αποκτήσει ώθηση, όταν κρατάει ο παίχτης πατημένο το κουμπί Space, πρέπει να ασκηθεί μία δύναμη σχετική με την φορά του πυραύλου, καθώς η ίδια δύναμη πρέπει να ασκείται με τον ίδιο τρόπο, είτε ο πύραυλος κοιτά προς τα πάνω/κάτω, είτε προς τα δεξιά/αριστερά.

Αυτή η δύναμη ασκείται μέσω της μεθόδου AddRelativeForce(). Για να μπορέσει να δημιουργηθεί επιτυχώς το συγκεκριμένο script, ακολούθησε η λογική που φαίνεται στο Σχήμα 3.1.



Σχήμα 3.1: *Λογική για το script Movement*

Αυτό που πρέπει να γίνει στην συνέχεια, είναι να δημιουργηθεί μέσω της Unity, το οπτικό εφέ για τους προωθητήρες και να προστεθεί το ηχητικό εφέ για τον ήχο του κύριου προωθητήρα. [40]

3.6 Αφετηρία, τερματισμός και σύγκρουση

Αφού δημιουργηθούν οι πλατφόρμες για την αφετηρία και για τον τερματισμό, τώρα πρέπει να δημιουργηθεί ένα script που να είναι υπεύθυνο για το πως διαχειρίζεται το παιχνίδι την επαφή με κάποια game objects και να προστεθεί και αυτό στα components του πυραύλου. Αν για παράδειγμα, ο πύραυλος ακουμπήσει κάποιο εμπόδιο που υπάρχει μέσα στην πίστα, ο πύραυλος καταστρέφεται, ενώ αν ακουμπήσει κάποια από τις πλατφόρμες δεν χάνει και ειδικά αν ακουμπήσει την πλατφόρμα του τερματισμού, κερδίζει την συγκεκριμένη πίστα και προχωράει στην επόμενη. Για να επιτευχθεί αυτό,

πρέπει να διαχωριστεί τι είναι εμπόδιο και τι όχι. Γι' αυτόν τον λόγο πρέπει να αλλαχτεί το tag της πλατφόρμας της αφετηρίας και του τερματισμού, σε «Friendly» και «Finish», αντίστοιχα. Οτιδήποτε άλλο υπάρχει μέσα στην πίστα είναι εμπόδιο και καταστρέφει τον πύραυλο, αν αυτός συγκρουστεί με αυτό. Μέσα στο ίδιο script μπαίνουν οι κατάλληλες εντολές για να φορτώνει η Unity την ίδια πίστα όταν ο παίχτης χάνει και για να φορτώνει η Unity την αμέσως επόμενη πίστα όταν ο παίχτης κερδίζει. Επιπλέον, κάτι ακόμα που θεωρείται βασικό για το συγκεκριμένο παιχνίδι και που πρέπει να ενταχθεί επίσης σε αυτό το script, είναι να παίζουν τα οπτικά εφέ, για όταν συγκρούεται ο πύραυλος με κάποιο εμπόδιο και για όταν προσγειώνεται ο πύραυλος στην πλατφόρμα τερματισμού και ουσιαστικά ο παίχτης περνάει επιτυχώς την παρούσα πίστα. Αυτά τα δύο οπτικά εφέ δημιουργούνται πάλι μέσω της Unity και στην συνέχεια προστίθενται οι κατάλληλες εντολές γι' αυτά μέσα στο script. Μαζί με τα οπτικά εφέ, εντάσσονται και τα ηχητικά εφέ για την επιτυχή προσγείωση στην πλατφόρμα τερματισμού [41] και για την σύγκρουση με το περιβάλλον. [42] Τρία τελευταία πράγματα πρέπει να προστεθούν στο συγκεκριμένο script, καθώς σχετίζονται το ένα με την φόρτωση της επόμενης πίστας, το άλλο με την απενεργοποίηση της σύγκρουσης του παίχτη με το περιβάλλον και το τελευταίο με την επαναφορά του πυραύλου, σε περίπτωση που αυτός, λόγω της απενεργοποιημένης σύγκρουσης με το περιβάλλον, βρεθεί σε θέση, όπου δεν μπορεί με κανέναν τρόπο να φτάσει στην πλατφόρμα τερματισμού. Αυτά τα τρία πράγματα είναι τρία κουμπιά, συγκεκριμένα το κουμπί L (Load next level), το κουμπί C (Collision disabled) και το κουμπί R (Reset rocket Starseeker), που χρησιμοποιούνται για να κάνουν το παιχνίδι πιο εύκολο, σε άτομα που το θεωρούν δύσκολο. Το κουμπί L, όταν πατηθεί από τον παίχτη, φορτώνει αυτόματα την επόμενη πίστα. Το κουμπί C, όταν πατηθεί από τον παίχτη, καταργεί την σύγκρουση του παίχτη με το περιβάλλον και μπορεί αυτός ανενόχλητος να συγκρουστεί με ό,τι θέλει και εν συνεχεία να φτάσει στο τέλος της πίστας και να κερδίσει (ο παίχτης όποτε θέλει κατά την διάρκεια της πίστας, αν έχει περάσει ένα δύσκολο εμπόδιο, έχοντας πατήσει το κουμπί C προηγουμένως, μπορεί να ξαναπατήσει το κουμπί C και να ενεργοποιήσει ξανά την σύγκρουση με τα εμπόδια). Το κουμπί R, όταν πατηθεί από τον παίχτη, επαναφέρει τον πύραυλο στην αρχή της πίστας, επαναφορτώνοντας την παρούσα πίστα. Το script ακολουθάει την λογική που υπάρχει στο Σχήμα 3.2, με σκοπό να δημιουργηθεί επιτυχώς.

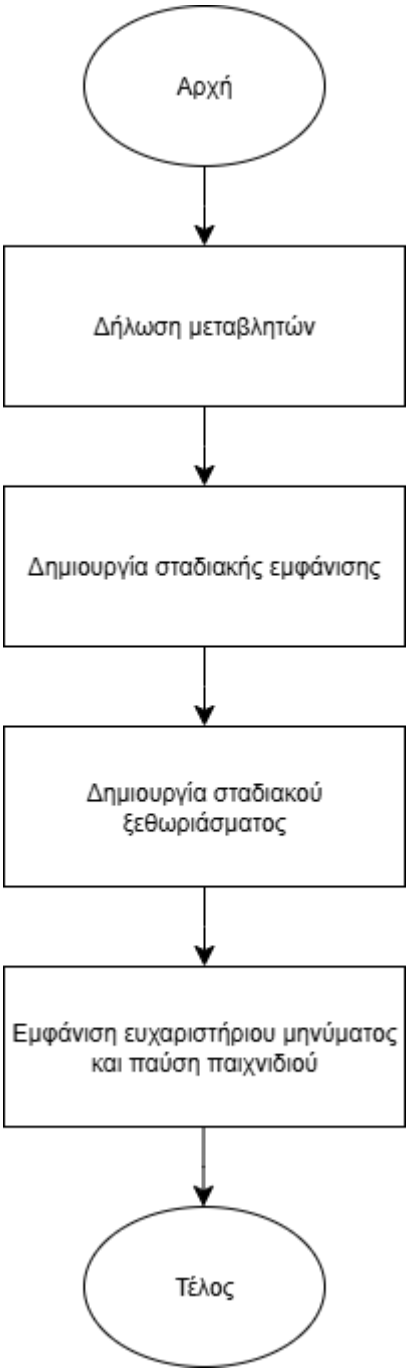
Αρχή
Δήλωση μεταβλητών
Δημιουργία σύγκρουσης του παίχτη με το περιβάλλον
Φόρτωση του επόμενου επιπέδου ή επαναφόρτωση του τωρινού επιπέδου
Αναπαραγωγή ηχητικών και οπτικών εφέ
Δημιουργία σκοπού των κουμπιών L, C και R
Απενεργοποίηση των κουμπιών L, C και R, όταν ο παίχτης βλέπει το ευχαριστήριο μήνυμα με το πέρας του παιχνιδιού
Τέλος

Σχήμα 3.2: *Λογική για το script CollisionHandler*

Επειδή πρέπει όμως να κερδίζει ο παίχτης μόνο όταν ακουμπάει ο κύριος προωθητήρας του Starseeker στην πλατφόρμα τερματισμού, χρειάζεται ένα ακόμα script, λειτουργώντας συνεργατικά με το παραπάνω script. Το συγκεκριμένο script εντάσσεται στα components του κύριου προωθητήρα του πυραύλου. Ο παίχτης χάνει αν ακουμπήσει με οποιοδήποτε άλλο μέρος του πυραύλου στην πλατφόρμα τερματισμού και για να τερματίσει επιτυχώς την κάθε πίστα, πρέπει να ακουμπήσει μόνο με τον κύριο προωθητήρα. Ουσιαστικά αυτό το script ελέγχει αν ο πύραυλος ακούμπησε όντως με τον κύριο προωθητήρα και η λογική που ακολουθείται μέχρι την υλοποίησή του, φαίνεται στο Σχήμα 3.3.

Σχήμα 3.3: *Λογική για το script ThrusterCollisionDetector*

Κάτι ακόμα που είναι όμορφο και είναι καλό να προστεθεί στο παιχνίδι, είναι να υπάρχει σταδιακή εμφάνιση της κάθε πίστας ή όταν επαναφορτώνεται η ίδια πίστα, να υπάρχει μία σταδιακή εξασθένιση της πίστας όταν χάνει ο παίχτης. Αυτό γίνεται με ένα script που προστίθεται στα components του game object με το όνομα «Fade Level Panel» και για να δημιουργηθεί ακολουθάει την λογική που βρίσκεται στο Σχήμα 3.4.

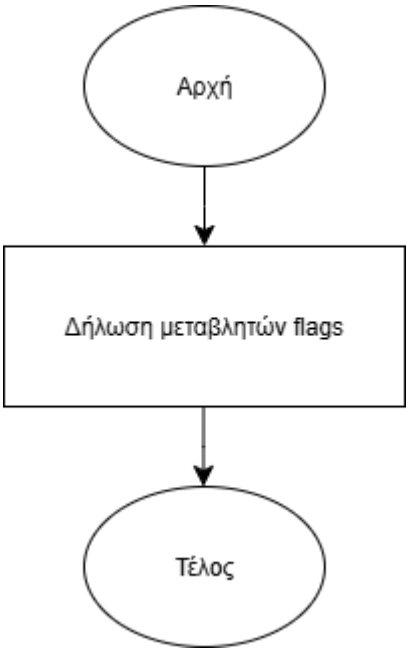


Σχήμα 3.4: *Λογική για το script ScreenFader*

Μαζί με το προηγούμενο script δημιουργείται και ένα νέο script που εντάσσεται στα components ενός panel με το όνομα «Thank You Message Panel». Το συγκεκριμένο script είναι υπεύθυνο για την εμφάνιση ενός ευχαριστήριου μηνύματος στον παίχτη, όταν αυτός κερδίζει και την τελευταία πίστα αυτού του demo και ξαναεμφανίζεται στην πρώτη πίστα. Το script αυτό για να δημιουργηθεί ακολουθάει την λογική που βρίσκεται στο Σχήμα 3.5.

Σχήμα 3.5: *Λογική για το script ThankYouMessage*

Για να λειτουργήσουν όλα αυτά, χρειάζεται και η δημιουργία ενός βοηθητικού script που όμως δεν εντάσσεται σε κάποιο game object. Αυτό το script ελέγχει αν ο παίχτης έφτασε στην πρώτη πίστα, αφού πρώτα τερμάτισε επιτυχώς την τελευταία πίστα. Κάνοντας αυτόν τον έλεγχο, μπορεί να εμφανιστεί το ευχαριστήριο μήνυμα, μόνο αν ο παίχτης έφτασε στην πρώτη πίστα, αμέσως μετά την τελευταία και όχι αν βρίσκεται ακόμα στην πρώτη πίστα. Η λογική που ακολουθείται για να δημιουργηθεί το script παρατίθεται στο Σχήμα 3.6.



Σχήμα 3.6: *Λογική για το script GameState*

3.7 Σχεδιασμός πιστών

3.7.1 Σχεδιασμός πίστας της Γης (Earth's level design)

Η πρώτη πίστα που δημιουργείται αφορά την Γη, καθώς στην Γη αρχικά ο πύραυλος Starseeker δοκιμάζει αν μπορεί να ανταπεξέλθει στην αποφυγή εμποδίων και στην αντιμετώπιση διαφόρων καταστάσεων. Αφού προστεθούν όλα τα βασικά στοιχεία του παιχνιδιού, ήρθε η ώρα για τον σχεδιασμό της πίστας (level design). Ακολουθείται η ίδια λογική ακριβώς, που ακολούθησε για την δημιουργία του πυραύλου. Όλα τα εμπόδια, όπως αναφέρθηκε και προηγουμένως, έχουν την μορφή βασικών game objects της Unity, όπως ο κύβος και η σφαίρα, τα οποία τροποποιούνται πολλές φορές για να τους δοθεί το επιθυμητό σχήμα.

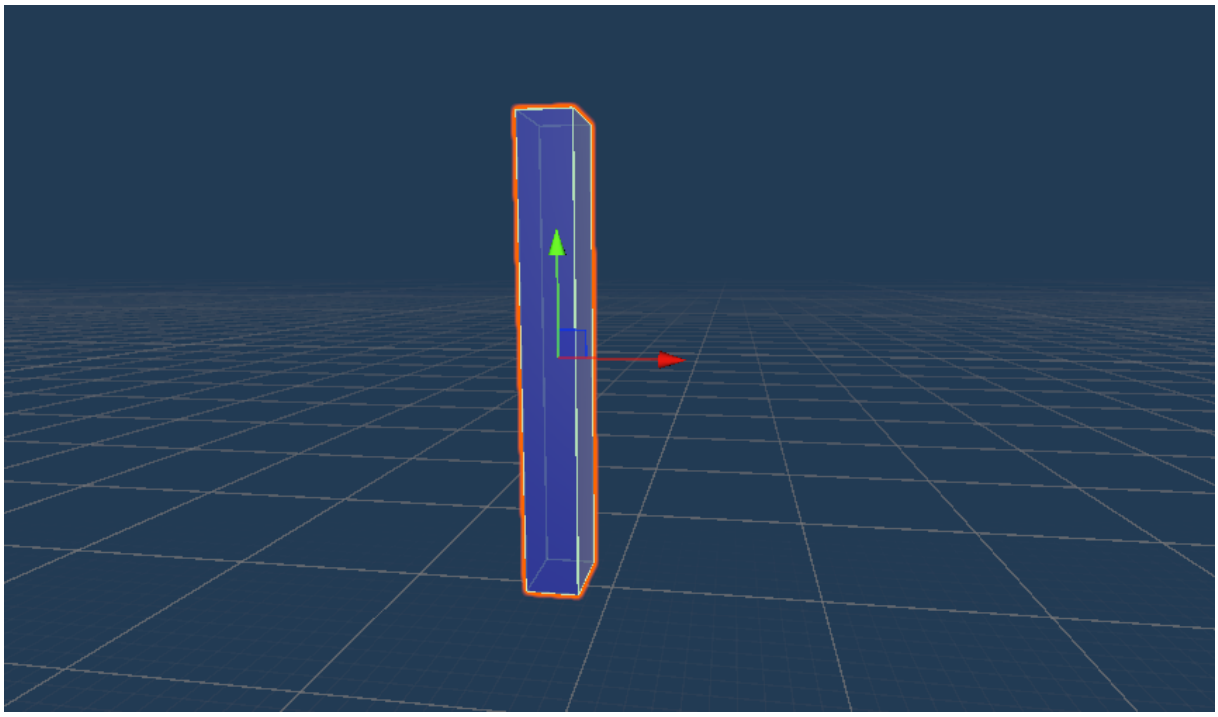
Τώρα αυτό που πρέπει να γίνει, είναι να δημιουργηθούν πολλά ορθογώνια παραλληλεπίπεδα που περιβάλλουν την πίστα και βρίσκονται και κατά μήκος της διαδρομής της πίστας. Αυτά τα εμπόδια δίνουν την αίσθηση ότι εκτός από εμπόδια, είναι και βράχοι, έδαφος, αλλά και γενικότερα το περιβάλλον γύρω από τον πύραυλο.

Επιπλέον, αυτό που έχει σειρά, είναι να δημιουργηθεί μία κάμερα που ακολουθάει τον πύραυλο σε κάθε του κίνηση.

Το πρώτο εμπόδιο που προστίθεται στην συγκεκριμένη πίστα, είναι ένα ορθογώνιο παραλληλεπίπεδο που μόνιμα περιστρέφεται, όντας σε σταθερή θέση. Περιστρέφεται βέβαια με διαφορετική ταχύτητα σε κάθε άξονα, κάνοντάς το λίγο απρόβλεπτο. Γι' αυτό το εμπόδιο χρειάζεται ένα script που το κάνει να περιστρέφεται με διαφορετική ταχύτητα στους άξονες x, y και z, όντας όμως σε ένα σταθερό σημείο στην πίστα. Αυτό το script εντάσσεται στο συγκεκριμένο εμπόδιο και για να δημιουργηθεί ακολουθεί την λογική που υπάρχει στο Σχήμα 3.7. Σε όλα αυτά τα εμπόδια τους δίνονται τρία χρώματα μόνο και αυτά είναι το καφέ, το μπλε και το πράσινο, καθώς αυτά είναι τα βασικά χρώματα που κυρίως βλέπει κάποιος στην Γη.

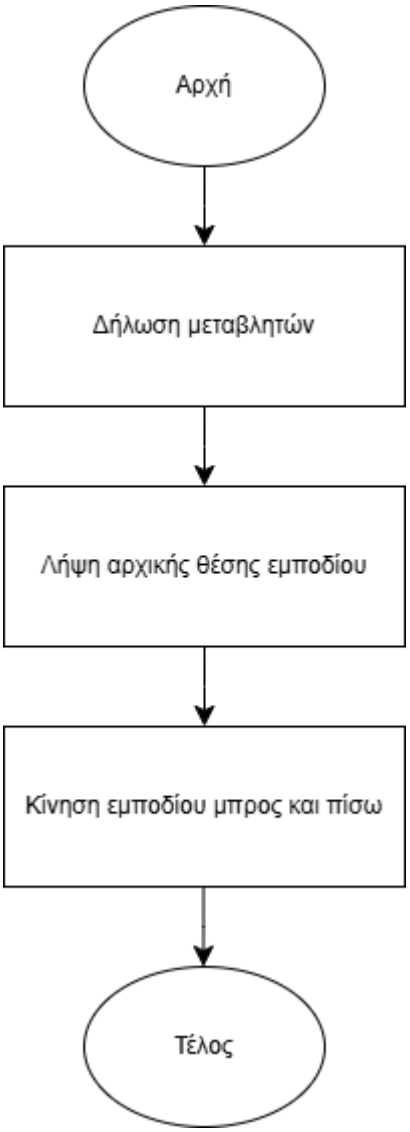
Σχήμα 3.7: *Λογική για το script ObstacleRotation*

Τώρα προστίθεται ένα ίδιο εμπόδιο, αλλά είναι διαφορετικού μεγέθους και αλλάζουν οι τιμές των μεταβλητών που υπάρχουν στο script, έτσι ώστε να περιστρέφεται με διαφορετική ταχύτητα σε κάθε άξονα.

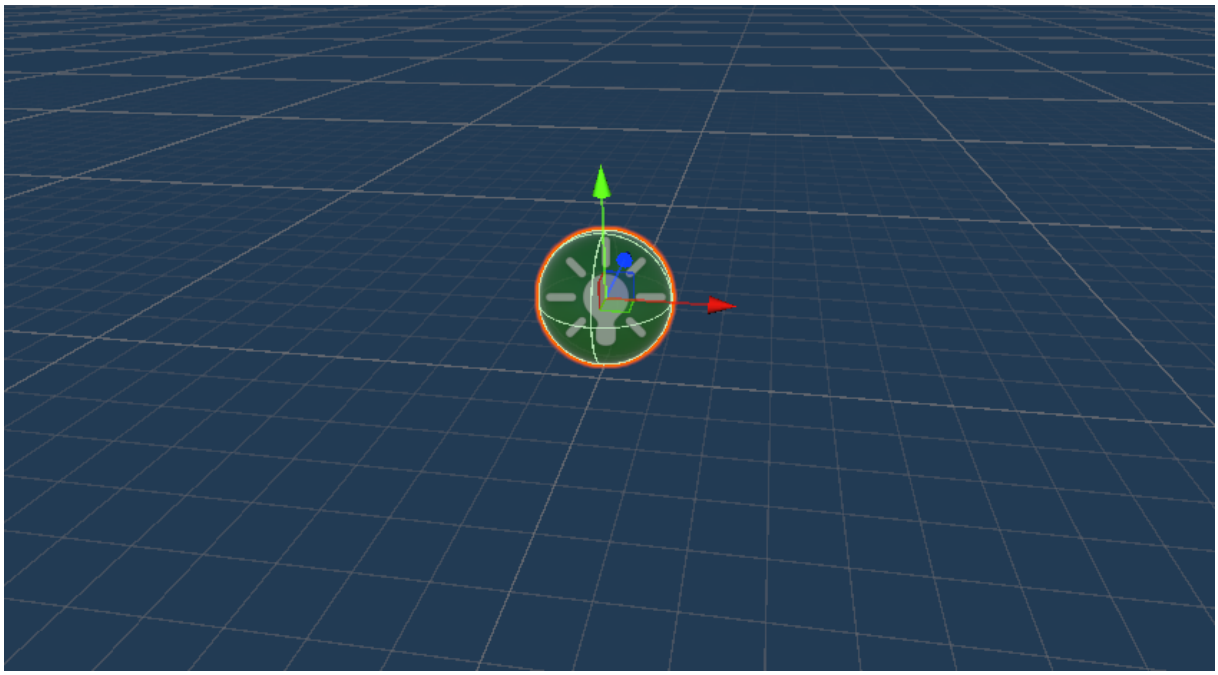


Εικόνα 3.4: *Rotating Obstacle*

Το τελευταίο εμπόδιο που προστίθεται στην παρούσα πίστα, είναι μία σφαίρα η οποία κάνει μία περιοδική κίνηση, πηγαίνοντας δεξιά και αριστερά και μετά πάλι από την αρχή, κινούμενη στον άξονα x. Η σφαίρα αυτή έχει τοποθετημένο πάνω της ένα point light. Γι' αυτό το εμπόδιο, χρειάζεται ένα script που προστίθεται πάνω στην σφαίρα και της επιτρέπει να κινηθεί με αυτόν τον τρόπο. Η λογική που ακολουθείται για να δημιουργηθεί επιτυχώς αυτό το script, παρατίθεται στο Σχήμα 3.8.



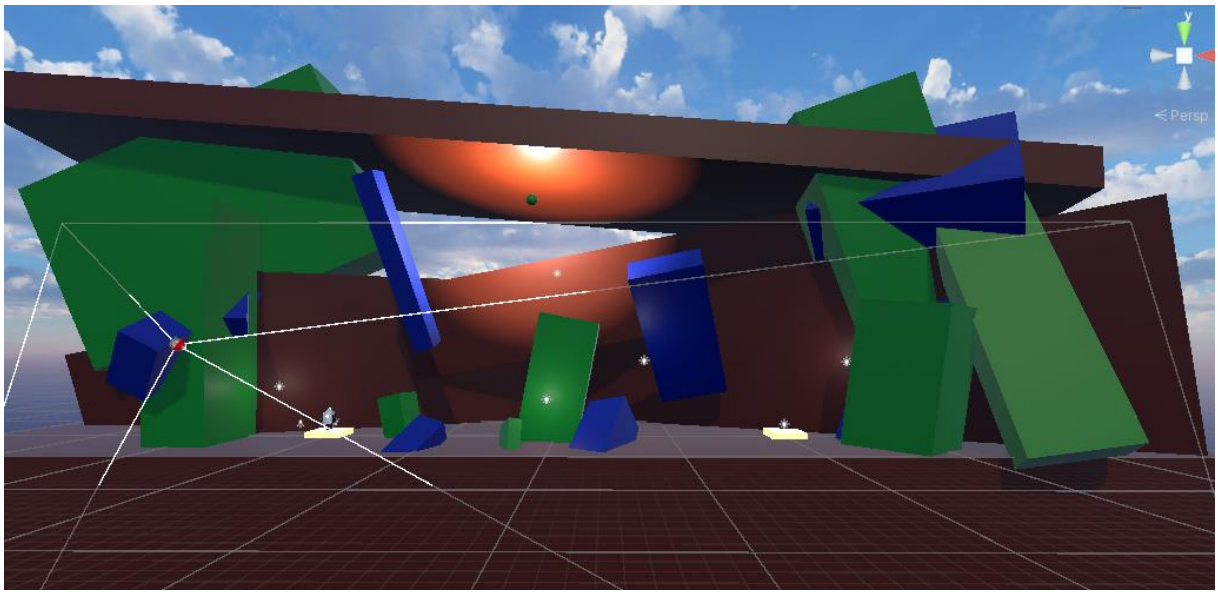
Σχήμα 3.8: *Λογική για το script ObstacleOscillation*



Εικόνα 3.5: *Back And Forth Obstacle*

Πέρα από αυτά προστίθεται και ο κατάλληλος φωτισμός σε συγκριμένα σημεία, μέσω διαφόρων point lights.

Η τελευταία ενέργεια είναι να τοποθετηθεί ένα παρασκήνιο πίσω από την πίστα, που δείχνει τον γαλάζιο ουρανό με τα σύννεφα. [43]

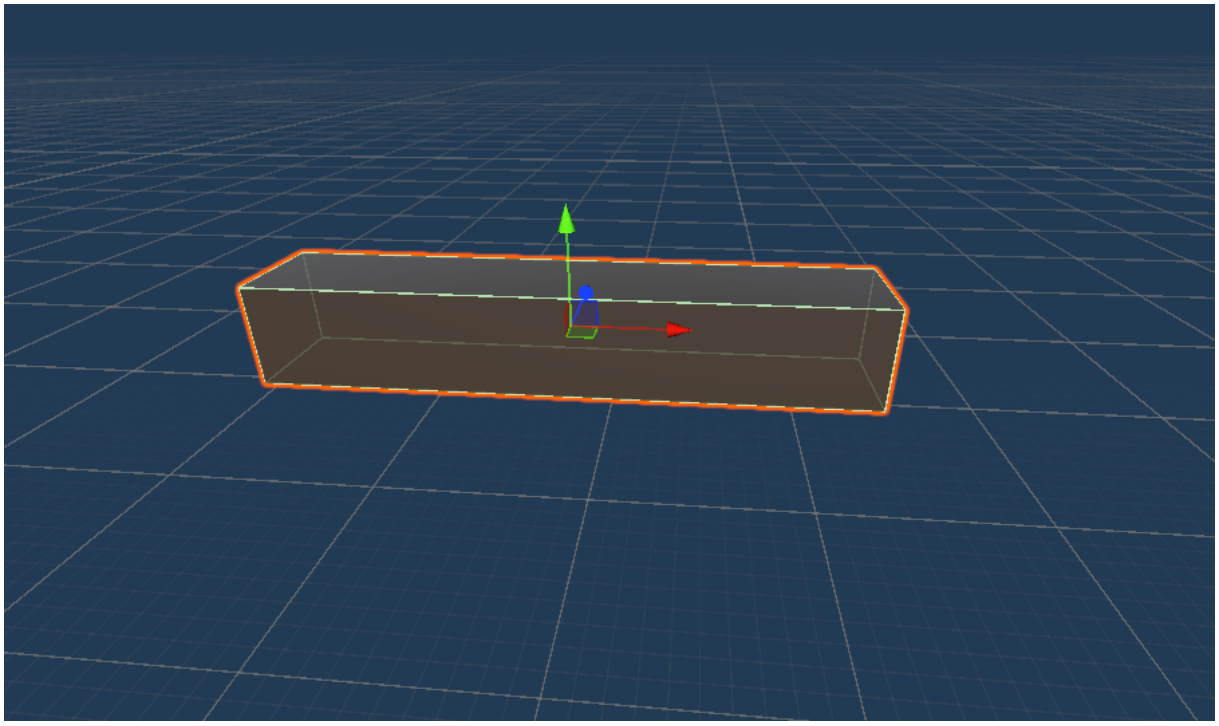


Εικόνα 3.6: *Πίστα της Γης*

3.7.2 Σχεδιασμός πίστας του Ερμή (Mercury's level design)

Εφόσον προηγουμένως δημιουργήθηκαν όλα όσα χρειάζονται, σειρά έχει να δημιουργηθεί μία νέα σκηνή και να τοποθετηθούν τα εμπόδια της πίστας. Γι' αυτόν τον λόγο λοιπόν, προστίθενται διαφόρων μεγεθών ακίνητα εμπόδια με σχήμα ορθογώνιου παραλληλεπίπεδου.

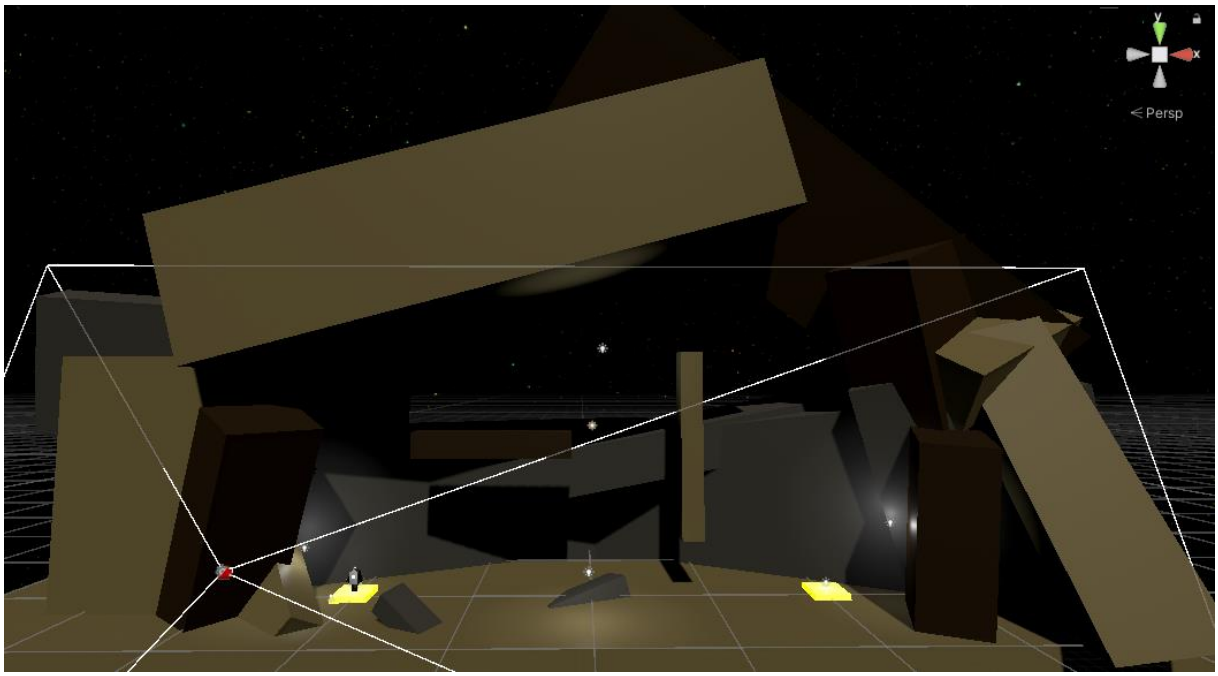
Τα δύο κινούμενα εμπόδια που δημιουργούνται, είναι του ίδιου είδους ακολουθώντας ακριβώς την ίδια λογική, αλλά η συμπεριφορά τους διαφέρει, καθώς έχουν άλλες τιμές το κάθε ένα σε κάθε μία μεταβλητή που επηρεάζει την συμπεριφορά τους. Αυτά τα δύο εμπόδια είναι μία μίξη της κινούμενης σφαίρας και των περιστρεφόμενων εμποδίων που αντιμετώπισε ο παίχτης κατά την δοκιμή του πυραύλου στην Γη. Με άλλα λόγια περιστρέφονται στον άξονα z και κάνουν το ένα μία μικρή περιοδική κίνηση στον άξονα x και το άλλο μία μικρή περιοδική κίνηση στον άξονα x, αλλά και στον y, καθιστώντας αρκετά δύσκολη την διέλευση του πυραύλου από αυτά. Αυτό που πρέπει να γίνει, είναι να δημιουργηθούν δύο ορθογώνια παραλληλεπίπεδα και να τους προστεθούν το ObstacleRotation script και το ObstacleOscillator script, που δημιουργήθηκαν νωρίτερα. Μετά απλά πρέπει να αλλάξουν οι τιμές των μεταβλητών για την συμπεριφορά των εμποδίων σε κάθε άξονα. Σε όλα τα εμπόδια δίνονται τρία συγκεκριμένα χρώματα, τα οποία είναι γκρι, σκούρο καφέ και ανοιχτό μπεζ.



