

ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

Σχολή Μηχανικών -Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Πρόγραμμα Σπουδών – Μηχανικών Πληροφορικής Τ.Ε.

Πτυχιακή Εργασία

CHARACTER BASED GAME ΣΤΗΝ UNITY

ΓΚΑΓΚΑΚΗΣ ΒΑΣΙΛΕΙΟΣ ΑΜ 4699

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ

ΠΑΧΟΥΛΑΚΗΣ ΙΩΑΝΝΗΣ

ΣΕΠΤΕΜΒΡΙΟΣ 2023

ΗΡΑΚΛΕΙΟ

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον υπεύθυνο καθηγητή μου κ. Παχουλάκη Ιωάννη για την καθοδήγηση του στην διάρκεια της πτυχιακής μου εργασίας. Επιπλέον θα ήθελα να ευχαριστήσω την οικογένεια μου για την στήριξη που μου παρείχε σε όλα τα ακαδημαϊκά μου χρόνια.

Abstract

This thesis aimed to create a Third-Person Action RPG zombie survival game using the Unity3D Game Engine and the C# programming language.

The primary objective of the game is for the player to survive as many waves of enemies as possible by using different weapons that are unlocked during the game. Each weapon has its own characteristics as well as each type of zombies which, using artificial intelligence, have the sole purpose of protecting the place and killing the player. As the player progresses through the game, bullets for weapons appear on the map as well as life packs that will help him survive as long as possible and make a new highscore.

Considerable attention was dedicated to rectifying both minor and substantial glitches during the design and implementation process to have a satisfactory level without affecting the player's experience. Moreover, some small details were added in several parts of the game for a better feel in the game such as the player's animations, the ability to interact with objects by having dynamic movements and for even more realism, animation rigging was added to the player's character for various body parts so that the character has restrictions on his movements. For this reason, there were several problems and difficulties that had to be solved with various techniques that often required a lot of digging, play-testing and complex solutions. The paper analyses the code with screenshots, the Behavior Trees for the enemies, the user interface, the management of all sound and visual effects and all the other parts that contributed to the final result.

Περίληψη

Η παρούσα πτυχιακή εργασία είχε ως στόχο την δημιουργία ενός Third-Person Action RPG παιχνιδιού επιβίωσης ζόμπι, χρησιμοποιώντας τη Unity3D Game Engine και την γλώσσα προγραμματισμού C#.

Σκοπός του παιχνιδιού είναι ο παίκτης να επιβιώσει όσα περισσότερα κύματα εχθρών μπορεί με την χρήση διαφορετικών όπλων που ξεκλειδώνει κατά την ροή του παιχνιδιού. Κάθε όπλο έχει τα δικά του χαρακτηριστικά όπως και κάθε είδος ζόμπι τα οποία με χρήση τεχνητής νοημοσύνης έχουν μοναδικό σκοπό να προστατέψουν το μέρος και να σκοτώσουν τον παίκτη. Όσο ο παίκτης προχωράει στο παιχνίδι εμφανίζονται στον χάρτη σφαίρες για τα όπλα καθώς και πακέτα ζωής που θα τον βοηθήσουν να επιβιώσει όσο περισσότερο μπορεί και να κάνει ένα νέο highscore.

Στον σχεδιασμό και την υλοποίηση δόθηκε μεγάλη προσοχή στα μικρά αλλά και στα πιο μεγάλα σφάλματα ώστε να υπάρχει ένα ικανοποιητικό επίπεδο χωρίς να επηρεάζεται η εμπειρία του παίκτη. Επίσης προστέθηκαν κάποιες μικρές λεπτομέρειες σε αρκετά σημεία του παιχνιδιού για καλύτερη αίσθηση στο παιχνίδι όπως τα animation του παίκτη, η ικανότητα αλληλεπίδρασης με αντικείμενα έχοντας δυναμικές κινήσεις και για ακόμα περισσότερο ρεαλισμό, προστέθηκε στον χαρακτήρα του παίκτη animation rigging για διάφορα σημεία του σώματος έτσι ώστε ο χαρακτήρας να έχει περιορισμούς στις κινήσεις του. Για τον λόγο αυτό, υπήρχαν αρκετά προβλήματα και δυσκολίες που έπρεπε να επιλυθούν με διάφορες τεχνικές που συχνά απαιτούσαν αρκετό ψάξιμο, δοκιμές στο παιχνίδι και περίπλοκες λύσεις. Στην εργασία αναλύεται ο κώδικας με στιγμιότυπα οθόνης, τα δέντρα συμπεριφοράς (Behavior Trees) για τους εχθρούς, η διεπαφή χρήστη, η διαχείριση όλων των ηχητικών και οπτικών εφέ και όλα τα υπόλοιπα κομμάτια που συνέβαλαν στο τελικό αποτέλεσμα.

Περιεχόμενα

Ευχαριστίες ii
Abstract iii
Περίληψη iv
1. Εισαγωγή 12
1.1 Περίληψη του παιχνιδιού 12
1.2 Κίνητρο για τη Διεξαγωγή της Εργασίας 12
1.3 Σκοπός και Στόχος της Εργασίας 12
2. Τεχνολογίες και έννοιες που χρησιμοποιήθηκαν 13
2.1 Γραφικά Υπολογιστών 13
2.2 Τι είναι τα Third Person Shooter RPG παιχνίδια 14
2.3 Τι είναι μια Παιχνιδομηχανή (Game Engine) 15
2.4 Τι είναι η Unity Game Engine 16
2.4.1 Εργαλεία της Unity που χρησιμοποιήθηκαν 16
2.4.2 Version και Render Pipeline που χρησιμοποιήθηκαν 17
3. Μεθοδολογία και Υλοποίηση 18
3.1 Επιλογή Παιχνιδομηχανής 18
3.2 Γλώσσα Προγραμματισμού και Σημαντικά Εργαλεία 18
3.2.1 Mixamo 18
3.2.2 Sketchfab 19
3.2.3 Unity Asset Store 20
3.3 Πρωτότυπο παιχνιδιού (Prototype) 21
4. Σχεδίαση του Παιχνιδιού 22
4.1 Διεπαφή Χρήστη (User Interface – UI) 22
4.1.1 Μενού 22
4.1.2 Ξεθώριασμα σκηνής (Scene Fader) 25
4.1.3 Φόρτωμα σκηνής (Loading Screen) 26
4.1.4 Gameplay UI 28
4.1.5 Εμφάνιση κειμένου ζημιάς (Damage Pop-Up Text) 29
4.1.6 Μενού Παύσης (Pause Menu) 30
4.1.7 Μενού Θανάτου (Death Menu) 31
4.1.8 Μπάρα ζωής (Healthbar) 31
4.1.9 Μπάρα Αντοχής Χτυπημάτων (Impact Resistance Bar) 32
4.1.10 Μπάρα Αντοχής Παίκτη (Player Stamina Bar) 33
4.1.11 Σημάδι Σφαίρας και Εμπόδιου για Όπλο (Decals Of Weapon) 33

4.1.12 Ραντάρ εχθρών και αντικειμένων 37
4.2 Χαρακτήρες 38
4.2.1 Χαρακτήρας Παίκτη και Συστατικά 38
4.2.1.1 Μοντέλο 38
4.2.1.2 Ελεγκτής κινήσεων χαρακτήρα (Animator controller) 39
4.2.1.3 Animations και Animation Events 43
4.2.1.4 Σύστημα Εισόδου (Input System) 45
4.2.1.5 Διαχείριση όπλων (Weapon Manager) 47
4.2.1.6 Διαχείριση Ιδιοτήτων (Attributes Manager) 48
4.2.1.7 Χειρισμός Third Person Shooter 49
4.2.1.8 Συλλογή Αντικειμένων (Grab Items) 53
4.2.1.9 Πέταγμα Αντικειμένων (Throw Items) 56
4.2.1.10 Διαχείριση ζωής (Health Controller) 57
4.2.1.11 Animation Rigging 58
4.2.2 Χαρακτήρες Εχθρών και Συστατικά 62
4.2.2.1 Μοντέλα 62
4.2.2.2 Μηχανή Καταστάσεων Εχθρών (State Machine) 63
4.2.2.3 Ζημιά Των Εχθρών 68
4.2.2.4 Ragdoll 69
4.2.2.5 Εφέ διάλυσης (Dissolve Effect) 70
4.2.2.6 Οπτικό Εφέ Αίματος (Blood VFX) 71
4.2.2.7 Ειδικός τύπος ζόμπι (Crawler Zombie) 72
5. Λοιπά Στοιχεία Παιχνιδιού 73
5.1 Διαχείριση δημιουργίας αντικειμένων – εχθρών (Spawn Manager) 73
5.2 NavMesh 77
5.3 Λεπτομέρειες στα όπλα 80
5.3.1 Κάλυκες 80
5.3.2 Γεμιστήρες 81
5.3.3 Line Renderer και Particles για τα όπλα 83
5.4 Cinemachine Impulse Listener 85
5.5 Διάφοροι ήχοι 86
5.6 Post-Processing εφέ για το νερό 88
5.7 PlayerPrefs 91
5.8 Βελτιστοποίηση Απόδοσης (Performance Optimization) 92
5.8.1 Baked Φωτισμός 92

5.8.2 Occlusion Culling 93
5.8.3 Static Batching 94
6. Επίλογος 96
6.1 Συμπεράσματα 96
6.2 Δυσκολίες 96
6.3 Μελλοντικές Βελτιώσεις και Επεκτάσεις 97
7. Βιβλιογραφία και Πηγές 98
8. Παραρτήματα 99

Λίστα Εικόνων

Εικόνα 1: Borderlands – Gameplay 14
Εικόνα 2: Monster Hunter: - Gameplay 14
Εικόνα 3: Tom Clancy's The Division - Gameplay 15
Εικόνα 4: Δημοφιλής Παιχνιδομηχανές 15
Εικόνα 5: Περιβάλλον Mixamo 19
Εικόνα 6: Περιβάλλον Sketchfab 19
Εικόνα 7: Περιβάλλον Unity Asset Store 20
Εικόνα 8: Στάδιο Δημιουργίας Πρωτότυπου 21
Εικόνα 9: Κεντρικό Μενού 22
Εικόνα 10: Λογική Κώδικα Για Τα Κουμπιά 23
Εικόνα 11: Gameobject Του Play Button 23
Εικόνα 12: Κεντρικό Μενού – Ρυθμίσεις 23
Εικόνα 13: Λογική Κώδικα Για Ήχους Των Κουμπιών 24
Εικόνα 14: Λογική Κώδικα Για Τα Sliders 24
Εικόνα 15: Gameobject Του SceneFader 25
Εικόνα 16: Φόρτωμα Σκηνής – Μπάρα Φόρτωσης 26
Εικόνα 17: Φόρτωμα Σκηνής - Έτοιμο για εκκίνηση 26
Εικόνα 18: Λογική Κώδικα Για Τις Οδηγίες 27
Εικόνα 19: Λογική Κώδικα Για Την Δημιουργία Του PopUp Κειμένου 29
Εικόνα 20: Λογική Κώδικα Για Το Animation Του PopUp Κειμένου 29
Εικόνα 21: Παράδειγμα Κειμένου Ζημιάς 29
Εικόνα 22: Μενού Παύσης - UI 30
Εικόνα 23: Button Animator 30
Εικόνα 24: Μενού Θανάτου – UI 31
Εικόνα 25: Λογική Κώδικα για Healthbar (Στιγμιότυπο 1) 31
Εικόνα 26: Λογική Κώδικα για HealthBar (Στιγμιότυπο 2) 32
Εικόνα 27: Λογική Κώδικα Για Impact Resistance Bar 32
Εικόνα 28: Textures Για Σημάδια Σφαίρας 34
Εικόνα 29: Λογική Κώδικα Για Σημάδια Σφαίρας 34
Εικόνα 30: Παράδειγμα Για Σημάδι Σφαίρας 35
Εικόνα 31: Decal Projector Του Εμποδίου 35
Εικόνα 32: Λογική Κώδικα Για Το Decal Εμποδίου 36
Εικόνα 33: Παράδειγμα Για Σημάδι Εμποδίου 36
Εικόνα 34: Prefab Locatable Player Icon 37
Εικόνα 35: Locatable Player Component 37
Εικόνα 36: Χαρακτήρας Παίκτη Από Mixamo 38
Εικόνα 37: Επίπεδα Animator Παίκτη 39
Εικόνα 38: Πρώτο Επίπεδο Animator Παίκτη 39
Εικόνα 39: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Punch 40
Εικόνα 40: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Axe 41
Εικόνα 41: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Pistol Και Rifle (Στιγμιότυπο 1) 41
Εικόνα 42: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Pistol και Rifle (Στιγμιότυπο 2) 41
Εικόνα 43: Λογική Κώδικα State MachineBehaviour – CanChangeWeapon 42
Εικόνα 44: Τρίτο Και Τέταρτο Επίπεδο Animator Παίκτη 42
Εικόνα 45: Μάσκα Avatar Για Το Πάνω Μέρος 42

Εικόνα 46: Πέμπτο Επίπεδο Animator Παίκτη 43
Εικόνα 47: Animation Events 44
Εικόνα 48: Λογική Κώδικα EnableWeaponCollider 44
Εικόνα 49: Animation Event EnableWeaponCollider 44
Εικόνα 50: Λογική Κώδικα SetHandIkTarget 45
Εικόνα 51: Input System Scheme 46
Εικόνα 52: Components Χαρακτήρα Για Τα Inputs 46
Εικόνα 53: Λογική Κώδικα WeaponManager (Στιγμιότυπο 1) 47
Εικόνα 54: Λογική Κώδικα WeaponManager (Στιγμιότυπο 2) 47
Εικόνα 55: Λογική Κώδικα WeaponManager (Στιγμιότυπο 3) 48
Εικόνα 56: Λογική Κώδικα για StopGameAfterDie 49
Εικόνα 57: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 1) 49
Εικόνα 58: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 2) 50
Εικόνα 59: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 3) 50
Εικόνα 60: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 4) 51
Εικόνα 61: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 5) 51
Εικόνα 62: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 6) 52
Εικόνα 63: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 7) 52
Εικόνα 64: GrabItems - Παράδειγμα Για Όπλα 53
Εικόνα 65: GrabItems - Παράδειγμα Για Σφαίρες 53
Εικόνα 66: Λογική Κώδικα GrabItems (Στιγμιότυπο 1) 54
Εικόνα 67: Λογική Κώδικα GrabItems (Στιγμιότυπο 2) 54
Εικόνα 68: Λογική Κώδικα GrabItems (Στιγμιότυπο 3) 55
Εικόνα 69: Λογική Κώδικα ThrowItems (Στιγμιότυπο 1) 56
Εικόνα 70: Λογική Κώδικα ThrowItems (Στιγμιότυπο 2) 57
Εικόνα 71: Λογική Κώδικα HealthController (Στιγμιότυπο 1) 57
Εικόνα 72: Λογική Κώδικα HealthController (Στιγμιότυπο 2) 58
Εικόνα 73: PistolRig Και Βάρος 59
Εικόνα 74: BodyAim Constraint 59
Εικόνα 75: HeadAim Constraint 60
Εικόνα 76: RightHand IK 60
Εικόνα 77: RightHand Constraint 61
Εικόνα 78: PickUp Rig 61
Εικόνα 79: Παράδειγμα PickUp Rig 62
Εικόνα 80: Πρώτο Μοντέλο Εχθρού 62
Εικόνα 81: Δεύτερο Μοντέλο Εχθρού 62
Εικόνα 82: Τρίτο Μοντέλου Εχθρού 62
Εικόνα 83: Τέταρτο Μοντέλο Εχθρού 62
Εικόνα 88: Μηχανή Καταστάσεων Εχθρών (State Machine) 63
Εικόνα 89: Animator Εχθρών 64
Εικόνα 90: Κατάσταση Εχθρών – Patrol 64
Εικόνα 91: Κατάσταση Εχθρών – Idle 65
Εικόνα 92: Κατάσταση Εχθρών – Chase 65
Εικόνα 93: Κατάσταση Εχθρών – Attack 66
Εικόνα 94: Παράδειγμα Μεταβλητών Για Ακινητοποίηση Εχθρού 66
Εικόνα 95: Λογική Κώδικα Για Την Ακινητοποίηση Εχθρού (Στιγμιότυπο 1) 67
Εικόνα 96: Λογική Κώδικα Για Την Ακινητοποίηση Εχθρού (Στιγμιότυπο 2) 67

Εικόνα 97: Συνάρτηση HandCollider Για Εχθρό 68
Εικόνα 98: Λογική Κώδικα Για ζημιά Στον Παίκτη 68
Εικόνα 99: Πρώτο Στάδιο Ragdoll - Άσκηση Δύναμης 69
Εικόνα 100: Δεύτερο Στάδιο Ragdoll - Επαφή Με Έδαφος 70
Εικόνα 101: Dissolve Shader 70
Εικόνα 102: Παράδειγμα Dissolve Στο Παιχνίδι 71
Εικόνα 103: Παράδειγμα Εφέ Αίματος 71
Εικόνα 104: Παράδειγμα Crawler Zombie 72
Εικόνα 105: Μηχανή Καταστάσεων Crawler Zombie (State Machine) 72
Εικόνα 106: Ξεκίνημα Παιχνιδιού (Στιγμιότυπο 1) 73
Εικόνα 107: Λογική Κώδικα SpawnManager (Στιγμιότυπο 1) 73
Εικόνα 108: Ξεκίνημα Παιχνιδιού (Στιγμιότυπο 2) 74
Εικόνα 109: Λογική Κώδικα SpawnManager (Στιγμιότυπο 2) 74
Εικόνα 110: Λογική Κώδικα SpawnManager (Στιγμιότυπο 3) 75
Εικόνα 111: Λογική Κώδικα SpawnManager (Στιγμιότυπο 4) 75
Εικόνα 112: Λογική Κώδικα SpawnManager (Στιγμιότυπο 5) 76
Εικόνα 113: Λογική Κώδικα SpawnManager (Στιγμιότυπο 6) 76
Εικόνα 114: Λογική Κώδικα SpawnManager (Στιγμιότυπο 7) 77
Εικόνα 115: NavMesh - Ρυθμίσεις Agents 78
Εικόνα 116: NavMesh - Ρυθμίσεις Object 78
Εικόνα 117: NavMesh - Ρυθμίσεις Areas 79
Εικόνα 118: NavMesh - Ρυθμίσεις Bake 79
Εικόνα 119: NavMesh - Παράδειγμα OffMeshLinks 79
Εικόνα 120: NavMeshAgent Component 80
Εικόνα 121: Λογική Κώδικα Για Κάλυκες Όπλων 81
Εικόνα 122: Παράδειγμα Παιχνιδιού Για Ρίψη Καλύκων 81
Εικόνα 123: Λογική Κώδικα Για Γεμιστήρες Όπλων 82
Εικόνα 124: Παράδειγμα Γεμίσματος Όπλου (Στιγμιότυπο 1) 82
Εικόνα 125: Παράδειγμα Γεμίσματος Όπλου (Στιγμιότυπο 2) 82
Εικόνα 126: Λογική Κώδικα για Μέθοδο SpawnBulletTrail 83
Εικόνα 127: Λογική Κώδικα LineFade 83
Εικόνα 128: Παράδειγμα LineRenderer 83
Εικόνα 129: Παράδειγμα Muzzle Flash 84
Εικόνα 130: Παράδειγμα Hit Effect 84
Εικόνα 131: Cinemachine Impulse Listener Component 85
Εικόνα 132: Λογική Κώδικα Cinemachine Impulse 85
Εικόνα 133: Λογική Κώδικα Για Ήχους Όπλων (Στιγμιότυπο 1) 86
Εικόνα 134: Λογική Κώδικα Για Ήχους Όπλων (Στιγμιότυπο 2) 86
Εικόνα 135: Εισαγωγή AudioClips Στο Component 87
Εικόνα 136: Λογική Κώδικα Για Ήχους Χαρακτήρα Παίκτη 87
Εικόνα 137: Volume Component - Channel Mixer 88
Εικόνα 138: Volume Component - Lift Gamma Gain 88
Εικόνα 139: Volume Component - White Balance 88
Εικόνα 140: Volume Component - Depth Of Field 89
Εικόνα 141: Volume Component – Vignette 89
Εικόνα 142: Film Grain 89
Εικόνα 143: Volume Component 89

Εικόνα 144: Παράδειγμα Παιχνιδιού Χωρίς Εφέ Νερού 90
Εικόνα 145: Παράδειγμα Παιχνιδιού Με Εφέ Νερού 90
Εικόνα 146: Λογική Κώδικα PlayerPrefs (Στιγμιότυπο 1) 91
Εικόνα 147: Λογική Κώδικα PlayerPrefs (Στιγμιότυπο 2) 91
Εικόνα 148: Mixed Lightning 92
Εικόνα 149: Occlusion Culling - Καρτέλα Object 93
Εικόνα 150: Occlusion Culling - Καρτέλα Bake 93
Εικόνα 151: Παράδειγμα Σκηνής Χωρίς Occlusion Culling 94
Εικόνα 152: Παράδειγμα Σκηνής Με Occlusion Culling 94
Εικόνα 153: Batching Static 95
Εικόνα 154: Στατιστικά Πριν Το Optimization 95
Εικόνα 155: Στατιστικά Μετά Το Optimization 96

Λίστα Παραρτημάτων

Παράρτημα 1 99
Παράρτημα 2 100
Παράρτημα 3 101
Παράρτημα 4 102
Παράρτημα 5 103
Παράρτημα 6 104
Παράρτημα 7 105
Παράρτημα 8 106
Παράρτημα 9 107
Παράρτημα 10 108

1. Εισαγωγή

1.1 Περίληψη του παιχνιδιού

Το "Zombie Uprising Last Stand!" είναι ένα Third Person Shooter RPG παιχνίδι στο οποίο έχεις μοναδικό σκοπό να επιβιώσεις στον κόσμο των ζόμπι όσο περισσότερο μπορείς, καθώς μεγαλύτερα και δυνατότερα κύματα εμφανίζονται στον χάρτη με την πρόοδο που κάνεις. Ο παίκτης πρέπει να παίξει έξυπνα και στρατηγικά δίνοντας προτεραιότητα στα όπλα και τα αντικείμενα που εμφανίζονται τα οποία θα τον βοηθήσουν να γίνει δυνατότερος και να ανταπεξέλθει στις ερχόμενες προκλήσεις.

Το παιχνίδι διεξάγεται στα πλημμυρισμένα εδάφη μιας μεγάλης πόλης η οποία αποτελείται από βίλες, σπίτια, εκκλησία, νεκροταφείο, αχυρώνες, γέφυρες, φάρους και αρκετή βλάστηση. Η πόλη αυτή είναι έρημη και ζουν πλέον μόνο ζόμπι τα οποία την προστατεύουν. Ο χαρακτήρας σου βρίσκεται εκεί τυχαία και προσπαθεί να επιβιώσει με ό,τι μέσα μπορεί να βρει γύρω γύρω σε φανερά αλλά και πιο απόκρυφα μέρη.

1.2 Κίνητρο για τη Διεξαγωγή της Εργασίας

Το κίνητρο που με ώθησε να ασχοληθώ με τη παιχνιδομηχανή Unity και να δημιουργήσω το πρώτο μου παιχνίδι ήταν ένα προπτυχιακό μάθημα στο οποίο ασχοληθήκαμε σε βασικό επίπεδο πάνω στην Unity. Η εκτενής έρευνα εκ τότε σε συνδυασμό με την αγάπη που έχω για τα παιχνίδια από πολύ μικρή ηλικία με οδήγησαν στην απόφαση να δημιουργήσω ένα παιχνίδι και βάση της κατηγορίας των παιχνιδιών που με ελκύουν καθώς και της δυσκολίας που ήθελα να αναπτύξω, πάρθηκε η τελική απόφαση για το συγκεκριμένο παιχνίδι.

1.3 Σκοπός και Στόχος της Εργασίας

Ο σκοπός της πτυχιακής εργασίας, είναι να δημιουργηθεί ένα παιχνίδι το οποίο δίνει την περισσότερη προσοχή στον χαρακτήρα που χειρίζεσαι και στις λεπτομέρειες ώστε να δώσει ρεαλισμό και ποιότητα. Κατά την πορεία, αναπτύχθηκαν ανάγκες ώστε να δοθεί ένα τελικό αποτέλεσμα και για τον λόγο αυτόν δημιουργήθηκαν εχθροί οι οποίοι χρησιμοποιούν τεχνητή νοημοσύνη και behavior trees ώστε να λειτουργούν έξυπνα και να δίνουν ένα επίπεδο δυσκολίας στον παίκτη. Επιπλέον έπρεπε να προστεθούν ήχοι, περιβάλλον, animation και διεπαφή χρήστη(UI) για το τελικό αποτέλεσμα.

Ο στόχος του παιχνιδιού όπως προαναφέρθηκε, είναι η επιβίωση όσων περισσότερων κυμάτων εχθρών μπορείς. Υπάρχουν 4 είδη εχθρών, κάθε ένα με διαφορετικό μέγεθος, ταχύτητα και δύναμη. Για να αντιμετωπιστούν, πρέπει ο παίκτης να βρει όπλα και σφαίρες που θα τον βοηθήσουν να σκοτώσει ευκολότερα τους εχθρούς που θα βρει στον δρόμο του. Επιπλέον, κάθε λίγο χρονικό διάστημα εμφανίζονται σφαίρες και ζωή που μπορεί να συλλέξει από το περιβάλλον. Η καλή διαχείριση των πόρων που δίνονται κατά την διάρκεια του παιχνιδιού θα κρίνει μέχρι ποιο επίπεδο θα καταφέρει ο παίκτης να φτάσει και να πετύχει μια νέα υψηλή βαθμολογία.

2. Τεχνολογίες και έννοιες που χρησιμοποιήθηκαν

2.1 Γραφικά Υπολογιστών

Τα γραφικά υπολογιστών ασχολούνται με την δημιουργία εικόνων και ταινιών στον υπολογιστή. Η είσοδος είναι συμβολικές περιγραφές της οπτικής σκηνής και η έξοδος είναι ψηφιακές εικόνες ή βίντεο. Έχουν εφαρμογές σε πολλούς τομείς όπως γραφικές διεπαφές χρήστη, βιντεοπαιχνίδια, εικονική πραγματικότητα, ταινίες, κινούμενα σχέδια και οπτικοποίηση δεδομένων. Επίσης χρησιμοποιούνται σε εικονογράφηση, αρχιτεκτονική σχεδίαση, δημιουργία λογοτύπων και ακριβή σχέδια. Τα γραφικά υπολογιστών μπορούν να χωριστούν σε δύο βασικές κατηγορίες:

Ανάλογα με τις διαστάσεις που συμμετέχουν στην απεικόνισή τους:

- **Δισδιάστατα (2D):** Τα δισδιάστατα γραφικά υπολογιστών απεικονίζουν γραφικά δύο διαστάσεων σε οθόνη ψηφιακής συσκευής, όπως ενός υπολογιστή. Συνήθως χρησιμοποιούνται για δημιουργία γραφικών διασυνδέσεων χρήστη (GUI) και εικονογραφήσεις σε περιοδικά, βιβλία και άλλα έντυπα. Αφού δημιουργηθούν, αποθηκεύονται σε ψηφιακά αρχεία εικόνας και η περαιτέρω επεξεργασία τους εμπίπτει στον τομέα της ψηφιακής επεξεργασίας εικόνας.

- **Τρισδιάστατα (3D):** Τα τρισδιάστατα γραφικά υπολογιστών απεικονίζουν γραφικά τριών διαστάσεων σε μια δισδιάστατη οθόνη ψηφιακής συσκευής. Η χρήση τριών διαστάσεων τα καθιστά ρεαλιστικά. Βασίζονται στη χωρική περιγραφή τρισδιάστατων αντικειμένων μέσω σημείων και μαθηματικών τύπων, προβάλλοντας τις συντεταγμένες των σημείων σε δύο διαστάσεις κατά την απόδοση. Συνήθως χρησιμοποιούνται σε παιχνίδια, εικονικούς κόσμους και στον κινηματογράφο για σύνθεση σκηνών και εφέ.

Ανάλογα με το χρόνο κατά τον οποίο λαμβάνει χώρα η απόδοσή τους (rendering):

- **Στατικά γραφικά υπολογιστών:** Τα στατικά γραφικά υπολογιστών είναι προ-επεξεργασμένα γραφικά που περιλαμβάνουν συντεταγμένες σημείων, επιφάνειες, χρωματισμούς, φωτισμούς και υφές. Δεν αποδίδονται κατά την προβολή, αλλά δημιουργούνται εκ των προτέρων και αποθηκεύονται ως αρχεία βίντεο. Δεν είναι διαδραστικά και παραμένουν αμετάβλητα κάθε φορά που παρακολουθούνται. Παραδείγματα περιλαμβάνουν τα μικρά βίντεο σε βιντεοπαιχνίδια. Για τη δημιουργία τους χρησιμοποιούνται προγράμματα όπως το Maya, το Blender, το Cinema4D κ.λπ.

- **Γραφικά υπολογιστών πραγματικού χρόνου:** Τα γραφικά πραγματικού χρόνου απεικονίζονται αμέσως κατά την εκτέλεση ενός προγράμματος υπολογιστή. Παραδείγματα περιλαμβάνουν τα γραφικά σε βιντεοπαιχνίδια. Χρησιμοποιούν μηχανές όπως η Ogre3D και το Crystal Space. Μπορεί να ανταποκριθούν σε εισόδους του χρήστη, αν και δεν είναι απαραίτητο. Για τον προγραμματισμό τους χρησιμοποιούνται βιβλιοθήκες όπως η OpenGL και το Direct3D.

2.2 Τι είναι τα Third Person Shooter RPG παιχνίδια

Τα Third Person Shooter RPG είναι ένα είδος βιντεοπαιχνιδιού που συνδυάζει στοιχεία των παιχνιδιών με πυροβολισμούς και των παιχνιδιών ρόλων. Σε αυτό το είδος, οι παίκτες υιοθετούν το ρόλο ενός χαρακτήρα σε ένα εικονικό κόσμο, συμμετέχοντας συνήθως σε σενάρια μάχης από μια οπτική γωνία τρίτου προσώπου, όπου η κάμερα είναι τοποθετημένη πίσω και ελαφρώς πάνω από τον χαρακτήρα. Αντίθετα με τους παραδοσιακούς πυροβολισμούς, τα Third Person Shooter RPGs περιλαμβάνουν στοιχεία RPG, όπως εξατομίκευση του χαρακτήρα, πρόοδο δεξιοτήτων και λήψη αποφάσεων που οδηγούν την αφήγηση, επιτρέποντας στους παίκτες να διαμορφώσουν την εξέλιξη και τις ικανότητες του ψηφιακού τους εαυτού. Η σύνθεση δράσης και βάθους αφήγησης σε αυτά τα παιχνίδια προσφέρει στους παίκτες μια δυναμική και εμβαθυμένη εμπειρία, τονίζοντας τόσο τις στρατηγικές μάχες όσο και την ανάπτυξη χαρακτήρα σε έναν πλούσιο και συχνά εκτενή κόσμο παιχνιδιού. Παρακάτω φαίνονται μερικά παραδείγματα γνωστών παιχνιδιών όπως το Borderlands, το Monster Hunter: World και το Tom Clancy's The Division.



Εικόνα 1: Borderlands – Gameplay



Εικόνα 2: Monster Hunter: - Gameplay



Εικόνα 3: Tom Clancy's The Division - Gameplay

2.3 Τι είναι μια Παιχνιδομηχανή (Game Engine)

Μια μηχανή παιχνιδιού είναι λογισμικό που σχεδιάζεται για τη δημιουργία και ανάπτυξη βιντεοπαιχνιδιών. Μπορεί να λειτουργεί σε κονσόλες βιντεοπαιχνιδιών και υπολογιστές με λειτουργικά συστήματα όπως τα Microsoft Windows, Linux και Mac OS X. Οι βασικές λειτουργίες περιλαμβάνουν απόδοση γραφικών (2D ή 3D), ανίχνευση συγκρούσεων, ήχο, scripting, animation, τεχνητή νοημοσύνη, δικτύωση, streaming, διαχείριση μνήμης, νήματα, και δομή σκηνής. Συχνά, η ίδια μηχανή παιχνιδιού χρησιμοποιείται για τη δημιουργία διαφορετικών παιχνιδιών.



Εικόνα 4: Δημοφιλής Παιχνιδομηχανές

2.4 Τι είναι η Unity Game Engine

Η Unity είναι μια πολλαπλών πλατφορμών μηχανή παιχνιδιών που αναπτύχθηκε από την Unity Technologies. Κυκλοφόρησε για πρώτη φορά τον Ιούνιο του 2005 στο Apple Worldwide Developers Conference ως μηχανή παιχνιδιών Mac OS X. Από τότε, έχει επεκταθεί για να υποστηρίζει πολλές πλατφόρμες, συμπεριλαμβανομένων επιτραπέζιων υπολογιστών, κινητών, κονσόλων και εικονικής πραγματικότητας. Είναι δημοφιλές για την ανάπτυξη παιχνιδιών για iOS και Android, θεωρείται εύκολο για αρχάριους προγραμματιστές και είναι αγαπητό για τη δημιουργία indie παιχνιδιών.

Η Unity μπορεί να χρησιμοποιηθεί για τη δημιουργία τόσο τρισδιάστατων (3D) όσο και δισδιάστατων (2D) παιχνιδιών, καθώς και διαδραστικών προσομοιώσεων και άλλων εμπειριών. Επίσης, έχει χρησιμοποιηθεί σε διάφορους τομείς εκτός των βιντεοπαιχνιδιών, όπως η μηχανική, ο κινηματογράφος, η αυτοκινητοβιομηχανία, η αρχιτεκτονική και οι κατασκευές. Η Unity περιλαμβάνει σημαντικά πλεονεκτήματα στα οποία γίνεται μια μικρή αναφορά παρακάτω:

- **Κατάστημα Πόρων (Asset Store):** Η Unity διαθέτει ένα εκτενές κατάστημα πόρων όπου οι προγραμματιστές μπορούν να βρουν προκατασκευασμένους πόρους, όπως 3D μοντέλα, textures, ήχους, και scripts.
- **Γλώσσες Προγραμματισμού:** Η Unity υποστηρίζει πολλές γλώσσες προγραμματισμού, συμπεριλαμβανομένων των C#, JavaScript, και Boo.
- **Ολοκληρωμένα Εργαλεία Ανάπτυξης:** Η Unity περιλαμβάνει μια πλήρη σειρά εργαλείων για την ανάπτυξη παιχνιδιών, συμπεριλαμβανομένων AI, Physics Engine, και πολλά άλλα.
- **Υποστήριξη Δικτύωσης και Multiplayer:** Η Unity παρέχει χαρακτηριστικά για τη δημιουργία online παιχνιδιών με πραγματικό χρόνο επικοινωνίας μεταξύ των παικτών.
- **Υποστήριξη για Virtual και Augmented Reality:** Η Unity υποστηρίζει ενεργά την ανάπτυξη παιχνιδιών για εικονική και επαυξημένη πραγματικότητα.
- **Κοινότητα και Forum:** Η Unity έχει μια μεγάλη και ενεργή κοινότητα προγραμματιστών, καλλιτεχνών, και ενθουσιαστών όπου οι χρήστες μπορούν να ανταλλάξουν απόψεις, να ζητήσουν βοήθεια, και να συνεργαστούν σε έργα.

2.4.1 Εργαλεία της Unity που χρησιμοποιήθηκαν

- **Animator:** Το σύστημα Animator της Unity επιτρέπει τη δημιουργία πολύπλοκων κινήσεων και μεταβάσεων για τους χαρακτήρες και τα αντικείμενα του παιχνιδιού, προσδίδοντας ζωντάνια και ρεαλισμό στην εμπειρία.
- **PlayerPrefs:** Το PlayerPrefs είναι μια απλή αλλά χρήσιμη βάση δεδομένων για την αποθήκευση και τη διατήρηση ρυθμίσεων και δεδομένων του χρήστη, όπως βαθμολογίες και πρόοδος σε παιχνίδια.
- **AudioMixer:** Το AudioMixer επιτρέπει τον έλεγχο του ήχου στο παιχνίδι με ακρίβεια, προσφέροντας ρύθμιση εφέ, ισορροπία ήχου και άλλες ηχητικές δυνατότητες.
- **Animation Rigging:** Το Animation Rigging επιτρέπει τη δημιουργία προηγμένων κινήσεων και μεταβάσεων μέσω προγραμματισμού, βοηθώντας στη δημιουργία ρεαλιστικών χαρακτήρων.

- **Cinemachine:** Το Cinemachine είναι ένα ισχυρό σύστημα για τη δημιουργία κινηματογραφικών σκηνών, επιτρέποντας την δυναμική ρύθμιση της κάμερας για εντυπωσιακές γωνίες λήψης.
- **Unity Asset Store:** Το Unity Asset Store είναι μια πλατφόρμα που προσφέρει προκατασκευασμένους πόρους και εργαλεία για τη δημιουργία των παιχνιδιών, εξοικονομώντας χρόνο και πόρους στους προγραμματιστές.
- **TextMesh Pro:** Το TextMesh Pro βελτιώνει την απεικόνιση κειμένου στο παιχνίδι με προηγμένη δυναμική δημιουργία και μορφοποίηση κειμένου.
- **Occlusion Culling:** Το Occlusion Culling βελτιώνει την απόδοση του παιχνιδιού με αυτόματο τρόπο, αποκρύπτοντας αόρατα αντικείμενα.
- **NavMesh:** Το NavMesh δημιουργεί αυτόματα μονοπάτια για την τεχνητή νοημοσύνη των χαρακτήρων, επιτρέποντάς τους να περιηγούνται στον χώρο με ευφυΐα.
- **Terrain:** Το Terrain είναι ένα ισχυρό εργαλείο που επιτρέπει τη δημιουργία, τον περιβάλλοντα σχεδιασμό και την αναπαράσταση εκτεταμένων φυσικών τοπίων σε παιχνίδια, προσθέτοντας ρεαλισμό και ποικιλία στον κόσμο του παιχνιδιού.
- **Particle System:** Το Particle System στην Unity είναι ένα εργαλείο που επιτρέπει τη δημιουργία και αναπαραγωγή εφέ σωματιδίων στο παιχνίδι. Μπορεί να χρησιμοποιηθεί για δημιουργία εντυπωσιακών γραφικών όπως σκόνη, φωτιές και άλλα εφέ που προσθέτουν ρεαλισμό και ατμόσφαιρα στον κόσμο του παιχνιδιού.
- **Input System:** Το Input System της Unity προσφέρει ενιαίο σύστημα διαχείρισης εισόδων από διάφορες πηγές, όπως πληκτρολόγιο, ποντίκι, χειριστήρια κ.α. Επιτρέπει εύκολη αντικατάσταση και διαχείριση των εισόδων ανάμεσα σε διάφορες πλατφόρμες και συσκευές.
- **Ragdoll:** Το Ragdoll είναι ένα σύστημα φυσικής που επιτρέπει στα χαρακτηριστικά ενός χαρακτήρα να αντιδρούν φυσικά στο περιβάλλον. Αυτό συνεπάγεται την προσομοίωση των κινήσεων και των αντιδράσεων του σώματος όταν εκτίθενται σε διάφορες δυνάμεις, προσφέροντας ρεαλιστικές κινήσεις και αντιδράσεις.

2.4.2 Version και Render Pipeline που χρησιμοποιήθηκαν

Για την δημιουργία του συγκεκριμένου παιχνιδιού χρησιμοποιήθηκε η έκδοση 2021.3.10f1. Η επιλογή έγινε γιατί ήταν η τελευταία έκδοση LTS που υπήρχε. Οι LTS (Long Term Support) εκδόσεις της Unity προσφέρουν σημαντικά οφέλη για τους προγραμματιστές και τις ομάδες ανάπτυξης. Αυτές οι εκδόσεις παρέχουν μακροπρόθεσμη υποστήριξη, περιλαμβάνοντας σταθερότητα, βελτιώσεις ασφάλειας και διορθώσεις σφαλμάτων για μεγαλύτερο χρονικό διάστημα. Αυτό επιτρέπει στους προγραμματιστές να δουλεύουν σε μια στιβαρή και αξιόπιστη πλατφόρμα, χωρίς την ανάγκη συχνών αλλαγών ή αναβαθμίσεων στον πυρήνα της μηχανής.

Το πρότζεκτ ξεκίνησε από το βασικό Built-in Render Pipeline. Η μετάβαση στο Universal Render Pipeline (URP) αποτέλεσε μια κρίσιμη απόφαση για τη βελτίωση της ποιότητας των γραφικών και των λειτουργιών του. Ενώ το Built-in λειτουργούσε άψογα, η ανάγκη για την προσθήκη "decals" οδήγησε στη μετάβαση. Τα "decals" στο Unity είναι 2D ή 3D αντικείμενα που χρησιμοποιούνται για να προσθέσουν δυναμικά λεπτομέρειες ή υφές σε επιφάνειες σε μια σκηνή. Συνήθως χρησιμοποιούνται για να προσομοιώσουν ατέλειες επιφανειών, όπως τρύπες από σφαίρες, λάσπες, αίματα ή άλλα σήματα. Εφαρμόζονται πάνω στην υπάρχουσα γεωμετρία των αντικειμένων και μπορούν να βελτιώσουν το ρεαλισμό και το οπτικό ενδιαφέρον μιας σκηνής.

3. Μεθοδολογία και Υλοποίηση

3.1 Επιλογή Παιχνιδομηχανής

Για την ανάπτυξη του παιχνιδιού με θέμα τα ζόμπι, επιλέχθηκε η Unity ως η μηχανή παιχνιδιών. Η επιλογή της Unity προήλθε φυσικά από την εξοικείωσή μου με τη μηχανή από προηγούμενες εμπειρίες στον ακαδημαϊκό χώρο. Οι βασικές γνώσεις που είχα λάβει, με οδήγησαν να χρησιμοποιήσω την Unity για να επιταχύνω τη διαδικασία ανάπτυξης και να αποφύγω την εκ νέου εκμάθηση μιας άλλης παιχνιδομηχανής. Η ευελιξία της Unity, η εκτενής υποστήριξη της κοινότητας και οι πολλαπλές δυνατότητές της συνέβαλαν στην καταλληλότητα της για το εργασία.

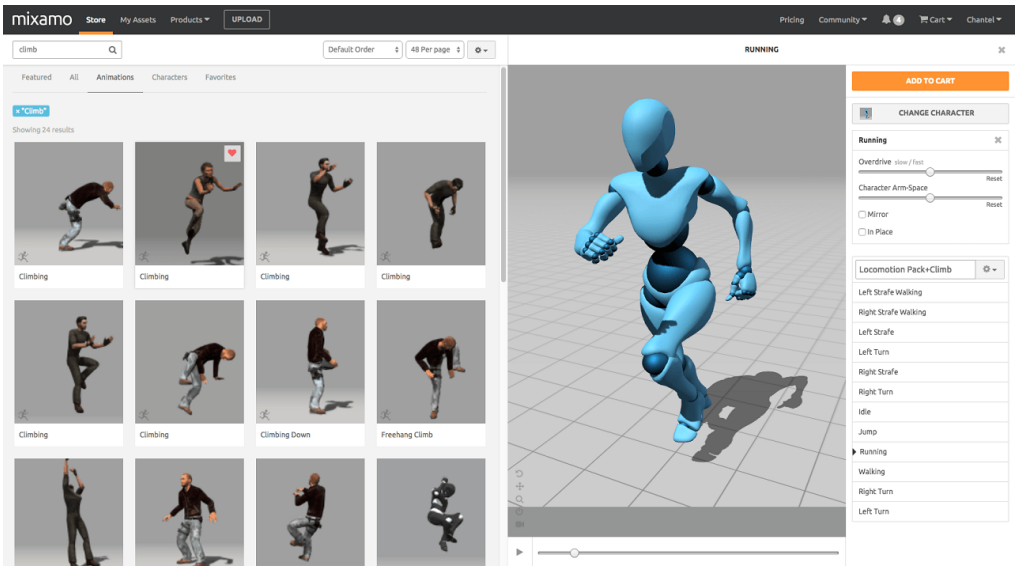
3.2 Γλώσσα Προγραμματισμού και Σημαντικά Εργαλεία

Η κύρια γλώσσα προγραμματισμού που χρησιμοποιήθηκε στην ανάπτυξη του παιχνιδιού ήταν η C#, καθώς είναι η κύρια γλώσσα σεναρίου που υποστηρίζεται από τη Unity. Στην επιλογή να χρησιμοποιηθεί η C# βοήθησε και η προηγούμενη εμπειρία μου με αντικειμενοστραφής γλώσσες όπως η Java, καθώς και η C# ανήκει σε αυτήν την κατηγορία. Έτσι επιτάχυνα τη διαδικασία κώδικα ώστε να έχω μια πιο αποτελεσματική ροή εργασίας.

Η ανάπτυξη ενός παιχνιδιού όταν είσαι ο μοναδικός δημιουργός μπορεί να αποτελεί πρόκληση, και για να επιτύχει κανείς ρεαλιστικά και ποιοτικά αποτελέσματα, είναι σημαντικό να αξιοποιεί τις κατάλληλες πηγές. Για τα 3D μοντέλα μου, απέφυγα τον χρονοβόρο και πολύπλοκο σχεδιασμό και χρησιμοποίησα κάποια εργαλεία τα οποία πρόσφεραν έτοιμα μοντέλα, animation, ήχους και διεπαφές χρήστη τα οποία έπρεπε να τροποποιήσω και να προσαρμόσω κατάλληλα για να ταιριάζουν στις ανάγκες μου. Οι τρεις βασικές πλατφόρμες που συνείσφεραν σε αυτόν ήταν το Mixamo, το sketchfab και η ηλεκτρονική αγορά της Unity (Unity Asset Store).

3.2.1 Mixamo

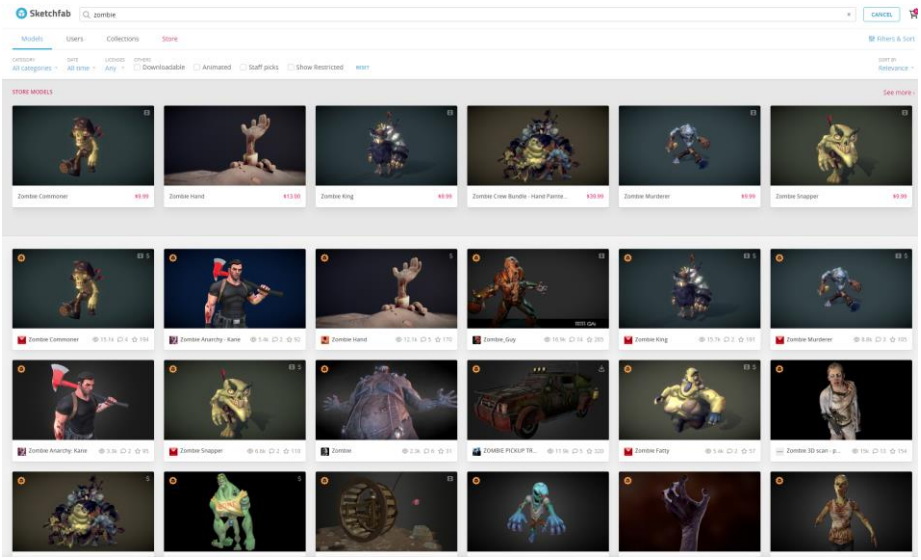
Το Mixamo αποτελεί ένα ανεκτίμητο εργαλείο για τους προγραμματιστές και τους δημιουργούς παιχνιδιών που αναζητούν γρήγορες και ρεαλιστικές λύσεις για τα 3D animations τους. Η πλατφόρμα παρέχει μια εκτενή συλλογή από κινήσεις χαρακτήρων (animations), καλύπτοντας ένα μεγάλο εύρος κοινού. Επιπλέον, σου επιτρέπει την αυτόματη προσθήκη σκελετού σε οποιοδήποτε 3D χαρακτήρα, εκτός αυτών που προσφέρονται ήδη δωρεάν, με εύκολο και γρήγορο τρόπο, και παρέχει μια φιλική προς τον χρήστη διεπαφή που επιτρέπει την προσαρμογή και εφαρμογή των animations. Η δυνατότητα εξαγωγής των animations σε διάφορες μορφές καθιστά συμβατό το Mixamo με διάφορα περιβάλλοντα ανάπτυξης. Με την ευελιξία και την ποικιλία που παρέχει, το Mixamo συμβάλλει σημαντικά στη δημιουργία ρεαλιστικών και εντυπωσιακών animation για τα παιχνίδια αλλά και άλλες εφαρμογές.



Εικόνα 5: Περιβάλλον Mixamo

3.2.2 Sketchfab

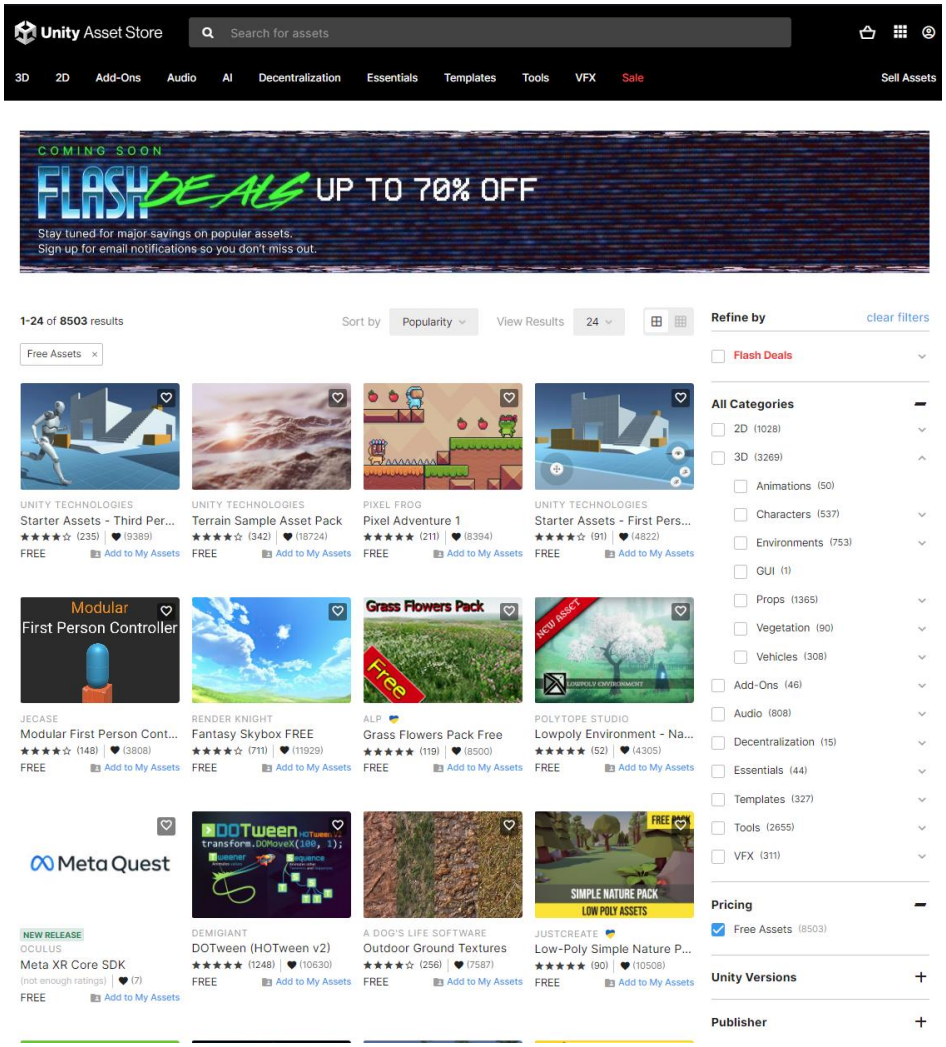
Το Sketchfab θεωρείται μια από τις κορυφαίες πλατφόρμες για τη δημιουργία, την κοινοποίηση και την αναζήτηση 3D περιεχομένου. Με έναν φιλικό προς τον χρήστη περιβάλλον, το Sketchfab επιτρέπει σε δημιουργούς από όλο τον κόσμο να παρουσιάζουν τα 3D μοντέλα τους. Η πλατφόρμα διαθέτει εκατομμύρια 3D μοντέλα, καλύπτοντας ένα ευρύ φάσμα κατηγοριών, από καλλιτεχνικά έργα μέχρι μοντέλα παιχνιδιών. Αξίζει να σημειωθεί ότι υπάρχει ένα πρόσθετο (plugin) του Sketchfab για τη Unity, που παρέχει μια ακόμη πιο άνετη διαδικασία ενσωμάτωσης 3D περιεχομένου στην παιχνιδομηχανή. Αυτό το πρόσθετο δίνει στους προγραμματιστές τη δυνατότητα να αναζητούν και να επιλέγουν 3D μοντέλα από την πλατφόρμα του Sketchfab απευθείας μέσα από το περιβάλλον του Unity.



Εικόνα 6: Περιβάλλον Sketchfab

3.2.3 Unity Asset Store

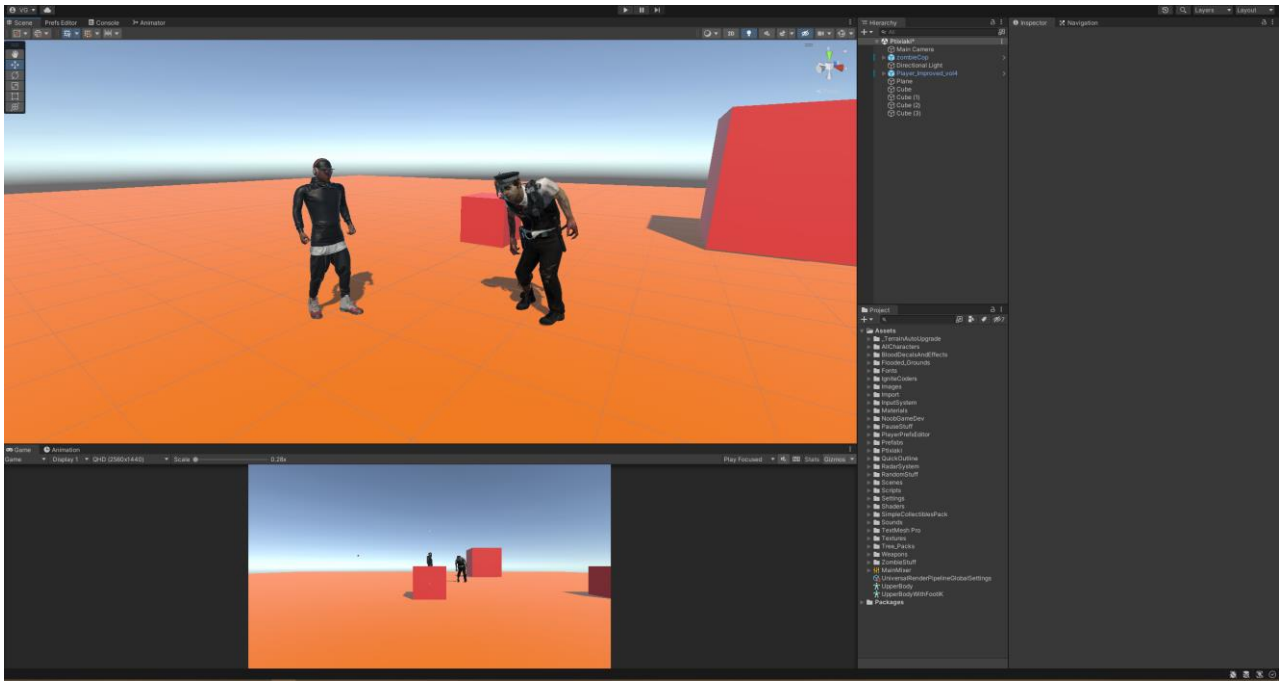
Το Unity Asset Store αντιπροσωπεύει έναν πολύτιμο πόρο για τους προγραμματιστές και τους δημιουργούς παιχνιδιών, προσφέροντας μια εκτενή συλλογή από πρόσθετα, εργαλεία που επεκτείνουν τις δυνατότητες της Unity. Με μια ποικιλία από assets όπως 3D μοντέλα, textures, animations, scripts, και πολλά άλλα, οι χρήστες μπορούν να ενισχύσουν τα παιχνίδια τους και να επιτύχουν εντυπωσιακά αποτελέσματα χωρίς την ανάγκη να δημιουργούν τα πάντα από την αρχή. Η διεπαφή του Unity Asset Store επιτρέπει μια εύκολη αναζήτηση, προεπισκόπηση και ενσωμάτωση των assets στο περιβάλλον ανάπτυξης της Unity, επιτρέποντας στους δημιουργούς να επωφεληθούν από την κοινότητα που μοιράζεται τις γνώσεις και τη δημιουργικότητά της. Μπορείς να βρεις asset τόσο δωρεάν όσο και επί πληρωμή όπως επίσης και να προσθέσεις δικά σου τα οποία μπορούν να χρησιμοποιήσουν ή να αγοράσουν οι χρήστες ανάλογα με τις ανάγκες τους.