Εικόνα 3.7: *Rotating Oscillator Obstacle*

Επίσης, προστίθενται διάφορα point lights σε συγκριμένα σημεία.

Η τελευταία κίνηση, είναι να προστεθεί ένα παρασκήνιο με έναν σκοτεινό ουρανό με άστρα. [44]

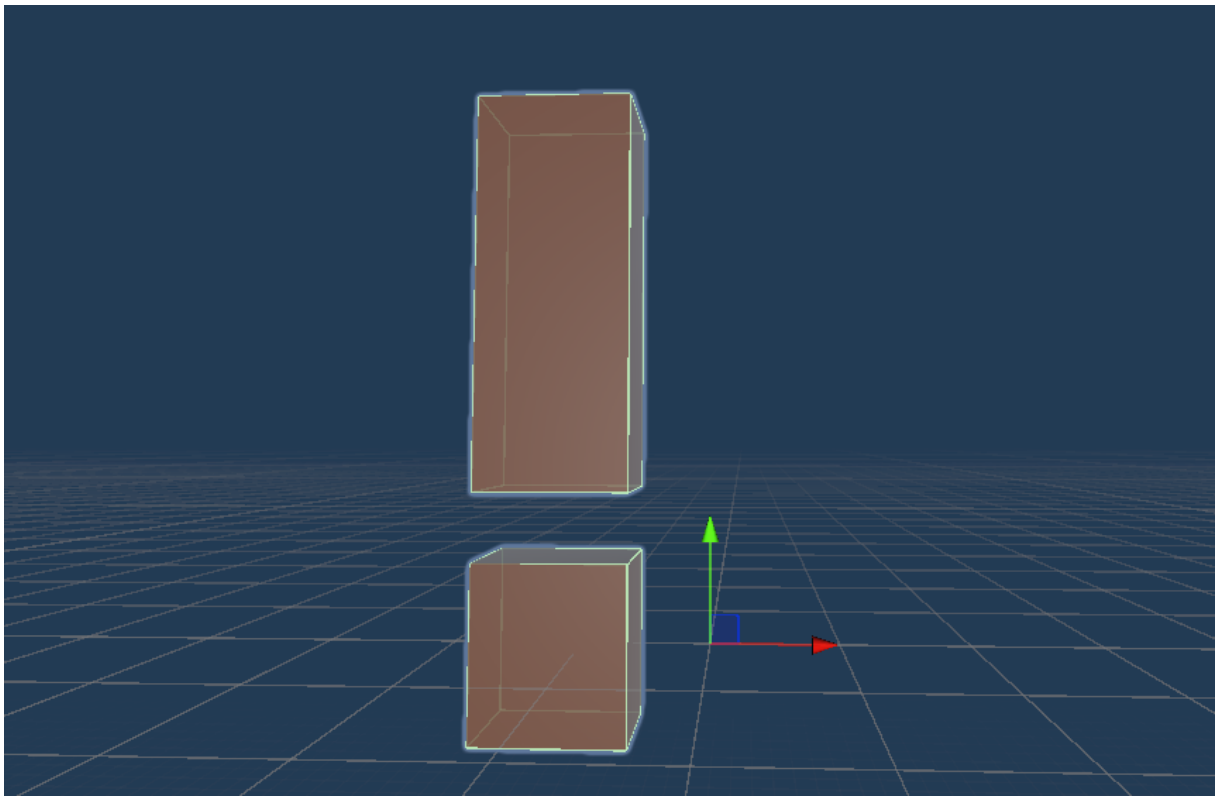


Εικόνα 3.8: *Πίστα του Ερμή*

3.7.3 Σχεδιασμός πίστας της Αφροδίτης (Venus' level design)

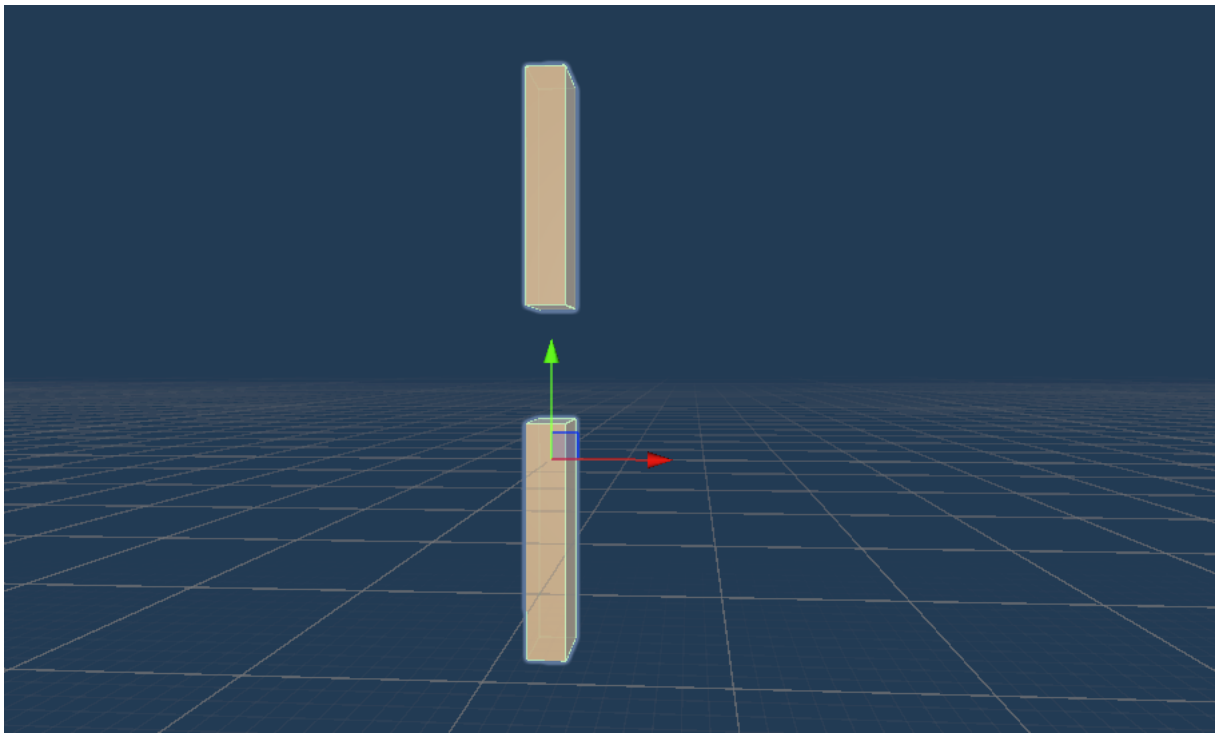
Παρόλο που τα περισσότερα που χρειάζονται γι' αυτή την πίστα είναι έτοιμα, χρειάζεται ακριβώς όπως και πριν, να δημιουργηθεί μία ακόμα σκηνή και να προστεθούν διάφορων μεγεθών ακίνητα εμπόδια με σχήμα ορθογώνιου παραλληλεπίπεδου.

Αυτή την φορά, εντάσσονται τρία κινούμενα εμπόδια στην συγκεκριμένη πίστα. Το πρώτο εμπόδιο είναι δύο μακρόστενα κάθετα ορθογώνια παραλληλεπίπεδα, τα οποία έχουν ένα μικρό πέρασμα ανάμεσά τους, από το οποίο πρέπει να περάσει ο πύραυλος. Αρχικά τους προστίθεται το ObstacleOscillator script. Πρέπει και οι δύο να κάνουν ακριβώς την ίδια κίνηση πάνω κάτω με τον ίδιο ρυθμό και ουσιαστικά το κενό που υπάρχει ανάμεσά τους, να παραμένει ίδιο και να «ανεβοκατεβαίνει» και αυτό μαζί τους. Για να υπάρξει αυτή η συμπεριφορά, πρέπει να αλλάξουν οι τιμές των μεταβλητών του script. Σε όλα τα εμπόδια δίνονται τρία συγκεκριμένα χρώματα, τα οποία είναι σκούρο καφέ, ανοιχτό καφέ και ένα ακόμα πιο σκούρο καφέ από το πρώτο.



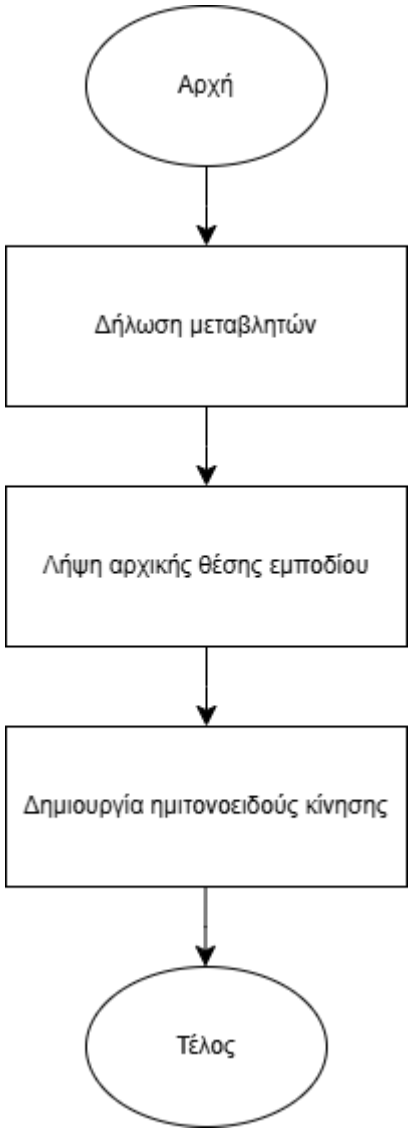
Εικόνα 3.9: *Gap Obstacle*

Το αμέσως επόμενο εμπόδιο, είναι παρόμοιο με το προηγούμενο, καθώς υπάρχουν πάλι δύο μακρόστενα κάθετα ορθογώνια παραλληλεπίπεδα, αλλά αυτή την φορά, με την βοήθεια του ObstacleOscillator script, που προστίθεται πάνω τους, τα δύο ορθογώνια παραλληλεπίπεδα, κάνουν μία κίνηση, σαν αυτή των Συμπληγάδων Πετρών. Με άλλα λόγια κάνουν μία αντίθετη μεταξύ τους κίνηση, κλείνοντας το πέρασμα που υπάρχει ανάμεσά τους και στην συνέχεια ανοίγοντάς το. Αυτή είναι μία περιοδική κίνηση, όπως και πριν, και πρέπει να αλλάξουν οι τιμές των μεταβλητών του script για να υπάρξει το επιθυμητό αποτέλεσμα.

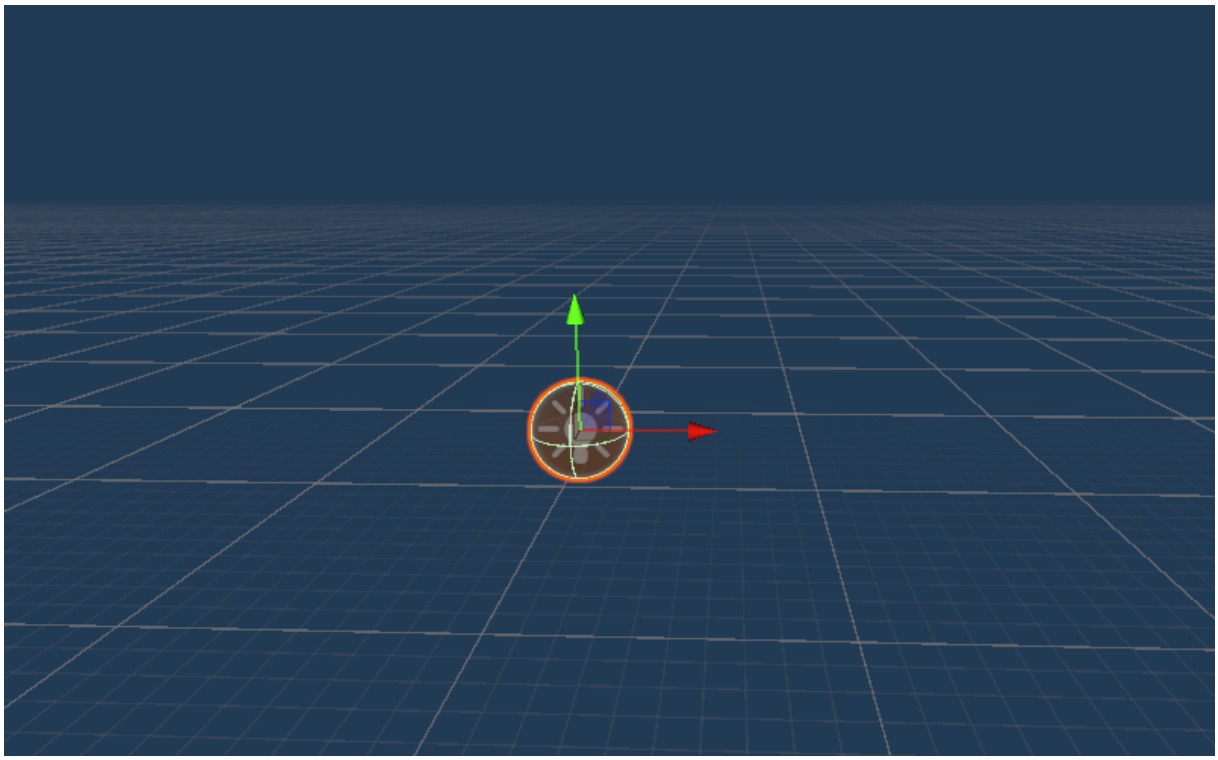


Εικόνα 3.10: *Up And Down Obstacle*

Το τρίτο και τελευταίο κινούμενο εμπόδιο, είναι μία σφαίρα, η οποία κάνει μία ημιτονοειδή κίνηση. Στη σφαίρα πάνω τοποθετείται ένα point light. Για την επίτευξη της επιθυμητής συμπεριφοράς της σφαίρας, δημιουργείται ένα script μέσω της λογικής που παρουσιάζεται στο Σχήμα 3.9 και αυτό το script εντάσσεται στην σφαίρα.



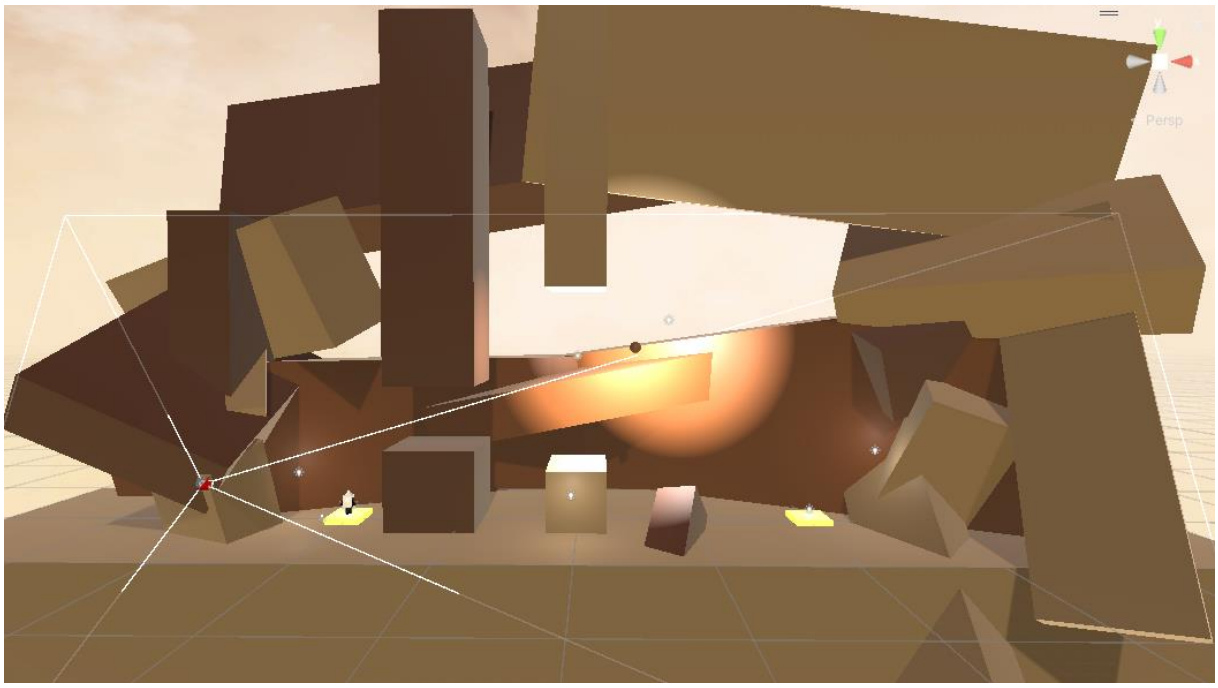
Σχήμα 3.9: *Λογική για το script SinusoidalWaveMovement*



Εικόνα 3.11: *Sine Obstacle*

Πέρα από αυτά, προστίθενται και point lights σε συγκριμένα σημεία.

Η τελευταία κίνηση, για την συγκεκριμένη πίστα, είναι να προστεθεί ένα παρασκήνιο με ομιχλώδη και συννεφιασμένο ουρανό. [45]



Εικόνα 3.12: *Πίστα της Αφροδίτης*

3.7.4 Σχεδιασμός πίστας του Άρη (Mars' level design)

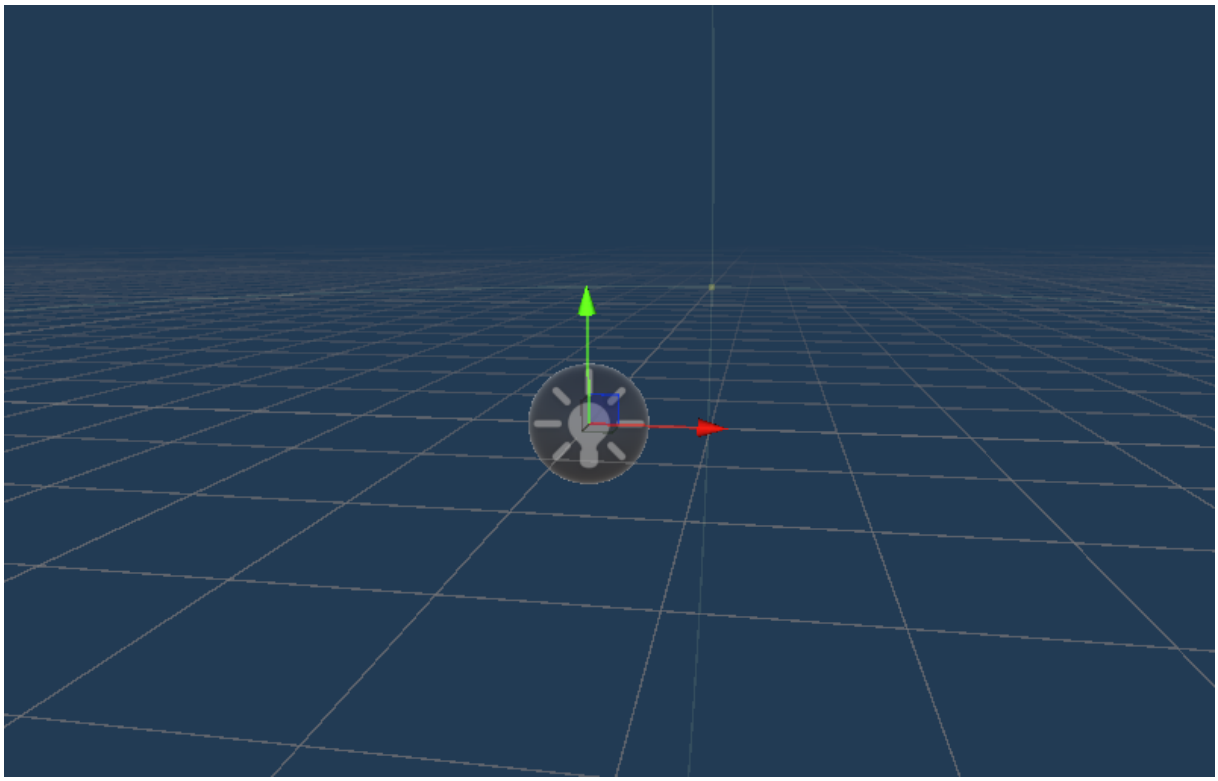
Η πίστα στον Άρη, είναι η τελευταία αυτού του παιχνιδιού, καθώς ο παίχτης, εφόσον ολοκληρώσει επιτυχώς την πίστα αυτή, επιστρέφει στην Γη, έχοντας ολοκληρώσει το ταξίδι του στους τρεις πλανήτες που έπρεπε να επισκεφτεί.

Το πρώτο που πρέπει να γίνει, είναι να δημιουργηθεί η τελευταία σκηνή του παιχνιδιού και να τοποθετηθούν ξανά ακίνητα εμπόδια κατά μήκος όλης της πίστας.

Αυτή την φορά, τα συνολικά κινούμενα εμπόδια είναι πέντε και είναι όλα του ίδιου είδους. Αυτό το είδος εμποδίου είναι μία σφαίρα, η οποία εμφανίζεται τυχαία μεταξύ ενός εύρους τιμών στον άξονα x και στον άξονα y. Ο τρόπος με τον οποίο εμφανίζεται, είναι μεγαλώνοντας, μέχρι το κανονικό της μέγεθος που είναι το x=1, y=1 και z=1. Αφού φτάσει στο μέγιστο μέγεθος, μετά από λίγο έχει την ακριβώς αντίθετη συμπεριφορά, καθώς συρρικνώνεται μέχρι να εξαφανιστεί. Στην συνέχεια, εξακολουθεί να έχει την ίδια συμπεριφορά, καθώς μόνιμα εμφανίζεται σε κάποιο τυχαίο σημείο, που είναι μέσα στο εύρος της, μεγαλώνει μέχρι το μέγιστο μέγεθος που μπορεί να πάρει και μετά από λίγο συρρικνώνεται μέχρι να εξαφανιστεί. Όσο διαρκεί η πίστα, η σφαίρα συνεχίζει να έχει αυτή την συμπεριφορά. Επίσης, η σφαίρα αυτή έχει τοποθετημένο πάνω της, ένα point light. Για να λειτουργήσει όμως αυτό το εμπόδιο, καθ' αυτόν τον τρόπο, πρέπει να δημιουργηθεί ένα script, ακολουθώντας την λογική που εμφανίζεται στο Σχήμα 3.10. Το script αυτό εντάσσεται στην σφαίρα της συγκεκριμένης πίστας. Όλα τα εμπόδια έχουν τρία συγκεκριμένα χρώματα, τα οποία είναι κόκκινο, πορτοκαλί και γκρι.

Αρχή
Δημιουργία λίστας όλων των ενεργών εμποδίων
Δήλωση μεταβλητών
Αρχή κύκλου τηλεμεταφοράς
Λήψη έγκυρης θέσης για το καινούριο εμπόδιο που είναι έτοιμο να εμφανιστεί
Συρρίκνωση εμποδίου
Μεγέθυνση εμποδίου
Τέλος

Σχήμα 3.10: *Λογική για το script ObstacleTeleportation*

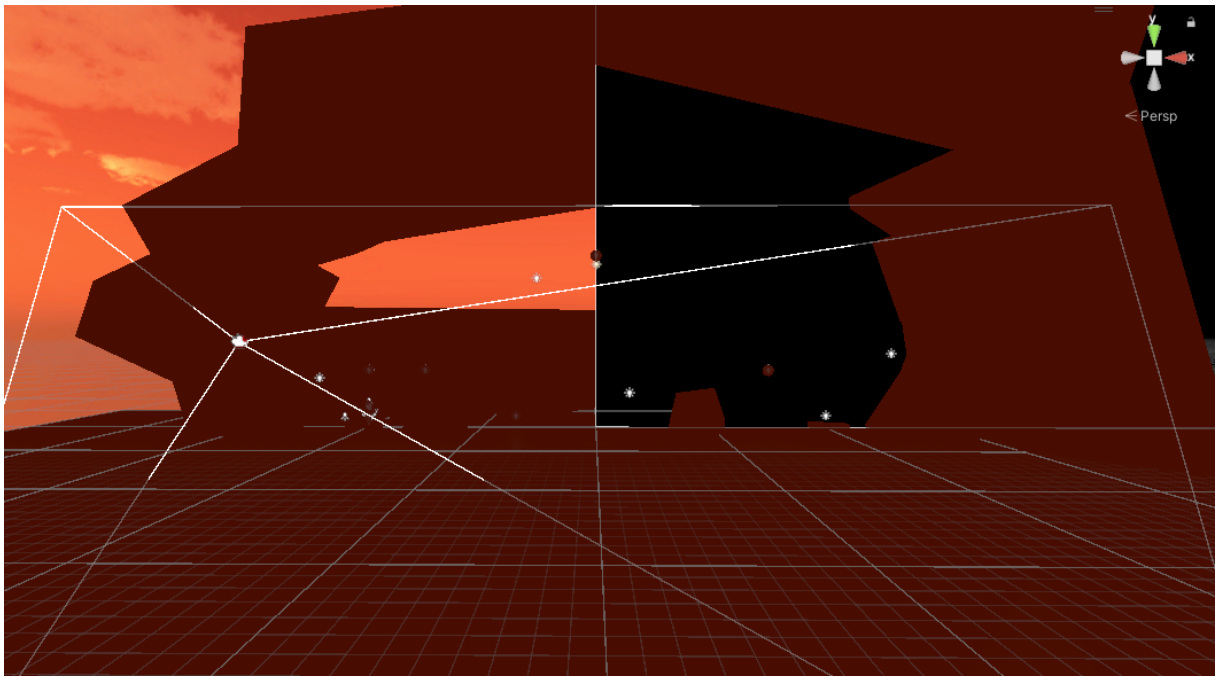


Εικόνα 3.13: *Teleporting Obstacle*

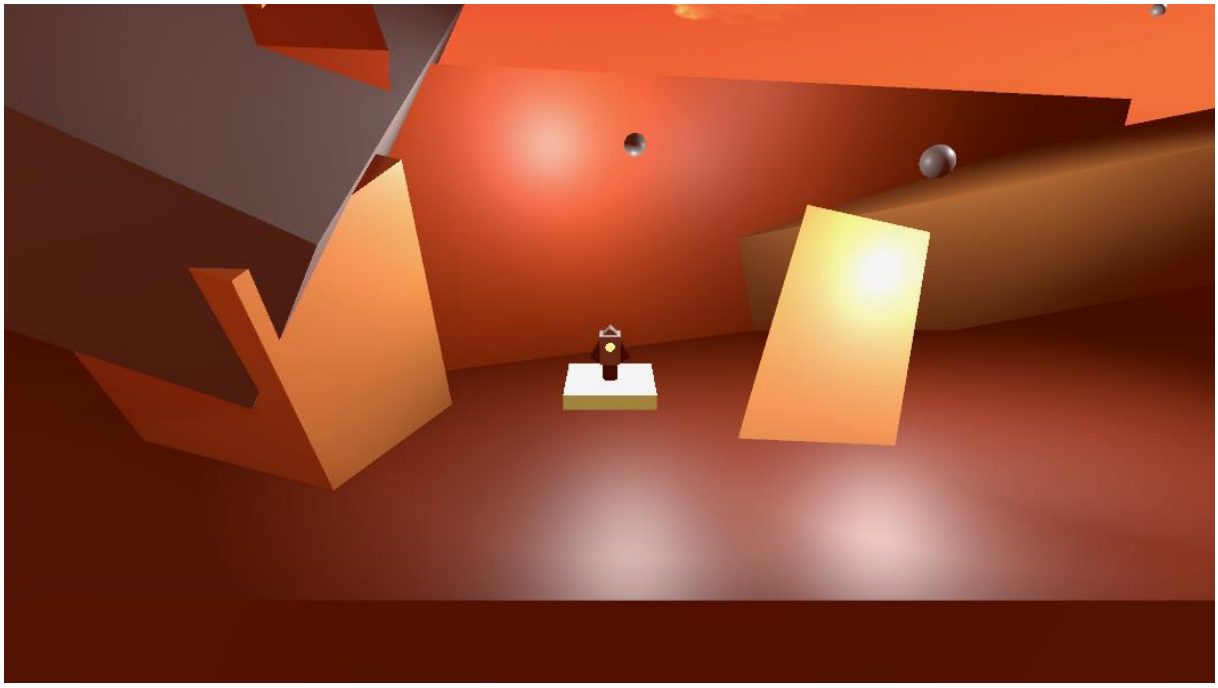
Αυτό που έχει σειρά τώρα, είναι να διασκορπιστούν άλλες τέσσερις παρόμοιες σφαίρες κατά μήκος της πίστας. Είναι ίδιου είδους, αλλά με παρόμοια συμπεριφορά, γιατί οι τιμές των μεταβλητών που αφορούν το εύρος των αξόνων x και y, είναι διαφορετικές, ο χρόνος που μένει ενεργή η κάθε σφαίρα είναι διαφορετικός και η ταχύτητα με την οποία συρρικνώνεται και μεγαλώνει, είναι επίσης διαφορετική. Κάθε νέα σφαίρα που είναι έτοιμη να εμφανιστεί σε κάποιο νέο σημείο, ελέγχει αν έχει τουλάχιστον τρία units απόσταση από κάποια άλλη σφαίρα που είναι ήδη ενεργή στην πίστα.

Όπως και στις προηγούμενες πίστες, έτσι και εδώ, προστίθενται point lights σε συγκριμένα σημεία.

Η τελευταία ενέργεια που πρέπει να γίνει, είναι να προστεθεί ένα παρασκήνιο κατάλληλο για τον συγκεκριμένο πλανήτη και είναι ένας κοκκινωπός ουρανός με σύννεφα και με πολλή σκόνη και ομίχλη. [46] Επιπλέον, στο περιβάλλον υπάρχει αρκετή κοκκινωπή ομίχλη, δίνοντας την αίσθηση της κοκκινωπής σκόνης που υπάρχει στον Άρη.



Εικόνα 3.14: *Πίστα του Άρη*



Εικόνα 3.15: Οπτική παίχτη στην *πίστα του Άρη*

3.8 Ρύθμιση βαρύτητας

Όπως αναφέρθηκε και στην αρχή αυτού του κεφαλαίου, ο πύραυλος εξερευνεί τους τρεις πλανήτες του ηλιακού μας συστήματος που έχουν έδαφος, πέρα από την Γη και μπορεί να προσεδαφιστεί επιτυχώς. Προφανώς σε κάθε πλανήτη υπάρχει διαφορετική βαρύτητα, οπότε σε κάθε πλανήτη ο παίχτης αντιμετωπίζει νέες προκλήσεις που ήταν άγνωστες, όσο γινόταν η δοκιμή στην Γη. Καθώς ο Ερμής

είναι ο πρώτος πλανήτης που επισκέπτεται ο παίχτης, αμέσως μετά την Γη, πρέπει να έχει κατά νου, ότι η βαρύτητα στον Ερμή, ισούται με 3,7 $\frac{m}{s^2}$ [47] και όχι με 9,81 $\frac{m}{s^2}$, που ισούται στην Γη. Το ίδιο ισχύει και για την Αφροδίτη και τον Άρη, όπου η βαρύτητά τους ισούται με 8,87 $\frac{m}{s^2}$ [48] και 3,73 $\frac{m}{s^2}$ [49], αντίστοιχα. Αυτό είναι κάτι που μπορεί να ωφελήσει τον παίχτη, αλλά μπορεί και να του σταθεί εμπόδιο.

Αυτό που πρέπει να γίνει τώρα, είναι να δημιουργηθεί ένα script, υπεύθυνο καθαρά για την αλλαγή της βαρύτητας σε κάθε πλανήτη. Καθώς το μόνο game object που υπάρχει σταθερά κάθε φορά σε κάθε πίστα, είναι ο πύραυλος, προστίθεται το συγκεκριμένο script πάνω του και αυτό το script για να υλοποιηθεί, ακολουθάει την λογική που παρατίθεται στο Σχήμα 3.11.