Εικόνα 7: Περιβάλλον Unity Asset Store

3.3 Πρωτότυπο παιχνιδιού (Prototype)

Το αρχικό πρωτότυπο του παιχνιδιού αποτελεί τον βασικό πυρήνα και την αφετηρία της ανάπτυξης. Σε αυτό το στάδιο, επικεντρώθηκα κυρίως στην ενσωμάτωση βασικών στοιχείων, με έμφαση στην κίνηση του χαρακτήρα. Για να επιτευχθεί αυτό γρήγορα χρησιμοποιήθηκε το asset [asset 1] της unity για χειρισμό χαρακτήρα σε τρίτο πρόσωπο (third person) που περιλαμβάνει και ένα προκαθορισμένο 3D μοντέλο. Μετά έγινε η αλλαγή του μοντέλου σε έναν χαρακτήρα που εισάχθηκε από το Mixamo, και φτιάχτηκε ένα αρχικό περιβάλλον με πάτωμα και κουτιά για εμπόδια για τις πρώτες δοκιμές. Εφόσον όλα λειτούργησαν, συνέχισα με προσθήκη του πρώτου εχθρού – ζόμπι όπου μέσω της χρήσης του NavMesh Agent της Unity προσαρμόστηκε ώστε να ακολουθάει τον παίκτη. Η ανάγκη για επίθεση στον παίκτη οδήγησε στην εισαγωγή των πρώτων animation. Αφού βρήκα κάποιο που ταίριαζε, το έβαλα στο ζόμπι και έτσι είχα το πρώτο στάδιο όπου υπάρχουν εχθροί που σε κυνηγάνε και σου επιτίθονται. Έτσι, έχει ολοκληρωθεί ένα πρωτότυπο στο οποίο ξεκίνησα να ‘χτίζω’ πάνω σε αυτό διάφορες ιδέες όπως όπλα τα οποία έχει δυνατότητα ο παίκτης να πάρει και να τα πετάξει, μηχανισμός ενέργειας, μηχανισμός ακινητοποίησης εχθρών, θεράπευση παίκτη, θάνατος και εξαφάνιση εχθρών με χρήση dissolve shaders και πολλά ακόμα που αναλύονται στο επόμενο κεφάλαιο.



Εικόνα 8: Στάδιο Δημιουργίας Πρωτότυπου

4. Σχεδίαση του Παιχνιδιού

4.1 Διεπαφή Χρήστη (User Interface – UI)

Η Διεπαφή Χρήστη αντιπροσωπεύει έναν σημαντικό παράγοντα στην επιτυχία και την απήχηση ενός παιχνιδιού. Η σχεδίαση του καθορίζει την εμπειρία του παίκτη, καθοδηγώντας τον και παρέχοντας ουσιαστικές πληροφορίες. Ένα καλά σχεδιασμένο UI απλοποιεί την πλοήγηση προσφέροντας μια καλή εμπειρία στον χρήστη. Συγχρόνως, με κατάλληλη τοποθέτηση στοιχείων όπως μετρητές, χάρτες, και εικονίδια, το UI διαμορφώνει την ατμόσφαιρα του παιχνιδιού. Παρακάτω αναλύεται πιο συγκεκριμένα που δημιουργήθηκε και πως συνδέεται με τον κώδικα ώστε να λειτουργεί και δυναμικά κατά την διάρκεια του παιχνιδιού.

4.1.1 Μενού

Το κεντρικό μενού και η διεπαφή χρήστη αναδεικνύονται ως καίριοι παράγοντες στον σχεδιασμό του παιχνιδιού, αποτελώντας την πρώτη επαφή του παίκτη με το παιχνίδι. Για την πτυχιακή εργασία το μενού που δημιούργησα αποτελείται από μια εικόνα φόντου που βρήκα με μια διαλυμένη πόλη για να ταιριάζει με το θέμα του παιχνιδιού και τρία κουμπιά στα οποία έχει προστεθεί λειτουργικότητα.



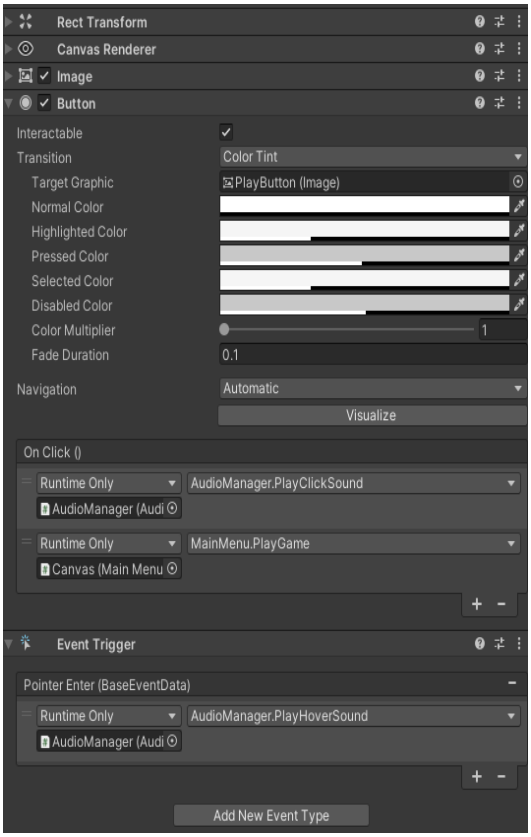
Εικόνα 9: Κεντρικό Μενού

Για αρχή δημιούργησα έναν καμβά από το Unity Editor στον οποίο έβαλα την εικόνα και τα τρία κουμπιά που χρειάζομαι. Η Unity επιτρέπει να βάλεις διάφορα event κάθε φορά που πατάς το συγκεκριμένο κουμπί. Ξεκινώντας από το πιο απλό, Το κουμπί ‘Quit’ τερματίζει το παιχνίδι. Το κουμπί ‘Settings’ κρύβει το Gameobject του κεντρικού μενού και ανοίγει το αντίστοιχο για τις ρυθμίσεις και το ‘Play’ φορτώνει την επόμενη σκήνη. Παρακάτω θα μιλήσουμε για αυτό.

Κάθε κουμπί έχει OnClick event στο οποίο έχει οριστεί μια μέθοδος που τρέχει. Παρακάτω φαίνεται για το κουμπί Play το οποίο τρέχει την μέθοδο PlayGame() και μέσω του Scene Fader αλλάζει σκηνή. Αντίστοιχα και τα υπόλοιπα κουμπιά τρέχουν τις δικές τους μεθόδους.

```
[SerializeField] GameObject mainMenu;
[SerializeField] GameObject settingsMenu;
[SerializeField] SceneFader fader;
private void Start()
{
    fader.FadeInGameOrMenu();
}
//μεθοδος για το κουμπι play
public void PlayGame()
{
    fader.FadeToGame("Game");
}
//μεθοδος για το κουμπι settings
public void SettingsOpen()
{
    mainMenu.SetActive(false);
    settingsMenu.SetActive(true);
}
public void SettingsClose()
{
    mainMenu.SetActive(true);
    settingsMenu.SetActive(false);
}
//μεθοδος για το κουμπι quit
public void QuitGame()
{
    Application.Quit();
}
```

Εικόνα 10: Λογική Κώδικα Για Τα Κουμπιά



Εικόνα 11: Gameobject Του Play Button

Όταν πατηθεί το κουμπί 'Settings' τότε το UI αλλάζει μορφή και ενεργοποιείται το άλλο Gameobject το οποίο περιέχει κουμπιά, sliders, dropdowns και οδηγίες για το παιχνίδι.



Εικόνα 12: Κεντρικό Μενού – Ρυθμίσεις

Μέσα στο Gameobject για τις ρυθμίσεις υπάρχουν δύο dropdown για την ποιότητα των γραφικών και την ανάλυση του παιχνιδιού, ένα κουμπί για να ενεργοποιήσει ή απενεργοποιήσει το fullscreen, δύο sliders που τροποποιούν την ένταση του ήχου και των διάφορων sound effects που υπάρχουν, ένα απλό κείμενο που δείχνει στον παίκτη πως να παίξει το παιχνίδι και τέλος ένα κουμπί για να γυρίσει στο κεντρικό μενού. [παράρτημα 1] Παρακάτω φαίνεται ο κώδικας που αναπαράγει τον ήχο των κουμπιών.

```
public void PlayHoverSound()
{
    if (pauseButtonsSource.gameObject.activeSelf)
        pauseButtonsSource.PlayOneShot(hoverButton);
    else
        gameOverButtonSource.PlayOneShot(hoverButton);
}
public void PlayClickSound()
{
    if (pauseButtonsSource.gameObject.activeSelf)
        pauseButtonsSource.PlayOneShot(clickButton);
    else
        gameOverButtonSource.PlayOneShot(clickButton);
}
public void PlayClickBackSound()
{
    if (pauseButtonsSource.gameObject.activeSelf)
        pauseButtonsSource.PlayOneShot(clickBackButton);
    else
        gameOverButtonSource.PlayOneShot(clickBackButton);
}
```

Εικόνα 13: Λογική Κώδικα Για Ήχους Των Κουμπιών

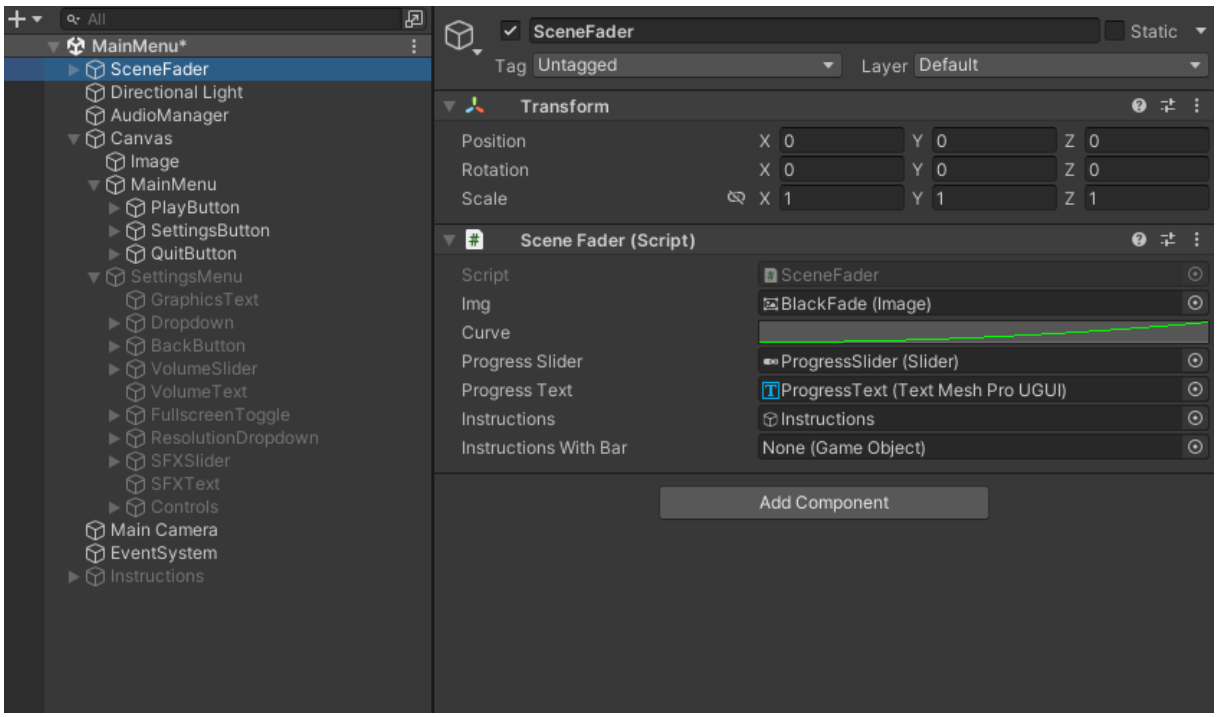
Επίσης, συγκεκριμένα για τα sliders, για να υπάρχει συσχέτιση με τον ήχο του παιχνιδιού, περνάμε το Audio Mixer, που είναι υπεύθυνο για τους ήχους, μέσα στον κώδικα και από εκεί μέσω events (συγκεκριμένα on Value Change) αλλάζουμε τις τιμές και έτσι πετυχαίνουμε να αλλάζει η ένταση ανάλογα το σημείο που αφήνουμε τα sliders.

```
public void SetMusicVolume(float value)
{
    audioMixer.SetFloat("MusicVolume", value);
}
public void SetSFXVolume(float value)
{
    audioMixer.SetFloat("SFXVolume", value);
    audioMixer.SetFloat("TerrainVolume", value);
}
```

Εικόνα 14: Λογική Κώδικα Για Τα Sliders

4.1.2 Ξεθώριασμα σκηνής (Scene Fader)

Το ξεθώριασμα σκηνής, γνωστό και ως ‘Scene Fader’ αποτελεί σημαντικό στοιχείο στον χώρο των παιχνιδιών, συμβάλλοντας στη δημιουργία μιας ομαλής μετάβασης μεταξύ σκηνών με τη χρήση χρωμάτων όπως το μαύρο. Αυτός ο τρόπος αναπαράστασης ομαλοποιεί τις μεταβάσεις, αποφεύγοντας απότομες αλλαγές και ενισχύοντας την συνοχή του παιχνιδιού. Επιπλέον, το ξεθώριασμα σκηνής μπορεί να χρησιμοποιηθεί για να δημιουργήσει ατμόσφαιρα, να αποκαλύψει σημαντικές στιγμές ή να ενισχύσει τον αφηγηματικό χαρακτήρα του παιχνιδιού. Για τον λόγο αυτόν, δημιούργησα την δική μου εκδοχή η οποία χρησιμοποιείται κάθε φορά που φορτώνει μια άλλη σκηνή ή φορτώνει από την αρχή η ίδια (π.χ. όταν κάνουμε retry στο παιχνίδι).

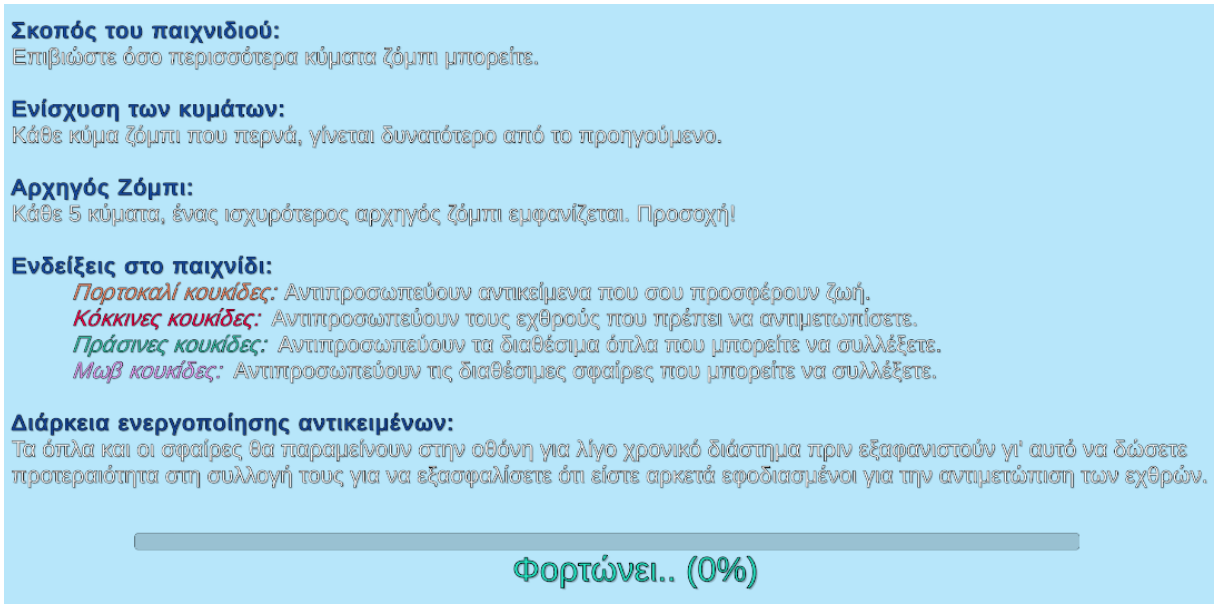


Εικόνα 15: Gameobject Του SceneFader

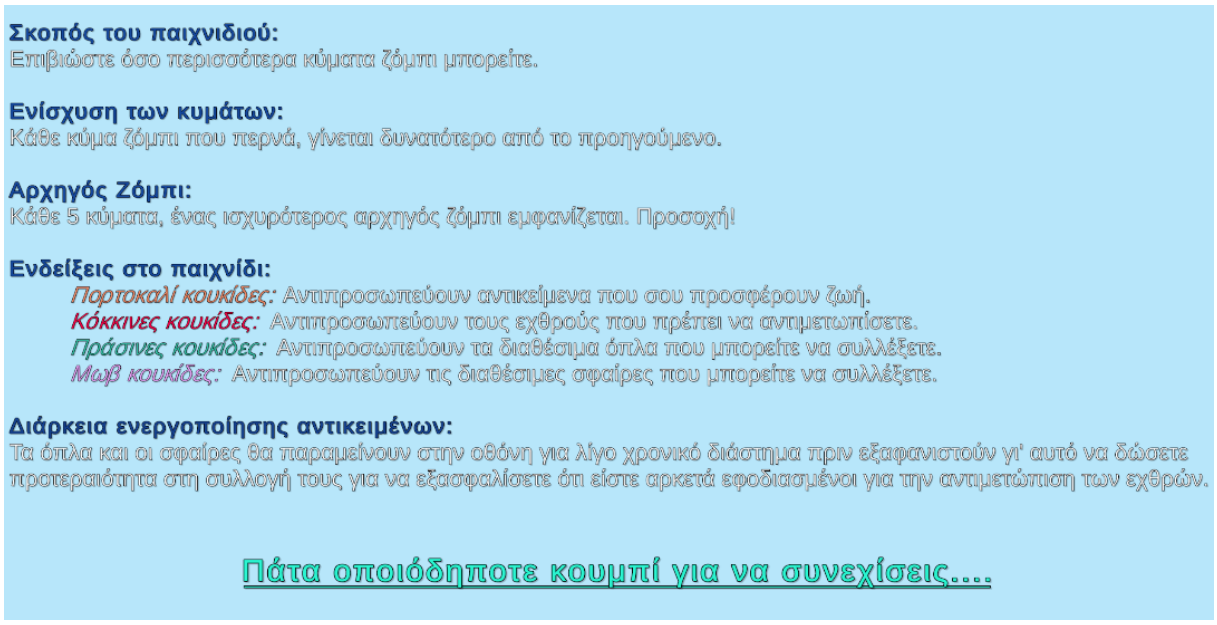
Όπως φαίνεται υπάρχει μια μαύρη εικόνα που χρησιμοποιείται πειράζοντας την alpha τιμή της και μια καμπύλη που καθορίζει το πόσο αργά ή γρήγορα γίνεται η μετάβαση. Στον κώδικα [παράρτημα 2] υπάρχουν δύο βασικές μέθοδοι (FadeIn – FadeOut) οι οποίες κάθε φορά που καλούνται στην ουσία εμφανίζουν ή εξαφανίζουν ομαλά την μαύρη εικόνα, πάνω από όλα τα άλλα, μέσα σε χρονικό διάστημα που έχω ορίσει και δίνει μια ωραία αίσθηση σε σχέση με όταν αλλάζει η σκήνη απότομα.

4.1.3 Φόρτωμα σκηνής (Loading Screen)

Όταν ο παίκτης πατήσει να παίξει, αρχίζει να φορτώνει η δεύτερη σκηνή που είναι η σκηνή του παιχνιδιού. Όσο φορτώνει, εμφανίζεται ένα νέο UI στο οποίο αναγράφονται περισσότερες πληροφορίες για το παιχνίδι. Παράλληλα, στο κάτω μέρος υπάρχει μια μπάρα φόρτωσης (loading bar) που γεμίζει ασύγχρονα ανάλογα με το πόσο έχει προχωρήσει το φόρτωμα της σκηνής. Είναι μια γνωστή τεχνική που χρησιμοποιούν από τα πιο μικρά μέχρι τα πιο μεγάλα παιχνίδια. Όταν πλέον τελειώσει το φόρτωμα, ο παίκτης έχει την επιλογή να πατήσει οποιοδήποτε πλήκτρο ώστε να προχωρήσει στο παιχνίδι ή να περιμένει λίγο μέχρι να είναι έτοιμος να ξεκινήσει.



Εικόνα 16: Φόρτωμα Σκηνής – Μπάρα Φόρτωσης



Εικόνα 17: Φόρτωμα Σκηνής - Έτοιμο για εκκίνηση

Η λογική του κώδικα είναι σχετικά απλή εδώ καθώς το μόνο που κάνουμε είναι να ορίζουμε τον χρόνο του παιχνιδιού ίσο με 0 ώστε να μην προχωράει πριν ο παίκτης πατήσει να παίξει. Όταν πατηθεί οποιοδήποτε πλήκτρο, όπως φαίνεται ξεκινάει ο χρόνος, γίνεται fade στην σκηνή και ξεκινάει το παιχνίδι.

```
private void Start()
    {
        DisableAfterDelay();
    }


    //χρησιμοποιω lateUpdate ωστε να τρεξει πρωτα το pause, να μην ανοιξει
    //εφοσον εχω ανοιχτα τα instructions και μετα να κλεισει τα instructions
    //το κανω αυτο για να μην ανοιγει το pause οταν κλεινω τα instructions με το escape
    void LateUpdate()
    {
        if (Input.anyKeyDown)
        {
            CloseInstructions();
        }
    }

    void CloseInstructions()
    {
        InputSystem.settings.updateMode = InputSettings.UpdateMode.ProcessEventsInFixedUpdate;
        //εξαφανιζω τον κερσορα
        _input.SetCursorState(true);
        //κανω τον χρονο 1 οποτε ξεκινανε ολα παλι να τρεχουνε
        Time.timeScale = 1f;
        fader.FadeInGameOrMenu();
        _playerInput.actions.Enable();
        gameObject.SetActive(false);
        spawnManager.StartFadeStartingText();
    }


    void DisableAfterDelay()
    {
        InputSystem.settings.updateMode = InputSettings.UpdateMode.ProcessEventsInDynamicUpdate;
        _input.SetCursorState(false);
        Time.timeScale = 0.0f;
        _playerInput.actions.Disable();
    }
```

Εικόνα 18: Λογική Κώδικα Για Τις Οδηγίες

Για να φορτώνει η μπάρα ασύγχρονα δημιουργήθηκε η παρακάτω μέθοδος στην οποία μέσω του IEnumerator interface, υπολογίζει πόσο έχει φορτώσει η σκηνή και αλλάζει το γέμισμα της μπάρας αναλόγως. Πέρα από αυτό, ενημερώνει και το κείμενο κάτω από την μπάρα στο οποίο αναγράφεται σε ποσοστό η πρόοδος του φορτώματος της σκηνής.

```
//συναρτηση που φορτωνει το loading bar αναλογα ποσο % εχει φορτωσει η σκηνη του παιχνιδιου
    IEnumerator LoadLevelAsync(string levelToLoad)
    {
        progressSlider.value = 0;
        AsyncOperation load = SceneManager.LoadSceneAsync(levelToLoad);
        load.allowSceneActivation = false;
        float progress = 0;
        while (!load.isDone)
        {
            progress = Mathf.MoveTowards(progress, load.progress, Time.deltaTime);
            progressSlider.value = progress;
            progressText.text = $"Φορτώνει.. ({Mathf.RoundToInt(progress * 100)}%)";
            if (progress >= 0.9f)
            {
                progressSlider.value = 1;
                progressText.text = "Φορτώνει.. (100%)";
                load.allowSceneActivation = true;
            }
            yield return null;
        }
    }
```

Εικόνα 19: Λογική Κώδικα Για Την Μπάρα Φόρτωσης

4.1.4 Gameplay UI

Η διεπαφή του χρήστη που αντικρίζει ο παίκτης μόλις μπαίνει στο παιχνίδι χωρίζεται σε δύο κομμάτια. Το πρώτο βρίσκεται στις τρεις γωνίες της οθόνης και το δεύτερο στο κέντρο της οθόνης. Η σχεδίαση του UI αναλύεται εκτενώς παρακάτω.



Εικόνα 20: Διεπαφή Χρήστη - Αρχή Παιχνιδιού

Βάση της εικόνας 20, υπάρχουν 4 σημεία που μπορεί να ξεχωρίσει ο παίκτης.

1: Η ζωή του χαρακτήρα, υπάρχει μια εικόνα καρδιάς στην οποία έχει υλοποιηθεί ένα μικρό animation στο οποίο μεγαλώνει και μικραίνει με γρήγορο ή αργό ρυθμό ανάλογα πόσο χαμηλή είναι η ζωή του παίκτη. Δίπλα υπάρχει κείμενο που ενημερώνεται δυναμικά σε περίπτωση που ο παίκτης χάσει ζωή.

2: Radar το οποίο λειτουργεί σαν χάρτης για τον παίκτη. Στο κέντρο βρίσκεται ο παίκτης και γύρω γύρω υπάρχουν διαφορετικά χρώματα τα οποία είναι ανάλογα το αντικείμενο που χαρακτηρίζουν. Στην παραπάνω εικόνα φαίνεται μόνο ο παίκτης καθώς το παιχνίδι μόλις ξεκίνησε και δεν έχει δημιουργηθεί ακόμα τίποτα.

3: Εικόνα που περιγράφει το όπλο που είναι ενεργό την δεδομένη στιγμή στον χαρακτήρα. Για τα όπλα (πέρα από τις μπουνιές που θεωρείται επίσης όπλο του χαρακτήρα) δημιουργήθηκε εικονίδιο από το συγκεκριμένο Gameobject του κάθε όπλου για μεγαλύτερη λεπτομέρεια στο παιχνίδι.

4: Αρχικό κείμενο που δίνει τελικές οδηγίες στον παίκτη. Μένει ενεργό για λίγο χρονικό διάστημα και μετά εξαφανίζεται και ξεκινάνε να βγαίνουν οι εχθροί.