Σχήμα 3.11: *Λογική για το script GravityHandler*

3.9 Έξοδος από το παιχνίδι

Καθώς το συγκεκριμένο κεφάλαιο φτάνει στο τέλος του, κάτι που του λείπει, είναι να μπορεί ο παίχτης να κλείσει το παιχνίδι, όταν θέλει να σταματήσει να παίζει. Εφόσον το παιχνίδι είναι ένα εκτελέσιμο αρχείο που υπάρχει στον υπολογιστή, μετά την εκτέλεσή του, πρέπει να μπορεί κάποιος και να το κλείσει πατώντας ένα κουμπί του πληκτρολογίου και όχι το X του παραθύρου του παιχνιδιού. Γι' αυτόν τον λόγο, εντάσσεται στο παιχνίδι ένα script που το μόνο που κάνει είναι να δίνει την δυνατότητα στον παίχτη να μπορεί να κλείσει το παιχνίδι, πατώντας ένα κουμπί και πιο συγκεκριμένα το κουμπί Escape. Το script αυτό προστίθεται πάνω στον πύραυλο και για να δημιουργηθεί, ακολουθάει την παρακάτω λογική που βρίσκεται στο Σχήμα 3.12.

Σχήμα 3.12: *Λογική για το script QuitApplication*

3.10 Επίλογος

Το κεφάλαιο αυτό έφτασε στο τέλος του και μπορεί να κατανοήσει κάποιος, ότι όσο προχωράει αυτή η διπλωματική εργασία, άλλο τόσο προστίθενται χρήσεις νέων στοιχείων της Unity και της C#. Εσκεμμένα έγινε η χρήση βασικών game objects της Unity, έτσι ώστε το κεφάλαιο αυτό να εμβαθύνει σε πιο σημαντικά πράγματα. Γίνεται λόγος για το onion design, κάτι που είναι απαραίτητο να γίνει πριν ξεκινήσει κάποιος τον σχεδιασμό ενός παιχνιδιού. Επιπλέον προστίθενται ηχητικά και οπτικά εφέ για συγκεκριμένες καταστάσεις κατά την διάρκεια του παιχνιδιού και χρησιμοποιείται ξανά η OnCollisionEnter() για την σύγκρουση του παίχτη με το περιβάλλον. Στο συγκεκριμένο παιχνίδι προστέθηκαν άλλες τρεις πίστες και δείχνεται ο τρόπος μετάβασης από την μία πίστα στην άλλη. Αυτό που κάνει όμως την διαφορά σε αυτό το κεφάλαιο, είναι η προσθήκη πολλών εμποδίων με χαρακτηριστική συμπεριφορά και με περιοδικές κινήσεις. Η επιλογή του να προστεθούν shortcuts από διάφορα κουμπιά είναι καθοριστική, καθώς δείχνουν την σημασία που έχουν στα παιχνίδια. Αυτό που είναι επίσης σημαντικό, είναι η αλλαγή της βαρύτητας σε κάθε πίστα, γιατί δημιουργεί άλλη αίσθηση στον παίχτη και τον κάνει να σκεφτεί και να αντιμετωπίσει τις επερχόμενες προκλήσεις διαφορετικά. Τέλος, δείχνεται ο τρόπος με τον οποίο μπορεί κάποιος να κλείσει ένα εκτελέσιμο αρχείο, όταν ο ίδιος το θέλει.

Κεφάλαιο 4ο: The Awakening

4.1 Εισαγωγή

Το παιχνίδι «The Awakening», είναι το τρίτο και το τελευταίο παιχνίδι αυτής της διπλωματικής εργασίας και είναι ένα first-person shooter (FPS) παιχνίδι, όπου ο παίχτης, χειρίζεται τον πρώην διοικητή Michael Shaw, ο οποίος κλήθηκε από την κυβέρνηση να ελέγξει τελείως μυστικά ένα απομακρυσμένο σημείο βαθιά μέσα στο δάσος, μετά από υποψίες παράνομων πειραμάτων σε ανθρώπους.

4.2 Σχεδιασμός παιχνιδιού του The Awakening (Game design of The Awakening)

Αντιθέτως με τα προηγούμενα δύο παιχνίδια, στο συγκεκριμένο παιχνίδι, για περισσότερο ρεαλισμό, χρησιμοποιούνται έτοιμα assets από το Asset Store. [4] Ο Michael Shaw έχει στην κατοχή του πέντε διαφορετικά όπλα, όπου το καθένα δέχεται διαφορετικές σφαίρες. Σε περίπτωση που τελειώσουν οι σφαίρες κάποιου όπλου, για να το χρησιμοποιήσει ξανά ο παίχτης, πρέπει να βρει τις κατάλληλες σφαίρες. Επιπλέον, ο Michael Shaw κρατάει έναν φακό, καθώς η αποστολή του είναι να πάει στο συγκεκριμένο μέρος αφότου δύσει ο ήλιος. Ο φακός έχει συγκεκριμένη διάρκεια και ο Michael Shaw χρειάζεται να βρίσκει μπαταρίες γι' αυτόν, αν θέλει να βλέπει μέσα στο δάσος κατά την διάρκεια της νύχτας. Μέσα σε όλο το δάσος υπάρχουν διασκορπισμένα πλάσματα, που είναι τα αποτελέσματα των πειραμάτων και αν ο παίχτης φτάσει σε μία συγκεκριμένη απόσταση από αυτά, αρχίζουν να τον καταδιώκουν με σκοπό να τον εξουδετερώσουν. Ο παίχτης πρέπει να τα αποφύγει ή να τα εξουδετερώσει, για να μπορέσει να προχωρήσει και να μάθει τι ακριβώς συμβαίνει σε εκείνο το μέρος. Ο χειρισμός του χαρακτήρα γίνεται με τα κουμπιά W, A, S και D, για μπροστά, αριστερά, πίσω και δεξιά, αντίστοιχα, με το Space για άλμα, με το αριστερό Shift πατημένο για τρέξιμο και κουνώντας το ποντίκι προς τα αριστερά ή προς τα δεξιά, στρέφεται η κάμερα αριστερά ή δεξιά, αντίστοιχα ή κουνώντας το προς τα πάνω ή προς τα κάτω, η κάμερα κοιτάζει προς τα πάνω ή προς τα κάτω, αντίστοιχα.

Ξεκινώντας με την εμπειρία του παίχτη (player's experience), το παιχνίδι προσφέρει μία αίσθηση έντονη και σκοτεινή.

Όσον αφορά τον κύριο μηχανισμό του παιχνιδιού (core mechanic), είναι απλός και είναι να πυροβολεί ο παίχτης τα εχθρικά πλάσματα και να μπορεί να τα αποφύγει, μέχρι να φτάσει στο τέλος της πίστας.

Ο σκοπός του παιχνιδιού (game loop) είναι να προχωρήσει ο παίχτης όσο πιο βαθιά γίνεται στο δάσος, να εξουδετερώσει τα εχθρικά πλάσματα, να μαζέψει σφαίρες/μπαταρίες και εν τέλει να φτάσει στο τέλος της πίστας.

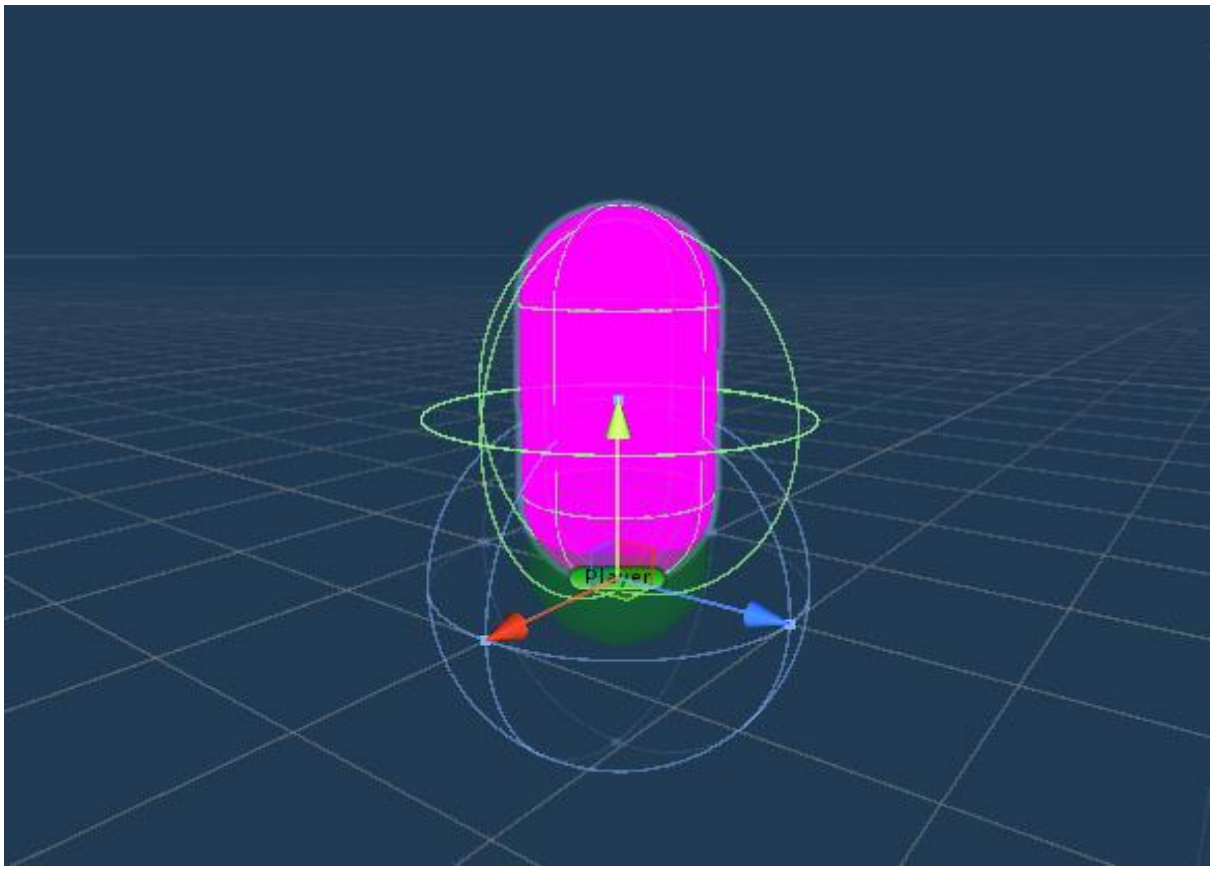
4.3 Onion design

Αυτό που έχει σειρά τώρα, είναι να γίνει το onion design. Το πρώτο και το σημαντικότερο χαρακτηριστικό που εντάσσεται στο κέντρο του onion design, είναι ότι το παιχνίδι είναι ένα first-person shooter. Το αμέσως επόμενο πιο σημαντικό χαρακτηριστικό του παιχνιδιού, είναι η προσθήκη πολλών ειδών όπλων. Το τρίτο πιο σημαντικό χαρακτηριστικό, είναι η προσθήκη διαφορετικών ειδών εχθρικών πλασμάτων σε όλη την πίστα, τα οποία μπορούν να κυνηγήσουν και να βλάψουν τον παίχτη. Το τέταρτο

πιο σημαντικό χαρακτηριστικό του παιχνιδιού είναι ο φακός του παίχτη, που με το πέρασμα του χρόνου το φως του εξασθενεί και γίνεται όλο και λιγότερο, ώσπου φτάνει να μην προσφέρει καθόλου φως στο τέλος. Και το πέμπτο και τελευταίο πιο σημαντικό χαρακτηριστικό αυτού του παιχνιδιού είναι ο ανεφοδιασμός σφαιρών για τα όπλα και μπαταριών για τον φακό του παίχτη.

4.4 Ελεγκτής πρώτου προσώπου (First-person controller)

Το πρώτο πράγμα που πρέπει να γίνει είναι να δημιουργηθεί ο χαρακτήρας του παιχνιδιού, με σκοπό στην συνέχεια να δημιουργηθεί και ο ελεγκτής πρώτου προσώπου (first-person controller), καθώς η οπτική πρώτου προσώπου (first-person view) είναι το πρώτο πιο σημαντικό χαρακτηριστικό, που εντάχθηκε στο onion design. Ο χαρακτήρας στην συγκεκριμένη περίπτωση είναι απλά μια κάψουλα, από τα έτοιμα game objects της Unity, κάτι που δεν επηρεάζει καθόλου το υπόλοιπο παιχνίδι, καθώς το παιχνίδι είναι first-person shooter και ουσιαστικά δεν φαίνεται ο χαρακτήρας που χειρίζεται ο παίχτης, αλλά φαίνεται μόνο τι βλέπει ακριβώς μπροστά του, ακριβώς σαν να είναι ο ίδιος ο παίχτης μέσα στο παιχνίδι και να κάνει ο ίδιος όλες τις ενέργειες.



Εικόνα 4.1: *Χαρακτήρας του παίχτη (Michael Shaw)*

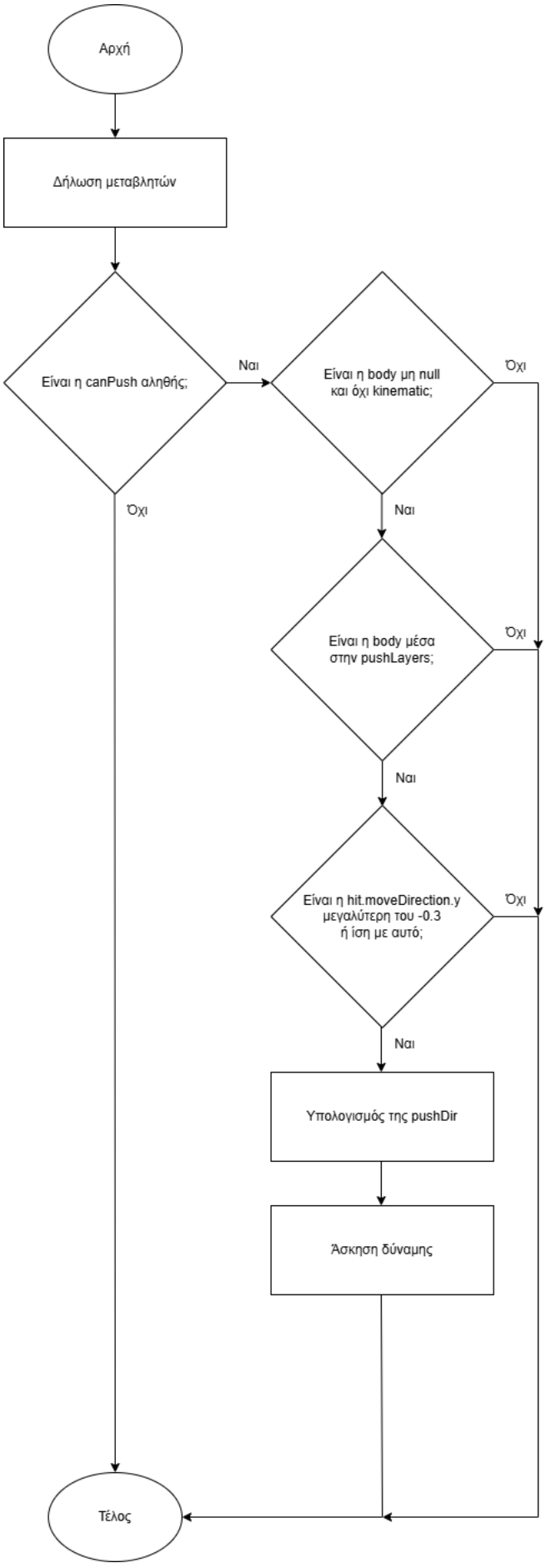
Πάνω στον χαρακτήρα προστίθεται μία κάμερα και είναι η κάμερα, η οποία δείχνει όλο το παιχνίδι στον παίχτη, ακριβώς σαν να κάνει αυτός όλες τις ενέργειες.

Τώρα πρέπει να δημιουργηθούν τρία scripts για τον χειρισμό του χαρακτήρα. Τα scripts ακολουθούν το καθένα την δική του λογική. Πρώτα δημιουργείται το script για το first-person controller και προστίθεται στον παίχτη. Το συγκεκριμένο script είναι υπεύθυνο για την κίνηση του παίχτη με οπτική

πρώτου προσώπου, για την περιστροφή του, για το άλμα του και για την συμπεριφορά της κάμερας. Η λογική που ακολουθείται για να δημιουργηθεί το script αυτό, εμφανίζεται στο Σχήμα 4.1.

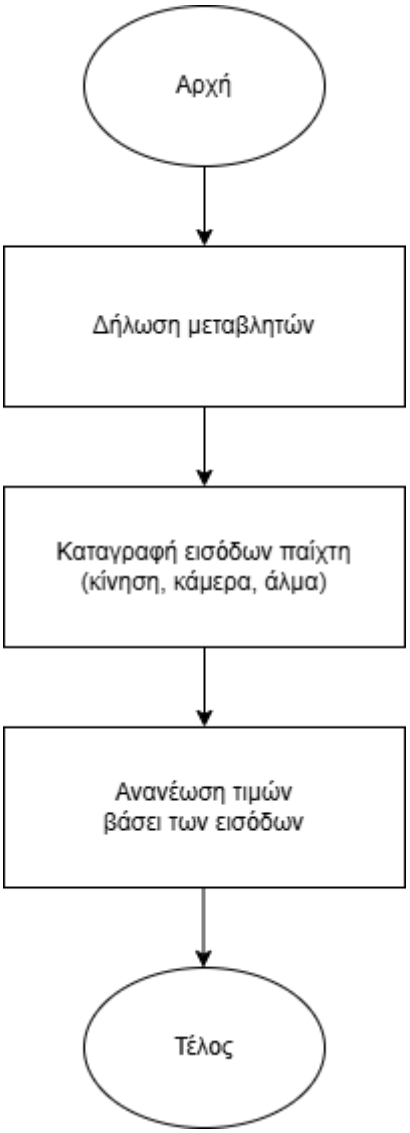
Σχήμα 4.1: *Λογική για το script FirstPersonController*

Εν συνεχεία, στο Σχήμα 4.2, παρουσιάζεται η λογική που ακολουθείται για να δημιουργηθεί το script που διαχειρίζεται την αλληλεπίδραση με αντικείμενα, που έχουν rigidbody, επιτρέποντας την ώθησή τους από τον παίχτη, όταν αυτός συγκρούεται με αυτά. Το script αυτό επίσης προστίθεται στα components του παίχτη.


Αρχή
Δήλωση μεταβλητών
Είναι η canPush αληθής;
Ναι
Είναι η body μη null
και όχι kinematic;
Όχι
Όχι
Ναι
Είναι η body μέσα
στην pushLayers;
Όχι
Ναι
Είναι η hit.moveDirection.y
μεγαλύτερη του -0.3
ή ίση με αυτό;
Όχι
Ναι
Υπολογισμός της pushDir
Άσκηση δύναμης
Τέλος

Σχήμα 4.2: *Λογική για το script BasicRigidBodyPush*

Το αμέσως επόμενο script, που χρειάζεται επίσης για τον χειρισμό τον παίχτη, εντάσσεται στα components του παίχτη και δέχεται τις εισόδους του παίχτη, μέσω των αντίστοιχων κουμπιών που πατιούνται και τις αποθηκεύει σε συγκεκριμένες μεταβλητές, έτσι ώστε να μπορούν τα άλλα scripts να τις χρησιμοποιήσουν. Παρακάτω, στο Σχήμα 4.3, δείχνεται η λογική που ακολουθείται για να δημιουργηθεί το συγκεκριμένο script..



Σχήμα 4.3: *Λογική για το script StarterAssetsInputs*

Πλέον ο παίχτης μπορεί να χειριστεί τον χαρακτήρα με τα κουμπιά W (μπροστά), A (αριστερά), S (πίσω), D (δεξιά), Space (άλμα), αριστερό Shift (τρέξιμο) και να περιστρέψει την κάμερα με την κίνηση του ποντικιού είτε αριστερά, είτε δεξιά, για να κοιτάξει αριστερά ή δεξιά, αντίστοιχα ή να κινήσει την κάμερα προς τα πάνω ή προς τα κάτω, για να κοιτάξει πάνω ή κάτω, αντίστοιχα.

Ένα μικρό πρόβλημα που παρατηρήθηκε, είναι ότι αν ο παίχτης ακουμπάει κάποιον τοίχο και παράλληλα προσπαθεί να κινηθεί, υπάρχει η αίσθηση πως ο παίχτης είναι σαν να κολλάει πάνω στον

τοίχο. Για να λυθεί αυτό το πρόβλημα πρέπει να προστεθεί ένα physics material στο capsule collider του παίχτη, να δοθεί η τιμή μηδέν σε όλες τις μεταβλητές του material αυτού και να αλλαχτεί η μεταβλητή friction combine από average σε minimum, έτσι ώστε ό,τι ακουμπάει ο παίχτης, να χρησιμοποιεί την ελάχιστη δυνατή τριβή.

4.5 Έδαφος, δέντρα και χόρτα πίστας

Το αμέσως επόμενο στοιχείο του παιχνιδιού που προστίθεται είναι το έδαφος της πίστας. Αυτό δημιουργείται με το terrain της Unity. Το έδαφος προφανώς δεν είναι τελείως επίπεδο και γι' αυτόν τον λόγο στην πίστα έχει λοφάκια, λακούβες και ανυψώσεις.

Εκτός από το έδαφος, στην πίστα υπάρχουν πολλά δέντρα και χόρτα. Για τα δέντρα και για τα χόρτα, χρησιμοποιείται και το κατάλληλο asset [50] από το Asset Store. [4]

4.6 Όπλα παίχτη

4.6.1 Sholo

Στην συνέχεια, καθώς το δεύτερο πιο σημαντικό χαρακτηριστικό αυτού του παιχνιδιού είναι να υπάρχουν πολλών ειδών όπλα, προστίθενται τα όπλα που χρησιμοποιεί ο παίχτης. Το πρώτο όπλο του παίχτη είναι ένα πιστόλι που ονομάζεται «Sholo». Για αυτό το πιστόλι χρησιμοποιείται το κατάλληλο asset [51] από το Asset Store. [4]

Για το συγκεκριμένο όπλο προστίθεται και το κατάλληλο ηχητικό εφέ για τον κρότο του. [52]

Επιπλέον, αυτό που πρέπει να προστεθεί είναι και ένα οπτικό εφέ για την εκπυρσοκρότηση του όπλου. [53]

Η τελευταία κίνηση που γίνεται, όσον αφορά το Sholo, είναι να γίνουν κάποιες αλλαγές στο material του όπλου, έτσι ώστε να έχει ένα έντονο χρυσαφί και μαύρο χρώμα.

Εικόνα 4.2: *Sholo*

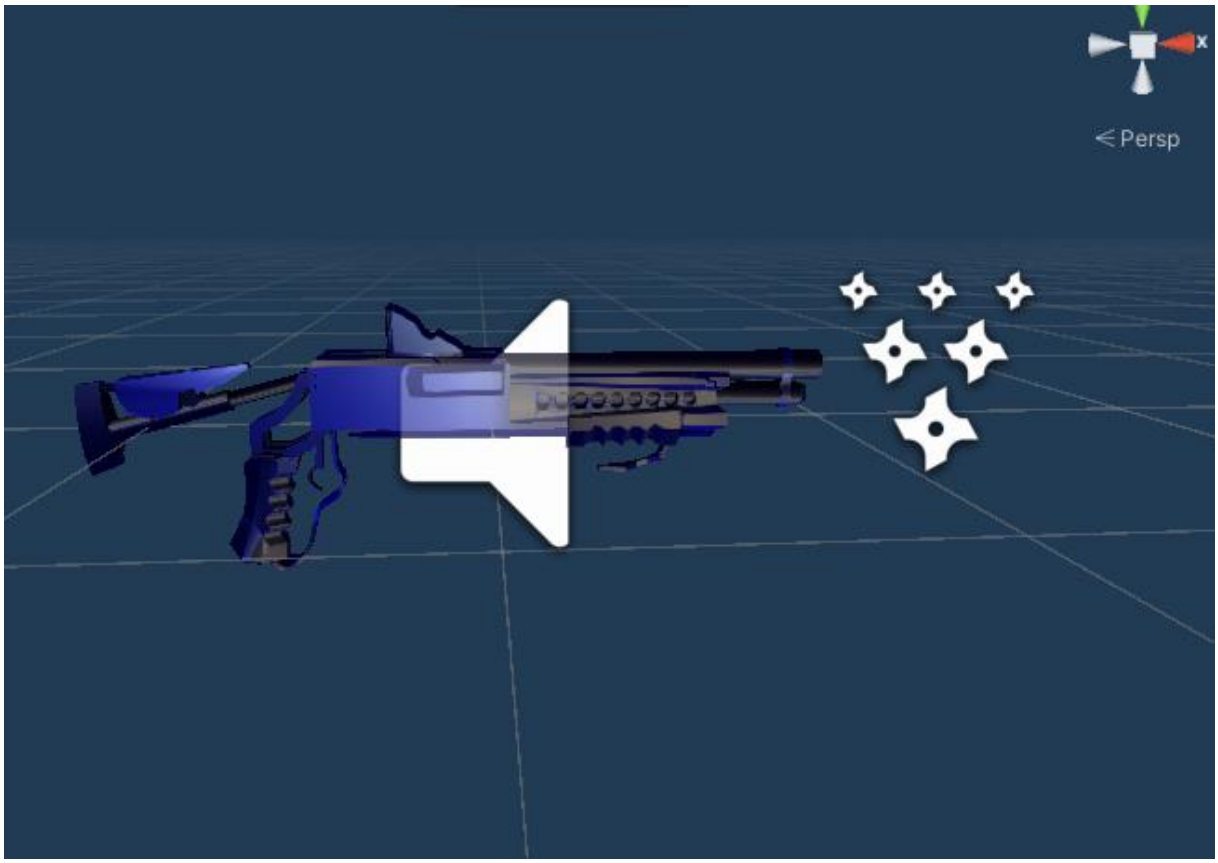
4.6.2 Buster

Το δεύτερο όπλο που προστίθεται είναι μία καραμπίνα με το όνομα «Buster». Για το όπλο αυτό χρησιμοποείται το κατάλληλο asset. [54]

Για το συγκεκριμένο όπλο προστίθεται και ένα ηχητικό εφέ για τον κρότο του. [55]

Ακόμα προστίθεται και ένα οπτικό εφέ για την εκπυρσοκρότηση του όπλου. [53]

Επίσης, αυτό που αλλάζει είναι το χρώμα του Buster που γίνεται μπλε με μαύρο.



Εικόνα 4.3: *Buster*

4.6.3 Gorlock

Το τρίτο όπλο που χειρίζεται ο παίχτης, ονομάζεται «Gorlock» και είναι ένα όπλο με λειτουργία ριπής τριών σφαιρών. Για ένα τέτοιο όπλο χρησιμοποιείται και το κατάλληλο asset. [56]

Για το Gorlock προστίθεται και το κατάλληλο ηχητικό εφέ για τον κρότο του. [57] Από το συγκεκριμένο ηχητικό εφέ, κόπηκε ένα μέρος του. Το μέρος αυτό είναι ο ήχος που κάνει η πρώτη σφαίρα. Ουσιαστικά ο ήχος αυτός ακούγεται τρεις φορές, εφόσον το όπλο Gorlock είναι όπλο με λειτουργία ριπής τριών σφαιρών.

Τώρα αυτό που έχει σειρά, είναι να προστεθεί ένα οπτικό εφέ για την εκπυρσοκρότηση του όπλου. [53]

Η τελευταία κίνηση που γίνεται για το όπλο Gorlock, είναι να γίνουν αλλαγές στο material του με σκοπό να έχει κόκκινο και μαύρο χρώμα.

Εικόνα 4.4: *Gorlock*

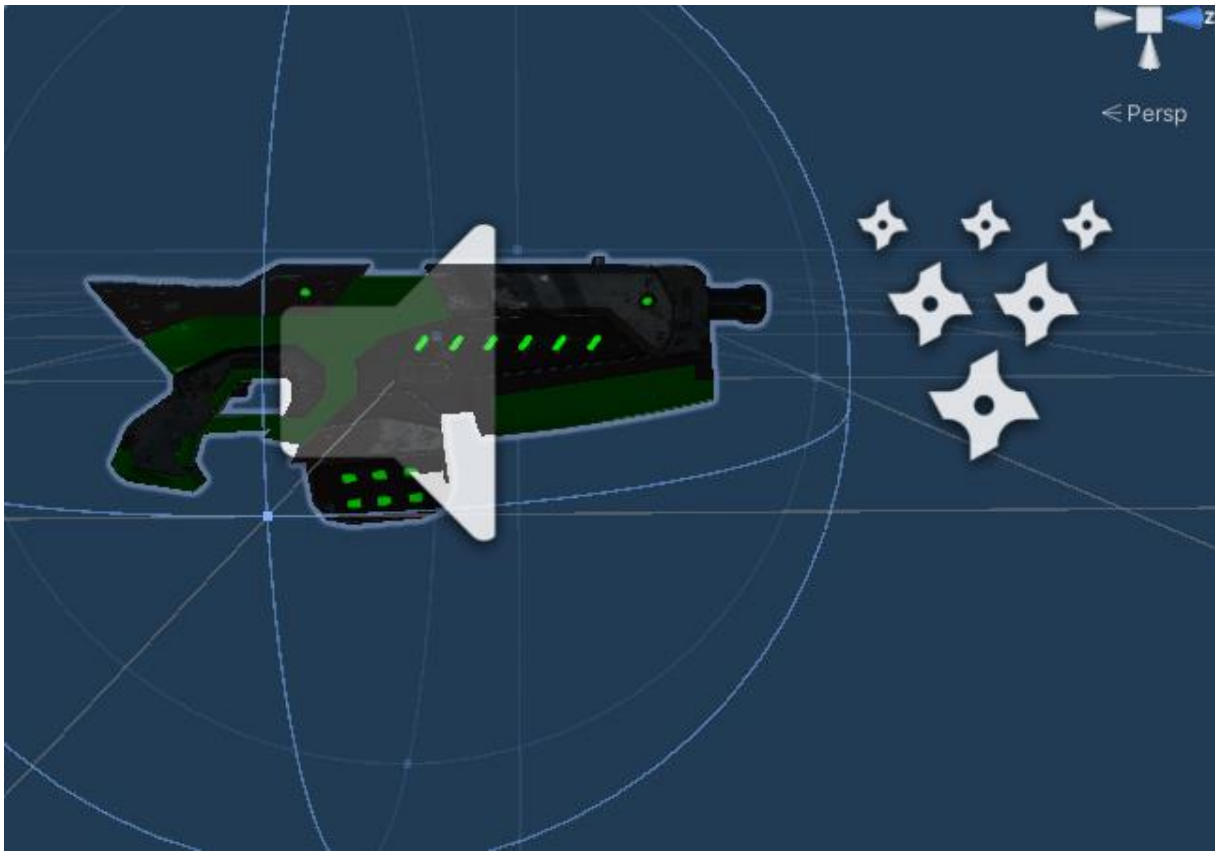
4.6.4 Plazmius

Το τέταρτο όπλο του παίχτη, ονομάζεται «Plazmius». Είναι ένα αυτόματο όπλο. Για ένα τέτοιο όπλο χρησιμοποιείται και το κατάλληλο asset. [58]

Εφόσον το Plazmius είναι ένα αυτόματο όπλο, για τον κρότο της κάθε σφαίρας, χρησιμοποιείται πάλι το ίδιο ηχητικό εφέ, που χρησιμοποιήθηκε και για το Gorlock. [57] Επειδή αυτό είναι άλλο όπλο, γίνονται αλλαγές στο ηχητικό εφέ για να δείχνει διαφορετικό από το ηχητικό εφέ που χρησιμοποιείται για το Gorlock.

Επιπλέον, προστίθεται και το κατάλληλο οπτικό εφέ για την εκπυρσοκρότησή του. [53]

Αυτό που χρειάζεται τώρα είναι να γίνουν αλλαγές στο material του όπλου, έτσι ώστε να έχει σκούρο πράσινο και μαύρο χρώμα.