4.1.5 Εμφάνιση κειμένου ζημιάς (Damage Pop-Up Text)

Όπως στα περισσότερα παιχνίδια, έτσι και εδώ έπρεπε να υλοποιηθεί ένα σύστημα όπου θα φαίνεται η ζημιά που προκαλείς στους εχθρούς. Έτσι, υλοποίησα ένα σύστημα στο οποίο κάθε φορά που προκαλείς ζημιά σε εχθρό εμφανίζεται στον 3D χώρο, δίπλα από τον εχθρό, η ζημιά που του προκάλεσες. Αφού μείνει για ένα πολύ μικρό διάστημα και συρρικνωθεί αρκετά με το animation που του έχει προστεθεί μέσω κώδικα εξαφανίζεται εντελώς και καταστρέφεται για να μην προκαλεί προβλήματα απόδοσης στο παιχνίδι.

Πιο αναλυτικά, κάθε φορά που προκαλείται ζημιά στον εχθρό καλείται η συνάρτηση CreatePopUp και η οποία αφού πάρει την τιμή δυναμικά την ορίζει στο κείμενο και το δημιουργεί. Η συνάρτηση καταστρέφει το κείμενο σε διάστημα μισού δευτερολέπτου από την στιγμή που δημιουργηθεί. Από την στιγμή που δημιουργηθεί το UI του κειμένου, τρέχει σε άλλο script, που είναι ενσωματωμένο, η update συνάρτηση η οποία συρρικνώνει το κείμενο όσο προλάβει πριν καταστραφεί.

```
public void CreatePopUp(Vector3 position, string text, Color color)
{
    var popup = Instantiate(prefab, position, Quaternion.identity, this.transform);
    var temp = popup.transform.GetChild(0).GetComponent<TextMeshProUGUI>();
    temp.text = text;
    temp.faceColor = color;

    //Destroy Timer
    Destroy(popup, 0.5f);
}
```

Εικόνα 19: Λογική Κώδικα Για Την Δημιουργία Του PopUp Κειμένου

```
//αλλαζει smoothly(κανει scale down) το νουμερο που δειχνει την ζημια
private void Update()
{
    tmp.color = new(1, 1, 1, opacityCurve.Evaluate(time));
    transform.localScale = (Vector3.one * scaleCurve.Evaluate(time)) / 2;
    transform.position = origin + new Vector3(0, 1 + heightCurve.Evaluate(time), 0);
    time += Time.deltaTime;
}
```

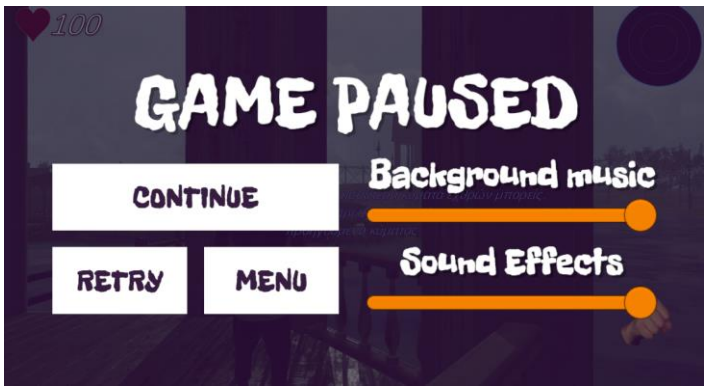
Εικόνα 20: Λογική Κώδικα Για Το Animation Του PopUp Κειμένου



Εικόνα 21: Παράδειγμα Κειμένου Ζημιάς

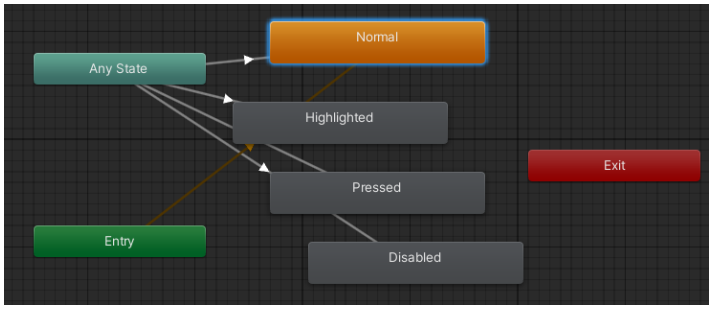
4.1.6 Μενού Παύσης (Pause Menu)

Το Μενού Παύσης αποτελεί ουσιαστικό στοιχείο σε κάθε παιχνίδι, προσφέροντας στους παίκτες την δυνατότητα για παύση από τη δράση, καθώς και πρόσβαση σε διάφορες λειτουργίες. Αν και η σχεδίαση του μενού μπορεί να ποικίλει ανάλογα με το είδος του παιχνιδιού, υπάρχουν κάποια κοινά στοιχεία που συνήθως περιλαμβάνονται και είναι αναγκαία για μια ολοκληρωμένη εμπειρία. Για το παιχνίδι μου δημιούργησα ένα μενού παύσης που σταματάει τη ροή του παιχνιδιού όταν ο παίκτης πατήσει το Escape. Το μενού παύσης περιλαμβάνει τρία κουμπιά τα οποία είναι για συνέχεια του παιχνιδιού, για επανεκκίνηση και για έξοδο στο κεντρικό μενού. Επιπλέον, όπως και στην αρχικό μενού, ο παίκτης μπορεί να αλλάξει την ένταση του ήχου μέσω των δύο slider που υπάρχουν. Από πίσω έχει επιλεχθεί μια μωβ ημι-διάφανη εικόνα για μια καλύτερη αίσθηση. Η λογική του κώδικα είναι σχεδόν ίδια με τα κουμπιά και τα sliders που υπάρχουν στο κεντρικό μενού καθώς χρησιμοποιούνται πάλι events για να γίνουν λειτουργικά. Σαν έξτρα έχει προστεθεί ένα μικρό animation στα κουμπιά τα οποία μεγεθύνονται όταν περνάμε το ποντίκι από πάνω (hover). Τέλος, υπάρχουν τα ίδια sound effects με το κεντρικό μενού στα κουμπιά για όταν γίνονται hover και clicked. Για να συνεχίσει ο παίκτης το παιχνίδι, μπορεί να πατήσει continue ή να ξαναπατήσει το Escape όπως ισχύει και στα περισσότερα παιχνίδια που κυκλοφορούν.



Εικόνα 22: Μενού Παύσης - UI

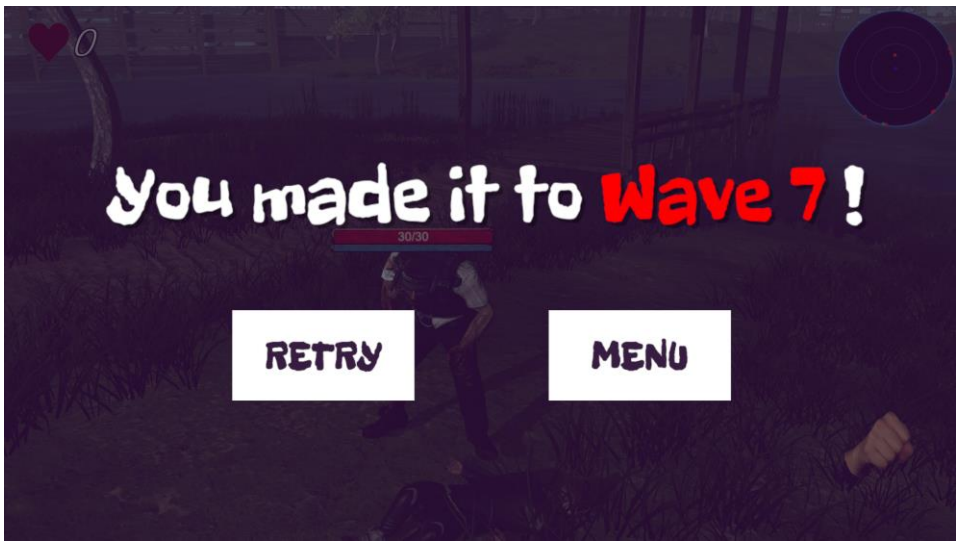
Σε κάθε κουμπί υπάρχει το εξής animator όπου το καθένα έχει ένα πολύ απλό animation στο οποίο αλλάζει το scale του Gameobject.



Εικόνα 23: Button Animator

4.1.7 Μενού Θανάτου (Death Menu)

Για την παρούσα εργασία φτιάχτηκε ένα απλοϊκό μενού θανάτου όπου ο παίκτης έχει την επιλογή ή να ξεκινήσει μια νέα προσπάθεια ή να επιστρέψει στο κεντρικό μενού. Το UI χτίστηκε πάνω στο ήδη υπάρχον UI του μενού παύσης. Διέγραψα τα δύο sliders για τους ήχους και το ένα κουμπί και άφησα τα άλλα δύο κουμπιά τα οποία είχαν ήδη την λειτουργικότητα για επανεκκίνηση του παιχνιδιού ή επιστροφή στο μενού. Η μόνη βασική αλλαγή ήταν στο κείμενο το οποίο ενημερώνεται δυναμικά και αναφέρει μέχρι ποιο κύμα έφτασε ο παίκτης. Έτσι, γρήγορα δημιουργήθηκε το UI ώστε να δοθεί έμφαση σε άλλα κομμάτια της εργασίας, πιο περίπλοκα και ουσιαστικά, καθώς ο σκοπός δεν ήταν η παραγωγή μιας εντυπωσιακής διεπαφής χρήστη.



Εικόνα 24: Μενού Θανάτου – UI

4.1.8 Μπάρα ζωής (Healthbar)

Για να γνωρίζει ο παίκτης πόση ζωή απομένει σε κάθε εχθρό, είναι σημαντικό να υπάρχει μια μπάρα ζωής ή κάτι παρόμοιο που να το επιδεικνύει. Για μια γρήγορη λύση χρησιμοποιήθηκε ένα asset [asset 2] από το Asset Store της Unity. Με τις κατάλληλες τροποποιήσεις και την σύνταξη κώδικα ώστε να αλλάζουν οι τιμές δυναμικά ανάλογα την ζωή που ξεκινάει ο κάθε εχθρός, είχα ένα αρκετά ικανοποιητικό αποτέλεσμα.

```
void Start()
    {
        //αλλαζω τις τιμες στο healthbar των εχθρων
        if (this.tag == "Enemy")
        {
            TryGetComponent<HealthSystemForDummies>(out healthSystem);
            healthSystem.CurrentHealth = (int)health;
            healthSystem.MaximumHealth = (int)health;
        }
    }
```

Εικόνα 25: Λογική Κώδικα για Healthbar (Στιγμιότυπο 1)

```
else if (target.tag == "Enemy" && !isDeadZombie)
{
    CheckForDizz(weaponTag);
    healthSystem.AddToCurrentHealth(-amount);
    sounds.OnDamaged();
    if (isCrit)
    {
        DamagePopUpGenerator.current.CreatePopUp(transform.position + randomness, amount.ToString(), Color.red);
    }
    else
    {
        DamagePopUpGenerator.current.CreatePopUp(transform.position + randomness, amount.ToString(), Color.yellow);
    }
}
```

Εικόνα 26: Λογική Κώδικα για HealthBar (Στιγμιότυπο 2)

Στην εικόνα 25 φαίνεται πως αρχικοποιούνται οι τιμές στους εχθρούς και στην εικόνα 26 τον κώδικα για όταν κάποιος εχθρός δέχεται ζημιά. Μέσα εκεί βλέπουμε ότι καλείται μια μέθοδος που αφαιρεί ζωή και ανανεώνει το UI της μπάρας. Στην εικόνα 21 υπάρχει ένας εχθρός ο οποίος έχει δεχτεί ζημιά. Εκεί βλέπουμε πόσο γεμάτη είναι η μπάρα το οποίο υπολογίζεται από το ποσοστό της τωρινής ζωής σε σχέση με την ανώτατο όριο.

4.1.9 Μπάρα Αντοχής Χτυπημάτων (Impact Resistance Bar)

Για τους εχθρούς αναπτύχθηκε μια περίπλοκη λογική στην οποία μετά από κάποιο αριθμό χτυπημάτων τα ζόμπι ‘ζαλίζονται’ και για λίγο διάστημα μένουν ακίνητα. Σε άλλη ενότητα αναλύεται η υλοποίηση και λειτουργία αυτή [βλ. 4.2.2.2]. Για να υπάρχει και το οπτικό κομμάτι στο οποίο ο παίκτης γνωρίζει πόσα ακόμα χτυπήματα χρειάζεται για να ‘ζαλιστεί’, έφτιαξα μια μικρή μπάρα με μπλε χρώμα, η οποία τοποθετήθηκε κάτω από την μπάρα ζωής για να παίρνει την πληροφορία ο παίκτης όσο πιο εύκολα γίνεται.

Η κύρια λογική για το UI βρίσκεται στην παρακάτω εικόνα. Υπάρχουν πολλά που το επηρεάζουν όπως ο τύπος του όπλου, ο χρόνος που έχει περάσει από το προηγούμενο χτύπημα κ.α. αλλά αναλύονται σε παρακάτω ενότητα.

```
IEnumerator DecreaseFillAmount(float targetFill)
{
    float currentFill = StaminaBarUI.fillAmount;
    float elapsedTime = 0f;

    while (elapsedTime < 0.2f && !isDeadZombie)
    {
        float fillAmount = Mathf.Lerp(currentFill, targetFill, elapsedTime / 0.2f);
        StaminaBarUI.fillAmount = fillAmount;
        elapsedTime += Time.deltaTime;
        yield return null;
    }
    if (!isDeadZombie)
        StaminaBarUI.fillAmount = targetFill;
}
```

Εικόνα 27: Λογική Κώδικα Για Impact Resistance Bar

4.1.10 Μπάρα Αντοχής Παίκτη (Player Stamina Bar)

Η χρήση της Μπάρας Αντοχής είναι βασικό στοιχεία σε πολλά παιχνίδια, ανεξαρτήτως είδους. Αναφέρεται σε ένα γραφικό στοιχείο, το οποίο αντιπροσωπεύει την σωματική αντοχή ή ενέργεια του χαρακτήρα που ελέγχει ο παίκτης. Οι παίκτες πρέπει να διαχειρίζονται προσεκτικά την αντοχή τους, αποφεύγοντας την εξάντληση της μπάρας με δραστηριότητες όπως τρέξιμο, άλματα, ή χρήση ειδικών ικανοτήτων, προκειμένου να διατηρήσουν τον χαρακτήρα τους σε καλή κατάσταση. Δημιούργησα την δική μου εκδοχή του UI με μια μπάρα χρώματος μπλε σκούρο η οποία εμφανίζεται μόνο όταν ο παίκτης καταναλώνει ενέργεια τρέχοντας ή κάνοντας κάποιο άλμα. Σε περίπτωση που τελειώσει η αντοχή, ο παίκτης τρέχει στον μισό σχεδόν ρυθμό από το φυσιολογικό μέχρι να ξαναγεμίσει, κάνοντας το έργο του ακόμα πιο δύσκολο [παράρτημα 3].

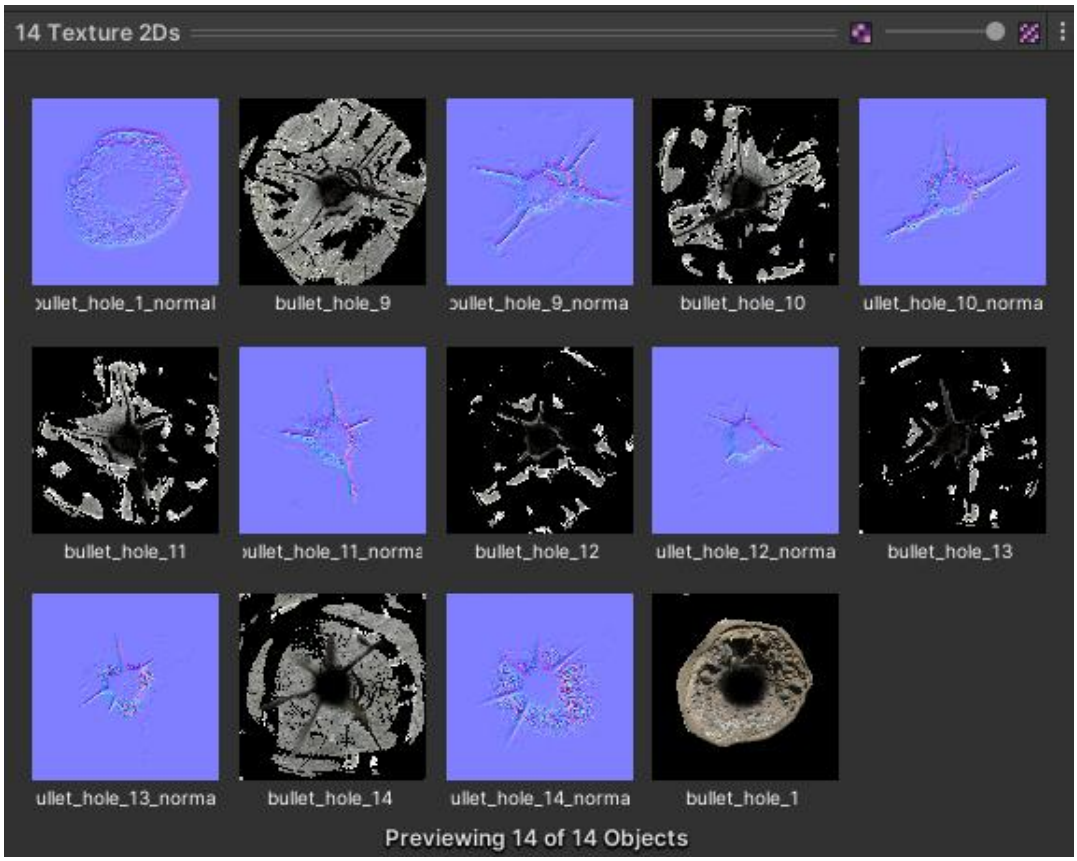


Εικόνα 30: Μπάρα Αντοχής Παίκτη

4.1.11 Σημάδι Σφαίρας και Εμπόδιου για Όπλο (Decals Of Weapon)

Η ένδειξη "X" σε πολλά Third Person Shooter παιχνίδια λειτουργεί ως κρίσιμο οπτικό σήμα, υποδεικνύοντας ότι η σφαίρα θα χτυπήσει σε διαφορετική θέση από αυτήν που σημαδεύει ο παίκτης. Αυτό το χαρακτηριστικό προσθέτει ένα στοιχείο ρεαλισμού στο παιχνίδι και παράλληλα προσφέρει μια καλύτερη εμπειρία στον παίκτη. Το ίδιο ισχύει και για τα σημάδια που αφήνει η σφαίρα του όπλου στους τοίχους ή σε άλλα σημεία. Εφόσον στην εργασία δόθηκε έμφαση στις λεπτομέρειες, τα δύο αυτά σημάδια ήταν μια πολύ σημαντική προσθήκη. Για την υλοποίηση του, έγινε χρήση των decals στην Unity URP όπου παρέχει μια σχετικά ανώδυνη λύση. Στην ουσία είναι ένα εργαλείο που επιτρέπει την δημιουργία γραφικών εφέ πάνω σε αντικείμενα. Μερικά παραδείγματα που χρησιμοποιείται στα βιντεοπαιχνίδια είναι το αίμα, τα σημάδια καταστροφής από εκρήξεις, σημάδια λάσπης κτλ.

Αρχικά για τα σημάδια σφαίρας βρήκα διάφορα σημάδια όπλου από τα οποία έφτιαξα materials για το καθένα ώστε να τα προσθέσω στα prefabs των Decals. Στο κομμάτι του κώδικα, βρίσκω το σημείο που χτύπησε η σφαίρα και αν είναι κάτι στο οποίο μπορεί να αφήσει σημάδι (έχω ορίσει χειροκίνητα τι θεωρείται περιβάλλον) τότε δημιουργεί σε εκείνο το σημείο ένα τυχαίο decal από τα 6 διαφορετικά που έχω. Μετά από 3 δευτερόλεπτα αυτό καταστρέφεται και εξαφανίζεται.



Εικόνα 28: Textures Για Σημάδια Σφαίρας

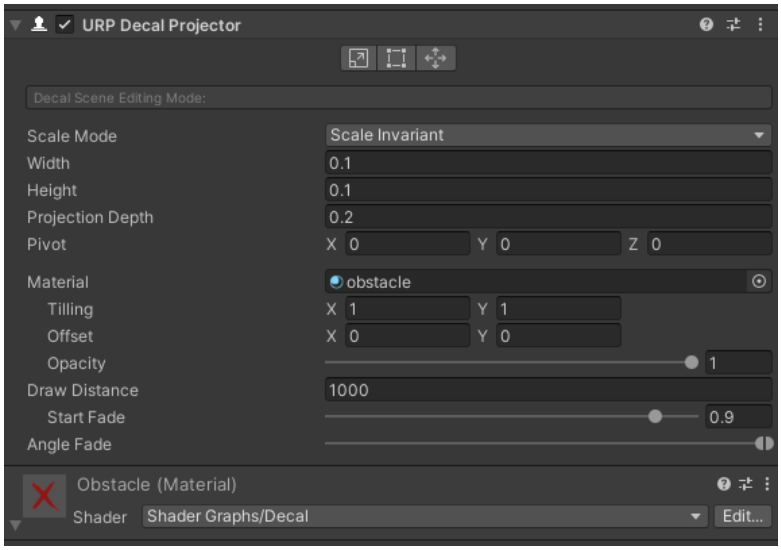
```
// αν εκανε κατι hit
if (canSpawnDecal)
{
    //παιρνω ενα random decal
    int randomDecal = (int)Random.Range(0f, decals.Length);
    //το δημιουργω στην θεση που εκανε hit το raycast
    GameObject decal = Instantiate(decals[randomDecal], hitPoint.point, Quaternion.identity);
    ParticleSystem hitEffectParticle = Instantiate(hitEffect, hitPoint.point, Quaternion.identity);
    //το βαζω με κατευθυνση αντιθετη του normal του raycast
    decal.transform.forward = -hitPoint.normal;
    hitEffectParticle.transform.forward = hitPoint.normal;
    hitEffectParticle.Emit(1);
    //το καταστρεφω 3 δευτερολεπτα μετα
    Destroy(decal, 3f);
    Destroy(hitEffectParticle.gameObject, hitEffectParticle.main.duration);
}
```

Εικόνα 29: Λογική Κώδικα Για Σημάδια Σφαίρας



Εικόνα 30: Παράδειγμα Για Σημάδι Σφαίρας

Όσων αφορά το σημάδι εμποδίου, βρήκα μια εικόνα με ‘X’ που μου ταίριαζε και έκανα transparent την υπόλοιπη εικόνα. Δημιούργησα ξανά ένα material που προστέθηκε στο decal του εμποδίου. Αν ο παίκτης στοχεύει αλλά το crosshair του όπλου σημαδεύει κάτι άλλο από αυτό που θα χτυπήσει κανονικά το όπλο, τότε ενεργοποιώ το Gamobject του decal και εμφανίζεται στο ‘X’ πάνω ακριβώς στο σημείο που θα χτυπήσει. Αυτό συμβαίνει συνήθως μόνο στα Third Person παιχνίδια καθώς η κάμερα πολλές φορές δημιουργεί προβλήματα γιατί βρίσκεται πίσω και πάνω από τον χαρακτήρα.



Εικόνα 31: Decal Projector Του Εμποδίου

```
//επειδή εχουμε TPS controller μπορει το raycast της καμερας να χτυπαει σε
//διαφορετικο σημειο απο οτι του οπλου οποτε βαζουμε ενα decal "X" πανω στο εμποδιο
private void CheckIfObstacle(RaycastHit hit)
{
    if (!hit.collider) return;
    obstacleDecal.transform.position = hit.point;
    obstacleDecal.transform.forward = -hit.normal;
    //μονο αν ειναι πανω σε περιβαλλον, κανουμε aim και το object που χτυπαει η καμερα ειναι διαφορετικο απο του οπλου
    if (starterAssetsInputs.aim && cameraRaycastHit.transform.name != hit.transform.name && hit.transform.CompareTag("Environment"))
    {
        obstacleDecal.SetActive(true);
    }
    else
    {
        obstacleDecal.SetActive(false);
    }
}
```

Εικόνα 32: Λογική Κώδικα Για Το Decal Εμποδίου

Στην εικόνα 32 φαίνεται η λογική του κώδικα. Επειδή έχω διαφορετικά raycast που φεύγουν από το όπλο και από το κέντρο της οθόνης, παίρνοντας απλά τα αντικείμενα και βλέποντας αν είναι τα ίδια ή όχι, μπορούμε να πούμε εύκολα αν υπάρχει εμπόδιο μπροστά από τον παίκτη.

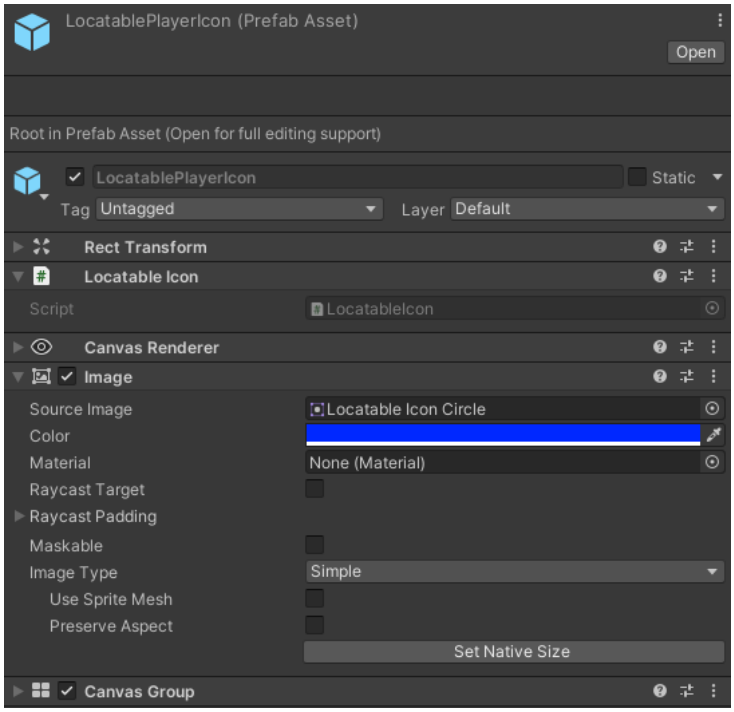


Εικόνα 33: Παράδειγμα Για Σημάδι Εμποδίου

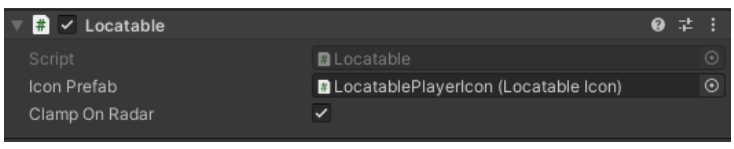
4.1.12 Ραντάρ εχθρών και αντικειμένων

Στα περισσότερα παιχνίδια υπάρχει ένας χάρτης ή κάτι παρόμοιο ώστε να καθοδηγεί τον παίκτη στις κινήσεις του. Καθώς το περιβάλλον που χρησιμοποίησα ήταν αρκετά μεγάλο και ο παίκτης πρέπει κάπως να λαμβάνει την πληροφορία για την τοποθεσία των εχθρών και λοιπών αντικειμένων, πρόσθεσα το ραντάρ που δείξαμε παραπάνω. Εφάρμοσα και τροποποίησα το asset [asset 3] από το Unity Asset Store για να ταιριάζει με τις ανάγκες μου. Όπως φαίνεται και στην εικόνα 33 υπάρχουν κουκίδες με διαφορετικά χρώματα που δείχνουν τις αποστάσεις και τις κατευθύνσεις από τον παίκτη. Με μπλε χρώμα είναι ο παίκτης και είναι σταθερός πάντα στο κέντρο και τα υπόλοιπα κινούνται δυναμικά πάνω στο ραντάρ ανάλογα την απόσταση από τον παίκτη. Έχω ορίσει κόκκινο χρώμα για τους εχθρούς, πράσινο για τα όπλα, πορτοκαλί για την ζωή και μωβ για τις σφαίρες.