Εικόνα 4.5: *Plazmius*

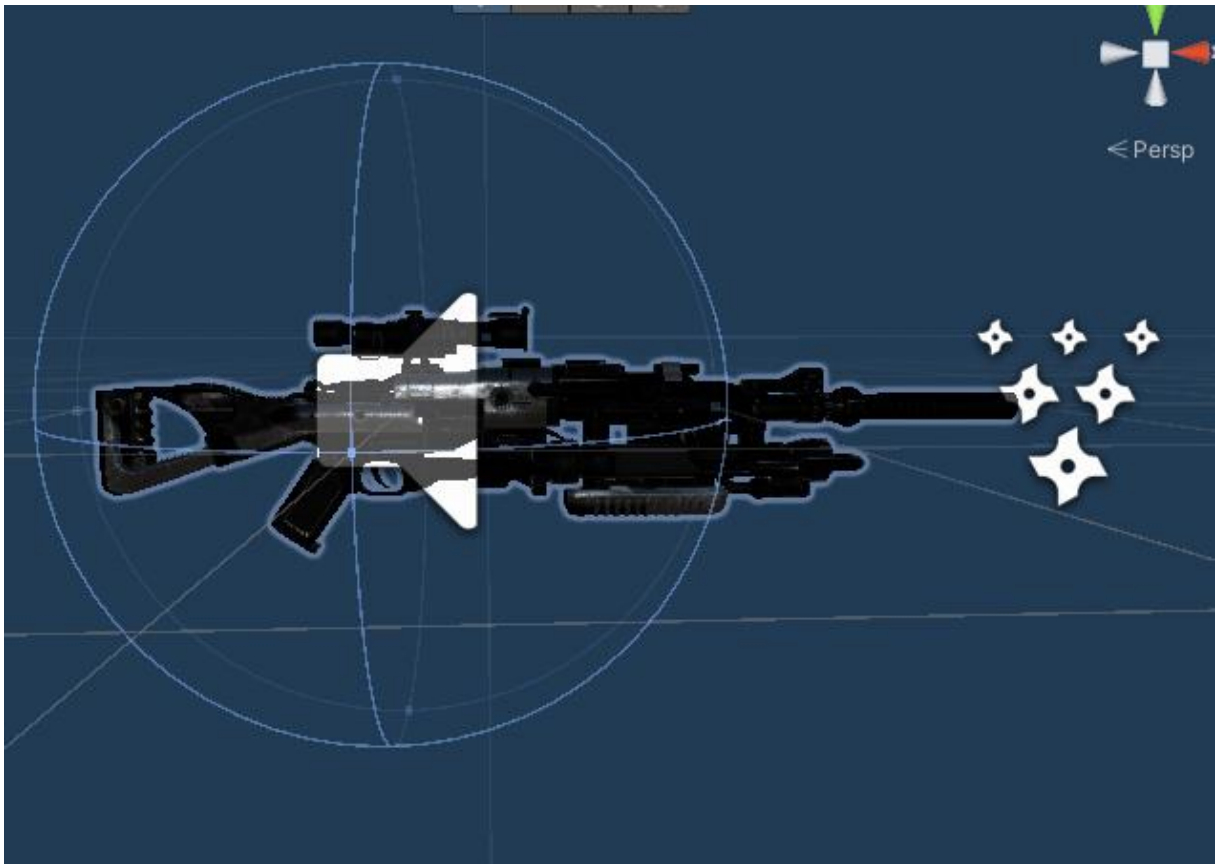
4.6.5 Soulpiercer

Το πέμπτο και το τελευταίο όπλο που υπάρχει στα χέρια του παίχτη, είναι το όπλο με το όνομα «Soulpiercer». Είναι ένα όπλο ελεύθερου σκοπευτή και γι' αυτό χρησιμοποιείται και το κατάλληλο asset [59] από το Asset Store. [4]

Για το Soulpiercer, στην συνέχεια, προστίθεται και ένα ηχητικό εφέ για τον κρότο του. [60]

Αμέσως μετά προστίθεται και το κατάλληλο οπτικό εφέ για την εκπυρσοκρότησή του. [53]

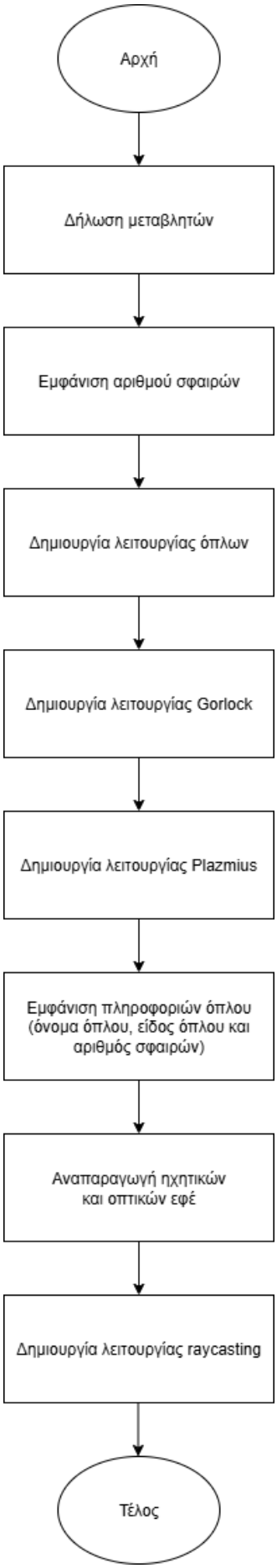
Και τέλος, γίνονται αλλαγές στο material του, έτσι ώστε να είναι ένα τελείως μαύρο όπλο, καθώς το μαύρο χρώμα ταιριάζει σε όπλο ελεύθερου σκοπευτή.



Εικόνα 4.6: *Soulpiercer*

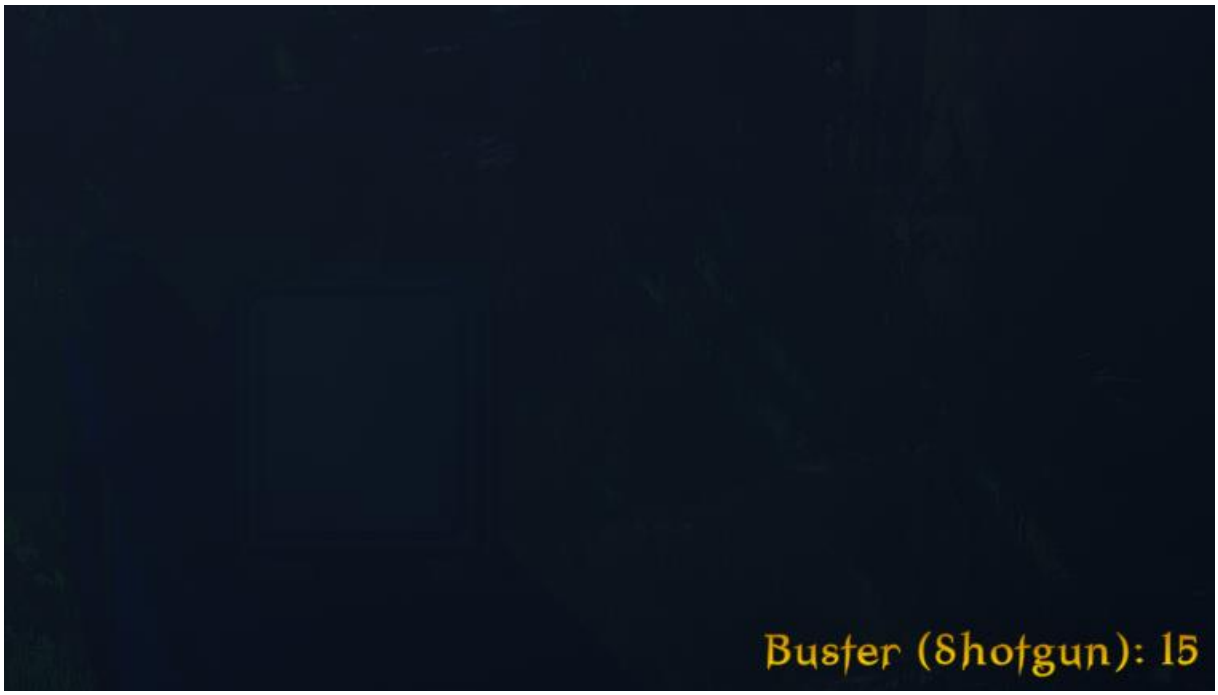
4.7 Physics.Raycast()

Για να μπορεί ο παίχτης να πυροβολήσει τα εχθρικά πλάσματα, χρησιμοποιείται το raycasting της Unity, που ουσιαστικά είναι μία νοητή ακτίνα που ενώνει δύο σημεία. Το ένα σημείο είναι αυτό από όπου ξεκινάει η ακτίνα και το άλλο είναι εκεί που καταλήγει. Για να λειτουργήσει το raycasting, πρέπει να υπάρχει κάποιου είδους collider στο game object που στοχεύει ο παίχτης. Έτσι λοιπόν, δημιουργείται ένα script που δίνει την δυνατότητα στον παίχτη με το πάτημα του αριστερού κουμπιού του ποντικιού (πατώντας το μία φορά ή κρατώντας το πατημένο σε περίπτωση που χρησιμοποιεί το αυτόματο όπλο Plazmius) να πυροβολήσει. Το συγκεκριμένο script προστίθεται σε κάθε όπλο που χρησιμοποιεί ο παίχτης και γίνονται αλλαγές στις μεταβλητές του, έτσι ώστε το κάθε όπλο να έχει τον δικό του χαρακτήρα και να χρησιμοποιείται για διαφορετική περίπτωση. Στο Σχήμα 4.4 παρουσιάζεται η λογική που ακολουθείται για την δημιουργία του συγκεκριμένου script.


Αρχή
Δήλωση μεταβλητών
Εμφάνιση αριθμού σφαιρών
Δημιουργία λειτουργίας όπλων
Δημιουργία λειτουργίας Gorlock
Δημιουργία λειτουργίας Plazmius
Εμφάνιση πληροφοριών όπλου (όνομα όπλου, είδος όπλου και αριθμός σφαιρών)
Αναπαραγωγή ηχητικών και οπτικών εφέ
Δημιουργία λειτουργίας raycasting
Τέλος

Σχήμα 4.4: *Λογική για το script Weapon*

Επιπλέον, δημιουργείται ένα κείμενο το οποίο δείχνει τις πληροφορίες του όπλου, που χρησιμοποιείται εκείνη την στιγμή. Για παράδειγμα, αν ο παίχτης χρησιμοποιεί το δεύτερο όπλο που είναι η καραμπίνα Buster, στο κάτω δεξιά μέρος της οθόνης του παίχτη, το κείμενο αυτό δείχνει το όνομα του όπλου, το είδος του όπλου από δίπλα μέσα σε παρένθεση και αμέσως μετά πόσες σφαίρες έχει μέσα στο όπλο, ακριβώς όπως δείχνει η παρακάτω εικόνα.



Εικόνα 4.7: *Πληροφορίες του όπλου Buster*

4.8 Εναλλαγή όπλων

Καθώς υπάρχουν πέντε είδη όπλων, πρέπει ο παίχτης να μπορεί να χειριστεί όποιο από αυτά θέλει και όποτε το θέλει. Γι' αυτόν τον λόγο πρέπει να δημιουργηθεί ένα script και να είναι υπεύθυνο για την εναλλαγή των όπλων κατά την διάρκεια του παιχνιδιού. Αυτό το script δημιουργείται με την λογική που παρατίθεται στο Σχήμα 4.5 και προστίθεται στα components ενός αρχικά κενού game object με το όνομα «Weapons», μέσα στο οποίο εν συνεχεία εντάσσονται και τα πέντε όπλα του παίχτη, σαν children του.

Σχήμα 4.5: *Λογική για το script WeaponSwitcher*

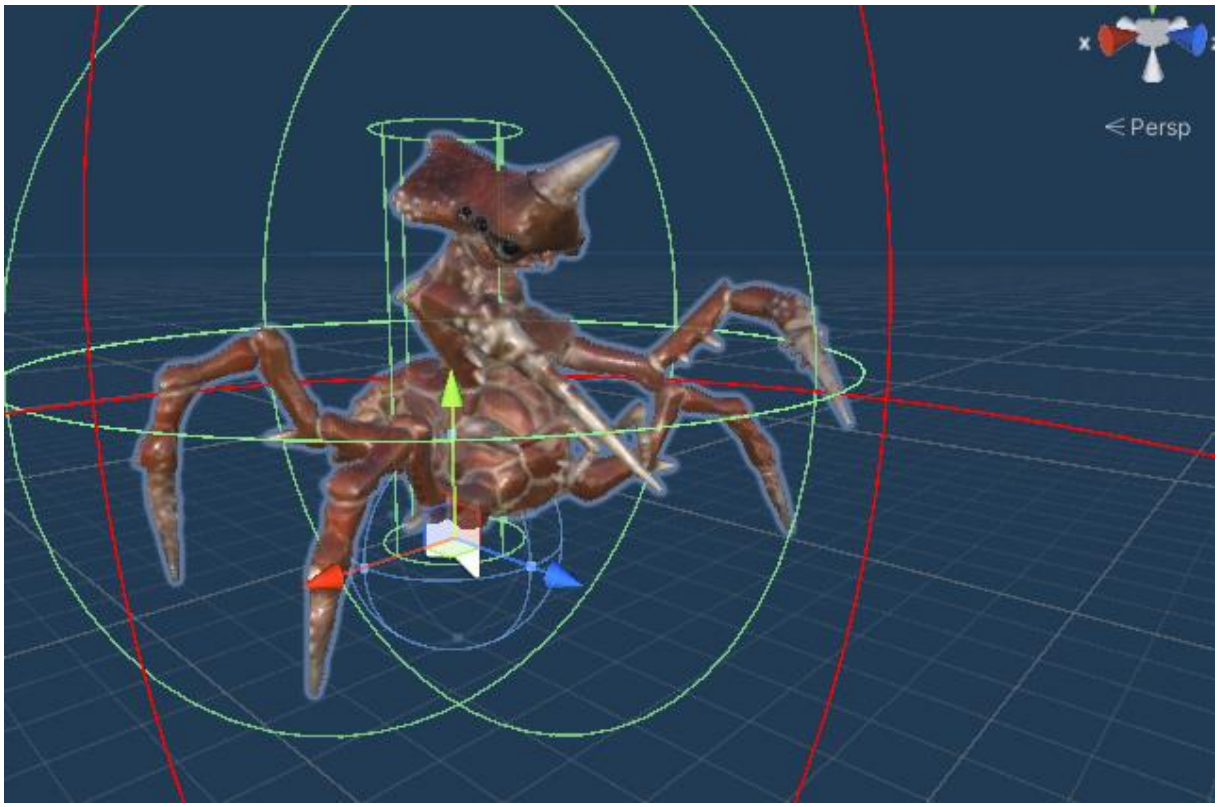
4.9 Εχθρικά πλάσματα

4.9.1 Shabumi

Αφού έγινε η προσθήκη πολλών ειδών όπλων, το οποίο είναι το δεύτερο πιο σημαντικό χαρακτηριστικό του παιχνιδιού, σειρά έχει να γίνει η προσθήκη διαφορετικών ειδών εχθρικών πλασμάτων, καθώς αυτό είναι το τρίτο πιο σημαντικό χαρακτηριστικό του συγκεκριμένου παιχνιδιού. Το πρώτο εχθρικό πλάσμα που προστίθεται στο παιχνίδι, είναι το Shabumi, που είναι το αποτέλεσμα ενός πειράματος πάνω σε άνθρωπο, αράχνη και καβούρι. Για το Shabumi, χρησιμοποιείται το κατάλληλο asset [61] από το Asset Store. [4] Μέσα στο συγκεκριμένο asset, υπάρχουν και τα animations για την κίνησή του, για τον τρόπο που επιτίθεται, αλλά και για την στάση που έχει όταν είναι σε αδράνεια. Αυτά τα τρία animations στην συνέχεια χρησιμοποιούνται στο παιχνίδι.

Εκτός από το μοντέλο αυτό και τα animations του, πρέπει να προστεθεί και ο ήχος που κάνει. [62]

Επιπλέον, γίνονται οι απαραίτητες ρυθμίσεις, όσον αφορά το script EnemyAI και τις μεταβλητές του, έτσι ώστε να μπορούν να λειτουργούν όλα όπως πρέπει και να δημιουργηθεί η επιθυμητή συμπεριφορά για το πλάσμα Shabumi.



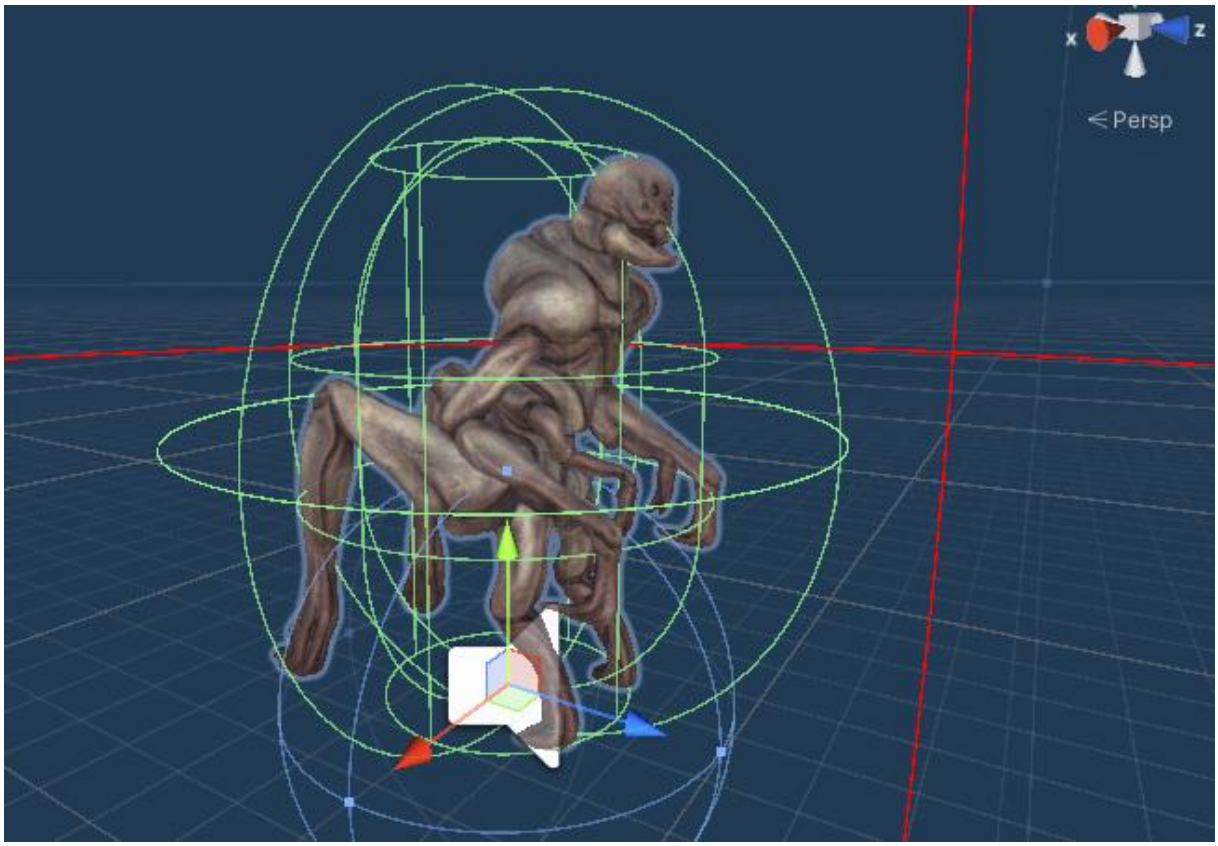
Εικόνα 4.8: *Shabumi*

4.9.2 Arach

Το δεύτερο εχθρικό πλάσμα που υπάρχει στην πίστα, είναι το Arach, που είναι αποτέλεσμα ενός πειράματος πάνω σε άνθρωπο, αράχνη και ακρίδα. Για το Arach, χρησιμοποιείται το κατάλληλο asset. [63] Μέσα στο συγκεκριμένο asset, υπάρχουν και τα animations για την κίνησή του και για τον τρόπο που επιτίθεται και αυτά τα animations στην συνέχεια χρησιμοποιούνται στο παιχνίδι.

Εκτός από το μοντέλο αυτό και τα animations του, πρέπει να προστεθεί και ο ήχος που κάνει το Arach. [64]

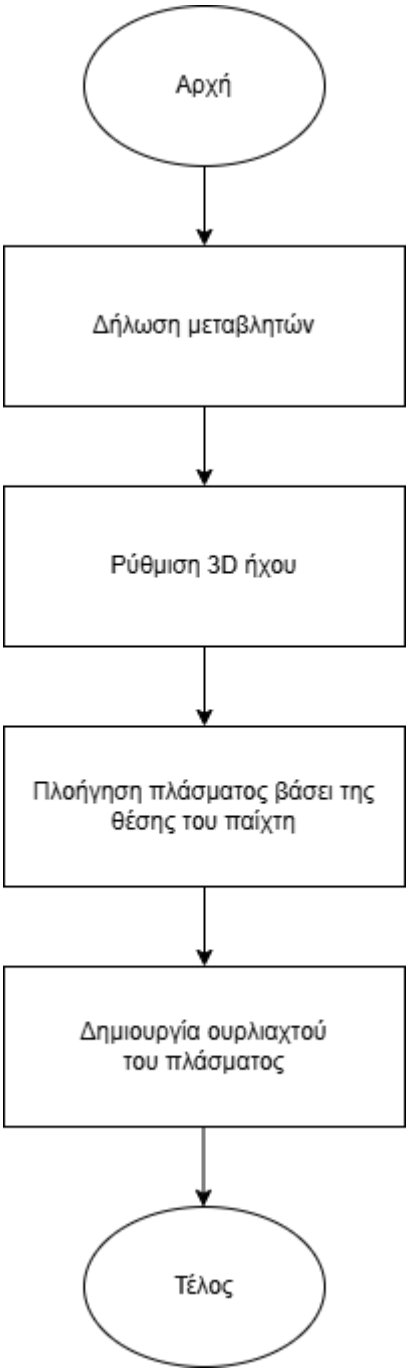
Επίσης, γίνονται οι απαραίτητες ρυθμίσεις, όσον αφορά το script EnemyAI και τις μεταβλητές του, για να μπορούν να λειτουργούν όλα στην εντέλεια και να δημιουργηθεί η επιθυμητή συμπεριφορά για το πλάσμα Arach.



Εικόνα 4.9: *Arach*

4.10 Τεχνητή νοημοσύνη εχθρικών πλασμάτων

Το βασικό script που χρειάζεται να προστεθεί στα εχθρικά πλάσματα, είναι το script που είναι υπεύθυνο για την κίνησή τους, όταν τα πλησιάσει ο παίχτης ή όταν τα επιτεθεί. Το script ουσιαστικά δίνει τεχνητή νοημοσύνη (artificial intelligence (AI)) στα εχθρικά πλάσματα και περιέχει μεταβλητές, που σχετίζονται με τα χαρακτηριστικά του κάθε είδους των εχθρικών πλασμάτων. Για παράδειγμα, μέσα σε αυτά τα χαρακτηριστικά είναι η απόσταση που υπάρχει ανάμεσα στον παίχτη και στο πλάσμα, η οποία αν ξεπεραστεί από τον παίχτη, το πλάσμα αντιλαμβάνεται ότι ο παίχτης είναι εντός της ακτίνας επίθεσής του και ξεκινάει να κυνηγήσει τον παίχτη. Το script εντάσσεται στα components του κάθε πλάσματος και για να δημιουργηθεί επιτυχώς, ακολουθάει την παρακάτω λογική που εμφανίζεται στο Σχήμα 4.6.

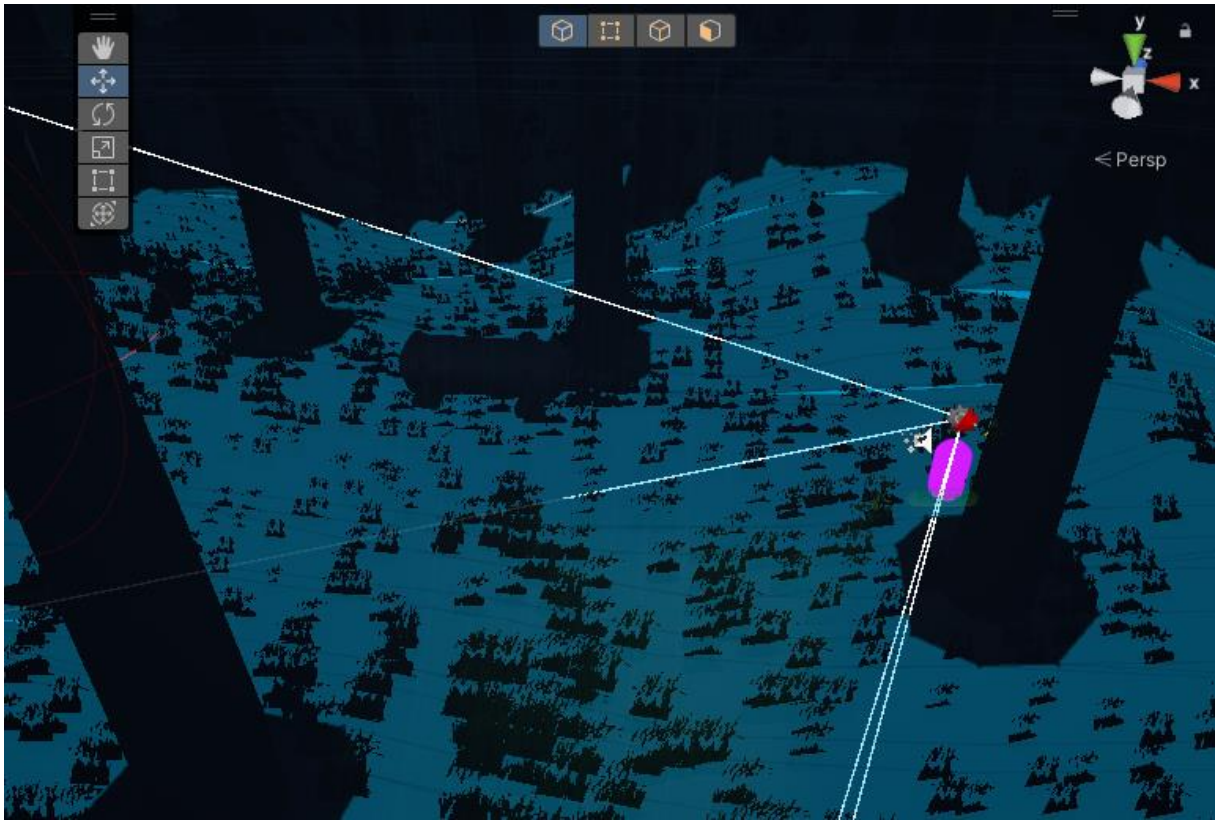


Σχήμα 4.6: *Λογική για το script EnemyAI*

4.11 NavMeshAgent

Το NavMeshAgent είναι η αμέσως επόμενη και μία από τις βασικότερες προσθήκες που γίνονται στο παιχνίδι. Προστίθεται στα components των εχθρικών πλασμάτων και με την βοήθειά του, τα εχθρικά πλάσματα μπορούν να κινηθούν σχεδόν σε όλη την πίστα. Γίνονται οι απαραίτητες ενέργειες και τροποποιήσεις στις ρυθμίσεις του NavMeshAgent, έτσι ώστε να υπάρχει ευελιξία των πλασμάτων στο μεγαλύτερο μέρος της πίστας, αλλά και για να είναι το παιχνίδι όσο πιο ρεαλιστικό γίνεται. Όπως φαίνεται και στην παρακάτω εικόνα, το NavMeshAgent δίνει αυτό το γαλάζιο χρώμα, σε όλα τα σημεία

της πίστας, στα οποία μπορούν να κινηθούν τα πλάσματα. Εννοείται πως το γαλάζιο χρώμα δεν φαίνεται όσο κάποιος παίζει το παιχνίδι και απλά φαίνεται το έδαφος με τα χόρτα και τα δέντρα.

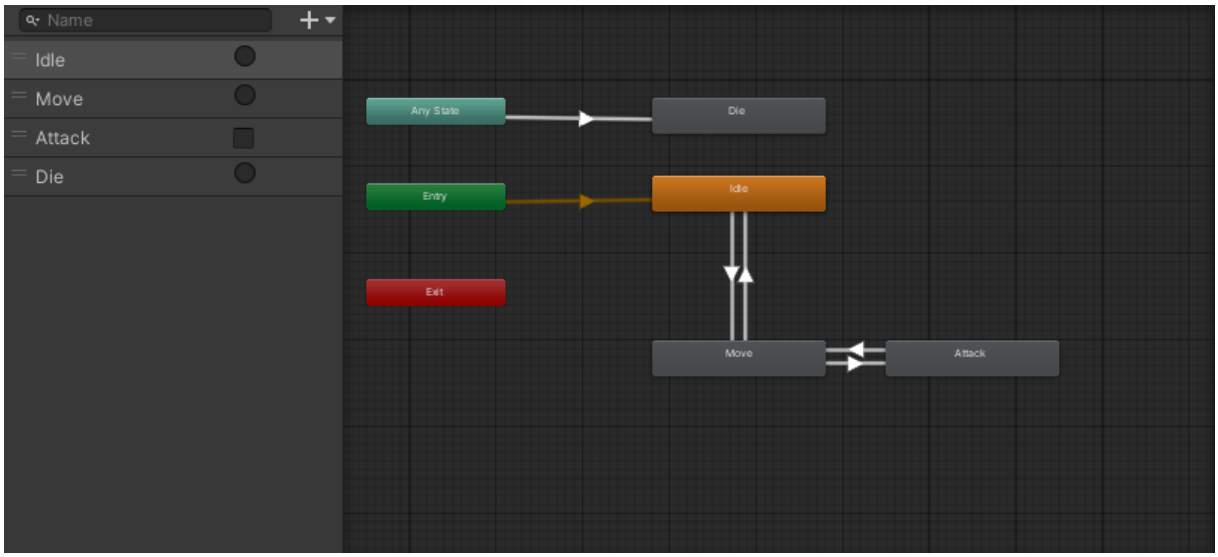


Εικόνα 4.10: *NavMeshAgent*

4.12 Animations εχθρικών πλασμάτων

Εφόσον πλέον έχουν προστεθεί τα κατάλληλα assets για τα εχθρικά πλάσματα, αυτή είναι η κατάλληλη στιγμή να ενταχθούν τα animations για τις κινήσεις τους. Τα συγκεκριμένα assets που χρησιμοποιούνται, έχουν στα περιεχόμενά τους ήδη δημιουργημένα animations για το κάθε πλάσμα, οπότε αυτό που πρέπει να γίνει είναι να χρησιμοποιηθούν κατάλληλα στο παιχνίδι. Έτσι, λοιπόν, δημιουργείται ένας animator και σε αυτόν εντάσσεται η παράμετρος Idle, η οποία είναι τύπου trigger, η παράμετρος Move, η οποία είναι τύπου trigger, η παράμετρος Attack, η οποία είναι τύπου bool και η παράμετρος Die, η οποία είναι τύπου trigger. Αμέσως μετά, δημιουργούνται τέσσερις καταστάσεις. Αυτές είναι η κατάσταση Idle, η οποία εμπεριέχει το animation του κάθε πλάσματος για όταν αυτό βρίσκεται σε αδράνεια, η κατάσταση Move, στην οποία εμπεριέχεται το animation του βαδίσματος ή του τρεξίματος του κάθε πλάσματος, η κατάσταση Attack, η οποία περιέχει το animation για κάθε πλάσμα όταν αυτό επιτίθεται και τέλος η κατάσταση Die, η οποία περιέχει το animation που χρησιμοποιείται όταν το πλάσμα εξουδετερώνεται.

Επίσης, δημιουργούνται και οι κατάλληλες μεταβάσεις από την κατάσταση Idle στην κατάσταση Move και το αντίστροφο και από την κατάσταση Move στην κατάσταση Attack και το αντίστροφο.

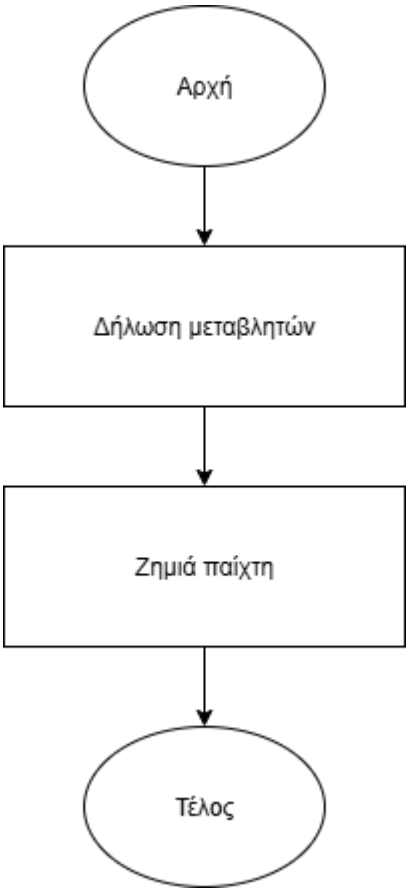


Εικόνα 4.11: *Παράθυρο του animator*

Στην συνέχεια, προστίθεται ο animator στα components του κάθε πλάσματος και συμπληρώνονται ανάλογα οι μεταβλητές του.