Στην παρακάτω εικόνα βλέπουμε το prefab που υπάρχει για την κουκίδα του παίκτη. Χρησιμοποιείται ένα κυκλικό εικονίδιο με χρώμα που όρισα από τον Editor. Αντίστοιχα prefabs έχουν και τα υπόλοιπα Gameobjects που όρισα ως Locatable, δηλαδή να τα αναγνωρίζει και να τα προσθέτει το ραντάρ.



Εικόνα 34: Prefab Locatable Player Icon



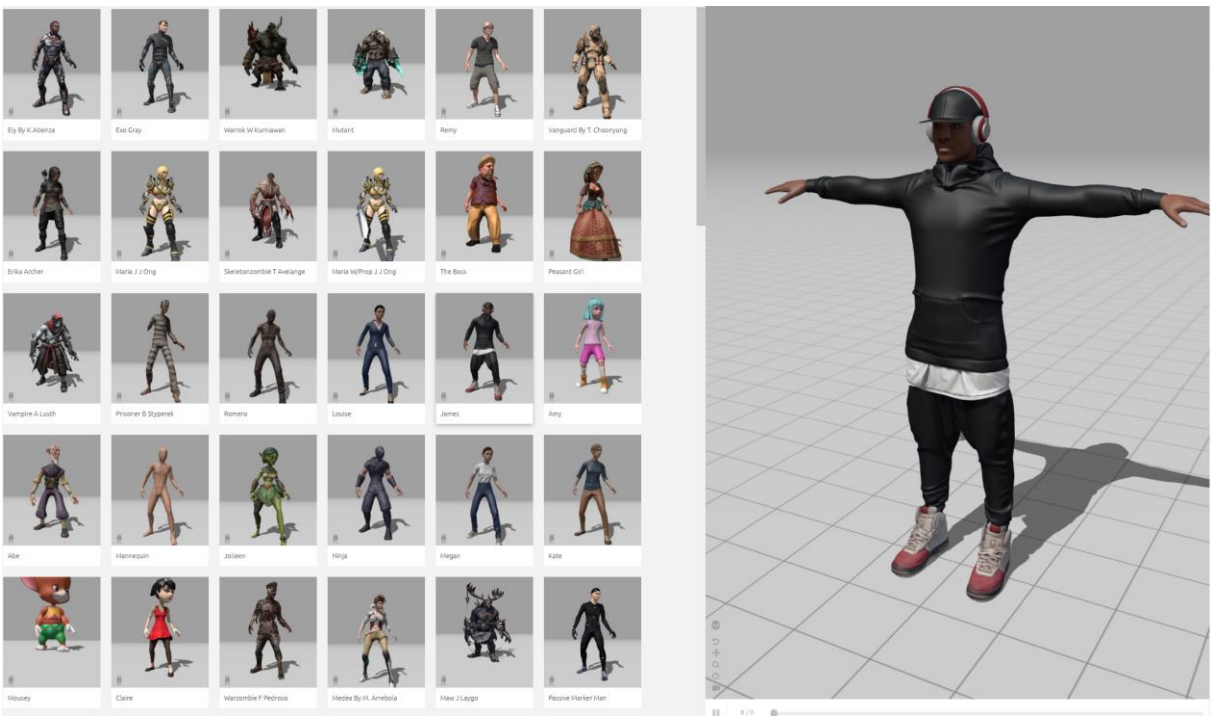
Εικόνα 35: Locatable Player Component

4.2 Χαρακτήρες

4.2.1 Χαρακτήρας Παίκτη και Συστατικά

4.2.1.1 Μοντέλο

Για το μοντέλο του χαρακτήρα που χειρίζεται ο παίκτης έβαλα ένα μοντέλο από το Mixamo. Η πλατφόρμα προσφέρει γρήγορες λύσεις για μοντέλα χαρακτήρων καθώς και animation αυτών σε πολύ καλή ποιότητα. Καθώς χρησιμοποίησα το Third Person Character Controller της Unity για να επιτύχω γρήγορη κίνηση στον χαρακτήρα, αλλάζοντας το προκαθορισμένο μοντέλο που είχε με το FBX αρχείο που κατέβασα είχα χωρίς καθυστερήσεις έναν πλήρες χαρακτήρα με animation και λειτουργίες όπως το τρέξιμο και το άλμα. Επίσης, με το συγκεκριμένο asset, υπάρχει ήδη κάμερα που χρησιμοποιεί το πακέτο Cinemachine της Unity η οποία έχει ήδη τις κατάλληλες ρυθμίσεις ώστε να δείχνει τον χαρακτήρα σε τρίτο πρόσωπο. Ακόμα, έχει τα βασικά animations τα οποία θα αναλυθούν περαιτέρω στην επόμενη ενότητα.

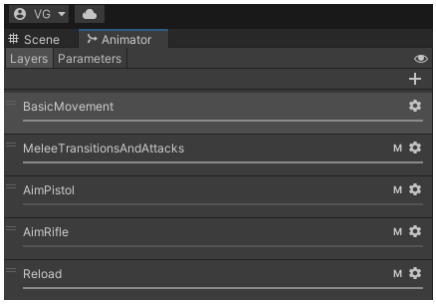


Εικόνα 36: Χαρακτήρας Παίκτη Από Mixamo

4.2.1.2 Ελεγκτής κινήσεων χαρακτήρα (Animator controller)

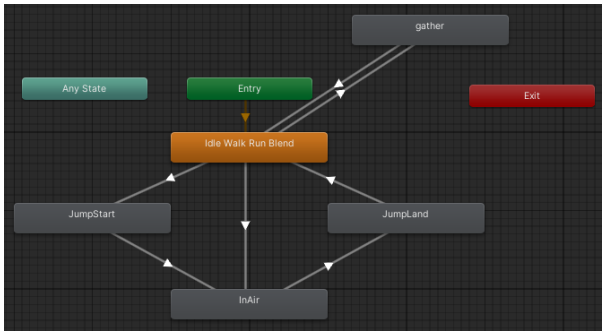
Ο Animator Controller αποτελεί ένα ισχυρό εργαλείο στον χώρο της ανάπτυξης παιχνιδιών, επιτρέποντας στους προγραμματιστές να διαχειρίζονται τη συμπεριφορά χαρακτήρων και αντικειμένων. Αυτό το εργαλείο επιτρέπει τόσο τη δημιουργία απλών όσο και πολύπλοκων κινήσεων και αντιδράσεων.

Ο προγραμματιστής μπορεί να καθορίσει τις συνθήκες και τις μεταβλητές που επηρεάζουν τον χαρακτήρα, συνήθως μέσω κώδικα, ώστε να γίνει εναλλαγή από ένα animation σε ένα άλλο. Για παράδειγμα, αν ο παίκτης είναι σταματημένος και θέλουμε να κάνει άλμα, θα κάνει trigger την μεταβλητή jump ώστε ο controller να κάνει εναλλαγή στο jump animation. Πέρα από απλά πράγματα, ο controller επιτρέπει να γίνεται override το ένα animation πάνω στο άλλο, αφήνει να ορίσεις μια ‘μάσκα’ που καθορίζει ποια σημεία του σκελετού επηρεάζονται από το animation και τέλος επιτρέπει χρήση πολλών επιπέδων (Layers) ώστε καθένα να λειτουργεί αυτόνομα. Στην εργασία έγινε χρήση πολλών Layer ώστε κάποια animation όπως το aim για το κάθε όπλο να επηρεάζει μόνο το πάνω σημείο του σώματος δίνοντας αίσθηση ρεαλισμού στο παιχνίδι. Συγκεκριμένα υπάρχουν πέντε διαφορετικά Layers για τον χαρακτήρα που φαίνονται και περιγράφονται παρακάτω.



Εικόνα 37: Επίπεδα Animator Παίκτη

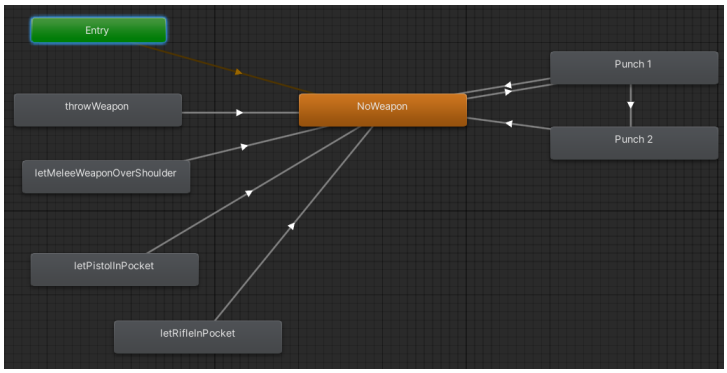
Το πρώτο επίπεδο είναι το BasicMovement το οποίο είναι αυτό που προϋπήρχε με το Asset που χρησιμοποίησα για τον χαρακτήρα. Περιλαμβάνει ένα BlendTree που κάνει ομαλές αλλαγές μεταξύ των animation idle, walk και run. Η Unity το χειρίζεται μόνη της με βάση έναν δεκαδικό αριθμό που δίνω, ο οποίος αφορά την ταχύτητα του παίκτη. Επίσης, υπάρχουν transitions για όταν ξεκινάει το άλμα, όταν βρίσκεται στον αέρα και όταν προσγειώνεται πάλι στο έδαφος. Σε αυτό το επίπεδο, πρόσθεσα ένα ακόμα animation που αφορά την συλλογή αντικειμένων [βλ. 4.2.1.8].



Εικόνα 38: Πρώτο Επίπεδο Animator Παίκτη

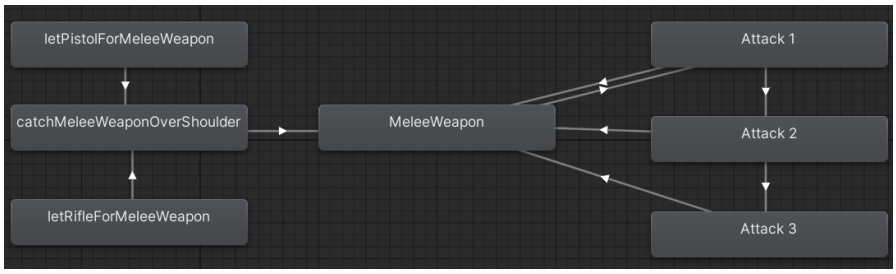
Το δεύτερο επίπεδο είναι το WeaponTransitionsAndMeleeAttacks. Είναι το πιο περίπλοκο επίπεδο που δημιούργησα καθώς περιλαμβάνει πολλά animations μέσα, τόσο για τις επιθέσεις του χαρακτήρα από κοντινή απόσταση (μπουνιές και τσεκούρι), όσο και για την αλλαγή όπλων. Για κάθε περίπτωση όπλου υπάρχει διαφορετική σειρά που ακολουθείται αφού ο χαρακτήρας πρέπει να αφήσει πρώτα το προηγούμενο όπλο και μετά να πιάσει το άλλο, αλλιώς αν δεν κρατάει όπλο να πιάσει κατευθείαν το επόμενο. Η λογική αυτή υπάρχει στον κώδικα ο οποίος ρυθμίζει ποια είναι η σωστή παράμετρος κάθε φορά. Επιπρόσθετα, χρησιμοποιήθηκαν StateMachineBehaviours για διορθώσεις σφαλμάτων και απενεργοποίηση ανεπιθύμητων ενεργειών κατά την διάρκεια ενός animation. Τα StateMachineBehaviours είναι scripts τα οποία επιτρέπουν την υλοποίηση κάποιας λογικής κατά την έναρξη, την διάρκεια και το τέλος ενός animation. Δημιούργησα δύο τα οποία ελέγχουν για το πότε μπορεί ή όχι ο χρήστης να αλλάξει και να πετάξει όπλο. Ένα παράδειγμα για να γίνει πιο κατανοητό είναι όταν ο παίκτης αλλάζει όπλο. Μέχρι να τελειώσει η συγκεκριμένη κίνηση, ο χρήστης δεν πρέπει να έχει την δυνατότητα να αλλάξει ξανά όπλο ή να το πετάξει καθώς οδηγεί σε ανεπιθύμητες ενέργειες.

Μπορούμε να χωρίσουμε το συγκεκριμένο επίπεδο σε τέσσερα κομμάτια. Το πρώτο που είναι και το default, είναι υπεύθυνο για τις μπουνιές του χαρακτήρα. Όταν τρέχει για πρώτη φορά, ο παίκτης μπαίνει στο NoWeapon State και μένει εκεί μέχρι να αλλάξει κάποια παράμετρος. Όταν πεταχτεί κάποιο όπλο μπαίνει πάντα σε αυτό το state. Επίσης αν γίνει trigger το attack τότε τρέχει το punch animation. Σε περίπτωση που ο χρήστης έχει καλό συγχρονισμό έχει την δυνατότητα να τρέξει και το δεύτερο punch animation το οποίο είναι από το άλλο χέρι.



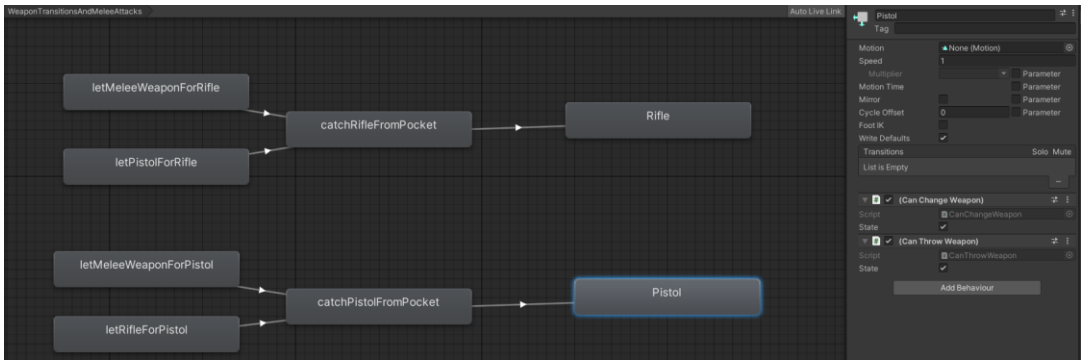
Εικόνα 39: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Punch

Το δεύτερο κομμάτι χειρίζεται το τσεκούρι. Εδώ έχουμε μια αλυσίδα επιθέσεων παρόμοια με τις μπουνιές αλλά με τρεις διαφορετικές επιθέσεις. Η λογική εδώ είναι ότι όταν κρατάμε ένα από τα όπλα π.χ. το πιστολάκι, πρέπει πρώτα να το αφήσει στην τσέπη οπότε θα τρέξει πρώτα το letPistolForMeleeWeapon animation και μετά το catchMeleeWeaponOverShoulder ώστε να πιάσει το τσεκούρι και να έρθει σε MeleeWeapon State. Αλλιώς αν δεν κρατάει κάποιο όπλο, από τον κώδικα που ελέγχεται η λογική θα τρέξει κατευθείαν το δεύτερο κομμάτι.

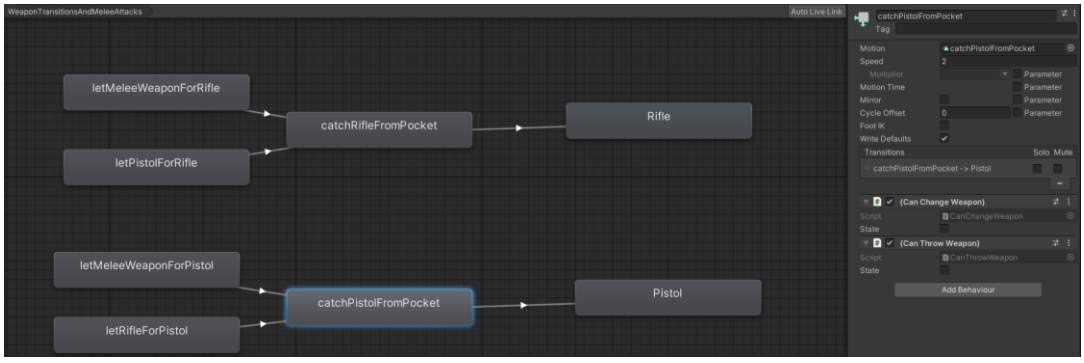


Εικόνα 40: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Axe

Η λογική για το τρίτο και το τέταρτο κομμάτι είναι η ίδια απλά με διαφορετικά animation καθώς έχουμε διαφορετικά όπλα και σημεία που τα αφήνει ο παίκτης. Στην εικόνα 41 βλέπουμε και τα State Machine Behaviors του Pistol τα οποία τα έχω ορίσει στον κώδικα ως ένα απλό Boolean το οποίο είναι true. Σε όλα τα άλλα states το έχω ορίσει ως false ώστε να μην μπορεί να αλλάξει ή να πετάξει όπλο εκείνη την στιγμή παρά μόνο όταν έχει φτάσει στο τελικό state και είναι έτοιμο για την επόμενη ενέργεια.



Εικόνα 41: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Pistol Και Rifle (Στιγμιότυπο 1)

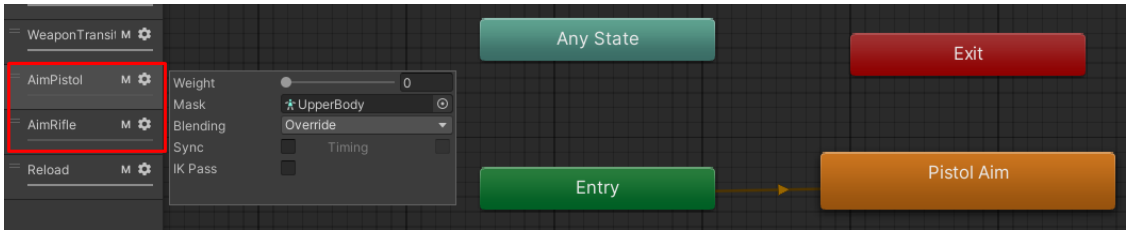


Εικόνα 42: Δεύτερο Επίπεδο Animator Παίκτη – Κομμάτι Pistol και Rifle (Στιγμιότυπο 2)

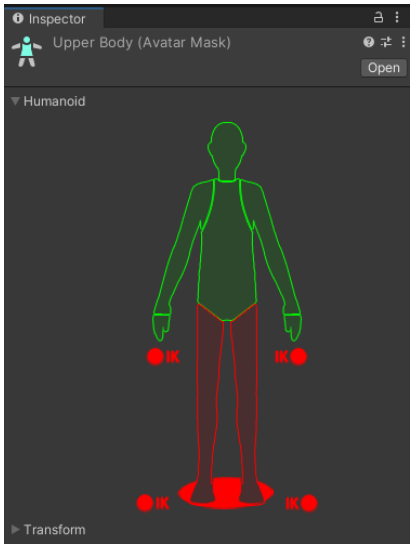
```
public class CanChangeWeapon : StateMachineBehaviour
{
    //χρησιμοποιειται στο animator ωστε να καθοριζει σε καθε state αν μπορω να αλλαξω το οπλο ή οχι
    [SerializeField] bool state;
    //τρέχει μια φορά όταν μπαίνει στο animation που έχω προσθέσει το script
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        //στο weaponManager script που είναι η λογική για τα όπλα
        // ενημερώνει αν μπορεί να αλλάξει όπλο την δεδομένη στιγμή
        WeaponManager.current.CanChangeWeapon(state);
    }
}
```

Εικόνα 43: Λογική Κώδικα State MachineBehaviour – CanChangeWeapon

Το τρίτο και το τέταρτο επίπεδο είναι για την στόχευση των όπλων (aim). Έχουν την ίδια λογική αλλά εδώ έχουμε διαφορετικό χειρισμό από τα προηγούμενα. Σε αυτά τα επίπεδα, αντί να αλλάζουμε παραμέτρους, αλλάζουμε το βάρος του επιπέδου. Το animator μας δίνει την δυνατότητα να καθορίζουμε το βάρος το οποίο αλλάζει το πόσο επηρεάζει τα animations που βρίσκονται μέσα στο συγκεκριμένο επίπεδο. Όποτε για αρχή έχουμε ορίσει την τιμή του βάρους σε μηδέν, και κάθε φορά που σημαδεύουμε με κάποιο όπλο, το αλλάζουμε ομαλά σε τιμή ένα. Έτσι πετυχαίνουμε ένα ρεαλιστικό αποτέλεσμα καθώς ο παίκτης πηγαίνει ομαλά σε θέση στόχευσης. Όταν σταματάμε να στοχεύουμε, η τιμή ορίζεται πάλι σε μηδέν. Στα δύο επίπεδα έχουμε ορίσει μάσκα (εικόνα 45) που επηρεάζει μόνο το πάνω μέρος του σώματος.

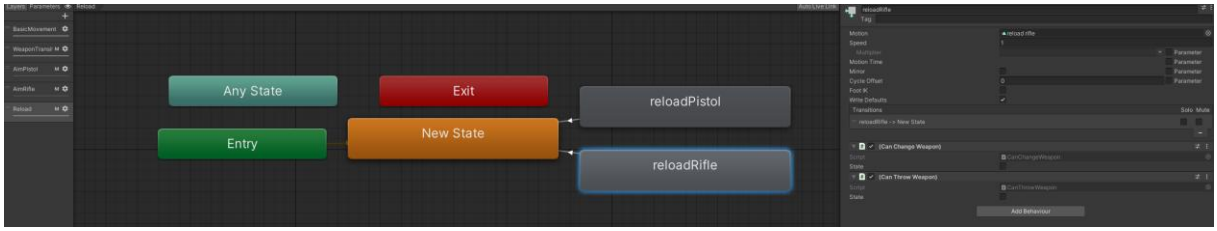


Εικόνα 44: Τρίτο Και Τέταρτο Επίπεδο Animator Παίκτη



Εικόνα 45: Μάσκα Avatar Για Το Πάνω Μέρος

Τέλος, το πέμπτο επίπεδο είναι υπεύθυνο για το γέμισμα (reload) των όπλων. Υπάρχουν δύο ξεχωριστά animations για το πιστολάκι και το πολυβόλο. Όταν ο παίκτης πατήσει να γεμίσει το όπλο, τρέχει το animation και μετά επιστρέφει στο αρχικό state. Κατά την διάρκεια του reload τρέχουν τα ίδια StateMachineBehaviours που είχαμε στο δεύτερο επίπεδο. Για μάσκα χρησιμοποιούμε την ίδια το πάνω μέρος του σώματος.

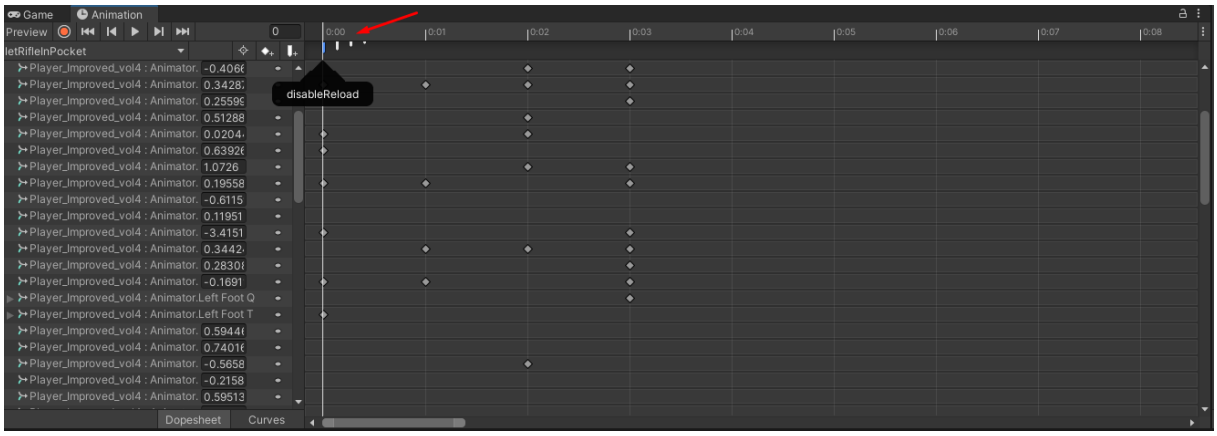


Εικόνα 46: Πέμπτο Επίπεδο Animator Παίκτη

4.2.1.3 Animations και Animation Events

Για τον χαρακτήρα χρησιμοποιήθηκαν περισσότερα από 20 διαφορετικά animations. Σε αυτά περιλαμβάνονται επιθέσεις, τρέξιμο, συλλογή και εκτόξευση όπλου, άλμα, στόχευση όπλου κ.α. Για τα περισσότερα animation χρησιμοποιήθηκε το Mixamo, καθώς όμως κάποια δεν ταίριαζαν στις ανάγκες μου έπρεπε να τα τροποποιήσω. Σε αυτό βοήθησε αρκετά το animation rigging το οποίο είναι ένα πακέτο της Unity που εκτός τις τεράστιες δυνατότητες που προσφέρει προσαρμόζοντας τις κινήσεις του χαρακτήρα με βάση διάφορες συνθήκες, έχει την ευκολία να προσθέτει έναν ‘εικονικό’ σκελετό (bone renderer) μέσα στο Unity Editor και να δημιουργήσεις κινήσεις από τα κόκκαλα του σκελετού. Έτσι, κατάφερα κάνοντας record και κουνώντας τα κόκκαλα να τροποποιήσω τα animation για πιο αληθοφανές κινήσεις. Σε άλλο κεφάλαιο [βλ. 4.2.1.11] θα αναλυθεί και πως αξιοποίησα το πακέτο της Unity για να βάλω περιορισμούς σε διάφορα σημεία του σώματος.

Τα Animation Events είναι σημεία στο χρονικό πλαίσιο ενός animation όπου μπορείτε να συνδέεται κώδικας ή άλλες ενέργειες. Αυτά τα events επιτρέπουν στον προγραμματιστή να ορίσει πότε κάποιος κώδικας θα εκτελεστεί κατά τη διάρκεια ενός animation. Έτσι, αν θέλουμε να τρέξει μια συνάρτηση σε ένα συγκεκριμένο καρέ (frame), τα events αποτελούν μια πολύ καλή λύση. Για την παρούσα εργασία χρησιμοποιήθηκαν σχεδόν σε κάθε animation τόσο του παίκτη όσο και των εχθρών. Κάποια υπάρχουν για να λύσουν κάποιο bug που υπήρχε καθώς ο χαρακτήρας έχει αρκετές ενέργειας και πρέπει να σκεφτούμε κάθε πιθανή εκδοχή για ένα καλό αποτέλεσμα. Ένα παράδειγμα είναι το disableReload που υπάρχει στο πρώτο frame του animation ‘letRifleInPocket’. Αν ο παίκτης πάταγε να γεμίσει το όπλο όσο άφηνε το όπλο στην τσέπη του τότε υπήρχε μια πολύ περίεργη κίνηση στον χαρακτήρα κάτι το οποίο δεν έπρεπε να συμβαίνει. Αυτό λύθηκε με το συγκεκριμένο event το οποίο απενεργοποιεί την δυνατότητα να γίνει reload όταν ξεκινάει να αφήσει το όπλο. Υπήρχαν πολλά και διάφορα τέτοια προβλήματα στα οποία χρησιμοποίησα events για να τα λύσω. Στην παρακάτω εικόνα φαίνονται με άσπρες γραμμές τα events τα οποία έχουν προστεθεί για το συγκεκριμένο μόνο animation.



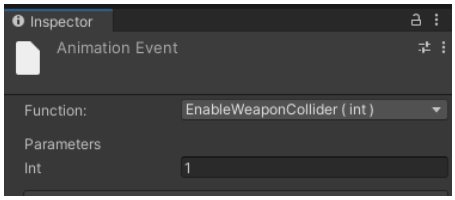
Εικόνα 47: Animation Events

Τρία events που θεωρώ ότι είναι αρκετά πιο σημαντικά και πρέπει να αναλυθούν παραπάνω είναι το EnableWeaponCollider, το SetHandIkTarget και το JobForItemGrabbed.

EnableWeaponCollider: Είναι υπεύθυνο για να ενεργοποιεί-απενεργοποιεί τα collider των όπλων στο κατάλληλο frame του animation ώστε να αναγνωριστεί η επαφή και να κάνουν ζημιά στους εχθρούς. Είναι σημαντικό να γίνει αυτή η ενέργεια αλλιώς αν είναι μόνιμα ανοιχτά τα colliders θα γίνεται πάντα ζημιά στην επαφή ακόμα και αν απλά τρέχω δίπλα από τους εχθρούς και ακουμπήσουν καταλάθος. Η λογική του κώδικα είναι αρκετά απλή αλλά έλυσε άλλο ένα πρόβλημα που υπήρχε στο πρωτότυπο του παιχνιδιού. Χρησιμοποιώ int αριθμό καθώς τα events δεν δέχονται Booleans. Το 1 είναι για ενεργοποίηση και το 0 για απενεργοποίηση.

```
//ενεργοποιει-απενεργοποιει τα colider στα οπλα
public void EnableWeaponCollider(int isEnable)
{
    if (meleeWeapon != null)
    {
        var col = meleeWeapon.GetComponent<Collider>();
        if (col != null)
        {
            if (isEnable == 1)
            {
                col.enabled = true;
            }
            else
            {
                col.enabled = false;
            }
        }
    }
}
```

Εικόνα 48: Λογική Κώδικα EnableWeaponCollider



Εικόνα 49: Animation Event EnableWeaponCollider

SetHandIkTarget: Το συγκεκριμένο event υπάρχει στο grab animation το οποίο εκτελείται στο πρώτο frame. Επειδή στο συγκεκριμένο action του χαρακτήρα χρησιμοποιώ animation rigging, έπρεπε να ορίσω το σημείο στο οποίο θα πάει το χέρι (καθώς κάνει override το animation). Στην ουσία βρίσκω το κεντρικό σημείο του αντικειμένου που πάω να πιάσω και ορίζω το χέρι να πάει σε αυτό το σημείο και να μην ακολουθήσει το animation. Με αυτόν τον τρόπο έχουμε πλέον δυναμικά animation κάτι που το κάνει αρκετά πιο περίπλοκο αλλά μας δίνει τρομερά αποτελέσματα ρεαλισμού. Το συγκεκριμένο είναι το πιο απλό που μπορούμε να πετύχουμε με το πακέτο αυτό καθώς απλά ορίζουμε ένα νέο transform στο IKTarget του χεριού.

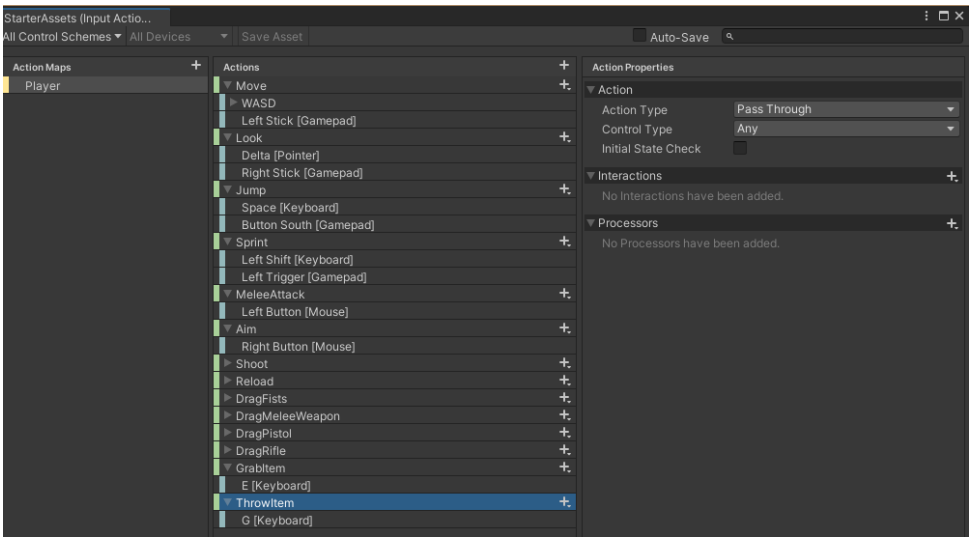
```
private void SetHandIkTarget()
{
    //τοποθετω το target που θα παει το χερι πανω στο αντικειμενο
    Vector3 center = moveToHit.collider.bounds.center;
    Vector3 topCenterPoint = center + (transform.up * (moveToHit.collider.bounds.size.y * 0.5f));
    handIKTarget.position = topCenterPoint;
}
```

Εικόνα 50: Λογική Κώδικα SetHandIkTarget

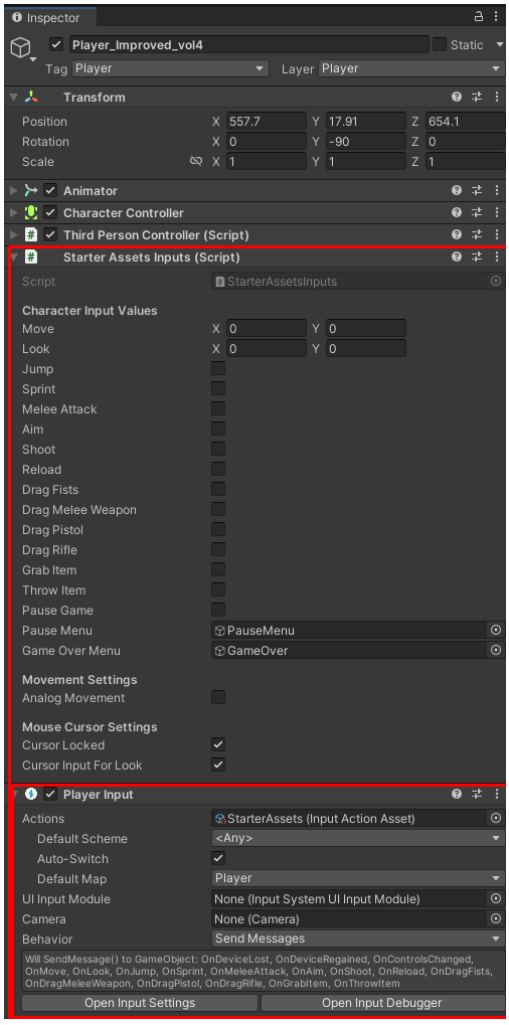
JobForItemGrabbed: Τρέχει στο frame που έχει πλέον σκύψει ο χαρακτήρας και πιάνει το αντικείμενο. Εκεί, επειδή το αντικείμενο που πιάνουμε εξαφανίζεται, είναι σημαντικό να το τρέξουμε στο συγκεκριμένο frame και όχι πριν σκύψει για να έχουμε ένα όμορφο αποτέλεσμα. Βλέπουμε λοιπόν ότι η χρήση των animation events είναι πολύ χρήσιμη και πολλές φορές είναι η μόνη λύση. [παράρτημα 4]

4.2.1.4 Σύστημα Εισόδου (Input System)

Με το Character Controller Asset ήταν ήδη εγκατεστημένο το νέο σύστημα εισόδου που έχει δημιουργήσει η Unity. Το Input System σχεδιάστηκε για να παρέχει μια πιο ευέλικτη και επεκτάσιμη λύση για τη διαχείριση της εισόδου του παίκτη σε παιχνίδια. Παρέχει καλύτερη υποστήριξη για διάφορες συσκευές εισόδου, συμπεριλαμβανομένων πληκτρολογίων, ποντικιών, χειριστηρίων κονσόλων, και χειριστήρια VR. Αυτό διευκολύνει την ανάπτυξη παιχνιδιών που μπορούν να παιχτούν σε διάφορες πλατφόρμες. Επιπλέον, εισάγει τον όρο ‘δράσεων (Actions)’. Στο Asset υπήρχαν ήδη οι βασικές δράσεις για κίνηση, άλμα, τρέξιμο και κίνηση κάμερας με το ποντίκι οπότε συνέχισα από εκεί. Προστέθηκαν δράσεις για attacks, aim, reload, grab, throw και drag(Weapon) στο οποίο βγάζει το συγκεκριμένο όπλο που διάλεξα. Σε κάθε action δίνεται ο τύπος που θα είναι και το πλήκτρο (καθώς το παιχνίδι είναι για υπολογιστή) που του αναλογεί. Μετά γίνεται η σύνδεση του με τον κώδικα και κάθε φορά που ενεργοποιείται ένα action, τρέχουμε τον κώδικα που του αναλογεί. [παράρτημα 5 και 6]



Εικόνα 51: Input System Scheme



Εικόνα 52: Components Χαρακτήρα Για Τα Inputs

4.2.1.5 Διαχείριση όπλων (Weapon Manager)

Το συγκεκριμένο script είναι υπεύθυνο για την διαχείριση των όπλων τόσο στο εικονικό κομμάτι (δηλαδή ποιο Gameobject είναι ορατό κάθε φορά) το οποίο συνδέεται με τα animation events που προαναφερθήκανε όσο και στο λειτουργικό κομμάτι στο οποίο αλλάζει το state του όπλου που έχει ο παίκτης. Επίσης τσεκάρει για εισόδους που δίνονται από τον παίκτη και τρέχει τα αντίστοιχα animation κάθε φορά. Τέλος, αλλάζει το UI στο οποίο φαίνεται το τωρινό όπλο που χρησιμοποιεί.

```
//αν θελω να βαλω μπουνιες και μπορω να αλλαξω οπλο τοτε παιζω το αναλογο animation
//αναλογως και στα αλλα 3
if (SAinputs.fistsInputOnly() && !CheckWeaponState("Fists") && canChangeWeapon)
{
    if (hasMeleeWeapon)
    {
        anim.Play("letMeleeWeaponOverShoulder");
    }
    if (hasPistol)
    {
        anim.Play("letPistolInPocket");
    }
    if (hasRifle)
    {
        anim.Play("letRifleInPocket");
    }
    ChangeTo("Fists");
    SetImageTo(0);
}
```

Εικόνα 53: Λογική Κώδικα WeaponManager (Στιγμιότυπο 1)

```
//αναλογα τι οπλο εβαλα ανανεωνω το UI
public void SetImageTo(int weapon)
{
    foreach (GameObject weaponImage in imageWeapons)
    {
        weaponImage.SetActive(false);
    }
    foreach (GameObject bulletText in bulletsText)
    {
        bulletText.SetActive(false);
    }

    switch (weapon)
    {
        case 0:
            imageWeapons[0].SetActive(true);
            break;
        case 1:
            imageWeapons[1].SetActive(true);
            break;
        case 2:
            imageWeapons[2].SetActive(true);
            bulletsText[0].SetActive(true);
            break;
        case 3:
            imageWeapons[3].SetActive(true);
            bulletsText[1].SetActive(true);
            break;
    }
}
```

Εικόνα 54: Λογική Κώδικα WeaponManager (Στιγμιότυπο 2)



```
//Αλλαζει το state του οπλου που εχω για να ξερω τι κραταει
public void ChangeTo(string weapon)
{
    hasFists = false;
    hasMeleeWeapon = false;
    hasPistol = false;
    hasRifle = false;
    switch (weapon)
    {
        case "Fists":
            hasFists = true;
            break;
        case "Melee":
            hasMeleeWeapon = true;
            break;
        case "Pistol":
            hasPistol = true;
            break;
        case "Rifle":
            hasRifle = true;
            break;
    }
}
//Γυρναει true-false αναλογα αν κραταει η οχι το οπλου που περασα στο ορισμα
public bool CheckWeaponState(string weapon)
{
    return weapon switch
    {
        "Fists" => hasFists,
        "Melee" => hasMeleeWeapon,
        "Pistol" => hasPistol,
        "Rifle" => hasRifle,
        _ => false,
    };
}
```

Εικόνα 55: Λογική Κώδικα WeaponManager (Στιγμιότυπο 3)

4.2.1.6 Διαχείριση Ιδιοτήτων (Attributes Manager)

Εδώ βρίσκεται η περισσότερη λογική τόσο για τον χαρακτήρα όσο και για τους εχθρούς καθώς υπάρχουν στοιχεία όπως για την ζωή, την ζημιά και την ακινητοποίηση των ζόμπι. Τα κομμάτια που αφορούν τον παίκτη αναλύονται παρακάτω.

Αρχικά, έχει φτιαχτεί η συνάρτηση DealDamage που ελέγχει για το πόση ζημιά προκάλεσε, με ποιο όπλο και αν ήταν καίριο χτύπημα (καθώς έχει κάποιες πιθανότητες). Οι πληροφορίες λαμβάνονται μέσω ορισμάτων που δέχεται η συνάρτηση. Το τελικό ποσό ζημιάς υπολογίζεται από το σημείο του σώματος που χτύπησε, από το πολλαπλασιαστή καίριου χτυπήματος και φυσικά από το όπλο που έκανε την ζημιά. [παράρτημα 7]

Μέσα στην συνάρτηση DealDamage υπάρχει και η TakeDamage [παράρτημα 8] η οποία αφού υπολογιστεί το τελικό νούμερο ζημιάς στην ουσία αλλάζει την τιμή της ζωής στο συγκεκριμένο Gameobject που καλέστηκε. Πέρα από την ζημιά περνάμε και άλλα ορίσματα όπως το σημείο επαφής για την ενεργοποίηση του ragdoll [βλ. 4.2.2.4] και την Boolean μεταβλητή για το καίριο χτύπημα η οποία καθορίζει το χρώμα που θα έχει το PopUp κείμενο για την ζημιά το οποίο αναφέραμε σε προηγούμενη ενότητα.

Τέλος, έχουμε την συνάρτηση StopGameAfterDie η οποία είναι τύπου IEnumerator. Εκεί εμφανίζουμε το UI για το τέλος του παιχνιδιού, αποθηκεύουμε μέσω PlayerPrefs [βλ. 5.7] το καινούργιο ρεκόρ (αν έγινε) και απενεργοποιεί τις εισόδους για το χαρακτήρα. Μετά από λίγο σταματάει και την ροή του χρόνου μέσα στο παιχνίδι.

```
//τρεχει αν πεθανει ο παιχτης
IEnumerator StopGameAfterDie()
{
    _playerInput.actions.Disable();
    _input.DisableActiveInputs();
    GetComponent<ThirdPersonController>().enabled = false;
    yield return new WaitForSeconds(4f);
    InputSystem.settings.updateMode = InputSettings.UpdateMode.ProcessEventsInDynamicUpdate;
    _input.SetCursorState(false);
    gameOver.SetActive(true);
    int waveNumber = SpawnManager.current.GetCurrentWave();
    //αν εκανα καλυτερο ρεκορ απο το προηγουμενο, το αποθηκευει
    if (waveNumber > PlayerPrefs.GetInt("BestWave"))
    {
        PlayerPrefs.SetInt("BestWave", waveNumber);
    }
    gameOverText.text = $"You made it to <color=red>Wave {waveNumber}</color> !";
    yield return new WaitForSeconds(6f);
    Ti
```

Εικόνα 56: Λογική Κώδικα για StopGameAfterDie

4.2.1.7 Χειρισμός Third Person Shooter

Το Third Person Shooter Controller αποτελεί το πιο περίπλοκο script του χαρακτήρα καθώς περιλαμβάνει την λογική για το πως πυροβολάει ο χαρακτήρας και πως συνδέεται με το animation rigging. Πέρα από αυτό, έχει διάφορες άλλες λειτουργίες στις οποίες γίνεται αναφορά παρακάτω.

Στα First Person Shooter η λογική είναι ευκολότερη καθώς το κέντρο της οθόνης είναι εκεί που θα πάει πάντα η σφαίρα. Όμως, στην περίπτωση μας, επειδή η κάμερα βρίσκεται πίσω και πάνω από τον χαρακτήρα, το κέντρο της οθόνης δεν ταυτίζεται πάντα με το τι βλέπει ο χαρακτήρας. Για τον λόγο αυτό, έχουμε άλλη λογική στον κώδικα μας. Θα αναλύσουμε πως λειτουργεί μόνο για το πιστολάκι καθώς είναι το ίδιο και για το αυτόματο. Αρχικά, βρίσκω το κέντρο της κάμερας και τσεκάρω αν το raycast ακουμπάει κάτι μέσα στο LayerMask που έχω ορίσει. Το raycast για λόγους ευκολίας, μπορούμε να το σκεφτούμε ως μία ευθεία γραμμή η οποία μας επιστρέφει το πρώτο αντικείμενο που βρίσκει μέσα στην μάσκα.

```
//οριζουμε τα βαρη για τα rigs αναλογα τις τιμες, χρησιμοποιω lerp για να γινει smooth
        aimPistolRig.weight = Mathf.Lerp(aimPistolRig.weight, aimPistolRigWeight, Time.deltaTime * 20f);
        riflePistolRig.weight = Mathf.Lerp(riflePistolRig.weight, aimRifleRigWeight, Time.deltaTime * 20f);
        Vector3 mouseWorldPosition = Vector3.zero;
        Vector2 screenCenterPoint = new(Screen.width / 2f, Screen.height / 2f);
        //ray που δειχνει απο το κεντρο της οθονης στο σημειο που χτυπησε
        cameraRay = Camera.main.ScreenPointToRay(screenCenterPoint);
        //αν χτυπησει κατι
        if (Physics.Raycast(cameraRay, out cameraRaycastHit, 999f, aimColliderLayerMask))
        {
            //παιρνω το σημειο που χτυπησε σε vector3
            mouseWorldPosition = cameraRaycastHit.point;
            //αφου εχω παρει το σημειο σε world position, τωρα δημιουργω αλλο raycast που ξεκιναει απο το πιστολακι και δειχνει εκει που χτυπησε το προηγουμενο raycast
            pistolRay = new Ray(spawnBulletPositionPistol.position, (cameraRaycastHit.point - spawnBulletPositionPistol.position).normalized);
            //αντιστοιχα για το rifle
            rifleRay = new Ray(spawnBulletPositionRifle.position, (cameraRaycastHit.point - spawnBulletPositionRifle.position).normalized);
        }
        //αν δεν σημαδευει καποιο collider απλα βαραει οπου βλεπει σε 1000 μοναδες μακρια
        else
        {
            hitPointWorld.position = cameraRay.GetPoint(1000);
            mouseWorldPosition = cameraRay.GetPoint(1000);
            //αφου εχω παρει το σημειο σε world position, τωρα δημιουργω αλλο raycast που ξεκιναει απο το πιστολακι και δειχνει εκει που χτυπησε το προηγουμενο raycast
            pistolRay = new Ray(spawnBulletPositionPistol.position, (cameraRaycastHit.point - spawnBulletPositionPistol.position).normalized);
            //αντιστοιχα για το rifle
            rifleRay = new Ray(spawnBulletPositionRifle.position, (cameraRaycastHit.point - spawnBulletPositionRifle.position).normalized);
        }
```

Εικόνα 57: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 1)

Έπειτα, δημιουργούμε άλλο ένα raycast, αυτή τη φορά από την άκρη του όπλου προς το σημείο που χτύπησε το raycast της κάμερας. Από εδώ παίρνουμε το τελικό σημείο που θα χτυπήσει η σφαίρα, το οποίο επειδή έχουμε Third Person κάμερα, μπορεί να διαφέρει από το αρχικό σημείο. Έτσι, αν υπάρχει κάποιο εμπόδιο όπως ένας τοίχος, η σφαίρα θα χτυπήσει τον τοίχο και δεν θα τον διαπεράσει για να πάει στο αρχικό σημείο που πήραμε από την κάμερα. Σε αυτό το στάδιο επίσης βλέπουμε το tag που έχει αυτό που χτυπήσαμε, αν είναι κομμάτι του χάρτη τότε κάνουμε την Boolean μεταβλητή CanSpawnDecal αληθή το οποίο θα μας χρειαστεί αργότερα για να δημιουργήσουμε τα Decals για τις τρύπες από τις σφαίρες.

```
//αν το raycast απο το πιστολακι χτυπησει κατι
if (Physics.Raycast(pistolRay, out pistolRaycastHit, 999f, aimColliderLayerMask))
{
    hitPointWorld.position = Vector3.Lerp(hitPointWorld.position, pistolRaycastHit.point, Time.deltaTime * smoothFactor);
    //παιρνω το σημειο που χτυπησε
    mouseWorldPosition = pistolRaycastHit.point;
    //κραταω αν χτυπησε το περιβαλλον για μετα ωστε να ξερω αν θα κανω spawn το decal ή οχι
    if (pistolRaycastHit.transform.tag == "Environment")
    {
        canSpawnDecal = true;
    }
    else
    {
        canSpawnDecal = false;
    }
}
```

Εικόνα 58: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 2)

Επόμενο στάδιο είναι να στοχεύσουμε με το όπλο. Όταν στοχεύουμε κάνουμε το βάρος για το aimPistol ίσο με 1 το οποίο ενεργοποιεί το animation rigging που έχουμε δημιουργήσει, καθώς και το βάρος για το animation του PistolAim. Μετά τσεκάρουμε αν πρέπει να εμφανίσουμε το Decal 'X' για το εμπόδιο (εικόνα 33) και τέλος στρίβουμε τον χαρακτήρα προς το σημείο που στοχεύει.

```
//αν κανουμε aim με πιστολακι, ειναι grounded και μπορει να σημαδεψει
if (starterAssetsInputs.aim && animator.GetBool("Grounded") && WeaponManager.current.CheckWeaponState("Pistol") && canAim)
{
    //αλλαζουμε το βαρος για το rig απο το πιστολακι
    aimPistolRigWeight = 1f;
    //ενεργοποιω το UI του crosshair
    crosshair.SetActive(true);
    //αλλαζω την καμερα στην aim
    aimVirtualCamera.gameObject.SetActive(true);
    //αλλαζω και το sensitivity
    thirdPersonController.SetSensitivity(aimSensitivity);
    //απενεργοποιω προσωρινο το rotate μεσω το thirdPersonController και το χειριζομαι παρακατω
    thirdPersonController.SetRotateOnMove(false);
    //στο layer που ειναι το animation του pistol aim, αυξανω το βαρος smooth σε 1 ωστε να παιζει το animation του
    animator.SetLayerWeight(2, Mathf.Lerp(animator.GetLayerWeight(2), 1f, Time.deltaTime * 10f));
    CheckIfObstacle(pistolRaycastHit);

    //ο χαρακτηρας στριβει smooth οπου στοχευει
    mouseWorldPosition.y = transform.position.y;
    Vector3 aimDirection = (mouseWorldPosition - transform.position).normalized;
    transform.forward = Vector3.Lerp(aimDirection, transform.forward, Time.deltaTime * smoothFactor);
}
```

Εικόνα 59: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 3)

Το τελικό στάδιο είναι όταν πυροβολάμε. Αφού τσεκάρουμε ότι μπορούμε να το κάνουμε (δηλαδή έχει σφαίρες και έχει περάσει ο χρόνος καθυστέρησης που έχουμε όρισει) τότε τρέχει το κομμάτι του κώδικα στο οποίο ελκύονται από τον ήχο οι εχθροί που βρίσκονται σε κοντινή απόσταση, παίζει του animation του shoot για το πιστολάκι στο οποίο φεύγει ο άδειος κάλυκας από το όπλο μέσα από μια σειρά από animation events, και τέλος εμφανίζουμε τα εφέ όπως το lineRenderer και το muzzleFlush του όπλου.

```
//αν στοχευουμε και πυροβλησουμε
if (starterAssetsInputs.aim && starterAssetsInputs.shoot && animator.GetBool("Grounded") && canAim)
{
    //αν πυροβλησω με πιστολακι και εχει περασει το διαστημα που εχω ως περιορισμο
    if (WeaponManager.current.CheckWeaponState("Pistol") && Time.time > nextFire)
    {
        //αν εχει σφαιρες
        if (PistolHandler.current.HasAmmo())
        {
            //ελκυει τους εχθρους σε αποσταση ιση με 80
            AttractZombiesWithSound(80);
            //καταναλωνει μια σφαιρα
            PistolHandler.current.ConsumeAmmo();
            //δημιουργει ενα impulse στην καμερα
            impulse.GenerateImpulse(1);
            //οριζει το χρονο για το επομενο δυνατο shoot
            nextFire = Time.time + pistolFireRate;
            //παιζει το animation του pistol shoot
            pistolAnimator.SetTrigger("Shoot");
            //δημιουργει το bullettrail της σφαιρας
            SpawnBulletTrail(pistolRaycastHit.point, spawnBulletPositionPistol);
            //παιζει το particle system του muzzle flash
            muzzleFlashPistol.Play();
            //αναλυεται παρακατω
            SpawnDecal(pistolRaycastHit, "Pistol");
            //αρχικοποιουμε το shoot
            starterAssetsInputs.shoot = false;
        }
        //αν δεν εχει σφαιρες απλα παιζει τον ηχο για αδειο οπλο
        else
        {
            PistolHandler.current.PlayEmptySound();
            starterAssetsInputs.shoot = false;
        }
    }
```

Εικόνα 60: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 4)

```
private void AttractZombiesWithSound(int zombiesHearSound)
{
    //βρισκω ολους τους εχθρους μεσω overlapsphere για καλυτερο optimize
    Collider[] enemiesInRange = Physics.OverlapSphere(transform.position, zombiesHearSound, zombiesLayer);
    List<GameObject> enemiesToHit = new();

    for (int i = 0; i < enemiesInRange.Length; i++)
    {
        enemiesToHit.Add(enemiesInRange[i].gameObject);
    }
    //για καθε εχθρο που βρηκαμε, παιζει το animation του hearSound και ξεκιναει να κυνηγαει τον παιχτη
    foreach (GameObject enemy in enemiesToHit)
    {
        enemy.GetComponent<Animator>().SetBool("HearSounds", true);
    }
}
```

Εικόνα 61: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 5)

```
private void SpawnBulletTrail(Vector3 hitPoint, Transform bulletStartPoint)
{
    GameObject bulletTrailEffect = Instantiate(bulletTrail.gameObject, bulletStartPoint.position, Quaternion.identity);
    LineRenderer lineR = bulletTrailEffect.GetComponent<LineRenderer>();
    lineR.SetPosition(0, bulletStartPoint.position);
    lineR.SetPosition(1, hitPoint);
    Destroy(bulletTrailEffect, 1f);
}
```

Εικόνα 62: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 6)

```
private void SpawnDecal(RaycastHit hitPoint, string weaponName)
{
    var hitGameobject = hitPoint.transform.gameObject;
    Vector3 directionToPlayer = (atm.transform.position - hitPoint.transform.position).normalized;
    Vector3 centerContactPoint = hitPoint.collider.bounds.center;
    Quaternion rotation = Quaternion.LookRotation(directionToPlayer);
    rotation.eulerAngles = new Vector3(0, rotation.eulerAngles.y, rotation.eulerAngles.z);
    if (hitGameobject.layer == 8)
    {
        Animator enemyAnim = hitGameobject.GetComponentInParent<Animator>();
        enemyAnim.SetBool("isPatrolling", false);
        enemyAnim.SetBool("isChasing", true);
        enemyAnim.Play("damageZombie");
        atm.DealDamage(hitGameobject, weaponName, hitPoint.point);
        GameObject blood = Instantiate(bloodEffect, centerContactPoint, rotation);
        blood.transform.SetParent(hitGameobject.transform);
        Destroy(blood, 2);
    }
    // αν εκανε κατι hit
    if (canSpawnDecal)
    {
        //παιρνω ενα random decal
        int randomDecal = (int)Random.Range(0f, decals.Length);
        //το δημιουργω στην θεση που εκανε hit το raycast
        GameObject decal = Instantiate(decals[randomDecal], hitPoint.point, Quaternion.identity);
        ParticleSystem hitEffectParticle = Instantiate(hitEffect, hitPoint.point, Quaternion.identity);
        //το βαζω με κατευθυνση αντιθετη του normal του raycast
        decal.transform.forward = -hitPoint.normal;
        hitEffectParticle.transform.forward = hitPoint.normal;
        hitEffectParticle.Emit(1);
        //το καταστρεφω 3 δευτερολεπτα μετα
        Destroy(decal, 3f);
        Destroy(hitEffectParticle.gameObject, hitEffectParticle.main.duration);
    }
}
```

Εικόνα 63: Λογική Κώδικα ThirdPersonShooterController (Στιγμιότυπο 7)

4.2.1.8 Συλλογή Αντικειμένων (Grab Items)

Μια άλλη ικανότητα που δόθηκε στον χαρακτήρα είναι να μπορεί να συλλέγει τα αντικείμενα. Έτσι, όταν βρίσκει στον χάρτη όπλα ή σφαίρες τρέχει η λογική από πίσω για την σειρά που θα ακολουθήσει και τις ενέργειες που θα κάνει. Υπάρχει η δυνατότητα για εντολή συλλογής αντικειμένων από κοντά ή από μακριά. Από την στιγμή που αρχίσει η κίνηση δεν υπάρχει τρόπος να σταματήσει μέχρι να τελειώσει, γι' αυτό ο παίκτης πρέπει να είναι ιδιαίτερα προσεκτικός. Τα αντικείμενα όταν τα κοιτάζουμε έχουν ένα κίτρινο περίγραμμα (outline) τα οποία σηματοδοτούν ότι από αυτό το σημείο μπορεί να δοθεί η εντολή. Για το περίγραμμα χρησιμοποιήθηκε ένα δωρεάν asset [asset 4] από το Unity Asset Store το οποίο επιτρέπει την γρήγορη προσθήκη περιγράμματος όπως και τροποποιήσεις σχετικά με το χρώμα με διαφορετικές λειτουργίες. Για τα όπλα συγκεκριμένα, κάθε φορά που τα κοιτάει ο χαρακτήρας, εμφανίζεται και ένα μικρό UI στον 3D χώρο όπου μας ενημερώνει για το πόσες σφαίρες έχει μέσα το συγκεκριμένο όπλο.