4.13 Επίθεση εχθρικών πλασμάτων

Για να μπορέσει το κάθε πλάσμα να βλάψει τον παίχτη με αποτέλεσμα να φτάσει στο σημείο να τον εξουδετερώσει, πρέπει να δημιουργηθεί και το κατάλληλο script. Αυτό το script είναι υπεύθυνο καθαρά για την ζημιά που προκαλεί το κάθε πλάσμα στον παίχτη και εντάσσεται στα components του κάθε πλάσματος. Η ζημιά που προκαλεί το κάθε πλάσμα είναι διαφορετική, για να μπορεί να είναι το κάθε πλάσμα ιδιαίτερο. Το script δημιουργείται ακολουθώντας την λογική που εμφανίζεται στο Σχήμα 4.7.

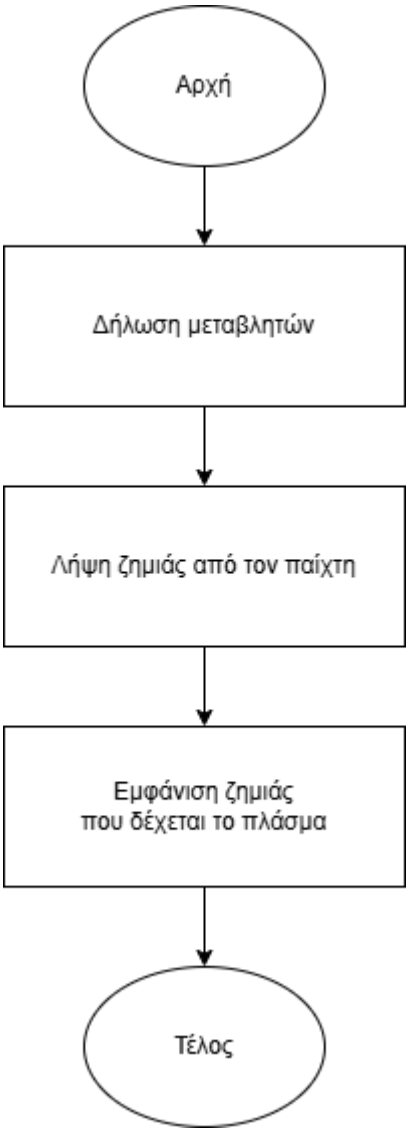


Σχήμα 4.7: *Λογική για το script EnemyAttack*

Για να μπορέσει να είναι τελείως λειτουργικό το script, που σχετίζεται με την επίθεση του κάθε πλάσματος στον παίχτη, πρέπει να επεξεργαστούν τα προϋπάρχοντα animations των πλασμάτων, με σκοπό την στιγμή που το κάθε πλάσμα φαίνεται ότι χτυπάει τον παίχτη, να δημιουργηθεί ένα animation event με το όνομα «AttackHitEvent», στο animation της επίθεσης του κάθε πλάσματος. Όπως φαίνεται και στο script, δημιουργείται μία μέθοδος ειδικά για αυτή την κατάσταση. Αυτή η μέθοδος είναι η AttackHitEvent(), η οποία δίνει την δυνατότητα στο κάθε πλάσμα να προκαλέσει ζημιά στον παίχτη.

4.14 Πόντοι ζωής εχθρικών πλασμάτων

Το αμέσως επόμενο στοιχείο που πρέπει να ενταχθεί στο παιχνίδι, είναι οι πόντοι ζωής των εχθρικών πλασμάτων. Με αυτόν τον τρόπο, μπορεί ο παίχτης να τα πυροβολήσει και εν τέλει να τα εξουδετερώσει. Για να γίνει όμως αυτό δημιουργείται ένα script, το οποίο προστίθεται σε κάθε πλάσμα και δίνει πόντους ζωής στο καθένα τους. Το script αυτό δημιουργείται, βάσει της λογικής που παρατίθεται στο Σχήμα 4.8.



Σχήμα 4.8: *Λογική για το script EnemyHealth*

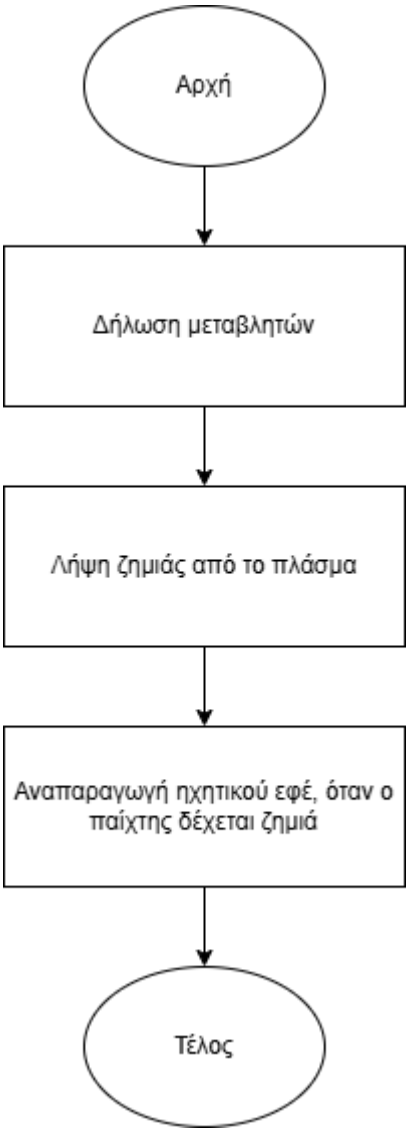
Οι πόντοι ζωής του πλάσματος Shabumi και του πλάσματος Arach διαφέρουν. Για να μπορούν να διαφέρουν οι πόντοι ζωής των πλασμάτων ρυθμίζεται η κατάλληλη μεταβλητή του script.

Κάτι ακόμα που προστίθεται, είναι μία ένδειξη η οποία δείχνει στον παίχτη πόση ζημιά κάνει το κάθε όπλο στο κάθε πλάσμα. Για να υπάρξει αυτή η λειτουργία δημιουργείται ένα κείμενο που δείχνει το νούμερο της ζημιάς που κάνει το κάθε όπλο. Στην παρακάτω εικόνα φαίνεται η ζημιά που κάνει το όπλο Gorlock, το οποίο κάνει ζημιά των δεκαπέντε πόντων με κάθε σφαίρα. Επειδή όμως αυτό το όπλο είναι όπλο με λειτουργία ριπής τριών σφαιρών, κάθε μία από τις τρεις σφαίρες που ρίχνει, κάνει δεκαπέντε πόντους ζημιά και το κάθε νούμερο εμφανίζεται το ένα ακριβώς μετά το άλλο.

Εικόνα 4.12: *Ένδειξη ζημιάς του όπλου Gorlock*

4.15 Πόντοι ζωής παίχτη

Κάτι που δεν μπορεί να παραληφθεί από το παιχνίδι, είναι οι πόντοι ζωής του παίχτη και πιο συγκεκριμένα του χαρακτήρα του παιχνιδιού, του Michael Shaw. Ο Michael Shaw δεν θα μπορούσε να μην έχει πόντους ζωής, καθώς είναι ο κύριος χαρακτήρας του παιχνιδιού. Εφόσον όμως ο Michael Shaw είναι άνθρωπος και έρχεται αντιμέτωπος με πλάσματα που έχουν προέλθει από πειράματα, είναι πολύ λογικό να μπορεί να δεχθεί πολύ εύκολα αρκετά μεγάλη ζημιά με μόνο ένα χτύπημα του κάθε πλάσματος. Για τον ήχο που ακούγεται όταν ο παίχτης δέχεται ζημιά, χρησιμοποιείται το κατάλληλο ηχητικό εφέ. [65] Έτσι, με ένα script που προστίθεται στα components του παίχτη, δίνονται κατάλληλα οι πόντοι ζωής στον Michael Shaw, κάνοντάς τον αρκετά ευάλωτο. Το script δημιουργείται με βάση την ακόλουθη λογική που παρουσιάζεται στο Σχήμα 4.9.



Σχήμα 4.9: *Λογική για το script PlayerHealth*

4.16 Οπτικά στοιχεία μάχης

Για την ενίσχυση του ρεαλισμού, της ανατροφοδότησης προς τον παίχτη και της συνολικής εμπειρίας του gameplay, γίνονται κάποιες βασικές προσθήκες. Η πρώτη βασική προσθήκη αφορά τον δείκτη σκόπευσης και είναι το μέσο με το οποίο ο παίχτης μπορεί με ευκολία να στοχεύσει τα εχθρικά πλάσματα. Για τον δείκτη σκόπευσης χρησιμοποιείται και το κατάλληλο asset. [66] Ο δείκτης σκόπευσης δημιουργείται μέσω της διεπαφής του χρήστη (user interface (UI)) και εφαρμόζεται και στα πέντε όπλα του παίχτη.

Εικόνα 4.13: *Δείκτης σκόπευσης*

Η επόμενη προσθήκη που γίνεται, είναι ένα οπτικό εφέ το οποίο εμφανίζεται κάθε φορά που το όπλο του παίχτη χτυπάει κάτι με το raycasting. Με αυτόν τον τρόπο, επιστρέφει στον παίχτη η πληροφορία ότι το όπλο του χτύπησε κάτι. Αυτό είναι ακόμα πιο χρήσιμο αν κάποιος σκεφτεί ότι κάθε όπλο μπορεί να πυροβολήσει μέχρι μία συγκεκριμένη απόσταση, η οποία είναι διαφορετική για το καθένα. Για παράδειγμα, το Soulpiercer (όπλο ελεύθερου σκοπευτή) μπορεί να πυροβολήσει πιο μακριά απ' ότι μπορεί το Sholo (πιστόλι). Για το συγκεκριμένο οπτικό εφέ, χρησιμοποιείται και το κατάλληλο asset. [53]

Η τελευταία απαραίτητη προσθήκη, όσον αφορά τα οπτικά στοιχεία μάχης, είναι να υπάρχει μία ένδειξη στον παίχτη, ότι δέχτηκε ζημιά από κάποιο από τα εχθρικά πλάσματα. Αυτό γίνεται πολύ εύκολα με την χρήση ενός script που εντάσσεται στα components του παίχτη και ενός συγκεκριμένου asset, στραμμένου στις μείον εκατόν τριάντα μοίρες. [67] Το script για να δημιουργηθεί, ακολουθάει ακριβώς την λογική που παρατίθεται στο Σχήμα 4.10.

Σχήμα 4.10: *Λογική για το script DisplayDamage*

Εικόνα 4.14: *Ένδειξη ζημιάς παίχτη*

4.17 Τέλος παιχνιδιού και προσθήκη ουρανού

Όταν ο παίχτης χάσει όλους τους πόντους ζωής του, αυτομάτως χάνει και εμφανίζεται ένα μενού, το οποίο δίνει δύο επιλογές στον παίχτη. Η μία επιλογή είναι να παίξει ξανά από την αρχή και η άλλη να κλείσει το παιχνίδι. Για να πραγματοποιηθεί αυτό πρέπει να δημιουργηθούν δύο κουμπιά και πιο συγκεκριμένα ένα που το πατάει ο παίχτης όταν θέλει να ξαναπαίξει και ένα που το πατάει όταν θέλει να κλείσει το παιχνίδι. Επίσης, δημιουργείται ένα κείμενο με την πρόταση «Massacred…» για να δείξει στον παίχτη όταν έχασε. Η λογική που ακολουθεί και φαίνεται στο Σχήμα 4.11, μπορεί να δημιουργήσει ένα script που εμφανίζει το προαναφερθέν μενού με τα δύο κουμπιά και το script αυτό, εντάσσεται στον παίχτη.

Σχήμα 4.11: *Λογική για το script DeathHandler*

Εικόνα 4.15: *Εμφάνιση μενού όταν χάνει ο παίχτης*

Επιπλέον, καθώς το παιχνίδι έχει σοβαρό και σκοτεινό ύφος, προστίθεται το κατάλληλο asset, όσον αφορά τον ουρανό της πίστας. [43]

Εικόνα 4.16: *Ουρανός πίστας*

Για να μπορεί όμως το παιχνίδι να ξαναφορτώσει την ίδια πίστα, εφόσον πιο πριν έχει πατηθεί το αντίστοιχο κουμπί ή για να μπορεί ο παίχτης να βγει από το παιχνίδι, εφόσον έχει πατήσει το αντίστοιχο κουμπί, πρέπει να δημιουργηθεί το κατάλληλο script. Το script αυτό εντάσσεται σε ένα κενό game object με το όνομα «Game Session» και για να μπορέσει να δημιουργηθεί ακολουθάει την λογική που δίνεται στο Σχήμα 4.12.

Σχήμα 4.12: *Λογική για το script SceneLoader*

4.18 Ζουμ όπλων

Μία προσθήκη που κάνει το παιχνίδι πιο ρεαλιστικό και προσφέρει μια ωραία αίσθηση στον παίχτη, είναι να μπορεί ο παίχτης να κάνει ζουμ με τα όπλα του. Το ζουμ του κάθε όπλου γίνεται κρατώντας πατημένο το δεξί κλικ του ποντικιού. Όταν ο παίχτης αφήνει το δεξί κλικ, τότε η οπτική του παίχτη επανέρχεται στο φυσιολογικό. Το script που δημιουργείται εντάσσεται στα components του κάθε όπλου. Η λογική που ακολουθεί στο Σχήμα 4.13, δείχνει τον τρόπο δημιουργίας αυτού του script.

Σχήμα 4.13: *Λογική για το script WeaponZoom*

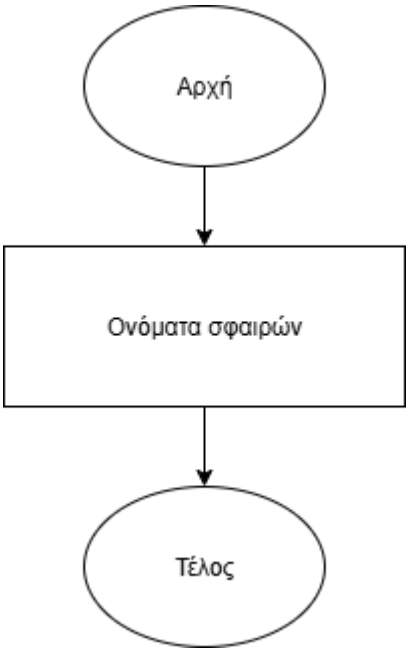
Το ζουμ του κάθε όπλου όμως είναι διαφορετικό, καθώς κάποιο όπλο παρόλο που έχει ενεργοποιημένο το ζουμ, μπορεί να βλέπει πιο κοντά σε σχέση με το όπλο ελεύθερου σκοπευτή Soulpiercer, που έχει το ανώτατο ζουμ από όλα αυτά τα όπλα. Γι' αυτόν τον λόγο οι μεταβλητές του script που σχετίζονται με το ζουμ του κάθε όπλου αλλάζουν για να υπάρχει το ανάλογο αποτέλεσμα.

4.19 Είδη σφαιρών

Όπως έγινε ήδη αναφορά στην αρχή του κεφαλαίου, το κάθε όπλο έχει διαφορετικό είδος σφαιρών. Ο παίχτης για να μπορέσει να χρησιμοποιήσει το κάθε όπλο, πρέπει να βρει τις κατάλληλες σφαίρες για αυτό, σε περίπτωση που δεν έχει άλλες. Ουσιαστικά το κάθε όπλο, όταν πυροβολάει, ξοδεύει έναν αριθμό από τις σφαίρες του και ο παίχτης πρέπει να τις ανανεώνει. Όταν ο παίχτης βρει το συγκεκριμένο είδος, οι σφαίρες προστίθενται στο όπλο. Για την επίτευξη αυτού του σκοπού, πρέπει να δημιουργηθεί ένα script με βάση την λογική που φαίνεται στο Σχήμα 4.14 και να ενταχθεί στα components του παίχτη.

Σχήμα 4.14: *Λογική για το script Ammo*

Στην συνέχεια, πρέπει να δημιουργηθεί ένα ακόμα script που είναι βοηθητικό και περιέχει τους τύπους των σφαιρών. Αυτό το script δημιουργείται ακολουθώντας την λογική που βρίσκεται στο Σχήμα 4.15. Επιπλέον, δεν χρειάζεται να προστεθεί σε κάποιο game object.



Σχήμα 4.15: *Λογική για το script AmmoType*

Πιο αναλυτικά, υπάρχουν οι σφαίρες του πιστολιού Sholo με το όνομα «Pistol Bullets», οι σφαίρες της καραμπίνας Buster με το όνομα «Shotgun Shells», οι σφαίρες του όπλου με λειτουργία ριπής τριών σφαιρών Gorlock με το όνομα «Light Rounds», οι σφαίρες του αυτόματου όπλου Plazmius με το όνομα «Plasma Ammo» και οι σφαίρες του όπλου ελεύθερου σκοπευτή Soulpiercer με το όνομα «Sniper Ammo».

4.20 Σχεδιασμός πίστας (Level design)

Πλέον, έχοντας σχεδόν τα πάντα έτοιμα, φτάνει η στιγμή που πρέπει να γίνει ο σχεδιασμός της πίστας. Όπως ειπώθηκε και στην αρχή, ο παίχτης βρίσκεται μέσα στην νύχτα, βαθιά μέσα στο δάσος. Καθώς ο παίχτης προχωράει όλο και πιο βαθιά μέσα στο δάσος, βρίσκει διάφορα σκόρπια αντικείμενα που υποδηλώνουν ότι εκεί βρίσκονται ή βρίσκονταν άνθρωποι. Κάποια είναι αντικείμενα που δείχνουν ότι γίνονται ή γίνονταν εργασίες σε αυτό το μέρος, κάποια άλλα είναι αντικείμενα που σχετίζονται με τον βιομηχανικό τομέα και κάποια άλλα είναι αντικείμενα τεχνολογίας. Δυστυχώς, δεν είναι ένας απλός περίπατος στο δάσος, βρίσκοντας διάφορα αντικείμενα που υπάρχουν τριγύρω. Στο δάσος είναι διασκορπισμένα πάρα πολλά εχθρικά πλάσματα και των δύο ειδών που αναφέρθηκαν προηγουμένως (Shabumi και Arach). [61][63] Με την πρώτη ευκαιρία αυτά τα πλάσματα κυνηγούν τον παίχτη με σκοπό να τον εξουδετερώσουν. Όσο προχωράει ο παίχτης, άλλο τόσο βρίσκει πλάσματα μπροστά του. Αν ψάξει αρκετά, μπορεί να βρει την είσοδο ενός εγκαταλελειμμένου κτηρίου, που πλέον είναι φωλιά αυτών των πλασμάτων. Όσο προχωράει μέσα στο κτήριο καταλαβαίνει ότι αυτό το κτήριο είναι η είσοδος ενός κρυφού και πλέον εγκαταλελειμμένου ασύλου, μέσα στο οποίο πραγματοποιούνταν παράνομα πειράματα πάνω σε ανθρώπους και σε ζώα/έντομα. Ο Michael Shaw δεν ξέρει τον λόγο αυτών των πειραμάτων, αλλά πρόκειται να τον μάθει, καθώς όλα όσα βλέπει είναι μόνο η κορυφή του παγόβουνου. Για να γίνει αυτό, πρέπει να προχωρήσει όλο και πιο βαθιά στο κτήριο αυτό.

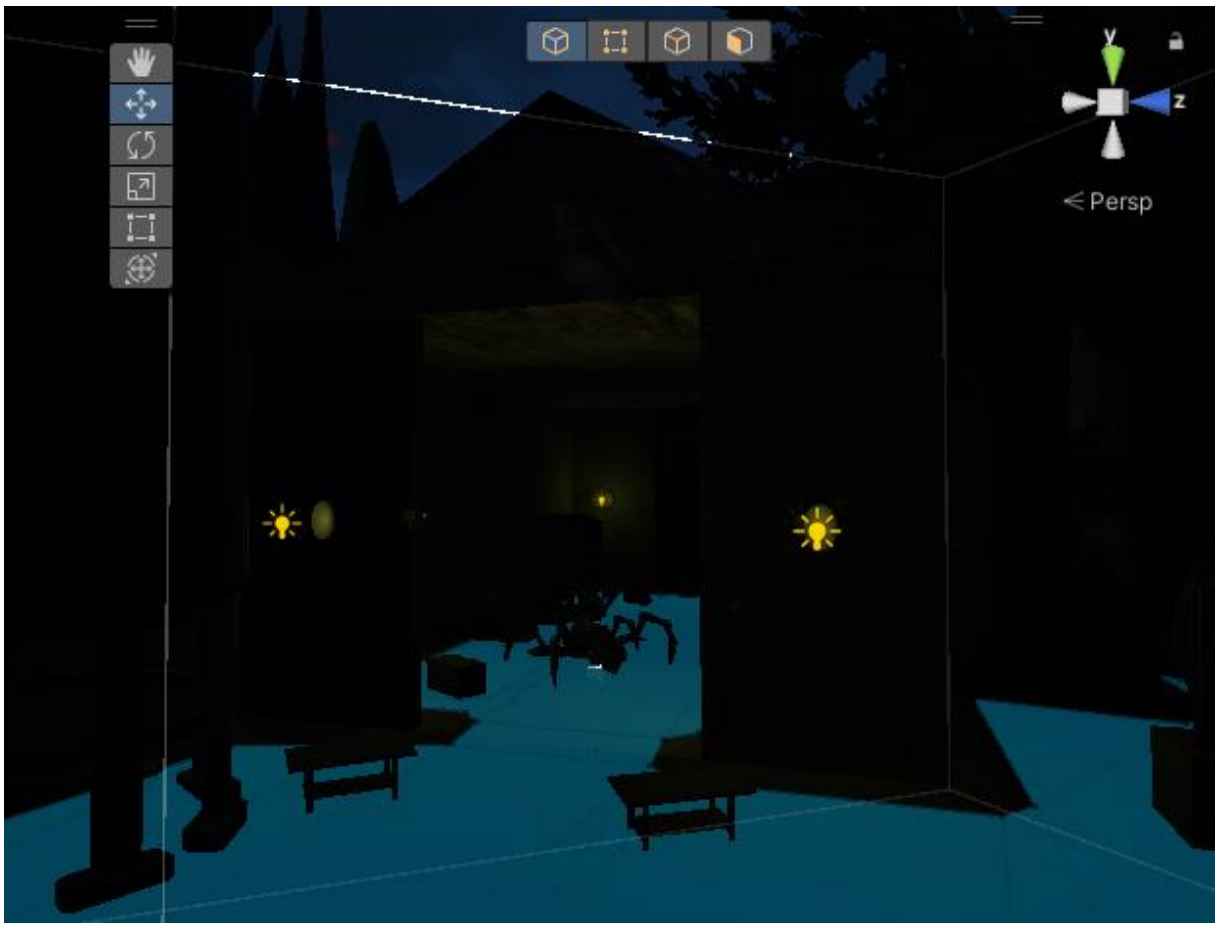
Για την δημιουργία της εισόδου του ασύλου, χρησιμοποιείται το εργαλείο ProBuilder της Unity και για τα φώτα χρησιμοποιείται μία σφαίρα από τα ήδη έτοιμα game objects της Unity, κατάλληλα τροποποιημένη για να μοιάζει οπτικά με φως τοίχου. Σε αυτό το φως τοίχου προστίθεται ένα material με κίτρινο χρώμα και ένα point light για να εκπέμπει φως.

Διασκορπισμένα στο δάσος, χρησιμοποιούνται αντικείμενα από κατάλληλα assets [68][69][70][71][72] που βρίσκονται στο Asset Store. [4]

Όσον αφορά τώρα την είσοδο του ασύλου, χρησιμοποιούνται κάποια assets που χρησιμοποιήθηκαν και σε προηγούμενο στάδιο της ανάπτυξης του παιχνιδιού [54][56][69][68][58][51], αλλά και κάποια νέα. [73][74][75] Για το χρώμα των τοίχων και του πατώματος του κτηρίου, χρησιμοποιείται το κατάλληλο asset [76], για να δώσει ένα ύφος εγκαταλελειμμένου κτηρίου, μέσα στο οποίο έχουν δοθεί μάχες ανάμεσα σε ανθρώπους και σε αυτά τα πλάσματα.

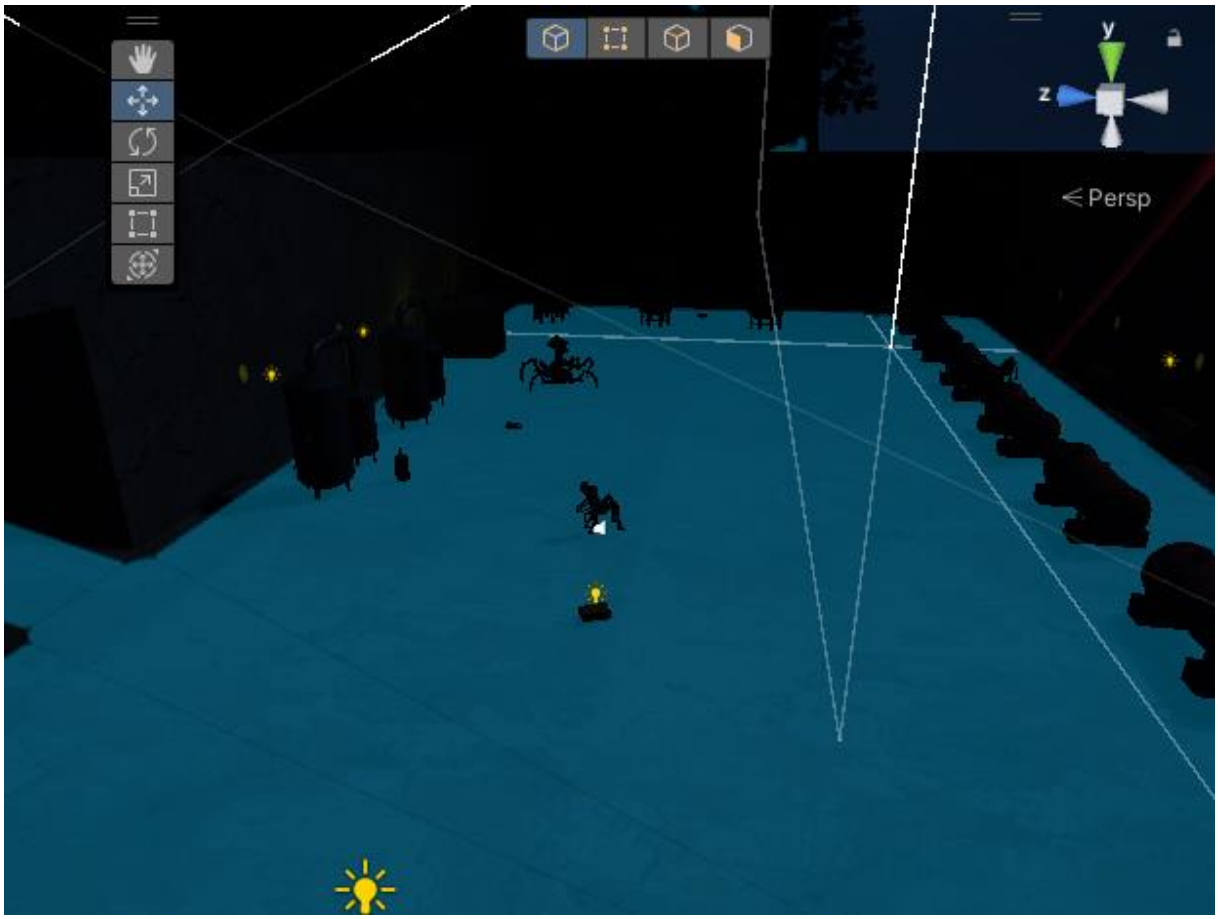
Τώρα είναι η κατάλληλη στιγμή να διασκορπιστούν σε όλη την πίστα, πολλά πλάσματα από τα δύο είδη πλασμάτων που προστέθηκαν στο παιχνίδι, έτσι ώστε ο παίχτης να είναι περιτριγυρισμένος από αυτά. [61][63]

Καθώς το παιχνίδι πρέπει να έχει ένα σκοτεινό ύφος, τροποποιούνται οι μεταβλητές της φωτεινότητας και της ομίχλης που υπάρχουν στην πίστα, για να μην μπορεί να δει πολύ μακριά ο παίχτης.



Εικόνα 4.17: *Είσοδος ασύλου v1*

Εικόνα 4.18: *Είσοδος ασύλου v2*



Εικόνα 4.19: *Είσοδος ασύλου v3*

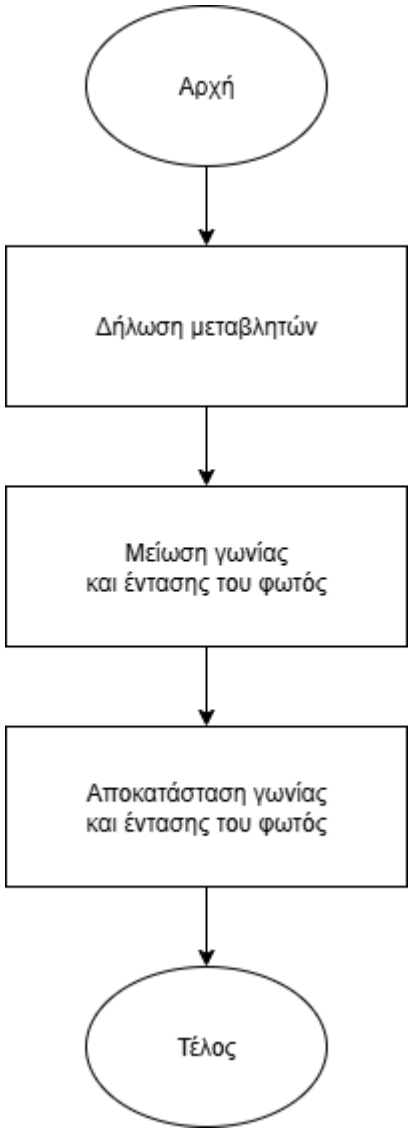
4.21 Οπτικό εφέ βροχής

Κάτι που κάνει το παιχνίδι ακόμα καλύτερο και δίνει πιο σκοτεινό ύφος σε αυτό, είναι η προσθήκη βροχής κατά την διάρκεια της πίστας. Η βροχή είναι ουσιαστικά ένα οπτικό εφέ που δημιουργείται μέσω της Unity και τροποποιείται κατάλληλα για να μοιάζει όσο περισσότερο γίνεται με βροχή. Η βροχή είναι στραμμένη σε γωνία είκοσι μοιρών στον άξονα x και καλύπτει ένα συγκεκριμένο σημείο πάνω από τον παίχτη και όχι όλη την πίστα, για να μειωθούν οι απαιτήσεις της απεικόνισής της. Ακριβώς όμως επειδή βρίσκεται πάνω από τον παίχτη, για να μην φαίνεται ότι κινείται μαζί του, πρέπει να δημιουργηθεί ένα script, το οποίο είναι υπεύθυνο για να κάνει την βροχή να ακολουθάει ομαλά τον παίχτη. Το script εντάσσεται στα components της βροχής και δημιουργείται με την λογική που παρατίθεται στο Σχήμα 4.16.

Σχήμα 4.16: *Λογική για το script LocalRainFollow*

4.22 Φακός παίχτη

Καθώς έχουν γίνει όλες οι απαραίτητες προσθήκες, τώρα είναι η κατάλληλη στιγμή να δοθεί στον παίχτη ένας φακός, ο οποίος είναι το τέταρτο πιο σημαντικό χαρακτηριστικό του παιχνιδιού. Για τον φακό αρχικά πρέπει να δημιουργηθεί ένα spot light, το οποίο ρυθμίζεται κατάλληλα έτσι ώστε να φαίνεται σαν να είναι αυτό το φως του φακού. Όπως αναφέρθηκε και νωρίτερα, το φως του φακού σιγά σιγά όσο περνάει η ώρα, μειώνεται μέχρις ότου δεν υπάρχει καθόλου φως από τον φακό. Η λειτουργία του φακού και το πως εξασθενεί το φως του, υλοποιείται μέσω ενός script, το οποίο δημιουργείται μέσω της λογικής που παρουσιάζεται στο Σχήμα 4.17. Το script αυτό εντάσσεται στα components του φακού.

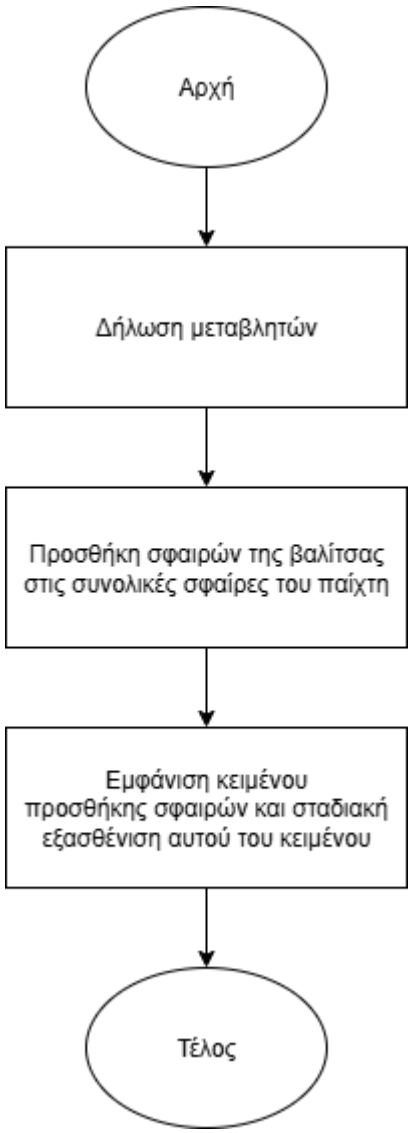


Σχήμα 4.17: *Λογική για το script FlashlightSystem*

4.23 Ανεφοδιασμός σφαιρών και μπαταριών

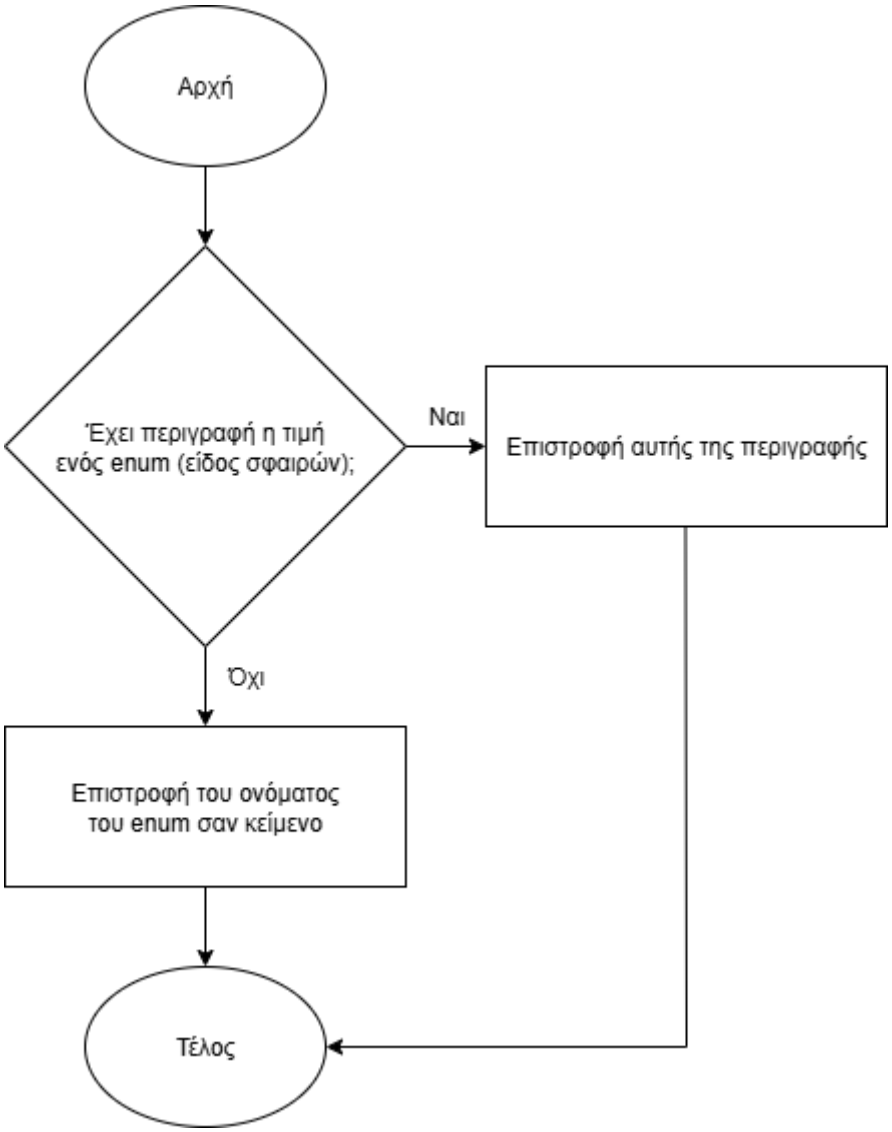
Τώρα είναι η σειρά της προσθήκης του πέμπτου και τελευταίου πιο σημαντικού χαρακτηριστικού του συγκεκριμένου παιχνιδιού, το οποίο είναι ο ανεφοδιασμός σφαιρών για τα όπλα του παίχτη και μπαταριών για τον φακό του. Το asset που χρησιμοποιείται για τον ανεφοδιασμό αυτών των αντικειμένων, είναι μία κλειστή βαλίτσα. [77] Η κάθε βαλίτσα αντιπροσωπεύει κάποιο όπλο. Για παράδειγμα, η βαλίτσα που προσφέρει σφαίρες για το πιστόλι του παίχτη, έχει material που δίνει χρυσό χρώμα στην βαλίτσα, ακριβώς όπως και το πιστόλι. Το ίδιο συμβαίνει και για τα υπόλοιπα όπλα, όπου οι βαλίτσες που σχετίζονται με αυτά, έχουν το ίδιο χρώμα με το βασικό χρώμα που υπάρχει στο κάθε όπλο. Για τις βαλίτσες που χρησιμοποιούνται για να ανεφοδιαστεί ο παίχτης με μπαταρίες για τον φακό, δίνεται ένα material που προσφέρει άσπρο χρώμα. Για να μπορέσει ο παίχτης να ανεφοδιαστεί κατάλληλα, πρέπει να περάσει πάνω από την κάθε βαλίτσα. Αυτό γίνεται δημιουργώντας ένα script για τον ανεφοδιασμό των σφαιρών και ένα για τον ανεφοδιασμό των μπαταριών, αντίστοιχα. Το πρώτο script που αφορά τον ανεφοδιασμό των σφαιρών, προστίθεται στα components της κάθε βαλίτσας που

σχετίζεται με σφαίρες και δημιουργείται με βάση την παρακάτω λογική, που παρουσιάζεται στο Σχήμα 4.18.



Σχήμα 4.18: *Λογική για το script AmmoPickup*

Για να μπορεί ο παίχτης να γνωρίζει με τι σφαίρες ανεφοδιάστηκε, πρέπει να χρησιμοποιηθεί μία διεπαφή χρήστη (user interface (UI)). Γι' αυτόν τον λόγο δημιουργείται ένα κείμενο, το οποίο δείχνει την ποσότητα και το είδος των σφαιρών που πήρε ο παίχτης από την κάθε μία βαλίτσα. Για να μπορέσει όμως ο παίχτης να το δει αυτό το κείμενο με έναν πιο φιλικό προς αυτόν τρόπο, πρέπει να δημιουργηθεί ένα script που λειτουργεί ως βοηθητικό script στο AmmoType enum, αλλά δεν εντάσσεται στα components κάποιου game object. Το script αυτό αφορά την εμφάνιση του ονόματος του είδους των σφαιρών και για να δημιουργηθεί ακολουθεί την λογική που υπάρχει στο Σχήμα 4.19.



Σχήμα 4.19: *Λογική για το script EnumExtensions*

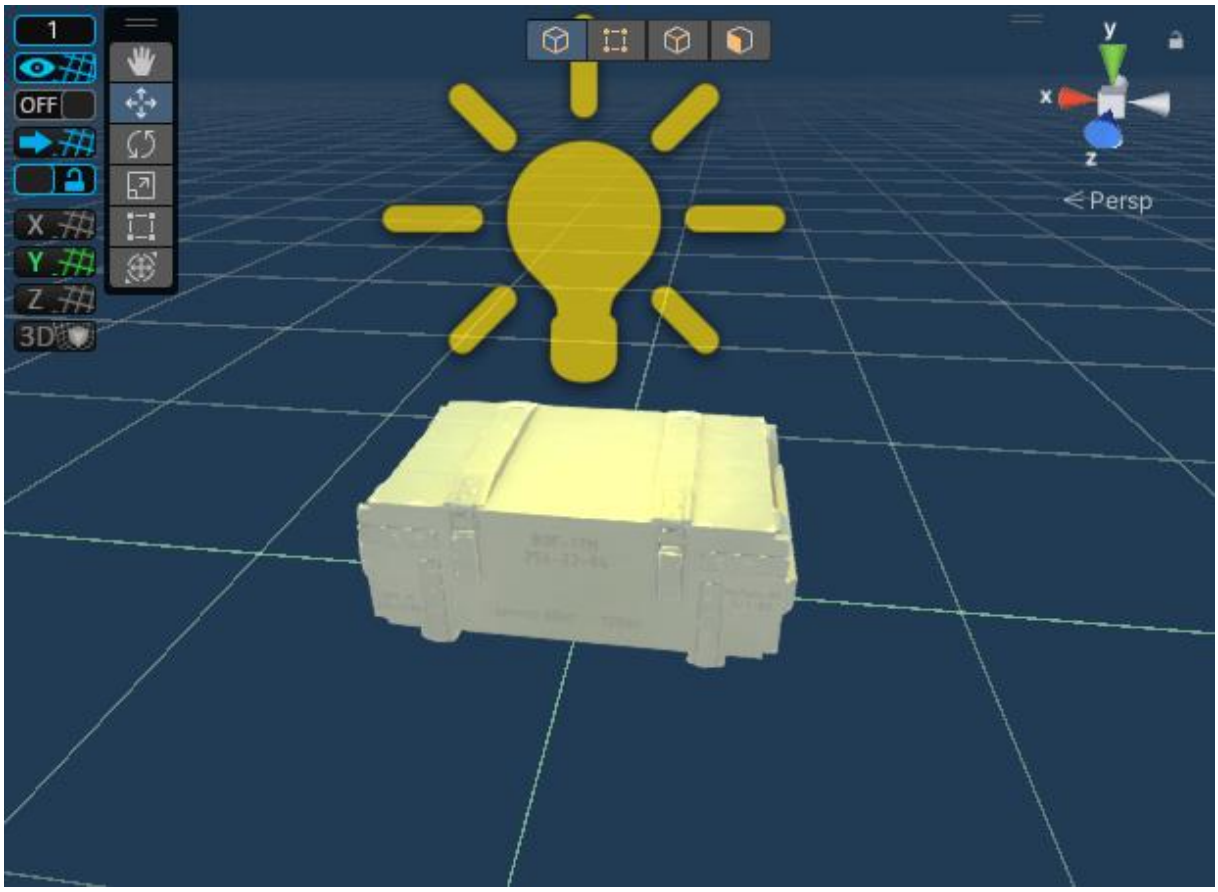
Εικόνα 4.20: *Βαλίτσα με σφαίρες για το Soulpiercer*

Εικόνα 4.21: *Ένδειξη ποσότητας και είδους σφαιρών που πάρθηκαν*

Το δεύτερο script που αφορά τον ανεφοδιασμό των μπαταριών του φακού και ουσιαστικά αποκαθιστά την γωνία και την ένταση του φωτός του φακού, προστίθεται στην βαλίτσα που δίνει μπαταρίες στον παίχτη και για να μπορέσει να υλοποιηθεί, ακολουθάει την παρακάτω λογική που παρουσιάζει το Σχήμα 4.20.

Σχήμα 4.20: *Λογική για το script BatteryPickup*

Αυτό που πρέπει να γίνει τώρα, είναι να διασκορπιστούν σε όλη την πίστα, πολλές από τις βαλίτσες που δίνουν στον παίχτη σφαίρες και μπαταρίες.



Εικόνα 4.22: *Βαλίτσα με μπαταρίες για τον φακό του παίχτη*

4.24 Post processing

Όλα είναι πλέον έτοιμα και ήρθε η στιγμή του post processing για να προστεθούν συγκεκριμένα εφέ που κάνουν ακόμα πιο σκοτεινή την αίσθηση του παιχνιδιού. Το post processing είναι η τελευταία ενέργεια που πρέπει να γίνει για να τελειοποιηθεί η αίσθηση και ο τόνος του παιχνιδιού.

4.25 Τέλος πίστας

Στο τέλος της πίστας, αφού ο παίχτης ψάξει αρκετά, πρέπει να φτάσει στην τελευταία πόρτα που βλέπει στον διάδρομο. Αφού φτάσει κοντά στην πόρτα αυτή, εμφανίζεται ένα μήνυμα το οποίο αναφέρεται στον Michael Shaw, κάνοντας νύξη για το τι ακολουθεί στην συνέχεια του παιχνιδιού. Στο συγκεκριμένο μήνυμα, στο τέλος, υπάρχει και ένα ευχαριστήριο μήνυμα προς τον παίχτη, που τελείωσε αυτό το demo. Για την δημιουργία αυτού του κειμένου και του ευχαριστήριου μηνύματος, πρέπει να δημιουργηθεί ένα script, βάσει της λογικής που εμφανίζεται στο Σχήμα 4.21 και να προστεθεί στο panel με το όνομα «Ending Message Panel».

Σχήμα 4.21: *Λογική για το script EndingMessage*

Αμέσως μετά πρέπει να δημιουργηθεί ένα ακόμα script που εντάσσεται στα componenets ενός διάφανου ορθογώνιου παραλληλεπίπεδου. Όταν ο παίχτης ακουμπήσει το ορθογώνιο παραλληλεπίπεδο, ξεθωριάζει σταδιακά η σκηνή και εμφανίζεται το μήνυμα. Το script δημιουργείται ακολουθώντας την λογική που υπάρχει στο Σχήμα 4.22.

Σχήμα 4.22: *Λογική για το script TriggerScrollingText*

4.26 Επίλογος

Το συγκεκριμένο παιχνίδι έφτασε στο τέλος του και μαζί με αυτό χρησιμοποιήθηκαν πολλά νέα πράγματα, που δεν χρησιμοποιήθηκαν στα προηγούμενα παιχνίδια. Αυτά είναι αρχικά η προσθήκη αρκετών έτοιμων assets από το Unity Asset Store [4] και η χρήση του NavMeshAgent για να μπορούν τα εχθρικά πλάσματα να κυνηγούν τον παίχτη, χρησιμοποιώντας την τεχνητή νοημοσύνη που τους δόθηκε. Επιπροσθέτως, εντάχθηκαν animations στα πλάσματα και κουμπιά που μπορεί να χρησιμοποιήσει ο παίχτης μέσω της διεπαφής του χρήστη (user interface (UI)), όταν χάνει. Στην συνέχεια, χρησιμοποιήθηκε ένα enum για τα είδη των σφαιρών και το εργαλείο ProBuilder της Unity για την δημιουργία της εισόδου του ασύλου. Στο τέλος, όλα αυτά μαζί και με εργαλεία, λειτουργίες και τεχνικές που χρησιμοποιήθηκαν και στα προηγούμενα κεφάλαια, βοήθησαν στην ανάπτυξη του τελευταίου παιχνιδιού της παρούσας διπλωματικής εργασίας.

Κεφάλαιο 5ο: Συμπεράσματα και προτάσεις βελτίωσης

Η παρούσα διπλωματική εργασία είχε ως στόχο την ανάπτυξη τριών τρισδιάστατων πρωτότυπων (demo) παιχνιδιών με τη χρήση της γλώσσας προγραμματισμού C# και της μηχανής παιχνιδιών Unity. Με την ολοκλήρωση της εργασίας αυτής, αφιερώθηκε χρόνος για να αναδειχθεί πόσο έχει εξελιχθεί τα τελευταία χρόνια η τεχνολογία και πιο συγκεκριμένα η βιομηχανία της ανάπτυξης παιχνιδιών. Αφότου καθιερώθηκε ως βιομηχανία, η ανάπτυξη παιχνιδιών παρουσίασε ραγδαία πρόοδο, τόσο σε επίπεδο τεχνικής υλοποίησης, όσο και σε εργαλεία και τεχνολογίες. Επιπλέον, μέσα από την υλοποίηση αυτής της διπλωματικής, αποκτήθηκε ουσιαστική εμπειρία στον σωστό και αποτελεσματικό προγραμματισμό με την γλώσσα προγραμματισμού C#, καθώς και στον σχεδιασμό και την υλοποίηση μηχανισμών gameplay. Καθ' όλη την διάρκεια της ανάπτυξης παρουσιάζονται διαφορετικές τεχνικές προγραμματισμού, όλες με θεμιτό αποτέλεσμα και γίνεται κατανοητός ο τρόπος με τον οποίο οργανώνεται και διαχειρίζεται το παιχνίδι μέσω του κώδικα. Συνολικά, η εργασία απέδειξε ότι είναι δυνατή η ανάπτυξη ολοκληρωμένων πρωτότυπων παιχνιδιών από ένα μόνο άτομο, αξιοποιώντας τα κατάλληλα εργαλεία και τις βασικές γνώσεις προγραμματισμού. Παράλληλα, προσέφερε μια ρεαλιστική εικόνα των προκλήσεων και των αποφάσεων που καλείται να αντιμετωπίσει ένας προγραμματιστής κατά την επαγγελματική ανάπτυξη παιχνιδιών. Παρόλο όμως που η εργασία ολοκληρώθηκε με επιτυχία, υπάρχουν αρκετές δυνατότητες για μελλοντική βελτίωση και εμπλουτισμό του περιεχομένου. Συγκεκριμένα, μπορούσαν να προστεθούν επιπλέον πρωτότυπα, που να αναδεικνύουν την χρήση του εργαλείου Timeline της Unity, για την δημιουργία animations. Επίσης, προτείνεται ο εμπλουτισμός της διεπαφής χρήστη (user interface (UI) με περισσότερα δυναμικά στοιχεία και μηνύματα ανατροφοδότησης προς τον παίχτη. Τέλος, μπορούσε να ενσωματωθεί ένα σύστημα ρυθμίσεων για τον παίχτη, όπως επιλογές για ήχο, γλώσσα και ευαισθησία κάμερας.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Βιβλία

[35] J. Skeet, *C# in Depth*, 4th ed. Shelter Island, NY: Manning, 2019.

Application Note

[36] Ecma International, *ECMA-334: C# Language Specification*, 4th ed., Geneva, Switzerland, Jun. 2006. [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-334_4th_edition_june_2006.pdf. [Accessed: Jun. 7, 2025].

Internet Site

[1] "Video game development," *Wikipedia, The Free Encyclopedia*, [Online]. Available: https://en.wikipedia.org/wiki/Video_game_development. [Accessed: Jun. 7, 2025].

[2] O. Carson, "Comparing Popular Game Engines," *PubNub*, Aug. 15, 2024. [Online]. Available: https://www.pubnub.com/blog/comparing-popular-game-engines/. [Accessed: Jun. 7, 2025].

[3] CB Insights, "The $120B Gaming Industry Is Being Built On The Backs Of These Two Engines," *CB Insights Research*, Sept. 20, 2018. [Online]. Available: https://www.cbinsights.com/research/game-engines-growth-expert-intelligence/. [Accessed: Jun. 7, 2025].

[4] Unity Technologies, "Unity's Asset Store," *Unity Manual*, [Online]. Available: https://docs.unity3d.com/Manual/AssetStore.html. [Accessed: Jun. 7, 2025].

[5] Unity Technologies, "Unity Engine," *Unity*, [Online]. Available: https://unity.com/products/unity-engine. [Accessed: Jun. 7, 2025].

[6] Epic Games, "Unreal Engine," *Unreal Engine*, [Online]. Available: https://www.unrealengine.com/en-US. [Accessed: Jun. 7, 2025].

[7] Godot Engine, "Godot Engine," *Godot Engine*, [Online]. Available: https://godotengine.org/features/. [Accessed: Jun. 7, 2025].

[8] Blender Foundation, "Blender," *Blender*, [Online]. Available: https://www.blender.org/features/. [Accessed: Jun. 7, 2025].

[9] Pixologic, "ZBrush," *Pixologic*, [Online]. Available: https://pixologic.com/features/. [Accessed: Jun. 7, 2025].

[10] Autodesk, "Autodesk 3ds Max," *Autodesk*, [Online]. Available: https://www.autodesk.com/products/3ds-max/overview. [Accessed: Jun. 7, 2025].

[11] Autodesk, "Autodesk Maya," *Autodesk*, [Online]. Available: https://www.autodesk.com/products/maya/overview. [Accessed: Jun. 7, 2025].

[12] Adobe, “Mixamo,” *Adobe*, [Online]. Available: https://www.mixamo.com/. [Accessed: Jun. 7, 2025].

[13] Microsoft, “Visual Studio,” *Microsoft*, [Online]. Available: https://visualstudio.microsoft.com/vs/. [Accessed: Jun. 7, 2025].

[14] Microsoft, “Visual Studio Code,” *Microsoft*, [Online]. Available: https://code.visualstudio.com/. [Accessed: Jun. 7, 2025].

[15] Unity Technologies, “Unity Profiler,” *Unity Manual*, [Online]. Available: https://docs.unity3d.com/Manual/Profiler.html. [Accessed: Jun. 7, 2025].

[16] Epic Games, “Unreal Insights Overview,” *Unreal Engine Documentation*, [Online]. Available: https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/PerformanceAndProfiling/UnrealInsights/Overview/. [Accessed: Jun. 7, 2025].

[17] Godot Engine, “The Profiler,” *Godot Documentation*, [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/debug/the_profiler.html. [Accessed: Jun. 7, 2025].

[18] M. Dealessandri, “What is the best game engine: is Unity right for you?,” *GamesIndustry.biz*, Nov. 23, 2023. [Online]. Available: https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you. [Accessed: Jun. 7, 2025].

[19] S. Axon, “Unity at 10: For better—or worse—game development has never been easier,” *Ars Technica*, Sept. 27, 2016. [Online]. Available: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/. [Accessed: Jun. 7, 2025].

[20] D. Takahashi, “John Riccitiello Q&A: How Unity CEO views Epic's Fortnite success,” *VentureBeat*, Sept. 15, 2018. [Online]. Available: https://venturebeat.com/business/john-riccitiello-interview-how-unity-ceo-views-epics-fortnite-success/. [Accessed: Jun. 7, 2025].

[21] Unity Technologies, “Government & Aerospace,” *Unity*, [Online]. Available: https://unity.com/solutions/government-aerospace. [Accessed: Jun. 7, 2025].

[22] Unity Technologies, “Introduction to scripting,” *Unity Manual*, [Online]. Available: https://docs.unity3d.com/6000.1/Documentation/Manual/intro-to-scripting.html. [Accessed: Jun. 7, 2025].

[23] Unity Technologies, “Programming in Unity,” *Unity*, [Online]. Available: https://unity.com/solutions/programming. [Accessed: Jun. 7, 2025].

[24] Unity Technologies, “Physics,” *Unity Manual*, [Online]. Available: https://docs.unity3d.com/Manual/PhysicsSection.html. [Accessed: Jun. 7, 2025].

[25] Unity Technologies, “Visual Scripting with Bolt,” *Unity Manual*, [Online]. Available: https://docs.unity3d.com/bolt/1.4/manual/index.html. [Accessed: Jun. 7, 2025].

[26] RocketBrush Studio, “Unity vs Unreal Engine: Which One to Choose in 2025,” *RocketBrush Blog*, Aug. 17, 2024. [Online]. Available: https://rocketbrush.com/blog/unity-vs-unreal-engine-which-one-should-you-choose-in-2024. [Accessed: Jun. 7, 2025].

[27] RocketBrush Studio, “Godot Engine vs Unity: Which One Suits You Best in 2025,” *RocketBrush Blog*, Jul. 28, 2024. [Online]. Available: https://rocketbrush.com/blog/godot-vs-unity. [Accessed: Jun. 7, 2025].

[28] A. Liptak, “How Neill Blomkamp and Unity are shaping the future of filmmaking with Adam: The Mirror,” *The Verge*, Nov. 30, 2017. [Online]. Available: https://www.theverge.com/2017/10/4/16409734/unity-neill-blomkamp-oats-studios-mirror-cinemachine-short-film. [Accessed: Jun. 7, 2025].

[29] S. Edelstein, “How gaming company Unity is driving automakers toward virtual reality,” *Digital Trends*, May 17, 2018. [Online]. Available: https://www.digitaltrends.com/cars/unity-automotive-virtual-reality-and-hmi/. [Accessed: Jun. 7, 2025].

[30] V. Savov, “Unity is the little game engine that could revolutionize animated movies,” *The Verge*, Jun. 30, 2017. [Online]. Available: https://www.theverge.com/2017/6/30/15899446/unity-cinemachine-unite-europe-2017-animation. [Accessed: Jun. 7, 2025].

[31] C. Giardina, “Disney Television Animation Launching ‘Big Hero 6’-Themed Shorts,” *The Hollywood Reporter*, Aug. 9, 2018. [Online]. Available: https://www.hollywoodreporter.com/movies/movie-news/disney-television-animation-launching-big-hero-6-themed-shorts-1133450/. [Accessed: Jun. 7, 2025].

[32] G. De Vynck, “Unity Technologies Aims to Bring Video Game Tools Into the Real World,” *Bloomberg*, May 7, 2020. [Online]. Available: https://www.bloomberg.com/news/articles/2020-05-07/unity-technologies-aims-to-bring-video-game-tools-into-the-real-world. [Accessed: Jun. 7, 2025].

[33] A. Oreskovic, “Why EA's former boss believes the 3D tech that powers video games will make way more money outside of gaming,” *Business Insider*, Sept. 14, 2018. [Online]. Available: https://www.businessinsider.com/unity-ceo-john-riccitiello-opportunity-beyond-gaming-2018-9?r=UK&IR=T. [Accessed: Jun. 7, 2025].

[34] S. Captain, “How Google’s DeepMind will train its AI inside Unity’s video game worlds,” *Fast Company*, Sept. 26, 2018. [Online]. Available: https://web.archive.org/web/20180927024638/https://www.fastcompany.com/90240010/deepminds-ai-will-learn-inside-unitys-video-game-worlds. [Accessed: Jun. 7, 2025].

[37] “C Sharp (programming language),” *Wikipedia, The Free Encyclopedia*, [Online]. Available: https://en.wikipedia.org/wiki/C_Sharp_(programming_language). [Accessed: Jun. 7, 2025].

[38] K. Dollard, “Announcing C# 12,” *.NET Blog*, Nov. 14, 2023. [Online]. Available: https://devblogs.microsoft.com/dotnet/announcing-csharp-12/. [Accessed: Jun. 7, 2025].

[39] G. Seth, “Announcing .NET 8,” *.NET Blog*, Nov. 14, 2023. [Online]. Available: https://devblogs.microsoft.com/dotnet/announcing-dotnet-8/. [Accessed: Jun. 7, 2025].

[40] TheLittleCrow, “Rocket Launch Boost and Burning (Version B),” *Freesound*, Dec. 11, 2024. [Online]. Available: https://freesound.org/people/TheLittleCrow/sounds/774270/. [Accessed: Jun. 7, 2025].

[41] lowve, “UI Notification - Success,” *Freesound*, Dec. 2, 2024. [Online]. Available: https://freesound.org/people/lowve/sounds/772277/. [Accessed: Jun. 7, 2025].

[42] SamsterBirdies, “Astronomical Explosion,” *Freesound*, Oct. 19, 2024. [Online]. Available: https://freesound.org/people/SamsterBirdies/sounds/760509/. [Accessed: Jun. 7, 2025].

[43] rpgwhitelock, “AllSky Free - 10 Sky / Skybox Set,” *Unity Asset Store*, May 17, 2019. [Online]. Available: https://assetstore.unity.com/packages/2d/textures-materials/sky/allsky-free-10-sky-skybox-set-146014. [Accessed: Jun. 7, 2025].

[44] Dogmatic, “*FREE* Skyboxes - Space,” *Unity Asset Store*, Dec. 10, 2020. [Online]. Available: https://assetstore.unity.com/packages/2d/textures-materials/sky/free-skyboxes-space-178953. [Accessed: Jun. 7, 2025].

[45] PULSAR BYTES, “WorldSkies Free Skybox Pack,” *Unity Asset Store*, Apr. 13, 2017. [Online]. Available: https://assetstore.unity.com/packages/2d/textures-materials/sky/worldskies-free-skybox-pack-86517. [Accessed: Jun. 7, 2025].

[46] BG Studio, “Free HDR Skyboxes Pack,” *Unity Asset Store*, Jul. 23, 2020. [Online]. Available: https://assetstore.unity.com/packages/2d/textures-materials/sky/free-hdr-skyboxes-pack-175525?srsltid=AfmBOopDT-sporyGluy3lmzrjWw1I8yz3jKmBHU6TBaJDHUy5pe3G77t. [Accessed: Jun. 7, 2025].

[47] “Mercury (planet),” *Wikipedia, The Free Encyclopedia*, [Online]. Available: https://en.wikipedia.org/wiki/Mercury_(planet). [Accessed: Jun. 7, 2025].

[48] “Venus,” *Wikipedia, The Free Encyclopedia*, [Online]. Available: https://en.wikipedia.org/wiki/Venus. [Accessed: Jun. 7, 2025].

[49] “Mars,” *Wikipedia, The Free Encyclopedia*, [Online]. Available: https://en.wikipedia.org/wiki/Mars. [Accessed: Jun. 7, 2025].

[50] IL.ranch, “Dream Forest Tree,” *Unity Asset Store*, Dec. 7, 2021. [Online]. Available: https://assetstore.unity.com/packages/3d/vegetation/trees/dream-forest-tree-105297. [Accessed: Jun. 20, 2025].

[51] NZ Bullet Studio, “Weapon NZ PT-9M Pistol,” *Unity Asset Store*, Oct. 24, 2022. [Online]. Available: https://assetstore.unity.com/packages/3d/props/weapons/weapon-nz-pt-9m-pistol-227889. [Accessed: Jun. 20, 2025].

[52] LeMudCrab, “Pistol Shot,” *Freesound*, Aug. 21, 2012. [Online]. Available: https://freesound.org/people/LeMudCrab/sounds/163456/. [Accessed: Jun. 20, 2025].

[53] Jean Moreno, “War FX,” *Unity Asset Store*, Mar. 21, 2019. [Online]. Available: https://assetstore.unity.com/packages/vfx/particles/war-fx-5669. [Accessed: Jun. 20, 2025].

[54] MLX, “Shotgun Sci-Fi Weapon,” *Unity Asset Store*, Oct. 5, 2017. [Online]. Available: https://assetstore.unity.com/packages/3d/props/guns/shotgun-sci-fi-weapon-100194. [Accessed: Jun. 20, 2025].

[55] hyperix6, “Shotgun Fire,” *Freesound*, Nov. 20, 2022. [Online]. Available: https://freesound.org/people/hyperix6/sounds/660299/. [Accessed: Jun. 20, 2025].

[56] HYPERION studio, “Sci-Fi Gun Light,” *Unity Asset Store*, Feb. 10, 2025. [Online]. Available: https://assetstore.unity.com/packages/3d/props/guns/sci-fi-gun-light-87916. [Accessed: Jun. 20, 2025].

[57] ngphil22, “m16 triburst.mp3,” *Freesound*, Apr. 13, 2014. [Online]. Available: https://freesound.org/people/ngphil22/sounds/233321/. [Accessed: Jun. 20, 2025].

[58] Pirrateam, “Plasma Guns Set Lite (SMG),” *Unity Asset Store*, Sep. 29, 2020. [Online]. Available: https://assetstore.unity.com/packages/3d/props/guns/plasma-guns-set-lite-smg-179430. [Accessed: Jun. 20, 2025].

[59] MAC, “MAC Sci-Fi Sniper Rifle,” *Unity Asset Store*, Feb. 8, 2021. [Online]. Available: https://assetstore.unity.com/packages/3d/props/weapons/mac-sci-fi-sniper-rifle-188585. [Accessed: Jun. 20, 2025].

[60] LilMati, “Shotgun Shot 03.wav,” *Freesound*, Jun. 1, 2019. [Online]. Available: https://freesound.org/people/LilMati/sounds/473846/. [Accessed: Jun. 20, 2025].

[61] LB3D, “Insectoid Crab Monster: Lurker of the Shores (20 Animations),” *Unity Asset Store*, Jan. 25, 2021. [Online]. Available: https://assetstore.unity.com/packages/3d/characters/insectoid-crab-monster-lurker-of-the-shores-20-animations-107223. [Accessed: Jun. 20, 2025].