Εικόνα 64: GrabItems - Παράδειγμα Για Όπλα



Εικόνα 65: GrabItems - Παράδειγμα Για Σφαίρες

Αναλυτικότερα τώρα, για τα αντικείμενα που θέλω να μπορεί να πιάνει ο χαρακτήρας, δημιούργησα ένα Layer (GrabbableItem) το οποίο χρησιμοποιείται στον κώδικα όταν ψάχνω να βρω αντικείμενα που πιάνονται στον χώρο. Στην αρχή λοιπόν δημιουργούμε ένα Boxcast (ίδιο με Raycast απλά αντί για γραμμή 'εκπέμπει' ένα κουτί) για μεγαλύτερο εύρος ώστε να σημαδεύονται τα αντικείμενα πιο εύκολα. Πρώτα κάνουμε έλεγχο αν το αντικείμενο πιάνεται, αν το έχω ήδη (καθώς δεν μπορώ να κρατάω πολλαπλά ίδια αντικείμενα) και αν δεν κουνιέται (εικόνα 66). Έπειτα ελέγχουμε αν κοιτάμε όπλο ώστε να εμφανίσουμε το UI για τις σφαίρες που έχει μέσα(εικόνα 67). Στην συνέχεια τσεκάρουμε για είσοδο από τον παίκτη. Αν πατήσει το πλήκτρο 'E' τότε ανάλογα την απόσταση από το αντικείμενο εκτελεί κάποιες ενέργειες. Αν βρίσκεται κοντά παίζει κατευθείαν το animation που σκύβει και το πιάνει αλλιώς πρώτα τρέχει αυτόματα προς το αντικείμενο και όταν φτάσει σε κοντινή απόσταση το πιάνει. Στην διάρκεια αυτή αφήνουμε ενεργό μόνο την είσοδο από το ποντίκι για να μπορεί να γυρίσει την κάμερα, όλα τα άλλα απενεργοποιούνται προσωρινά(εικόνα 68). Τέλος ανάλογα τι πιάσει, εκτελεί και τις κατάλληλες εντολές που θέλουμε. Πιο συγκεκριμένα, για τις σφαίρες γεμίζει και τα δύο όπλα, για το τσεκούρι απλά το παίρνει και για τα δύο όπλα διαβάζει τις σφαίρες που έχει μέσα για μελλοντική χρήση και το παίρνει.

```
//αν κοιταω grabbable item, ειμαι κοντα στο να το πιασω, δεν κουνιεται και δεν το εχω ηδη
            if (selectedGameobject && selectedGameobject.GetComponent<Rigidbody>().velocity == new Vector3(0, 0, 0)
             && distanceCharacterToItem < grabDistance && !CheckIfAlreadyHave(selectedGameobject.tag))
            {
                moveToObject = selectedGameobject;
                moveToObjectTag = moveToObject.tag;
                isClose = true;
                isCloseEnough = false;
                //αν ειχα προηγουμενο hightlighted, το βγαζω
                if (justForHideGameobject && justForHideGameobject != selectedGameobject)
                {
                    justForHideGameobject.OverlayHide();
                    if (justForHideGameobject.CompareTag("Pistol") || justForHideGameobject.CompareTag("Rifle"))
                    {
                        justForHideGameobject.transform.Find("Canvas").gameObject.SetActive(false);
                    }
                }
```

Εικόνα 66: Λογική Κώδικα GrabItems (Στιγμιότυπο 1)

```
if (selectedGameobject.CompareTag("Pistol"))
{
    selectedGameobject.OverlayShow();
    //το UI για τις σφαιρες πρεπει να κοιταει παντα πανω οτι rotation και να εχει το οπλο
    pistolAmmoUI = selectedGameobject.transform.Find("Canvas").gameObject;
    var centerPoint = selectedGameobject.transform.Find("Sketchfab_model");
    Vector3 desiredPosition = centerPoint.position + Vector3.up * 0.6f;
    pistolAmmoUI.transform.position = desiredPosition;
    pistolAmmoUI.transform.forward = Camera.main.transform.forward;
    pistolAmmoUI.SetActive(true);
    pistolAmmoTextUI = pistolAmmoUI.transform.Find("Ammo").GetComponent<TextMeshProUGUI>();
    int pistolCurrentAmmo = selectedGameobject.GetComponent<AmmoInside>().GetPistolCurrentAmmo();
    int pistolTotalAmmo = selectedGameobject.GetComponent<AmmoInside>().GetPistolTotalAmmo();
    if (pistolCurrentAmmo == 0 && pistolTotalAmmo == 0)
    {
        pistolAmmoTextUI.color = Color.red;
    }
    else
    {
        pistolAmmoTextUI.color = Color.green;
    }
    pistolAmmoTextUI.text = pistolCurrentAmmo.ToString() + " / " + pistolTotalAmmo.ToString();
}
```

Εικόνα 67: Λογική Κώδικα GrabItems (Στιγμιότυπο 2)

```
//αν ειναι κοντα βρισκω το rotation
            if (isClose)
            {
                if (selectedGameobject != null)
                {
                    targetRotation = Quaternion.LookRotation(selectedGameobject.transform.position - transform.position);
                    isRotatingWhenClose = true;
                    //για λιγο μπορω μονο να γυρναω το ποντικι
                    DisableAllActionsExceptLook();
                }
            }
            //αν ειναι στα ορια που φτανει βρισκω το rotation
            else if (isCloseEnough)
            {
                targetRotation = Quaternion.LookRotation(moveToObject.transform.position - transform.position);
                targetRotation.x = 0;
                targetRotation.z = 0;
                isRotatingWhenCloseEnough = true;
                DisableAllActionsExceptLook();
            }
        }
        if (isRotatingWhenClose)
        {
            //στριβω τον χαρακτηρα προς το αντικειμενο
            transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation, rotationSpeed * Time.deltaTime);
            Vector3 currentEulerAngles = transform.rotation.eulerAngles;
            Vector3 targetEulerAngles = targetRotation.eulerAngles;
            //και οταν πλεον το κοιταζει
            if (Mathf.Abs(currentEulerAngles.y - targetEulerAngles.y) < 0.5f)
            {
                isRotatingWhenClose = false;
                //παιζει το animation του grab
                anim.SetTrigger("GrabItem");
            }
        }
        else if (isRotatingWhenCloseEnough)
        {
            moveForAmmo = true;
            //στριβω τον χαρακτηρα προς το αντικειμενο
            transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation, rotationSpeed * Time.deltaTime);
            Vector3 currentEulerAngles = transform.rotation.eulerAngles;
            Vector3 targetEulerAngles = targetRotation.eulerAngles;
            //και οταν πλεον το κοιταζει
            if (Mathf.Abs(currentEulerAngles.y - targetEulerAngles.y) < 0.5f)
            {
                //ανεβαζω το speed value στο animator ωστε να "τρεχει" προς την κατευθυνση
                anim.SetFloat("Speed", SprintSpeed);
                //ομαλα μετακινω τον χαρακτηρα προς το αντικειμενο μεχρι να ερθει αρκετα κοντα και να το πιασει
                Vector3 newPosition = Vector3.Lerp(transform.position, moveToObject.transform.position, SprintSpeed * Time.deltaTime);
                transform.position = newPosition;
                if (Vector3.Distance(transform.position, moveToObject.transform.position) < grabDistance)
                {
                    isRotatingWhenCloseEnough = false;
                    anim.SetTrigger("GrabItem");
                    moveForAmmo = false;
                }
            }
```

Εικόνα 68: Λογική Κώδικα GrabItems (Στιγμιότυπο 3)

4.2.1.9 Πέταγμα Αντικειμένων (Throw Items)

Καθώς ο χαρακτήρας έχει την ικανότητα να πιάνει αντικείμενα, κάπως πρέπει και να τα πετάει. Γι' αυτό το λόγο, δημιούργησα άλλο ένα script το οποίο διαχειρίζεται αυτή την κίνηση. Αρχικά περιμένουμε για είσοδο με το πλήκτρο 'G'. Αφού τσεκάρουμε ότι κρατάει κάτι και ότι μπορεί να πετάξει το αντικείμενο, τρέχει το animation που έχουμε ορίσει και προχωράμε στην βασική συνάρτηση Throw όπου ενεργοποιείται μέσω animation event. Εκεί, ανάλογα με το όπλο που κρατάμε περνάνε διαφορετικά ορίσματα. Στην συνέχεια, η ThrowWeapon παίρνει το velocity του χαρακτήρα ώστε η δύναμη που θα πεταχτεί το όπλο να εξαρτάται και από την ταχύτητα μας. Μετά απενεργοποιείται το Gameobject του όπλου και αντίστοιχα ενεργοποιείται το visual όπλο αφού πρώτα γίνει αντιγραφή το transform ώστε να εμφανιστεί στο ίδιο σημείο. Τέλος, περνάει η πληροφορία για τις διαθέσιμες σφαίρες που υπάρχουν στο visual όπλο για αργότερα αν το ξαναπάρουμε και εφαρμόζουμε την τελική δύναμη στο όπλο αφού πρώτα γίνει detach από τον χαρακτήρα. Η λογική των βασικών κομματιών του κώδικα φαίνεται παρακάτω στις εικόνες.

```
void Update()
{
    if (!starterAssetsInputs.throwItem
    || WeaponManager.current.CheckWeaponState("Fists")
    || !WeaponManager.current.GetThrowWeaponState())
        return;

    anim.Play("throwWeapon");
}
public void Throw()
{
    //αναλογα τι κραταω για να πεταξω, βρισκω το αντικειμενο και τρεχω την ThrowWeapon
    if (WeaponManager.current.CheckWeaponState("Melee"))
    {
        Transform currentWeaponToThrow = FindChildWithKeyword(WeaponsSlot.transform, "axe");
        CopyTransformAndScale(weapons[0].transform, currentWeaponToThrow.transform);
        ThrowWeapon(weapons[0], currentWeaponToThrow, "Melee");
    }
    else if (WeaponManager.current.CheckWeaponState("Pistol"))
    {
        Transform currentWeaponToThrow = FindChildWithKeyword(WeaponsSlot.transform, "colt");
        CopyTransformAndScale(weapons[1].transform, currentWeaponToThrow.transform);
        ThrowWeapon(weapons[1], currentWeaponToThrow, "Pistol");
    }
    else if (WeaponManager.current.CheckWeaponState("Rifle"))
    {
        Transform currentWeaponToThrow = FindChildWithKeyword(WeaponsSlot.transform, "rifle");
        CopyTransformAndScale(weapons[2].transform, currentWeaponToThrow.transform);
        ThrowWeapon(weapons[2], currentWeaponToThrow, "Rifle");
    }
}
```

Εικόνα 69: Λογική Κώδικα ThrowItems (Στιγμιότυπο 1)

```
private void ThrowWeapon(GameObject WeaponInHand, Transform VisualWeapon, String weaponType)
    {
        //κραταω το speed για να παρει το οπλο το velocity μετα που το πεταω
        float playerSpeed = anim.GetFloat("Speed");
        Rigidbody weaponRB = VisualWeapon.GetComponent<Rigidbody>();
        WeaponInHand.SetActive(false);
        VisualWeapon.gameObject.SetActive(true);
        VisualWeapon.SetParent(null);
        //αν ειναι pistol ή rifle ανανεωνω τις σφαιρες που εχει μεσα
        switch (weaponType)
        {
            case "Pistol":
                VisualWeapon.GetComponent<AmmoInside>().SetPistolCurrentAmmo(PistolHandler.current.currentPistolAmmo);
                VisualWeapon.GetComponent<AmmoInside>().SetPistolTotalAmmo(PistolHandler.current.pistolTotalAmmo);
                PistolHandler.current.UpdateText(false);
                break;
            case "Rifle":
                VisualWeapon.GetComponent<AmmoInside>().SetRifleCurrentAmmo(RifleHandler.current.currentRifleAmmo);
                VisualWeapon.GetComponent<AmmoInside>().SetRifleTotalAmmo(RifleHandler.current.rifleTotalAmmo);
                RifleHandler.current.UpdateText(false);
                break;
        }
        //κανω apply force λιγο προς τα πανω και προς τα μπροστα(επηρεαζεται και απο την ταχυτητα του παιχτη)
        weaponRB.AddForce(transform.up * dropUpwardForce, ForceMode.Impulse);
        playerSpeed = Mathf.Max(2f, playerSpeed);
        weaponRB.AddForce(transform.forward * dropForwardForce * playerSpeed, ForceMode.Impulse);
        float random = UnityEngine.Random.Range(-1f, 1f);
        weaponRB.AddTorque(new Vector3(random, random, random) * 50);
        //αλλαζει το state σε μπουνιες και ανανεωνει το UI
        WeaponManager.current.ChangeTo("Fists");
        WeaponManager.current.SetImageTo(0);
        WeaponManager.current.DisableCarryWeapon(weaponType);
        starterAssetsInputs.meleeAttack = false;
        starterAssetsInputs.throwItem = false;
    }
```

Εικόνα 70: Λογική Κώδικα ThrowItems (Στιγμιότυπο 2)

4.2.1.10 Διαχείριση ζωής (Health Controller)

Για την διαχείριση της ζωής στο UI έφτιαξα ένα μικρό script που ενημερώνει δυναμικά την ζωή του παίκτη καθώς είναι η πιο σημαντική πληροφορία στο παιχνίδι. Για να δοθεί και περισσότερο έμφαση, δίπλα υπάρχει μια εικόνα καρδιάς στην οποία προστέθηκε ένα μικρό animation το οποίο μεγαλώνει-μικραίνει την εικόνα με συχνότητα ανάλογη της ζωής την δεδομένη στιγμή.

```
void Update()
{
    currentHealth = GetHealth();
    if (currentHealth > 1)
        health_text.text = currentHealth.ToString();
    else
        health_text.text = "0";

    if (!animating)
    {
        //αν ειναι πιο κατω απο 20 χτυπαει πολυ γρηγορα
        if (currentHealth < healthThreshold1)
        {
            StartCoroutine(ScaleAnimation(animationSpeed1));
        }
        //κατω απο 50 λιγο πιο αργα
        else if (currentHealth < healthThreshold2)
        {
            StartCoroutine(ScaleAnimation(animationSpeed2));
        }
    }
}
```

Εικόνα 71: Λογική Κώδικα HealthController (Στιγμιότυπο 1)

```
IEnumerator ScaleAnimation(float animationSpeed)
{
    animating = true;

    while (GetHealth() < healthThreshold1 || GetHealth() < healthThreshold2)
    {
        float t = 0f;
        while (t < 1f)
        {
            t += Time.deltaTime * animationSpeed;
            float scale = Mathf.Lerp(minScale, maxScale, t);
            healthBarTransform.localScale = new Vector3(scale, scale, scale);
            yield return null;
        }

        t = 0f;
        while (t < 1f)
        {
            t += Time.deltaTime * animationSpeed;
            float scale = Mathf.Lerp(maxScale, minScale, t);
            healthBarTransform.localScale = new Vector3(scale, scale, scale);
            yield return null;
        }
    }

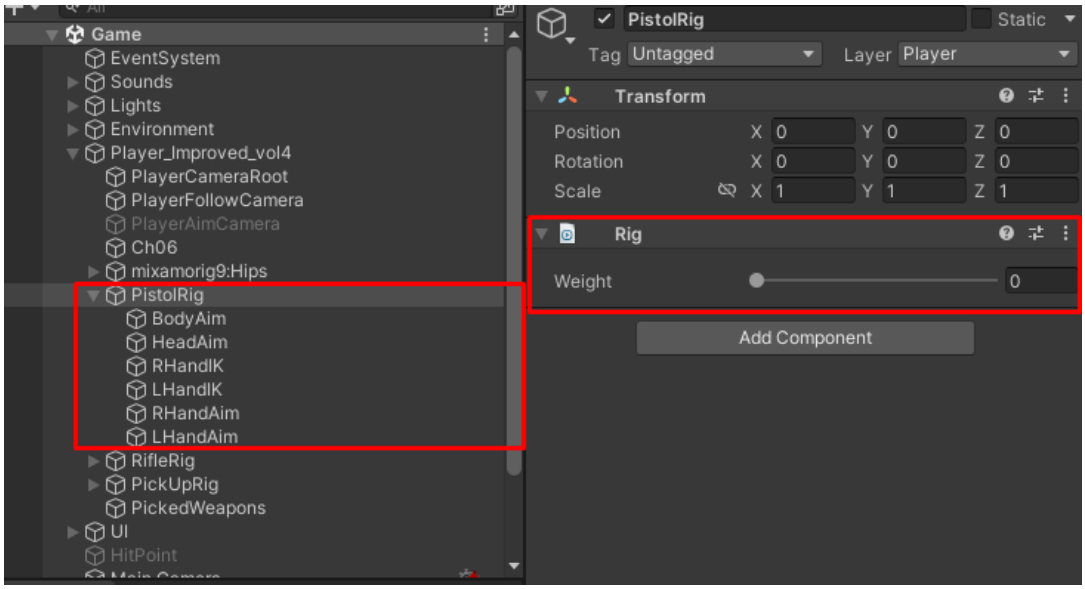
    animating = false;
}
```

Εικόνα 72: Λογική Κώδικα HealthController (Στιγμιότυπο 2)

4.2.1.11 Animation Rigging

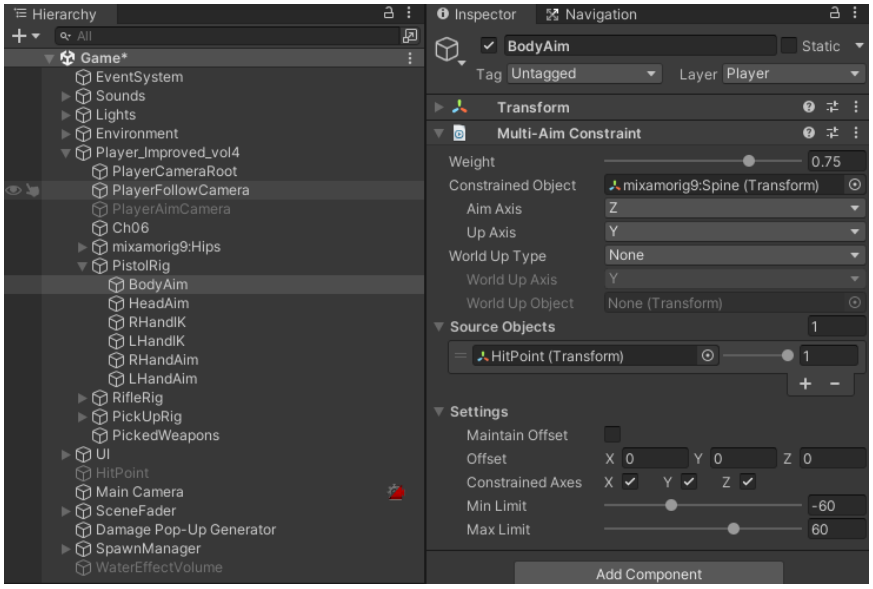
Το animation Rigging είναι ένα πακέτο της Unity που επιτρέπει την δημιουργία προηγμένων συστημάτων για τον έλεγχο των κινήσεων κατά την αναπαραγωγή ενός animation. Μπορεί να θέσει περιορισμούς που θα ενταχθούν πάνω από το εκάστοτε animation που παίζει, ώστε να προσφέρει περισσότερο ρεαλισμό στις κινήσεις. Παραδείγματα που βλέπουμε να χρησιμοποιείται συχνά σε παιχνίδια είναι το Foot Inverse Kinematics (Foot IK) που τοποθετεί το πόδι του χαρακτήρα στην σωστή θέση ανάλογα το έδαφος καθώς και οι περιορισμοί στόχου (Aim Constraints) που ορίζουμε σε ένα οποιοδήποτε αντικείμενο ώστε να περιστρέφεται προς ένα συγκεκριμένο στόχο. Ειδικά στα Shooter παιχνίδια, η περιστροφή τόσο του όπλου όσο και του χαρακτήρα προς το σημείο που σημαδεύει, δίνει ένα εντυπωσιακό τελικό αποτέλεσμα. Για την εργασία, το πακέτο χρησιμοποιήθηκε σε δύο σημεία που αναλύονται στην συνέχεια.

Αρχικά, χρησιμοποιήθηκε το πακέτο για τον χαρακτήρα όταν σημαδεύει με κάποιο όπλο. Δημιουργήθηκε ένα καινούριο Rig το οποίο είναι υπεύθυνο για το πιστολάκι. Κάθε φορά που ο χαρακτήρας σημαδεύει, ενεργοποιείται αυτόματα αλλάζοντας την τιμή του βάρους του από 0 σε 1. Το βάρος καθορίζει το ποσοστό επίδρασης που έχει κάθε bone του μοντέλου. Οπότε όταν δεν στοχεύουμε είναι πάντα 0 και μετά το αλλάζουμε ομαλά προς το 1 μέσω κώδικα. Στο Rig αυτό, επηρεάζονται τέσσερα διαφορετικά στοιχεία: το σώμα, το κεφάλι και τα δύο χέρια του χαρακτήρα. Η σειρά που τα βάζουμε έχει μεγάλη σημασία καθώς είναι η σειρά που επηρεάζονται οπότε πρέπει να δώσουμε μεγάλη προσοχή.



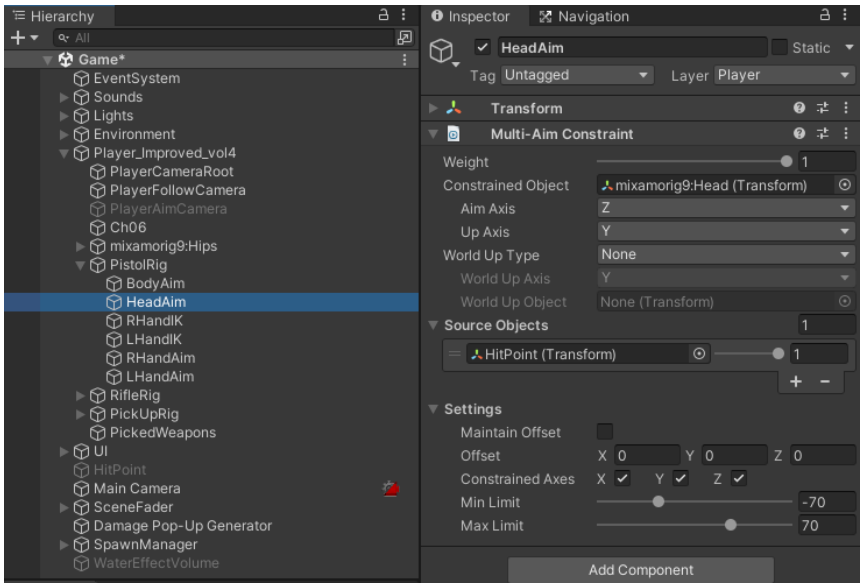
Εικόνα 73: PistolRig Και Βάρος

Για πρώτο περιορισμό βάλαμε το σώμα και συγκεκριμένα το κόκκαλο Spine ώστε να περιστρέφεται η μέση του χαρακτήρα προς τον στόχο. Μέσω ενός component, το Multi-Aim Constraint, μπορούμε να δώσουμε κάποιες λεπτομέρειες όπως το ποιο κόκκαλο επηρεάζουμε, προς τα που θα κοιτάει, περιορισμούς για την ελάχιστη και την μέγιστη περιστροφή αλλά και το βάρος. Στην συγκεκριμένη περίπτωση ορίσαμε ως αντικείμενο που θα κοιτάει το HitPoint που είναι το τελικό world space σημείο που στοχεύει ο χαρακτήρας.



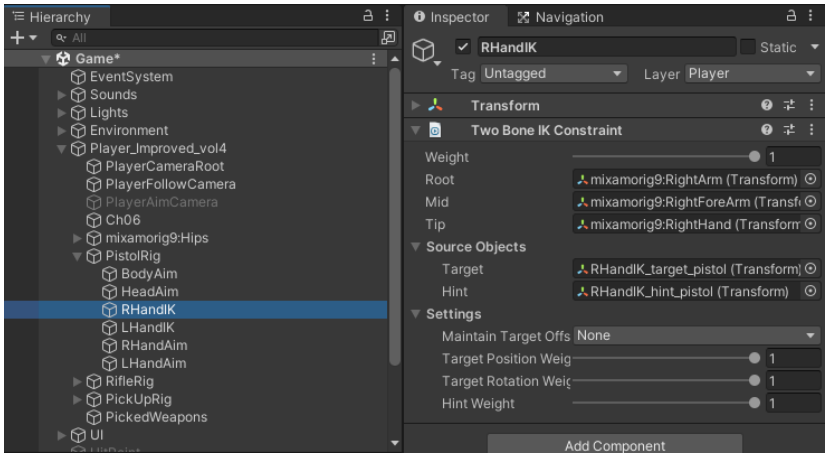
Εικόνα 74: BodyAim Constraint

Επόμενος περιορισμός είναι η περιστροφή για το κεφάλι. Παρόμοια με το σώμα, το κεφάλι του χαρακτήρα ‘κοιτάει’ προς τον τελικό στόχο. Ορίσαμε επίσης περιορισμούς ώστε να είναι όσο πιο κοντά γίνεται στον πραγματικό κόσμο.