[62] Darsycho, “Angry spider monster,” *Freesound*, Feb. 5, 2020. [Online]. Available: https://freesound.org/people/Darsycho/sounds/505185/. [Accessed: Jun. 20, 2025].

[63] GrigoriyArx, “Fantastic Creature #1,” *Unity Asset Store*, Nov. 3, 2017. [Online]. Available: https://assetstore.unity.com/packages/3d/characters/creatures/fantastic-creature-1-103074. [Accessed: Jun. 20, 2025].

[64] ztrees1, “Monster Growl.wav,” *Freesound*, Nov. 16, 2011. [Online]. Available: https://freesound.org/people/ztrees1/sounds/134931/. [Accessed: Jun. 20, 2025].

[65] ttttrr, “hurts.wav,” *Freesound*, Mar. 19, 2020. [Online]. Available: https://freesound.org/people/ttttrr/sounds/509566/. [Accessed: Jun. 20, 2025].

[66] Clean Shirt Labs, “Too Many Crosshairs,” *Unity Asset Store*, May 2, 2022. [Online]. Available: https://assetstore.unity.com/packages/2d/gui/icons/too-many-crosshairs-126069. [Accessed: Jun. 20, 2025].

[67] “Blood Splatter PNG Background - Pngsource,” *ngsource*, [Online]. Available: https://pngsource.in/download/blood-splatter-png-background-pngsource-8zwzagqj. [Accessed: Jun. 20, 2025].

[68] Osmanov, “Industrial Props PBR,” *Unity Asset Store*, Jan. 9, 2019. [Online]. Available: https://assetstore.unity.com/packages/3d/props/industrial/industrial-props-pbr-130438. [Accessed: Jul. 7, 2025].

[69] Abandoned World, “Industrial Tanks Package,” *Unity Asset Store*, Jul. 24, 2018. [Online]. Available: https://assetstore.unity.com/packages/3d/props/industrial/industrial-tanks-package-122228. [Accessed: Jul. 7, 2025].

[70] Sagital3D, “SCI-Fi Battery Pack Free,” *Unity Asset Store*, Dec. 16, 2015. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-battery-pack-free-19738. [Accessed: Jul. 7, 2025].

[71] Dmitrii Kutsenko, “RPG/FPS Game Assets for PC/Mobile (Industrial Set v1.0),” *Unity Asset Store*, Aug. 5, 2019. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/industrial/rpg-fps-game-assets-for-pc-mobile-industrial-set-v1-0-87024. [Accessed: Jul. 7, 2025].

[72] Duane's Mind, “Industrial Storage Tanks,” *Unity Asset Store*, Sep. 8, 2017. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/industrial/industrial-storage-tanks-6276. [Accessed: Jul. 7, 2025].

[73] Lukas Bobor, “Industrial Machine Models,” *Unity Asset Store*, Mar. 26, 2019. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/industrial/industrial-machine-models-66217. [Accessed: Jul. 7, 2025].

[74] voynov_v, “Transformer substation booth,” *Unity Asset Store*, Jun. 12, 2023. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/industrial/transformer-substation-booth-253838. [Accessed: Jul. 7, 2025].

[75] FoeJofMay, “Hospital Horror Pack,” *Unity Asset Store*, Aug. 20, 2015. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/hospital-horror-pack-44045. [Accessed: Jul. 7, 2025].

[76] IndieChest, “One-Click Wall Materials,” *Unity Asset Store*, Apr. 21, 2018. [Online]. Available: https://assetstore.unity.com/packages/2d/textures-materials/one-click-wall-materials-117123. [Accessed: Jul. 7, 2025].

[77] MyxerMan, “Ammo Crate Wood Ammunition Box,” *Unity Asset Store*, Sep. 15, 2017. [Online]. Available: https://assetstore.unity.com/packages/3d/props/weapons/ammo-crate-wood-ammunition-box-90071. [Accessed: Jul. 7, 2025].

ΠΑΡΑΡΤΗΜΑ: ΚΩΔΙΚΑΣ C#

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

public class Mover : MonoBehaviour
{
    [SerializeField] float moveSpeed = 10f;

    // Start is called before the first frame update
    void Start()
    {
        PrintInstructions();
    }

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

    void PrintInstructions()
    {
        Debug.Log("Welcome to the game!");
        Debug.Log("Move your player with WASD or arrow keys.");
        Debug.Log("Don't hit the walls!");
        Debug.Log("Reach the finish before time runs out!");
    }

    void MovePlayer()
    {
        float xValue = Input.GetAxis("Horizontal") * Time.deltaTime *
moveSpeed;
        float zValue = Input.GetAxis("Vertical") * Time.deltaTime * moveSpeed;

        transform.Translate(xValue,0,zValue);
    }
}
```

Mover script

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

public class ObjectHit : MonoBehaviour
{
```

```
    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            GetComponent<MeshRenderer>().material.color = Color.red;
        }
    }
}
```

ObjectHit script

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

public class Scorer : MonoBehaviour
{
    int hits = 0;
    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag != "Hit" && collision.gameObject.tag != 
"Finish" && collision.gameObject.tag != "Start")
        {
            hits++;
            collision.gameObject.tag = "Hit";
            Debug.Log("You've bumped into a thing this many times: " + hits);
        }
    }
}
```

Scorer script

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

public class Dropper : MonoBehaviour
{
    MeshRenderer meshRenderer;
    Rigidbody rigidBody;
    [SerializeField] float timeToWait = 5f; // Time to wait before Dropper 
falls

    // Start is called before the first frame update
    void Start()
    {
        meshRenderer = GetComponent<MeshRenderer>();
        rigidBody = GetComponent<Rigidbody>();
        meshRenderer.enabled = false;
```

```
        rigidBody.useGravity = false;
    }

    // Update is called once per frame
    void Update()
    {
        if (Time.time > timeToWait)
        {
            meshRenderer.enabled = true;
            rigidBody.useGravity = true;
        }
    }
}
```

Dropper script

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

public class Spinner : MonoBehaviour
{
    [SerializeField] float xAngle = 0f;
    [SerializeField] float yAngle = 0f;
    [SerializeField] float zAngle = 0f;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(xAngle,yAngle,zAngle);
    }
}
```

Spinner script

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

public class EndGameCondition : MonoBehaviour
{
    [SerializeField] float timer = 60f;
```

```
    [SerializeField] TextMeshProUGUI countdownText;
    [SerializeField] ThankYouMessage thankYouMessage;

    bool gameEnded = false;
    float pulseTimer = 0f;
    int lastSecond = 0;

    void Update()
    {
        if (gameEnded) return;

        timer -= Time.deltaTime;

        if (timer <= 0f)
        {
            timer = 0f;
            LoseGame();
        }

        UpdateCountdownUI();
        UpdateTimerStyle();
    }

    void UpdateCountdownUI()
    {
        if (!gameEnded)
        {
            int minutes = Mathf.FloorToInt(timer / 60f);
            int seconds = Mathf.FloorToInt(timer % 60f);
            countdownText.text = string.Format("{0:00}:{1:00}", minutes,
seconds);
        }
    }

    void UpdateTimerStyle()
    {
        int currentSecond = Mathf.FloorToInt(timer);

        if (currentSecond <= 10)
        {
            countdownText.color = Color.red;

            if (currentSecond != lastSecond)
            {
                pulseTimer = 0f;
                lastSecond = currentSecond;
            }

            pulseTimer += Time.deltaTime;
```

```
            float fontSize = Mathf.Lerp(80f, 100f, pulseTimer / 1f);
            countdownText.fontSize = fontSize;
        }
    }

    void OnCollisionEnter(Collision collision)
    {
        if (gameEnded) return;

        switch (collision.gameObject.tag)
        {
            case "Finish":
                WinGame();
                break;
        }
    }

    void LoseGame()
    {
        gameEnded = true;

        countdownText.text = "Time's up!";
        countdownText.color = Color.red;
        countdownText.fontSize = 110f;

        // Disable movement
        Mover mover = GetComponent<Mover>();
        if (mover != null)
        {
            mover.enabled = false;
        }

        // Force stop movement instantly
        transform.Translate(Vector3.zero);

        StartCoroutine(ShowThankYouMessage());
    }

    void WinGame()
    {
        gameEnded = true;

        countdownText.text = "You won!";
        countdownText.fontSize = 110f;

        // Disable movement
        Mover mover = GetComponent<Mover>();
        if (mover != null)
        {
```

```
            mover.enabled = false;
        }

        transform.Translate(Vector3.zero);

        StartCoroutine(ShowThankYouMessage());
    }

    IEnumerator ShowThankYouMessage()
    {
        yield return new WaitForSeconds(1f);

        StartCoroutine(thankYouMessage.FadeOut());
    }
}
```

EndGameCondition script

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

public class ThankYouMessage : MonoBehaviour
{
    [SerializeField] float fadeDuration = 1f;

    private CanvasGroup canvasGroup;

    private void Awake()
    {
        canvasGroup = GetComponent<CanvasGroup>();

        gameObject.SetActive(false);
    }

    void Start()
    {
        FadeOut();
    }

    public IEnumerator FadeOut()
    {
        gameObject.SetActive(true); // Reactivate before fading

        canvasGroup.blocksRaycasts = true;
        float timer = 0f;
        while (timer < fadeDuration)
        {
            timer += Time.deltaTime;
```

```
            canvasGroup.alpha = timer / fadeDuration;
            yield return null;
        }
        canvasGroup.alpha = 1;
    }
}
```

ThankYouMessage script

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

public class Movement : MonoBehaviour
{
    // PARAMETERS - for tuning, typically set in the editor ([SerializeField]
and so on)
    // CACHE - e.g. references for readability or speed (Cache the rigidbody
and the audio source)
    // STATE - private instance (member) variables (bool variables, like bool
isAlive)

    [SerializeField] float mainThrust = 100f;
    [SerializeField] float rotationThrust = 70f;
    [SerializeField] AudioClip mainEngine;
    [SerializeField] ParticleSystem mainThrusterParticles;
    [SerializeField] ParticleSystem leftThrusterParticles;
    [SerializeField] ParticleSystem rightThrusterParticles;

    Rigidbody rb; // rb stands for rigidbody
    AudioSource audioSource;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody>();
        audioSource = GetComponent<AudioSource>();
    }

    // Update is called once per frame
    void Update()
    {
        ProcessThrust();
        ProcessRotation();
    }
    void ProcessThrust()
    {
        if (Input.GetKey(KeyCode.Space))
        {
```

```
            StartThrusting();
        }
        else
        {
            StopThrusting();
        }
    }

    void ProcessRotation()
    {
        if (Input.GetKey(KeyCode.A))
        {
            RotateLeft();
        }
        else if (Input.GetKey(KeyCode.D))
        {
            RotateRight();
        }
        else
        {
            StopRotating();
        }
    }

    void StartThrusting()
    {
        rb.AddRelativeForce(Vector3.up * mainThrust * Time.deltaTime);
        if (!audioSource.isPlaying)
        {
            audioSource.PlayOneShot(mainEngine);
        }
        if (!mainThrusterParticles.isPlaying)
        {
            mainThrusterParticles.Play();
        }
    }

    void StopThrusting()
    {
        audioSource.Stop();
        mainThrusterParticles.Stop();
    }

    void RotateLeft()
    {
        ApplyRotation(rotationThrust);
        if (!rightThrusterParticles.isPlaying)
        {
            rightThrusterParticles.Play();
```

```
        }
    }

    void RotateRight()
    {
        ApplyRotation(-rotationThrust);
        if (!leftThrusterParticles.isPlaying)
        {
            leftThrusterParticles.Play();
        }
    }

    void StopRotating()
    {
        rightThrusterParticles.Stop();
        leftThrusterParticles.Stop();
    }

    void ApplyRotation(float rotationThisFrame)
    {
        rb.freezeRotation = true; // freezing rotation so I can manually
rotate
        transform.Rotate(Vector3.forward * rotationThisFrame *
Time.deltaTime);
        rb.freezeRotation = false; // unfreezing rotation so the physics
system can take over
    }
}
```

Movement script

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

public class CollisionHandler : MonoBehaviour
{
    [SerializeField] AudioClip success;
    [SerializeField] AudioClip crash;
    [SerializeField] ParticleSystem successParticles;
    [SerializeField] ParticleSystem crashParticles;
    [SerializeField] ScreenFader screenFader;

    AudioSource audioSource;
    Rigidbody rb; // rb stands for rigidbody

    bool isTransitioning = false; // It keeps track of whether the rocket is
already in a win or crash sequence. If we are transitioning, then don't do
```

```
something (e.g. do not play collision audio or do not allow the player to move
the rocket). It's used to prevent multiple triggers (like winning or crashing
twice, or triggering them while transitioning between levels)
    bool collisionDisabled = false;

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        rb = GetComponent<Rigidbody>();

        if (GameState.cameFromFinalLevel)
        {
            enabled = false; // Disable this script entirely
        }
    }
    void Update()
    {
        if (isTransitioning || GameState.cameFromFinalLevel)
        {
            return;
        }

        RespondToDebugKeys();
    }

    void RespondToDebugKeys()
    {
        if (Input.GetKeyDown(KeyCode.L))
        {
            StartCoroutine(FadeAndLoadNextLevel());
        }
        else if (Input.GetKeyDown(KeyCode.C))
        {
            collisionDisabled = !collisionDisabled; // This will toggle
collision

            if (collisionDisabled)
            {
                // Lock position in Z (so it can't be pushed forward/backward)
                // Lock rotation in X and Y (so it can only rotate on Z)
                rb.constraints = RigidbodyConstraints.FreezePositionZ |
                                 RigidbodyConstraints.FreezeRotationX |
                                 RigidbodyConstraints.FreezeRotationY;
            }
            else
            {
                // Keep Z position locked but allow X/Y rotation again
                rb.constraints = RigidbodyConstraints.FreezePositionZ;
            }
```

```
        }
        else if (Input.GetKeyDown(KeyCode.R))
        {
            StartCoroutine(FadeAndReloadCurrentLevel());
        }
    }

    void OnCollisionEnter(Collision collision)
    {
        if (isTransitioning)
        {
            return;
        }

        // Always allow Friendly tag to print
        if (collision.gameObject.CompareTag("Friendly"))
        {
            Debug.Log("This thing is friendly.");
            return;
        }

        // Always allow Finish pad to behave correctly
        if (collision.gameObject.CompareTag("Finish"))
        {
            if (collisionDisabled)
            {
                Debug.Log("Finish pad touched, but collision is disabled.
Ignoring.");
                return;
            }

            Debug.Log("Non-thruster part touched the finish. Crashing...");
            StartCrashSequence();
            return;
        }

        // For all other objects
        if (collisionDisabled)
        {
            Debug.Log("Collision disabled — touched obstacle but no crash.");
            return;
        }

        // Only crash on unknown objects if collision is enabled
        StartCrashSequence();
    }

    void ReloadCurrentLevel()
    {
```

```
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }

    void LoadNextLevel()
    {
        int currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
        int nextSceneIndex = currentSceneIndex + 1;

        if (nextSceneIndex == SceneManager.sceneCountInBuildSettings)
        {
            GameState.cameFromFinalLevel = true;
            nextSceneIndex = 0;
        }

        SceneManager.LoadScene(nextSceneIndex); // This also triggers FadeIn
from ScreenFader
    }

    void StartCrashSequence()
    {
        isTransitioning = true;
        audioSource.Stop();
        audioSource.PlayOneShot(crash);
        crashParticles.Play();
        GetComponent<Movement>().enabled = false;
        StartCoroutine(FadeAndReloadCurrentLevel());
    }

    void StartSuccessSequence()
    {
        isTransitioning = true;
        audioSource.Stop();
        audioSource.PlayOneShot(success);
        successParticles.Play();
        GetComponent<Movement>().enabled = false;
        StartCoroutine(FadeAndLoadNextLevel());
    }

    public void TriggerWinFromThruster()
    {
        if (isTransitioning)
        {
            return; // If the rocket is already transitioning (or debugging is
on), then skip this logic
        }

        StartSuccessSequence();
    }
```

```
    IEnumerator FadeAndReloadCurrentLevel()
    {
        int currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
        yield return screenFader.FadeOut(currentSceneIndex);

        ReloadCurrentLevel();
    }

    IEnumerator FadeAndLoadNextLevel()
    {
        int currentSceneIndex = SceneManager.GetActiveScene().buildIndex;
        int nextSceneIndex = currentSceneIndex + 1;

        if (nextSceneIndex == SceneManager.sceneCountInBuildSettings)
        {
            nextSceneIndex = 0; // Loop back to scene 0
        }

        yield return screenFader.FadeOut(nextSceneIndex);

        LoadNextLevel();
    }
}
```

CollisionHandler script

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

public class ThrusterCollisionDetector : MonoBehaviour
{
    private CollisionHandler collisionHandler; // Declare a private variable named collisionHandler of type CollisionHandler. The variable collisionHandler is a reference to an object (a component) that uses the class CollisionHandler

    void Start()
    {
        collisionHandler = GetComponentInParent<CollisionHandler>(); // Goes to parent object (Rocket) and finds a component of type CollisionHandler. It stores that found component in the collisionHandler variable
    }

    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Finish") && collisionHandler != null)
        {
            Debug.Log("Main Thruster touched Finish. Winning...");
            collisionHandler.TriggerWinFromThruster();
```

```
        }
    }
}
```

ThrusterCollisionDetector script

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

public class ScreenFader : MonoBehaviour
{
    [SerializeField] float fadeDuration = 2f;
    [SerializeField] ThankYouMessage thankYouMessage;

    CanvasGroup canvasGroup;

    void Awake()
    {
        if (FindObjectsOfType<ScreenFader>().Length > 1)
        {
            Destroy(gameObject); // Avoid duplicates
            return;
        }

        canvasGroup = GetComponent<CanvasGroup>();
    }

    void Start()
    {
        StartCoroutine(FadeIn()); // Always fade in after scene load
    }

    public IEnumerator FadeIn()
    {
        Debug.Log("FadeIn started");
        float timer = 0f;
        while (timer < fadeDuration)
        {
            timer += Time.deltaTime;
            canvasGroup.alpha = 1 - (timer / fadeDuration);
            yield return null;
        }
        canvasGroup.alpha = 0;
        canvasGroup.blocksRaycasts = false; // Make UI click-through
        Debug.Log("FadeIn ended");

        if (SceneManager.GetActiveScene().buildIndex == 0 &&
```

```
            GameState.cameFromFinalLevel &&
            !GameState.alreadyShown)
        {
            GameState.alreadyShown = true;
            StartCoroutine(ShowThankYouMessage());
        }
    }

    public IEnumerator FadeOut(int sceneIndex)
    {
        Debug.Log("FadeOut started");
        canvasGroup.blocksRaycasts = true;
        float timer = 0f;
        while (timer < fadeDuration)
        {
            timer += Time.deltaTime;
            canvasGroup.alpha = timer / fadeDuration;
            yield return null;
        }
        canvasGroup.alpha = 1;
        Debug.Log("FadeOut ended");

        SceneManager.LoadScene(sceneIndex);
    }

    IEnumerator ShowThankYouMessage()
    {
        if (thankYouMessage != null)
        {
            yield return StartCoroutine(thankYouMessage.FadeIn());
        }

        GameState.cameFromFinalLevel = false;

        Time.timeScale = 0f; // Fully pause the game after showing message
    }
}
```

ScreenFader script

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

public class ThankYouMessage : MonoBehaviour
{
    [SerializeField] float fadeDuration = 1f;

    private CanvasGroup canvasGroup;
```

```
    private void Awake()
    {
        canvasGroup = GetComponent<CanvasGroup>();
        canvasGroup.alpha = 0f; // Fully transparent initially
        canvasGroup.blocksRaycasts = false;

        gameObject.SetActive(true);
    }

    public IEnumerator FadeIn()
    {
        float timer = 0f;
        canvasGroup.blocksRaycasts = true;
        while (timer < fadeDuration)
        {
            timer += Time.deltaTime;
            canvasGroup.alpha = timer / fadeDuration;
            yield return null;
        }
        canvasGroup.alpha = 1;
    }
}
```

ThankYouMessage script

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

public class GameState : MonoBehaviour
{
    public static bool cameFromFinalLevel = false;
    public static bool alreadyShown = false;
}
```

GameState script

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

public class ObstacleRotation : MonoBehaviour
{
    [SerializeField] float xAngle = 0f;
    [SerializeField] float yAngle = 0f;
    [SerializeField] float zAngle = 0f;

    // Start is called before the first frame update
```

```
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(xAngle,yAngle,zAngle);
    }
}
```

ObstacleRotation script

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

public class ObstacleOscillation : MonoBehaviour
{
    [SerializeField] Vector3 movementVector;
    [SerializeField] float period = 2f;

    Vector3 startingPosition;
    float movementFactor;

    // Start is called before the first frame update
    void Start()
    {
        startingPosition = transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        if (period <= Mathf.Epsilon)
        {
            return;
        }

        float cycles = Time.time / period; // Variable "cycles" counts how
many complete laps have been done around our circle and it continually grows
over time

        const float tau = Mathf.PI * 2; // Tau is a constant value of 2 * pi
(6.283)
        float rawSinWave = Mathf.Sin(cycles * tau); // Going from -1 to 1
```

```
        movementFactor = (rawSinWave + 1f) / 2f; // Recalculated to go from 0
to 1

        Vector3 offset = movementVector * movementFactor;
        transform.position = startingPosition + offset;
    }
}
```

ObstacleOscillation script

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

public class SinusoidalWaveMovement : MonoBehaviour
{
    [SerializeField] float moveSpeed = 5f; // This is responsible for
horizontal speed
    [SerializeField] float frequency = 5f; // This represents how fast the
game object goes up and down in certain amount of time
    [SerializeField] float amplitude = 0.5f; // This implies the difference
between the peak and the equilibrium position
    [SerializeField] float travelDistance = 5f; // How far to move left/right
before returning

    Vector3 startingPosition;

    bool goingForward = true;
    float traveled = 0f;

    void Start()
    {
        startingPosition = transform.position;
    }

    void Update()
    {
        float direction = goingForward ? 1f : -1f;

        float deltaX = direction * moveSpeed * Time.deltaTime;
        traveled += Mathf.Abs(deltaX);

        float yOffset = Mathf.Sin(Time.time * frequency) * amplitude;

        Vector3 newPosition = transform.position + new Vector3(deltaX, yOffset
- transform.position.y + startingPosition.y, 0);
        transform.position = newPosition;

        if (traveled >= travelDistance)
```

```
        {
            goingForward = !goingForward;
            traveled = 0f;
        }
    }
}
```

SinusoidalWaveMovement script

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

public class ObstacleTeleportation : MonoBehaviour
{
    public static List<ObstacleTeleportation> activeObstacles = new
List<ObstacleTeleportation>(); // A shared list to keep track of all active
obstacles in the scene

    [SerializeField] public float rangeX1 = -20f;
    [SerializeField] public float rangeX2 = -15f;
    [SerializeField] public float rangeY1 = 5f;
    [SerializeField] public float rangeY2 = 10f;
    [SerializeField] public float waitTime = 2f; // It is measured in seconds
    [SerializeField] public float shrinkSpeed = 2f; // The higher the value,
the faster the obstacle shrinks
    [SerializeField] public float growSpeed = 2f; // The higher the value, the
faster the obstacle grows
    [SerializeField] public float minDistanceFromOthers = 3f;

    private Vector3 targetScale; // The original size of the sphere (used for
shrinking/growing)
    private bool isVisible = false; // Whether the sphere is currently active
and taking up space in the scene (used in spacing checks)

    void Start()
    {
        targetScale = transform.localScale; // Save original size
        activeObstacles.Add(this); // Add this obstacle to the shared list
        StartCoroutine(TeleportCycle()); // Begin the cycle
    }

    void OnDestroy()
    {
        activeObstacles.Remove(this); // Clean up when object is destroyed
    }

    IEnumerator TeleportCycle()
    {
```

```
        while (true)
        {
            yield return StartCoroutine(Shrink());

            // Find a valid position
            Vector3 newPosition = GetValidPosition(); // Find a safe random
position
            transform.position = newPosition;

            yield return StartCoroutine(Grow());

            yield return new WaitForSeconds(waitTime);
        }
    }

    Vector3 GetValidPosition()
    {
        Vector3 candidate;
        bool isValid = false;
        int maxAttempts = 100; // Try up to 100 times to find a valid position

        // Generate a random candidate position in the defined range
        do
        {
            candidate = new Vector3(
                Random.Range(rangeX1, rangeX2),
                Random.Range(rangeY1, rangeY2),
                0f
            );

            // Loop through all other spheres. If any are too close to the
candidate position (less than 3 units), reject it and try again
            isValid = true;

            foreach (var other in activeObstacles)
            {
                if (other != this && other.isVisible) // If another sphere is
in the list but is not marked as visible, it will be ignored during collision-
distance checks
                {
                    float distance = Vector3.Distance(candidate,
other.transform.position);
                    if (distance < minDistanceFromOthers)
                    {
                        isValid = false;
                        break;
                    }
                }
            }
```

```
            maxAttempts--;

        } while (!isValid && maxAttempts > 0);

        // Return the safe position to be used for teleporting
        return candidate;
    }

    IEnumerator Shrink()
    {
        while (transform.localScale.x > 0.05f)
        {
            transform.localScale = Vector3.Lerp(transform.localScale,
Vector3.zero, Time.deltaTime * shrinkSpeed);
            yield return null;
        }
        transform.localScale = Vector3.zero; // Force final scale
    }

    IEnumerator Grow()
    {
        while (transform.localScale.x < targetScale.x * 0.95f)
        {
            transform.localScale = Vector3.Lerp(transform.localScale,
targetScale, Time.deltaTime * growSpeed);
            yield return null;
        }
        transform.localScale = targetScale; // Force final scale
    }
}
```

ObstacleTeleportation script

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

public class GravityHandler : MonoBehaviour
{
    private Dictionary<string, Vector3> lut = new Dictionary<string, Vector3>{
        {"Level 2 (Mercury)", new Vector3(0, -3.7f, 0)},
        {"Level 3 (Venus)",   new Vector3(0, -8.87f, 0)},
        {"Level 4 (Mars)",    new Vector3(0, -3.73f, 0)},
    }; // lut - Lookup table that maps each scene name to a gravity vector

    private string lastSceneLogged = ""; // Track the last scene to prevent
duplicate logs
```

```
    void OnEnable() // This method runs automatically when the script is
enabled
    {
        SceneManager.sceneLoaded += OnSceneLoaded; // It subscribes to the
sceneLoaded event, so that whenever a new scene is loaded, the method
OnSceneLoaded will be called

        // Handle the current scene manually (only once)
        ApplyGravity(SceneManager.GetActiveScene().name);
    }

    void OnDisable()
    {
        SceneManager.sceneLoaded -= OnSceneLoaded; // When the object is
disabled or destroyed, it unsubscribes from the event
    }

    private void OnSceneLoaded(Scene scene, LoadSceneMode mode) // This method
is automatically called every time a new scene is loaded
    {
        ApplyGravity(scene.name); // It passes the name of the new scene to
ApplyGravity() so it can change the gravity
    }

    private void ApplyGravity(string sceneName)
    {
        Vector3 gravity = lut.GetValueOrDefault(sceneName, new Vector3(0, -
9.81f, 0)); // Looks up the sceneName in the dictionary. If the scene exists
in lut, it uses the custom gravity. If not, it uses Earth's default gravity:
Vector3(0, -9.81f, 0)
        Physics.gravity = gravity; // It applies the gravity using
Physics.gravity

        if (sceneName != lastSceneLogged)
        {
            Debug.Log("Gravity set to " + gravity + " in scene: " + sceneName
+ "."); // Prints the result in the Console, so I can verify it's working
during play

            lastSceneLogged = sceneName;
        }
    }
}
```

GravityHandler script

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

```
using UnityEngine;

public class QuitApplication : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            Application.Quit();
        }
    }
}
```

QuitApplication script

```
using UnityEngine;
#if ENABLE_INPUT_SYSTEM
using UnityEngine.InputSystem;
#endif

namespace StarterAssets
{
    [RequireComponent(typeof(CharacterController))]
#if ENABLE_INPUT_SYSTEM
    [RequireComponent(typeof(PlayerInput))]
#endif
    public class FirstPersonController : MonoBehaviour
    {
        [Header("Player")]
        [Tooltip("Move speed of the character in m/s")]
        public float MoveSpeed = 4.0f;
        [Tooltip("Sprint speed of the character in m/s")]
        public float SprintSpeed = 7.0f;
        [Tooltip("Rotation speed of the character")]
        public float RotationSpeed = 1.0f;
        [Tooltip("Acceleration and deceleration")]
        public float SpeedChangeRate = 10.0f;

        [Space(10)]
        [Tooltip("The height the player can jump")]
        public float JumpHeight = 1.2f;
        [Tooltip("The character uses its own gravity value. The engine default
is -9.81f")]
        public float Gravity = -15.0f;

        [Space(10)]
        [Tooltip("Time required to pass before being able to jump again. Set
to 0f to instantly jump again")]
        public float JumpTimeout = 0.1f;
```

```
        [Tooltip("Time required to pass before entering the fall state. Useful
for walking down stairs")]
        public float FallTimeout = 0.15f;

        [Header("Player Grounded")]
        [Tooltip("If the character is grounded or not. Not part of the
CharacterController built in grounded check")]
        public bool Grounded = true;
        [Tooltip("Useful for rough ground")]
        public float GroundedOffset = -0.14f;
        [Tooltip("The radius of the grounded check. Should match the radius of
the CharacterController")]
        public float GroundedRadius = 0.5f;
        [Tooltip("What layers the character uses as ground")]
        public LayerMask GroundLayers;

        [Header("Cinemachine")]
        [Tooltip("The follow target set in the Cinemachine Virtual Camera that
the camera will follow")]
        public GameObject CinemachineCameraTarget;
        [Tooltip("How far in degrees can you move the camera up")]
        public float TopClamp = 90.0f;
        [Tooltip("How far in degrees can you move the camera down")]
        public float BottomClamp = -90.0f;

        // cinemachine
        private float _cinemachineTargetPitch;

        // player
        private float _speed;
        private float _rotationVelocity;
        private float _verticalVelocity;
        private float _terminalVelocity = 53.0f;

        // timeout deltatime
        private float _jumpTimeoutDelta;
        private float _fallTimeoutDelta;


#if ENABLE_INPUT_SYSTEM
        private PlayerInput _playerInput;
#endif
        private CharacterController _controller;
        private StarterAssetsInputs _input;
        private GameObject _mainCamera;

        private const float _threshold = 0.01f;

        private bool IsCurrentDeviceMouse
```

```
        {
            get
            {
                #if ENABLE_INPUT_SYSTEM
                return _playerInput.currentControlScheme == "KeyboardMouse";
                #else
                return false;
                #endif
            }
        }

        private void Awake()
        {
            // get a reference to our main camera
            if (_mainCamera == null)
            {
                _mainCamera = GameObject.FindGameObjectWithTag("MainCamera");
            }
        }

        private void Start()
        {
            _controller = GetComponent<CharacterController>();
            _input = GetComponent<StarterAssetsInputs>();
#if ENABLE_INPUT_SYSTEM
            _playerInput = GetComponent<PlayerInput>();
#else
            Debug.LogError( "Starter Assets package is missing dependencies.
Please use Tools/Starter Assets/Reinstall Dependencies to fix it");
#endif

            // reset our timeouts on start
            _jumpTimeoutDelta = JumpTimeout;
            _fallTimeoutDelta = FallTimeout;
        }

        private void Update()
        {
            JumpAndGravity();
            GroundedCheck();
            Move();
        }

        private void LateUpdate()
        {
            CameraRotation();
        }

        private void GroundedCheck()
```

```
        {
            // set sphere position, with offset
            Vector3 spherePosition = new Vector3(transform.position.x,
transform.position.y - GroundedOffset, transform.position.z);
            Grounded = Physics.CheckSphere(spherePosition, GroundedRadius,
GroundLayers, QueryTriggerInteraction.Ignore);
        }

        private void CameraRotation()
        {
            // if there is an input
            if (_input.look.sqrMagnitude >= _threshold)
            {
                //Don't multiply mouse input by Time.deltaTime
                float deltaTimeMultiplier = IsCurrentDeviceMouse ? 1.0f :
Time.deltaTime;

                _cinemachineTargetPitch += _input.look.y * RotationSpeed *
deltaTimeMultiplier;
                _rotationVelocity = _input.look.x * RotationSpeed *
deltaTimeMultiplier;

                // clamp our pitch rotation
                _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch,
BottomClamp, TopClamp);

                // Update Cinemachine camera target pitch
                CinemachineCameraTarget.transform.localRotation =
Quaternion.Euler(_cinemachineTargetPitch, 0.0f, 0.0f);

                // rotate the player left and right
                transform.Rotate(Vector3.up * _rotationVelocity);
            }
        }

        private void Move()
        {
            // set target speed based on move speed, sprint speed and if
sprint is pressed
            float targetSpeed = _input.sprint ? SprintSpeed : MoveSpeed;

            // a simplistic acceleration and deceleration designed to be easy
to remove, replace, or iterate upon

            // note: Vector2's == operator uses approximation so is not
floating point error prone, and is cheaper than magnitude
            // if there is no input, set the target speed to 0
            if (_input.move == Vector2.zero) targetSpeed = 0.0f;
```

```
            // a reference to the players current horizontal velocity
            float currentHorizontalSpeed = new Vector3(_controller.velocity.x,
0.0f, _controller.velocity.z).magnitude;

            float speedOffset = 0.1f;
            float inputMagnitude = _input.analogMovement ?
_input.move.magnitude : 1f;

            // accelerate or decelerate to target speed
            if (currentHorizontalSpeed < targetSpeed - speedOffset ||
currentHorizontalSpeed > targetSpeed + speedOffset)
            {
                // creates curved result rather than a linear one giving a
more organic speed change
                // note T in Lerp is clamped, so we don't need to clamp our
speed
                _speed = Mathf.Lerp(currentHorizontalSpeed, targetSpeed *
inputMagnitude, Time.deltaTime * SpeedChangeRate);

                // round speed to 3 decimal places
                _speed = Mathf.Round(_speed * 1000f) / 1000f;
            }
            else
            {
                _speed = targetSpeed;
            }

            // normalise input direction
            Vector3 inputDirection = new Vector3(_input.move.x, 0.0f,
_input.move.y).normalized;

            // note: Vector2's != operator uses approximation so is not
floating point error prone, and is cheaper than magnitude
            // if there is a move input rotate player when the player is
moving
            if (_input.move != Vector2.zero)
            {
                // move
                inputDirection = transform.right * _input.move.x +
transform.forward * _input.move.y;
            }

            // move the player
            _controller.Move(inputDirection.normalized * (_speed *
Time.deltaTime) + new Vector3(0.0f, _verticalVelocity, 0.0f) *
Time.deltaTime);
        }

        private void JumpAndGravity()
```

```
        {
            if (Grounded)
            {
                // reset the fall timeout timer
                _fallTimeoutDelta = FallTimeout;

                // stop our velocity dropping infinitely when grounded
                if (_verticalVelocity < 0.0f)
                {
                    _verticalVelocity = -2f;
                }

                // Jump
                if (_input.jump && _jumpTimeoutDelta <= 0.0f)
                {
                    // the square root of H * -2 * G = how much velocity
needed to reach desired height
                    _verticalVelocity = Mathf.Sqrt(JumpHeight * -2f *
Gravity);
                }

                // jump timeout
                if (_jumpTimeoutDelta >= 0.0f)
                {
                    _jumpTimeoutDelta -= Time.deltaTime;
                }
            }
            else
            {
                // reset the jump timeout timer
                _jumpTimeoutDelta = JumpTimeout;

                // fall timeout
                if (_fallTimeoutDelta >= 0.0f)
                {
                    _fallTimeoutDelta -= Time.deltaTime;
                }

                // if we are not grounded, do not jump
                _input.jump = false;
            }

            // apply gravity over time if under terminal (multiply by delta
time twice to linearly speed up over time)
            if (_verticalVelocity < _terminalVelocity)
            {
                _verticalVelocity += Gravity * Time.deltaTime;
            }
        }
```

```
        private static float ClampAngle(float lfAngle, float lfMin, float
lfMax)
        {
            if (lfAngle < -360f) lfAngle += 360f;
            if (lfAngle > 360f) lfAngle -= 360f;
            return Mathf.Clamp(lfAngle, lfMin, lfMax);
        }

        private void OnDrawGizmosSelected()
        {
            Color transparentGreen = new Color(0.0f, 1.0f, 0.0f, 0.35f);
            Color transparentRed = new Color(1.0f, 0.0f, 0.0f, 0.35f);

            if (Grounded) Gizmos.color = transparentGreen;
            else Gizmos.color = transparentRed;

            // when selected, draw a gizmo in the position of, and matching
radius of, the grounded collider
            Gizmos.DrawSphere(new Vector3(transform.position.x,
transform.position.y - GroundedOffset, transform.position.z), GroundedRadius);
        }
    }
}
```

FirstPersonController script

```
using UnityEngine;

public class BasicRigidBodyPush : MonoBehaviour
{
    public LayerMask pushLayers;
    public bool canPush;
    [Range(0.5f, 5f)] public float strength = 1.1f;

    private void OnControllerColliderHit(ControllerColliderHit hit)
    {
        if (canPush) PushRigidBodies(hit);
    }

    private void PushRigidBodies(ControllerColliderHit hit)
    {
        //
https://docs.unity3d.com/ScriptReference/CharacterController.OnControllerColli
derHit.html

        // make sure we hit a non kinematic rigidbody
        Rigidbody body = hit.collider.attachedRigidbody;
        if (body == null || body.isKinematic) return;
```

```
        // make sure we only push desired layer(s)
        var bodyLayerMask = 1 << body.gameObject.layer;
        if ((bodyLayerMask & pushLayers.value) == 0) return;

        // We dont want to push objects below us
        if (hit.moveDirection.y < -0.3f) return;

        // Calculate push direction from move direction, horizontal motion
only
        Vector3 pushDir = new Vector3(hit.moveDirection.x, 0.0f,
hit.moveDirection.z);

        // Apply the push and take strength into account
        body.AddForce(pushDir * strength, ForceMode.Impulse);
    }
}
```

BasicRigidBodyPush script

```
using UnityEngine;
#if ENABLE_INPUT_SYSTEM
using UnityEngine.InputSystem;
#endif

namespace StarterAssets
{
    public class StarterAssetsInputs : MonoBehaviour
    {
        [Header("Character Input Values")]
        public Vector2 move;
        public Vector2 look;
        public bool jump;
        public bool sprint;

        [Header("Movement Settings")]
        public bool analogMovement;

        [Header("Mouse Cursor Settings")]
        public bool cursorLocked = true;
        public bool cursorInputForLook = true;

#if ENABLE_INPUT_SYSTEM
        public void OnMove(InputValue value)
        {
            MoveInput(value.Get<Vector2>());
        }

        public void OnLook(InputValue value)
```

```
        {
            if(cursorInputForLook)
            {
                LookInput(value.Get<Vector2>());
            }
        }

        public void OnJump(InputValue value)
        {
            JumpInput(value.isPressed);
        }

        public void OnSprint(InputValue value)
        {
            SprintInput(value.isPressed);
        }
#endif


        public void MoveInput(Vector2 newMoveDirection)
        {
            move = newMoveDirection;
        }

        public void LookInput(Vector2 newLookDirection)
        {
            look = newLookDirection;
        }

        public void JumpInput(bool newJumpState)
        {
            jump = newJumpState;
        }

        public void SprintInput(bool newSprintState)
        {
            sprint = newSprintState;
        }

        private void OnApplicationFocus(bool hasFocus)
        {
            SetCursorState(cursorLocked);
        }

        private void SetCursorState(bool newState)
        {
            Cursor.lockState = newState ? CursorLockMode.Locked :
CursorLockMode.None;
        }
```

```
    }

}
```

StarterAssetsInputs script

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

public class Weapon : MonoBehaviour
{
    [SerializeField] Camera FPCamera;
    [SerializeField] float range = 100f;
    [SerializeField] float damage = 30f;
    [SerializeField] ParticleSystem muzzleFlash;
    [SerializeField] GameObject hitEffect;
    [SerializeField] Ammo ammoSlot;
    [SerializeField] AmmoType ammoType;
    [SerializeField] float timeBetweenShots = 0.5f;
    [SerializeField] TextMeshProUGUI weaponInfoText;
    [SerializeField] string weaponName;
    [SerializeField] string weaponType;

    AudioSource gunShot;

    bool canShoot = true;
    bool isAutomated = false;
    bool isBurstMode = false;

    private void OnEnable()
    {
        canShoot = true;
        StartCoroutine(Waiter());
        DisplayWeaponInfo();
    }

    private void Start()
    {
        gunShot = GetComponent<AudioSource>();
    }

    // Update is called once per frame
    void Update()
    {
        DisplayWeaponInfo();
        if (Input.GetMouseButtonDown(0) && canShoot == true &&
ammoSlot.GetCurrentAmmo(ammoType) > 0)
```

```
        {
            if (WeaponSwitcher.currentWeapon == 2)
            {
                isBurstMode = true;
                StartCoroutine(BurstShoot());
            }
            else if (WeaponSwitcher.currentWeapon == 3)
            {
                isAutomated = true;
                StartCoroutine(AutoShoot());
            }
            else
            {
                isBurstMode = false;
                isAutomated = false;
                StartCoroutine(Shoot());
            }
        }
        else if (Input.GetMouseButtonUp(0) && isBurstMode)
        {
            isBurstMode = false;
        }
        else if (Input.GetMouseButtonUp(0) && isAutomated)
        {
            isAutomated = false;
            StopMuzzleFlash();
            gunShot.Stop();
        }
    }

    private void DisplayWeaponInfo()
    {
        int currentAmmo = ammoSlot.GetCurrentAmmo(ammoType);
        weaponInfoText.text = $"{weaponName} ({weaponType}): {currentAmmo}";
    }

    IEnumerator Waiter()
    {
        canShoot = false;
        yield return new WaitForSeconds(0.1f);
        canShoot = true;
    }

    IEnumerator Shoot()
    {
        canShoot = false;
        if (ammoSlot.GetCurrentAmmo(ammoType) > 0)
        {
            PlayMuzzleFlash();
```

```
            gunShot.Play();
            ProcessRaycast();
            ammoSlot.DecreaseCurrentAmmo(ammoType);
        }
        else
        {
            StopMuzzleFlash();
            gunShot.Stop();
        }
        yield return new WaitForSeconds(timeBetweenShots);
        canShoot = true;
    }

    IEnumerator BurstShoot()
    {
        canShoot = false;
        if (ammoSlot.GetCurrentAmmo(ammoType) > 0)
        {
            for (int i = 0; i <= 2; i++)
            {
                PlayMuzzleFlash();
                gunShot.Play();
                ProcessRaycast();
                ammoSlot.DecreaseCurrentAmmo(ammoType);
                yield return new WaitForSeconds(0.131f);
                StopMuzzleFlash();
            }
        }
        else
        {
            StopMuzzleFlash();
            gunShot.Stop();
        }
        yield return new WaitForSeconds(timeBetweenShots);
        canShoot = true;
    }

    IEnumerator AutoShoot()
    {
        canShoot = false;
        while (isAutomated)
        {
            if (ammoSlot.GetCurrentAmmo(ammoType) > 0)
            {
                PlayMuzzleFlash();
                gunShot.Play();
                ProcessRaycast();
                ammoSlot.DecreaseCurrentAmmo(ammoType);
            }
```

```
            else
            {
                StopMuzzleFlash();
                gunShot.Stop();
            }
            yield return new WaitForSeconds(timeBetweenShots);
            canShoot = true;
        }
    }

    private void PlayMuzzleFlash()
    {
        muzzleFlash.Play();
    }

    private void StopMuzzleFlash()
    {
        muzzleFlash.Stop();
    }

    private void ProcessRaycast()
    {
        RaycastHit hit;
        if (Physics.Raycast(FPCamera.transform.position,
FPCamera.transform.forward, out hit, range))
        {
            CreateHitImpact(hit);
            EnemyHealth target = hit.transform.GetComponent<EnemyHealth>();
            if (target == null)
            {
                return;
            }
            target.TakeDamage(damage);
        }
        else
        {
            return;
        }
    }

    private void CreateHitImpact(RaycastHit hit)
    {
        GameObject impact = Instantiate(hitEffect, hit.point,
Quaternion.LookRotation(hit.normal));
        Destroy(impact, 0.1f);
    }
}
```

Weapon script

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

public class WeaponSwitcher : MonoBehaviour
{
    [SerializeField] public static int currentWeapon = 0;

    // Start is called before the first frame update
    void Start()
    {
        SetWeaponActive();
    }

        // Update is called once per frame
    void Update()
    {
        int previousWeapon = currentWeapon;

        ProcessKeyInput();
        ProcessScrollWheel();

        if (previousWeapon != currentWeapon)
        {
            SetWeaponActive();
        }
    }

    private void ProcessKeyInput()
    {
        if (Input.GetKeyDown(KeyCode.Alpha1))
        {
            currentWeapon = 0;
        }
        else if (Input.GetKeyDown(KeyCode.Alpha2))
        {
            currentWeapon = 1;
        }
        else if (Input.GetKeyDown(KeyCode.Alpha3))
        {
            currentWeapon = 2;
        }
        else if (Input.GetKeyDown(KeyCode.Alpha4))
        {
            currentWeapon = 3;
        }
        else if (Input.GetKeyDown(KeyCode.Alpha5))
```

```
        {
            currentWeapon = 4;
        }
    }

    private void ProcessScrollWheel()
    {
        if (Input.GetAxis("Mouse ScrollWheel") < 0)
        {
            if (currentWeapon >= transform.childCount - 1)
            {
                currentWeapon = 0;
            }
            else
            {
                currentWeapon++ ;
            }
        }

        if (Input.GetAxis("Mouse ScrollWheel") > 0)
        {
            if (currentWeapon <= 0)
            {
                currentWeapon = transform.childCount - 1;
            }
            else
            {
                currentWeapon-- ;
            }
        }
    }

    private void SetWeaponActive()
    {
        int weaponIndex = 0;

        foreach (Transform weapon in transform)
        {
            if (weaponIndex == currentWeapon)
            {
                weapon.gameObject.SetActive(true);
            }
            else
            {
                weapon.gameObject.SetActive(false);
            }
            weaponIndex++ ;
        }
    }
```

```
}
```

WeaponSwitcher script

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

public class EnemyAI : MonoBehaviour
{
    [SerializeField] float chaseRange = 5f; // This shows how close to the
enemy do we get, before the enemy starts chasing
    [SerializeField] float turnSpeed = 5f;
    [SerializeField] AudioClip howling;
    [SerializeField] float minDistance = 1f; // Minimum distance for full
volume of the howling
    [SerializeField] float maxDistance = 4f; // Maximum distance for zero
volume of the howling

    NavMeshAgent navMeshAgent;
    float distanceToTarget = Mathf.Infinity; // How far is the enemy from the
target. It is on purpose a gigantic number, so the player can move freely and
will not be chased immediately from the other side of the map, by the enemy
    bool isProvoked = false;
    EnemyHealth health;
    Transform target; // This shows to the enemy, where the target (player) is
    AudioSource audioSource;
    Coroutine howlingCoroutine;

    void Start()
    {
        if (maxDistance == 0) maxDistance = chaseRange - 1; // Set maxDistance
based on chaseRange if not manually set

        navMeshAgent = GetComponent<NavMeshAgent>();
        health = GetComponent<EnemyHealth>();
        target = FindObjectOfType<PlayerHealth>().transform;
        audioSource = GetComponent<AudioSource>();

        if (audioSource != null)
        {
            audioSource.spatialBlend = 1f; // Ensure the audio source is set
to 3D
            audioSource.rolloffMode = AudioRolloffMode.Linear; // Optional:
Adjust this to fit your needs
            audioSource.minDistance = minDistance; // Full volume within this
distance
```

```
            audioSource.maxDistance = maxDistance; // No volume beyond this
distance
        }
    }

    void Update()
    {
        if (health.IsDead())
        {
            enabled = false;
            navMeshAgent.enabled = false;
            if (howlingCoroutine != null)
            {
                StopCoroutine(howlingCoroutine);
                howlingCoroutine = null;
            }
        }

        distanceToTarget = Vector3.Distance(target.position,
transform.position); // This is how the distance to target is being measured.
target.position is the target's position and transform.position is what our
position is
        if (isProvoked)
        {
            EngageTarget();
        }
        else if (distanceToTarget <= chaseRange)
        {
            Provoke();
        }
    }

    public void OnDamageTaken()
    {
       Provoke();
    }

    private void Provoke()
    {
        if (isProvoked == false)
        {
            isProvoked = true;
            if (howlingCoroutine == null)
            {
                howlingCoroutine = StartCoroutine(PlayHowlSoundRepeatedly());
            }
        }
    }
```

```
    private IEnumerator PlayHowlSoundRepeatedly()
    {
        while (isProvoked && !health.IsDead())
        {
            if (howling != null && audioSource != null)
            {
                audioSource.PlayOneShot(howling);
                yield return new WaitForSeconds(howling.length);
            }
        }
    }

    private void EngageTarget()
    {
        FaceTarget();
        if (distanceToTarget >= navMeshAgent.stoppingDistance &&
!health.IsDead())
        {
            ChaseTarget();
        }
        if (distanceToTarget <= navMeshAgent.stoppingDistance)
        {
            AttackTarget();
        }
    }

    private void ChaseTarget()
    {
        GetComponent<Animator>().SetBool("Attack", false);
        GetComponent<Animator>().SetTrigger("Move");
        navMeshAgent.SetDestination(target.position);
    }

    private void AttackTarget()
    {
        GetComponent<Animator>().SetBool("Attack", true);
    }

    private void FaceTarget()
    {
        Vector3 direction = (target.position - transform.position).normalized;
        Quaternion lookRotation = Quaternion.LookRotation(new
Vector3(direction.x, 0, direction.z));
        transform.rotation = Quaternion.Slerp(transform.rotation,
lookRotation, Time.deltaTime * turnSpeed);
    }

    void OnDrawGizmosSelected()
    {
```

```
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, chaseRange);
    }
}
```

EnemyAI script

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

public class EnemyAttack : MonoBehaviour
{
    PlayerHealth target;

    [SerializeField] float damage = 40f;

    // Start is called before the first frame update
    void Start()
    {
        target = FindObjectOfType<PlayerHealth>();
    }

    public void AttackHitEvent()
    {
        if (target == null)
        {
            return;
        }
        target.TakeDamage(damage);
        target.GetComponent<DisplayDamage>().ShowDamageImpact();
    }
}
```

EnemyAttack script

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

public class EnemyHealth : MonoBehaviour
{
    [SerializeField] float hitPoints = 100f;
    [SerializeField] TextMeshProUGUI damageDisplayText;

    bool isDead = false;

    public bool IsDead()
```

```
    {
        return isDead;
    }

    public void TakeDamage(float damage)
    {
        BroadcastMessage("OnDamageTaken");
        hitPoints -= damage;

        DisplayDamage(damage);

        if (hitPoints <= 0)
        {
            Die();
        }
    }

    private void Die()
    {
        if (isDead)
        {
            return;
        }

        isDead = true;

        GetComponent<Animator>().SetTrigger("Die");
    }

    private void DisplayDamage(float damage)
    {
        if (isDead == false)
        {
            // Set the damage text
            damageDisplayText.text = "-" + damage.ToString("F0"); // Display
the damage with no decimal places

            // Start the floating and fading coroutine
            StartCoroutine(FloatingDamageText(damageDisplayText));
        }
    }

    private IEnumerator FloatingDamageText(TextMeshProUGUI text)
    {
        CanvasGroup canvasGroup = text.GetComponent<CanvasGroup>();
        if (canvasGroup == null)
        {
            canvasGroup = text.gameObject.AddComponent<CanvasGroup>();
        }
```

```
        // Set initial alpha and position
        canvasGroup.alpha = 1f;
        RectTransform rectTransform = text.GetComponent<RectTransform>();
        Vector2 startPosition = new Vector2(rectTransform.anchoredPosition.x,
75);
        Vector2 finalPosition = new Vector2(rectTransform.anchoredPosition.x,
100);
        float duration = 1f; // Duration of the float animation

        float elapsedTime = 0f;

        while (elapsedTime < duration)
        {
            elapsedTime += Time.deltaTime;
            float t = elapsedTime / duration;

            // Move the text upwards
            rectTransform.anchoredPosition = Vector2.Lerp(startPosition,
finalPosition, t);

            // Fade out the text
            canvasGroup.alpha = Mathf.Lerp(1f, 0f, t);

            yield return null;
        }

        // Ensure the final position and alpha are set
        rectTransform.anchoredPosition = finalPosition;
        canvasGroup.alpha = 0f;
    }
}
```

EnemyHealth script

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

public class PlayerHealth : MonoBehaviour
{
    [SerializeField] float hitPoints = 100f;
    [SerializeField] AudioClip damageReceived;

    AudioSource audioSource;

    private void Start()
    {
        audioSource = GetComponent<AudioSource>();
```

```
    }

    public void TakeDamage(float damage)
    {
        hitPoints -= damage;
        audioSource.PlayOneShot(damageReceived);
        if (hitPoints <= 0)
        {
            GetComponent<DeathHandler>().HandleDeath();
        }
    }
}
```

PlayerHealth script

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

public class DisplayDamage : MonoBehaviour
{
    [SerializeField] Canvas impactCanvas;
    [SerializeField] float impactTime = 0.3f;

    // Start is called before the first frame update
    void Start()
    {
        impactCanvas.enabled = false;
    }

    public void ShowDamageImpact()
    {
        StartCoroutine(ShowSplatter());
    }

    IEnumerator ShowSplatter()
    {
        impactCanvas.enabled = true;
        yield return new WaitForSeconds(impactTime);
        impactCanvas.enabled = false;
    }
}
```

DisplayDamage script

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

```
public class DeathHandler : MonoBehaviour
{
    [SerializeField] Canvas gameOverCanvas;
    [SerializeField] Canvas gunReticleCanvas;
    [SerializeField] Canvas ammoDisplayCanvas;
    [SerializeField] Canvas damageReceivedCanvas;
    [SerializeField] Canvas damageDisplayCanvas;
    [SerializeField] Canvas ammoPickedUpDisplayCanvas;
    [SerializeField] float fadeInDuration = 1f;

    private CanvasGroup gameOverCanvasGroup;

    private void Start()
    {
        // Get the CanvasGroup component from the gameOverCanvas
        gameOverCanvasGroup = gameOverCanvas.GetComponent<CanvasGroup>();

        gunReticleCanvas.enabled = true;
        ammoDisplayCanvas.enabled = true;
        damageReceivedCanvas.enabled = false;
        damageDisplayCanvas.enabled = true;
        ammoPickedUpDisplayCanvas.enabled = true;

        // Ensure the canvas starts invisible
        gameOverCanvasGroup.alpha = 0f;
        gameOverCanvas.enabled = false;
    }

    public void HandleDeath()
    {
        GetComponent<StarterAssets.FirstPersonController>().enabled = false;
        FindObjectOfType<WeaponSwitcher>().enabled = false;
        FindObjectOfType<Weapon>().enabled = false;

        // Start the fade-in coroutine
        StartCoroutine(FadeInCanvas());

        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;

        // Pause the game
        Time.timeScale = 0;
    }

    private IEnumerator FadeInCanvas()
    {
        gunReticleCanvas.enabled = false;
        ammoDisplayCanvas.enabled = false;
        damageDisplayCanvas.enabled = false;
```

```
        ammoPickedUpDisplayCanvas.enabled = false;

        // Enable and show the canvas
        gameOverCanvas.enabled = true;

        // Gradually increase alpha to 1 over fadeInDuration seconds
        float elapsedTime = 0f;
        while (elapsedTime < fadeInDuration)
        {
            gameOverCanvasGroup.alpha = Mathf.Lerp(0f, 1f, elapsedTime /
fadeInDuration);
            elapsedTime += Time.unscaledDeltaTime; // Use unscaledDeltaTime to
ignore time scale
            yield return null;
        }

        // Ensure alpha is exactly 1 at the end
        gameOverCanvasGroup.alpha = 1f;

        // Now fade out the damageReceivedCanvas
        CanvasGroup damageReceivedCanvasGroup =
damageReceivedCanvas.GetComponent<CanvasGroup>();
        float fadeOutDuration = 1f; // Adjust the duration as needed

        float fadeElapsedTime = 0f;
        while (fadeElapsedTime < fadeOutDuration)
        {
            damageReceivedCanvasGroup.alpha = Mathf.Lerp(1f, 0f,
fadeElapsedTime / fadeOutDuration);
            fadeElapsedTime += Time.unscaledDeltaTime; // Use
unscaledDeltaTime to ignore time scale
            yield return null;
        }

        // Ensure alpha is exactly 0 at the end
        damageReceivedCanvasGroup.alpha = 0f;

        // Disable damageReceivedCanvas after fading out
        damageReceivedCanvas.enabled = false;
    }
}
```

DeathHandler script

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

```
public class SceneLoader : MonoBehaviour
{
    public void ReloadGame()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1;
    }

    public void QuitGame()
    {
        Application.Quit();
    }
}
```

SceneLoader script

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

public class WeaponZoom : MonoBehaviour
{
    [SerializeField] CinemachineVirtualCamera fpsCamera;
    [SerializeField] FirstPersonController fpsController;
    [SerializeField] float zoomedOutFOV = 40;
    [SerializeField] float zoomedInFOV = 20;
    [SerializeField] float zoomedOutSensitivity = 1f;
    [SerializeField] float zoomedInSensitivity = 0.25f;

    private void OnDisable()
    {
        ZoomOut();
    }

    private void Update()
    {
        if (Input.GetMouseButton(1)) // While right mouse button is held down
        {
            ZoomIn();
        }
        else
        {
            ZoomOut();
        }
    }

    private void ZoomIn()
```

```
    {
        fpsCamera.m_Lens.FieldOfView = zoomedInFOV;
        fpsController.RotationSpeed = zoomedInSensitivity;
    }

    private void ZoomOut()
    {
        fpsCamera.m_Lens.FieldOfView = zoomedOutFOV;
        fpsController.RotationSpeed = zoomedOutSensitivity;
    }
}
```

WeaponZoom script

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

public class Ammo : MonoBehaviour
{
    [SerializeField] AmmoSlot[] ammoSlots;

    [System.Serializable]
    private class AmmoSlot
    {
        public AmmoType ammoType;
        public int ammoAmount;
    }

    public int GetCurrentAmmo(AmmoType ammoType)
    {
        return GetAmmoSlot(ammoType).ammoAmount;
    }

    public void DecreaseCurrentAmmo(AmmoType ammoType)
    {
        GetAmmoSlot(ammoType).ammoAmount-- ;
    }

    public void IncreaseCurrentAmmo(AmmoType ammoType, int ammoAmount)
    {
        GetAmmoSlot(ammoType).ammoAmount += ammoAmount;
    }

    private AmmoSlot GetAmmoSlot(AmmoType ammoType)
    {
        foreach (AmmoSlot slot in ammoSlots)
        {
            if (slot.ammoType == ammoType)
```

```
            {
                return slot;
            }
        }
        return null;
    }
}
```

Ammo script

```
using System.ComponentModel;

public enum AmmoType
{
    [Description("Pistol Bullets")]
    PistolBullets,

    [Description("Shotgun Shells")]
    ShotgunShells,

    [Description("Light Rounds")]
    LightRounds,

    [Description("Plasma Ammo")]
    PlasmaAmmo,

    [Description("Sniper Ammo")]
    SniperAmmo
}
```

AmmoType script

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

public class LocalRainFollow : MonoBehaviour
{
    [SerializeField] Transform player;
    [SerializeField] float followSpeed = 10f;

    void Update()
    {
        // Follow only on X and Z (horizontal)
        Vector3 targetPos = new Vector3(player.position.x,
transform.position.y, player.position.z);
        transform.position = Vector3.Lerp(transform.position, targetPos,
followSpeed * Time.deltaTime);
    }
```

```
}
```

LocalRainFollow script

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

public class FlashlightSystem : MonoBehaviour
{
    [SerializeField] float lightDecay = 0.1f;
    [SerializeField] float angleDecay = 1f;

    Light myLight;

    private void Start()
    {
        myLight = GetComponent<Light>();
    }

    private void Update()
    {
        DecreaseLightAngle();
        DecreaseLightIntensity();
    }

    public void RestoreLightAngle(float restoreAngle)
    {
        myLight.spotAngle = restoreAngle;
    }

    public void RestoreLightIntensity(float intensityAmount)
    {
        myLight.intensity = intensityAmount;
    }

    private void DecreaseLightAngle()
    {
        myLight.spotAngle -= angleDecay * Time.deltaTime;
    }

    private void DecreaseLightIntensity()
    {
        myLight.intensity -= lightDecay * Time.deltaTime;
    }
}
```

FlashlightSystem script

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

public class AmmoPickup : MonoBehaviour
{
    [SerializeField] int ammoAmount = 5;
    [SerializeField] AmmoType ammoType;
    [SerializeField] TextMeshProUGUI AmmoPickedUpDisplayText;
    [SerializeField] float displayDuration = 1f;
    [SerializeField] float fadeDuration = 1f;

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            FindObjectOfType<Ammo>().IncreaseCurrentAmmo(ammoType,
ammoAmount);

            AmmoPickedUpDisplayText.text = "+" + ammoAmount.ToString("F0") + "
" + ammoType.GetDescription();

            CanvasGroup canvasGroup =
AmmoPickedUpDisplayText.GetComponent<CanvasGroup>();
            if (canvasGroup != null)
            {
                canvasGroup.alpha = 1f;

                // Start fading on a persistent object (like the UI Text
itself)
                AmmoPickedUpDisplayText.StartCoroutine(FadeOutAmmoPickedUpText
(canvasGroup));
            }

            // Destroy this gameObject after 0.5 seconds
            Destroy(gameObject, 0.5f);
        }
    }

    private IEnumerator FadeOutAmmoPickedUpText(CanvasGroup canvasGroup)
    {
        yield return new WaitForSeconds(displayDuration);

        float fadeSpeed = 1f / fadeDuration;
        while (canvasGroup.alpha > 0f)
        {
            canvasGroup.alpha -= fadeSpeed * Time.deltaTime;
            yield return null;
```

```
        }

        canvasGroup.alpha = 0f;
    }
}
```

AmmoPickup script

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

public static class EnumExtensions
{
    public static string GetDescription(this Enum value)
    {
        FieldInfo fi = value.GetType().GetField(value.ToString());
        DescriptionAttribute[] attributes =
(DescriptionAttribute[])fi.GetCustomAttributes(typeof(DescriptionAttribute),
false);

        if (attributes != null && attributes.Length > 0)
        {
            return attributes[0].Description;
        }
        else
        {
            return value.ToString();
        }
    }
}
```

EnumExtensions script

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

public class BatteryPickup : MonoBehaviour
{
    [SerializeField] float restoreAngle = 70;
    [SerializeField] float intensityAmount = 5;

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
```

```
        {
            other.GetComponentInChildren<FlashlightSystem>().RestoreLightAngle
(restoreAngle);
            other.GetComponentInChildren<FlashlightSystem>().RestoreLightInten
sity(intensityAmount);
            Destroy(gameObject);
        }
    }
}
```

BatteryPickup script

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

public class EndingMessage : MonoBehaviour
{
    public RectTransform textToScroll;
    public float scrollSpeed = 20f;
    public float fadeDuration = 2f;
    public float panelHeight = 376f;

    private CanvasGroup canvasGroup;
    private bool isScrolling = false;

    private void Awake()
    {
        canvasGroup = GetComponent<CanvasGroup>();

        if (canvasGroup != null)
        {
            canvasGroup.alpha = 0f;
            canvasGroup.interactable = false;
            canvasGroup.blocksRaycasts = false;
        }

        if (textToScroll != null)
        {
            textToScroll.anchoredPosition = new Vector2(0, -panelHeight);
        }

        gameObject.SetActive(false);
    }

    public void ShowAndScroll()
    {
        gameObject.SetActive(true);
        StartCoroutine(FadeAndScroll());
```

```
    }

    private IEnumerator FadeAndScroll()
    {
        // Fade In with unscaled time
        float timer = 0f;
        while (timer < fadeDuration)
        {
            canvasGroup.alpha = Mathf.Lerp(0f, 1f, timer / fadeDuration);
            timer += Time.unscaledDeltaTime;
            yield return null;
        }
        canvasGroup.alpha = 1f;

        isScrolling = true;
    }

    private void Update()
    {
        if (isScrolling && textToScroll != null)
        {
            textToScroll.anchoredPosition += Vector2.up * scrollSpeed *
Time.unscaledDeltaTime;
        }
    }
}
```

EndingMessage script

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

public class TriggerScrollingText : MonoBehaviour
{
    public EndingMessage endingMessage;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            Time.timeScale = 0f; // Pause the game
            endingMessage.ShowAndScroll();

            gameObject.SetActive(false);
        }
    }
}
```

TriggerScrollingText script