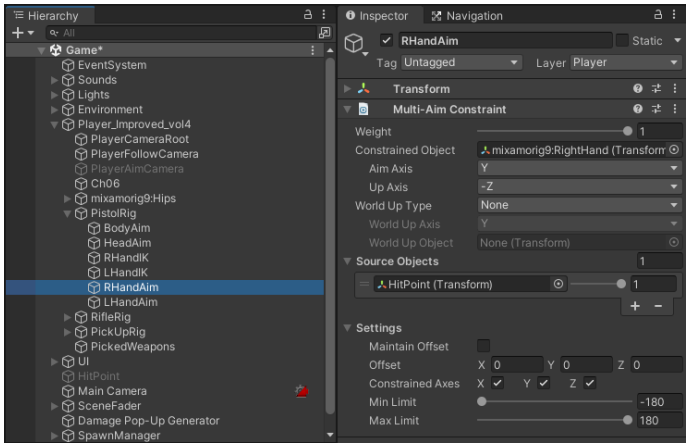
Εικόνα 75: HeadAim Constraint

Στην συνέχεια είναι οι πιο σημαντικοί περιορισμοί καθώς επηρεάζουν τα χέρια. Έχουμε δύο διαφορετικά, ένα το οποίο επηρεάζει όλο το χέρι και ένα που επηρεάζει μόνο το άκρο. Για όλο το χέρι αυτή τη φορά χρησιμοποιήσαμε ένα διαφορετικό component, το Two Bone IK Constraint. Με αυτό, καθορίζοντας τρία κόκκαλα, root-mid-tip, καθώς και δύο νέα Gameobject που φτιάχνουμε, target-hint, εφαρμόζουμε Inverse Kinematics στο χέρι. Inverse Kinematics (IK) είναι μια τεχνική όπου αντί να καθορίζουμε θέση για κάθε bone ξεχωριστά, καθορίζουμε μόνο το τελικό σημείο (target) και το σύστημα υπολογίζει αυτόματα τις θέσεις των κόμβων. Οπότε είναι μια πολύ καλή τεχνική για τα χέρια του χαρακτήρα, την οποία εφαρμόσαμε και στα δύο χέρια για να αλλάξουμε την θέση που είχαν από το αρχικό animation και να τα τοποθετήσουμε εμείς στο σημείο που θέλαμε.



Εικόνα 76: RightHand IK

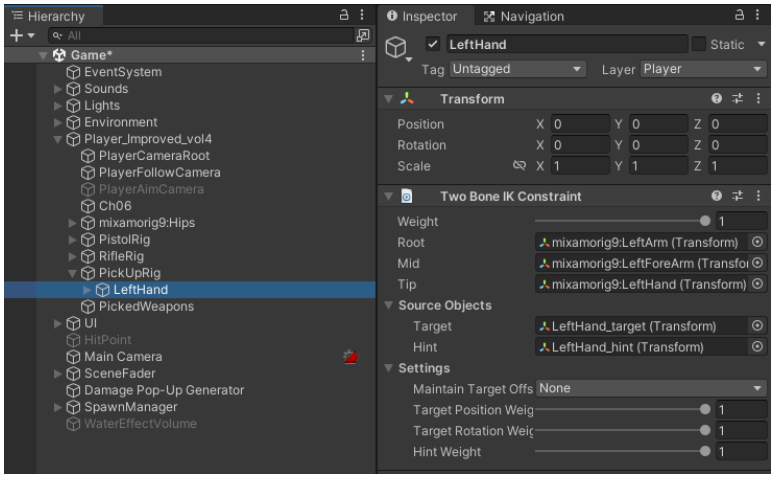
Τέλος, προστέθηκε ένας περιορισμός στα άκρα των χεριών. Με το ίδιο component που χρησιμοποιήσαμε στο σώμα και στο κεφάλι, ορίσαμε έναν περιορισμό στους καρπούς των χεριών ώστε να κοιτάνε προς τον στόχο. Έτσι, αφού και το όπλο είναι child στο χέρι, καταφέρνουμε να στρίβουμε και το όπλο μαζί ώστε να κοιτάει τον στόχο προσδίδοντας ένα αρκετά ρεαλιστικό αποτέλεσμα.



Εικόνα 77: RightHand Constraint

Έτσι, ολοκληρώθηκε το Animation Rigging για τα όπλα, καθώς το Rig για το αυτόματο όπλο είναι παρόμοιο με μόνη διαφορά τις διαφορετικές θέσεις για τα IK των χεριών. Τα υπόλοιπα ελέγχονται μέσω script που έχουμε ήδη αναφέρει στα οποία αλλάζοντας το βάρος τους, τα ενεργοποιούμε ή απενεργοποιούμε αντίστοιχα.

Ένα τρίτο απλό Rig που έχουμε ορίσει είναι το PickUp. Όταν ο χαρακτήρας πιάνει κάτι, τρέχει το animation που σκύβει να το πιάσει, όμως ανάλογα το αντικείμενο και την περιστροφή πολλές φορές η θέση του χεριού δεν ταίριαζε με το αντικείμενο. Το πρόβλημα αυτό λύθηκε περιορίζοντας το χέρι στο σωστό σημείο. Με ένα Two Bone IK Constraint όπως και στο aim, το δεξί χέρι πηγαίνει στο target που καθορίζεται κάθε φορά δυναμικά μέσω script. Έτσι, το χέρι εκτελεί μια αρμονική κίνηση πάνω στο αντικείμενο, προσφέροντας μια ακόμα εξαιρετική λεπτομέρεια στο παιχνίδι. Στην εικόνα 79, βλέπουμε με μπλε σφαίρα το position του target και πως το χέρι πηγαίνει προς το συγκεκριμένο σημείο, κάνοντας override το animation.



Εικόνα 78: PickUp Rig



Εικόνα 79: Παράδειγμα PickUp Rig

4.2.2 Χαρακτήρες Εχθρών και Συστατικά

4.2.2.1 Μοντέλα

Για τα μοντέλα των εχθρών – ζόμπι χρησιμοποιήθηκε τόσο το Mixamo όσο και το Sketchfab. Για το παιχνίδι έβαλα τέσσερα διαφορετικά μοντέλα ώστε να υπάρχει ποικιλία τόσο στο οπτικό επίπεδο όσο και στα χαρακτηριστικά και τις κινήσεις του.



Εικόνα 80: Πρώτο Μοντέλο Εχθρού



Εικόνα 81: Δεύτερο Μοντέλο Εχθρού



Εικόνα 82: Τρίτο Μοντέλου Εχθρού



Εικόνα 83: Τέταρτο Μοντέλο Εχθρού

4.2.2.2 Μηχανή Καταστάσεων Εχθρών (State Machine)

Οι εχθροί είναι υλοποιημένοι ως NPC (Non-Player Characters) και λειτουργούν με βάση ένα σύστημα συμπεριφοράς βασισμένο σε ένα state machine (μηχανή καταστάσεων). Το state machine επιτρέπει στους εχθρούς να αλλάζουν δυναμικά συμπεριφορά ανάλογα με το περιβάλλον τους, τις ενέργειες του παίκτη, και άλλες παραμέτρους. Αυτό δημιουργεί ένα πιο ρεαλιστικό και δυναμικό περιβάλλον παιχνιδιού χωρίς οι εχθροί να έχουν πάντα συγκεκριμένες κινήσεις και ενέργειες. Μπορούν να υπάρξουν σε μία από τις εξής καταστάσεις: Idle, Patrol, Chase, Scream, Attack1-Attack2, Dizzy Idle. Ας αναλύσουμε την κάθε μια:

Idle: Κατάσταση στην οποία ο εχθρός κάθεται ακίνητος για λίγα δευτερόλεπτα και παρατηρεί. Αν ο παίκτης έρθει σε πολύ κοντινή απόσταση ή αν ακούσει πυροβολισμούς τότε βγαίνει πρόωρα από την κατάσταση και κυνηγάει τον παίκτη.

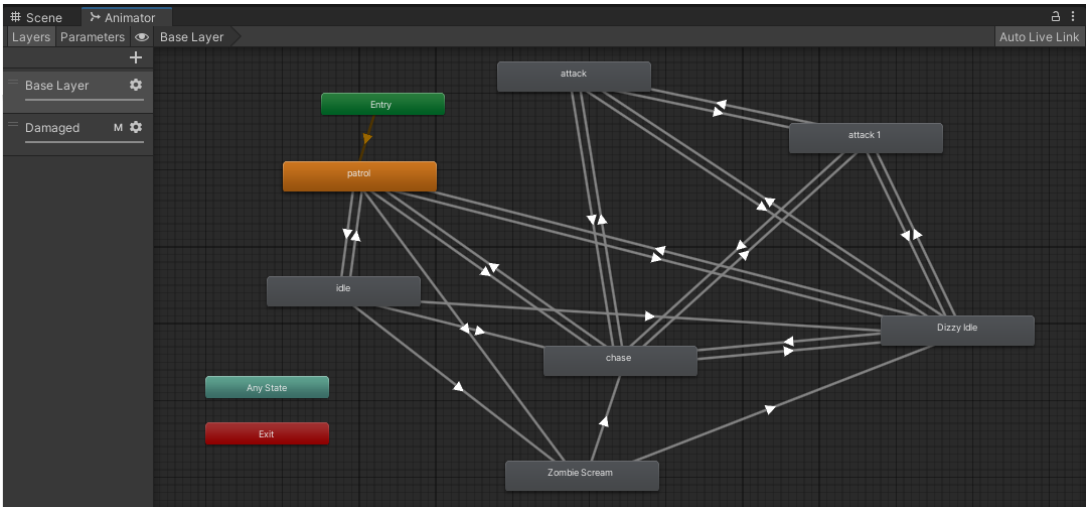
Patrol: Κατάσταση στην οποία ο εχθρός περπατάει αργά προς ένα τυχαίο σημείο που ορίζεται στο περιβάλλον. Όπως και στο idle, ο παίκτης μπορεί να διακόψει αυτήν την κατάσταση.

Chase: Κατάσταση στην οποία ο εχθρός κυνηγάει τον παίκτη. Δεν μπορεί να διακοπεί εκτός αν πληρούνται οι προϋποθέσεις για το Dizzy Idle ή φυσικά αν πεθάνει ο εχθρός.

Scream: Αν ο εχθρός ακούσει πυροβολισμούς από μακριά, τότε τρέχει ένα animation που 'ουρλιάζει' και στην συνέχεια αρχίζει να κυνηγάει τον παίκτη, όσο μακριά και αν βρίσκεται.

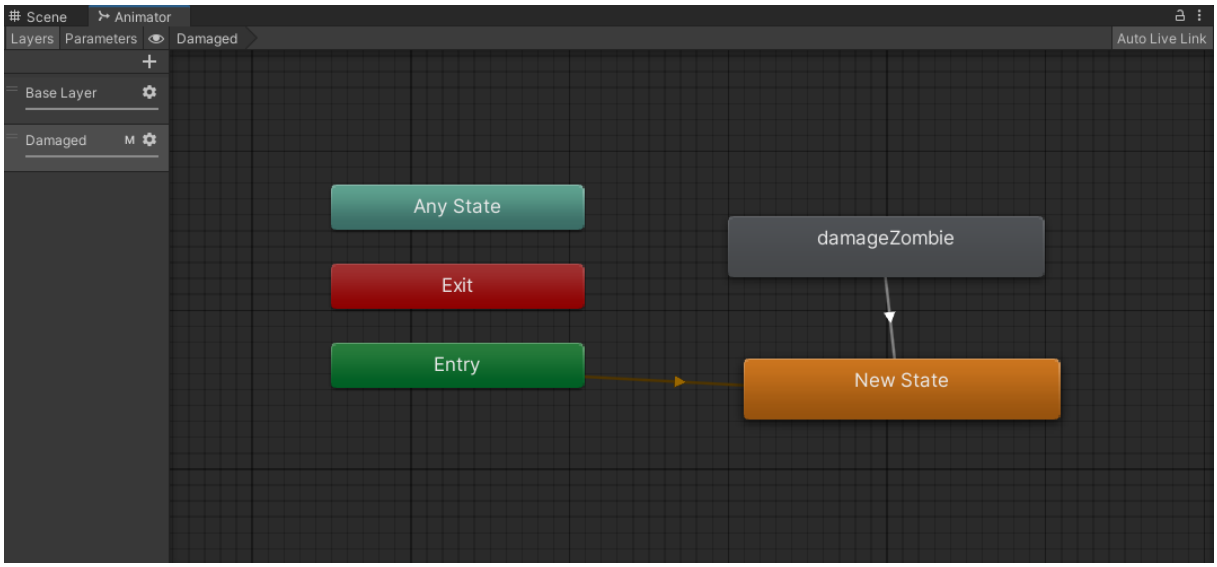
Attack1-Attack2: Όταν ο εχθρός έρθει αρκετά κοντά στον παίκτη ώστε να φτάνει να τον χτυπήσει, τότε τυχαία παίζει ένα από τα δύο animations που υπάρχουν για επίθεση. Κατά την διάρκεια της επίθεσης δεν διακόπτεται εκτός και αν μπει σε κατάσταση Dizzy.

Dizzy Idle: Κατάσταση στην οποία ο εχθρός βρίσκεται ζαλισμένος (stunned). Η κατάσταση αυτή δεν διακόπτεται με τίποτα μέχρι να περάσει ο χρόνος που αντιστοιχεί. Υλοποιήθηκε κυρίως για να κάνει το παιχνίδι ελαφρώς πιο εύκολο καθώς και να δώσει πολυπλοκότητα στην λειτουργία των εχθρών.



Εικόνα 84: Μηχανή Καταστάσεων Εχθρών (State Machine)

Όπως και στον χαρακτήρα μας που έχουμε διάφορα επίπεδα στο Animator, έτσι και στους εχθρούς έχουμε δύο διαφορετικά. Το πρώτο αφορά όλη τη λογική και τις μεταβάσεις από την μία κατάσταση στην άλλη, και το άλλο είναι για το animation που παίζει όταν δέχεται ζημιά. Έχουμε ορίσει μάσκα που επηρεάζει μόνο το πάνω μέρος του σώματος και κάνει override τα άλλα animation όπως ακριβώς λειτουργούν και τα περισσότερα παιχνίδια ίδιου τύπου.



Εικόνα 85: Animator Εχθρών

Ας δούμε τώρα την λογική του κώδικα που τρέχει για κάθε κατάσταση των εχθρών. Ξεκινώντας από το Patrol που είναι και η κατάσταση που ξεκινάνε οι εχθροί, η λογική είναι σχετικά απλή καθώς το μόνο που κάνουμε είναι να κοιτάμε την απόσταση με τον παίκτη. Αν γίνει αρκετά μικρή (15 στην περίπτωση μας), τότε αλλάζει τα Booleans που έχουμε για παραμέτρους στο Animator και έτσι αλλάζει και η κατάσταση. Επιπλέον, αν μετά από 20 δευτερόλεπτα είμαστε ακόμα στην ίδια κατάσταση, τότε μπαίνει σε Idle.

```
override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    if (agent.remainingDistance < 1f)
    {
        //βρισκει τυχαιο σημειο στο περιβαλλον
        GetNewRandomWaypoint(animator);
    }
    timer += Time.deltaTime;
    //μετα απο 20 δευτερολεπτα μπαινει παλι σε idle state
    if (timer > 20)
    {
        animator.SetBool("isPatrolling", false);
    }
    float distance = Vector3.Distance(player.position, animator.transform.position);
    //αν ειναι σχετικα κοντα και μπορει να παει τοτε κυνηγαει τον παιχτη
    if (distance <= chaseRange)
    {
        animator.SetBool("isPatrolling", false);
        animator.SetBool("isChasing", true);
    }
}
```

Εικόνα 86: Κατάσταση Εχθρών – Patrol

Στην κατάσταση Idle μένει μόνο για 3 δευτερόλεπτα πριν ξεκινήσει πάλι το Patrol. Αν στο διάστημα αυτό βρεθεί σε κοντινή απόσταση με τον παίκτη, μπαίνει στην κατάσταση Chase.

```
override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    timer += Time.deltaTime;
    //μετα απο 3 δευτερολεπτα μπαινει σε patrol state
    if (timer > 3)
    {
        animator.SetBool("isPatrolling", true);
    }
    float distance = Vector3.Distance(player.position, animator.transform.position);
    //αν ειναι σχετικα κοντα και μπορει να παει τοτε κυνηγαει τον παιχτη
    if (distance <= chaseRange )
    {
        animator.SetBool("isPatrolling", false);
        animator.SetBool("isChasing", true);
    }
}
```

Εικόνα 87: Κατάσταση Εχθρών – Idle

Στην κατάσταση Chase το μόνο που κάνουμε είναι να ορίζουμε συνεχώς στον εχθρό τον προορισμό για τον χαρακτήρα μας. Εδώ γίνεται χρήση του NavMeshAgent που θα αναλύσουμε παρακάτω (βλ. NavMesh). Εφόσον έρθει σε πολύ κοντινή απόσταση, τότε επιλέγει τυχαία μία από τις δύο επιθέσεις που έχει και μπαίνει σε κατάσταση Attack.

```
override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    //τρεχει προς τον παιχτη
    agent.SetDestination(player.position);
    float distance = Vector3.Distance(player.position, animator.transform.position);
    //αν ειναι κοντα στον παιχτη επιτιθεται
    if (distance <= attackDistance)
    {
        animator.SetBool("isChasing", false);

        int randomChoice = Random.Range(0, 2);
        if (randomChoice == 0)
        {
            animator.SetBool("isAttacking", true);
            animator.SetBool("isAttacking1", false);
        }
        else
        {
            animator.SetBool("isAttacking", false);
            animator.SetBool("isAttacking1", true);
        }
    }
}
```

Εικόνα 88: Κατάσταση Εχθρών – Chase

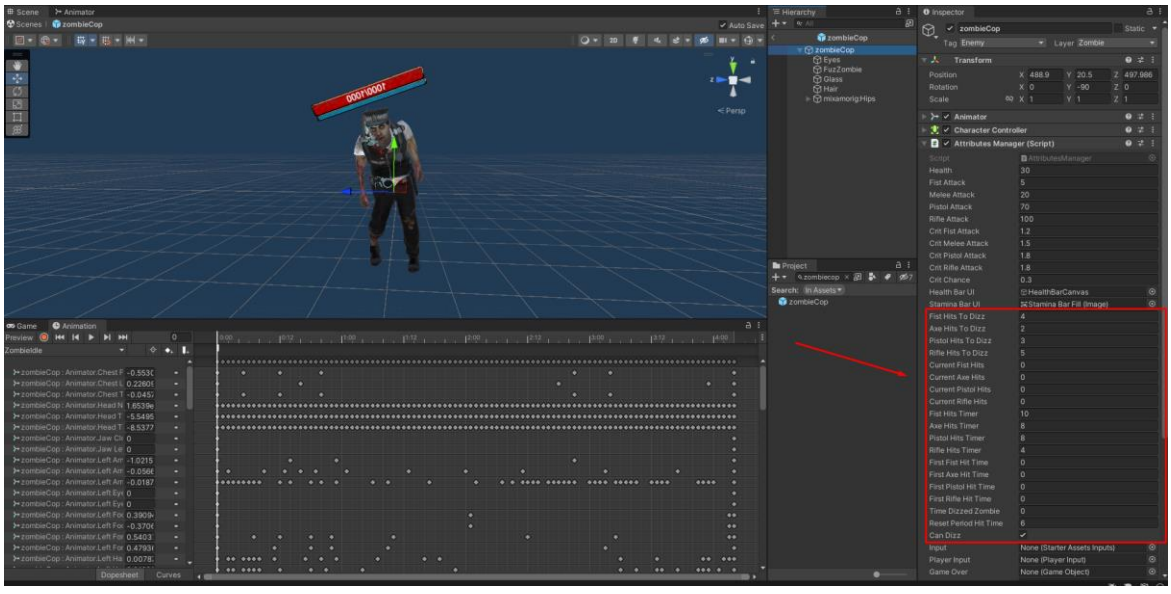
Στην κατάσταση Attack γυρνάμε τον εχθρό προς το σημείο του παίκτη ώστε να επιτεθεί προς αυτόν και ξαναελέγχουμε την απόσταση. Αν ο παίκτης παρέμεινε κοντά,

παίρνουμε μία τυχαία πάλι επίθεση και την εκτελούμε, αλλιώς αν έφυγε πιο μακριά, γυρνάει πίσω στην κατάσταση Chase.

```
override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
    //δεν αλλαζω το y ωστε να γυρναει μονο αριστερα-δεξια
    Vector3 playerPosition = new(player.transform.position.x, animator.transform.position.y, player.transform.position.z);
    //κοιταει προς τον παιχτη οταν επιτιθεται
    animator.transform.LookAt(playerPosition);
    float distance = Vector3.Distance(player.position, animator.transform.position);
    //αν φυγει πιο μακρια σταματαει να επιτιθεται και τον κυνηγαει
    if (distance >= attackDistance)
    {
        animator.SetBool("isChasing", true);
        animator.SetBool("isAttacking", false);
        animator.SetBool("isAttacking1", false);
    }
    //αν ειναι ακομα κοντα ισως αλλαξει attack animation
    else
    {
        int randomChoice = Random.Range(0, 2);
        if (randomChoice == 0)
        {
            animator.SetBool("isAttacking", true);
            animator.SetBool("isAttacking1", false);
        }
        else
        {
            animator.SetBool("isAttacking", false);
            animator.SetBool("isAttacking1", true);
        }
    }
}
```

Εικόνα 89: Κατάσταση Εχθρών – Attack

Η λογική για την ακινητοποίηση των εχθρών δεν είναι σε StateMachineBehaviour script όπως τα υπόλοιπα, αντίθετα υλοποιείται μέσω στο AttributesManager για ευκολία. Κάθε εχθρός διαφέρει όσων αφορά τα πόσα χτυπήματα χρειάζεται με κάθε όπλο και μετά από πόσο χρόνο κάνει restart ώστε να μπορεί να ακινητοποιηθεί ξανά. Αυτές οι μεταβλητές έχουν οριστεί όλες ως SerializeFields ώστε να τις χειριζόμαστε μέσω του Editor για κάθε prefab εχθρού ξεχωριστά.



Εικόνα 90: Παράδειγμα Μεταβλητών Για Ακινητοποίηση Εχθρού

Όπως φαίνεται, υπάρχουν διαφορετικές τιμές για μπουνιές, τσεκούρι, πιστόλι ή αυτόματο όπως και διαφορετικά χρονόμετρα που καθορίζουν το διάστημα που έχει ο παίκτης να πετύχει τα απαιτούμενα χτυπήματα για να ακινητοποιήσει τον στόχο. Στο κομμάτι του κώδικα, έχουμε μια μεταβλητή (για κάθε όπλο) που ανεβαίνει κατά ένα για κάθε χτύπημα που δέχεται. Αν αυτό φτάσει τα χτυπήματα που απαιτούνται τότε ξεκινάει το Dizz Animation αλλιώς αν περάσει ο χρόνος χάνονται τα προηγούμενα χτυπήματα και ξεκινάει από την αρχή. Παρακάτω φαίνεται η λογική για τις μπουνιές, αντίστοιχα είναι και για τα άλλα όπλα.

```
if (canDizz)
{
    UpdateDizzBar(weaponTag);
    //τσεκαρω αν εφτασε το απαραιτητα χτυπηματα
    if (currentFistHits >= fistHitsToDizz
    || currentAxeHits >= axeHitsToDizz
    || currentPistolHits >= pistolHitsToDizz
    || currentRifleHits >= rifleHitsToDizz)
    {
        canDizz = false;
        StartDizzyAnimation();
        //κανω το timer για το reset ισο με τον πραγματικο χρονο ωστε να
        // μπορω να κανω dizz μετα απο τον χρονο που χρειαζεται ο εχθρος
        timeDizzedZombie = Time.time;
    }
}
```

Εικόνα 91: Λογική Κώδικα Για Την Ακινητοποίηση Εχθρού (Στιγμιότυπο 1)

```
void CheckForDizz(string weaponTag)
{
    switch (weaponTag)
    {
        case "LeftFist":
        case "RightFist":
            //αν ειναι το πρωτο hit στο διαστημα που εχω να χτυπησω,
            // κανω το χρονο του πρωτου χτυπηματος ισο με τον πραγματικο χρονο
            if (currentFistHits == 0)
            {
                firstFistHitTime = Time.time;
            }
            //στο πρωτο hit μπαινει παντα στην if
            //αν περασει ο χρονος που εχω για μπουνιες(10 δευτερολεπτα) τοτε θα
            //μπει στην else και θα ξαναρχισει το timer και θα αυξησει τα hit κατα 1
            if (Time.time - firstFistHitTime <= FistHitsTimer)
            {
                currentFistHits++;
            }
            else
            {
                firstFistHitTime = Time.time;
                currentFistHits = 1;
            }
            break;
```

Εικόνα 92: Λογική Κώδικα Για Την Ακινητοποίηση Εχθρού (Στιγμιότυπο 2)

4.2.2.3 Ζημιά Των Εχθρών

Οι εχθροί προκαλούν ζημιά στον παίκτη με το collider που έχουν στο δεξί τους χέρι. Το Collider αυτό ενεργοποιείται - απενεργοποιείται μέσω Animation Event όπως και στον παίκτη μας για τον ίδιο λόγο. Η λογική του κώδικα είναι πολύ απλή καθώς απλά τσεκάρουμε για triggers μεταξύ των colliders του παίκτη με το χέρι του ζόμπι. Αν αυτό που χτυπήσει έχει το tag Player σημαίνει ότι χτύπησε τον παίκτη και έτσι του προκαλεί ζημιά καλώντας την DealDamage που αναλύσαμε σε προηγούμενο κεφάλαιο.

```
1  private void Awake() {
2      EnableHandCollider(0);
3  }
4
5  //για χρηση animation event, ενεργοποιει-απενεργοποιει το collider του χεριου
6  public void EnableHandCollider(int isEnable)
7  {
8      if (handCollider != null)
9      {
10             if (isEnable == 1)
11             {
12                 handCollider.enabled = true;
13             }
14             else
15             {
16                 handCollider.enabled = false;
17             }
18     }
19 }
```

Εικόνα 93: Συνάρτηση HandCollider Για Εχθρό

```
1  //αν κανει trigger στον παιχτη του κανει ζημια και δημιουργει ενα impulse στην καμερα
2  private void OnTriggerEnter(Collider other)
3  {
4      //εχουμε timer 0.5sec για να μην κανει πολλαπλα χτυπηματα μεσα σε μια κινηση
5      if (other.tag == "Player" && Time.time - lastHitTime >= 0.5f)
6      {
7          lastHitTime = Time.time;
8          impulse.GenerateImpulse(1);
9          atm.DealDamage(other.gameObject, this.tag, Vector3.zero);
10     }
11 }
```

Εικόνα 94: Λογική Κώδικα Για ζημιά Στον Παίκτη

4.2.2.4 Ragdoll

Στην Unity, ένα ragdoll αναφέρεται σε ένα σύστημα που καθοδηγείται από τη φυσική που προσομοιώνει τη συμπεριφορά των άκρων και των αρθρώσεων ενός χαρακτήρα ή αντικειμένου, επιτρέποντάς του να αντιδρά δυναμικά σε εξωτερικές δυνάμεις όπως η βαρύτητα, οι συγκρούσεις και άλλες φυσικές αλληλεπιδράσεις. Ουσιαστικά, ένα ragdoll είναι ένας τρόπος να μιμηθείς τη ρεαλιστική κίνηση και απόκριση ενός χαρακτήρα ή αντικειμένου όταν επηρεάζεται από τους νόμους της φυσικής.

Η μηχανή Unity παρέχει ένα ενσωματωμένο σύστημα για τη δημιουργία ragdolls, διευκολύνοντας τους προγραμματιστές να εφαρμόσουν ρεαλιστικά Animations και αντιδράσεις. Για να φτιάξουμε ragdoll σε κάποιον εχθρό, ορίζουμε σωστά τα μέρη του σώματος και η Unity κάνει για μας τα υπόλοιπα, βάζοντας τα colliders και τα joints εκεί που χρειάζονται. Επίσης, κάθε μέρος αποκτάει ένα rigidbody στοιχείο για να αντιδρά με την φυσική. Όταν ξεκινάει το παιχνίδι, για κάθε εχθρό που δημιουργούμε, απενεργοποιούμε τα ragdolls καθώς θέλουμε να ενεργοποιηθούν μόνο όταν πεθαίνει. Αυτό σημαίνει ότι ορίζουμε τα rigidbody σε kinematic οπότε δεν αντιδράνε με τις φυσικές δυνάμεις και συγκρούσεις, μπορούν μόνο να μετακινηθούν μέσω κώδικα. Όταν δώσουμε το τελειωτικό χτύπημα σε έναν εχθρό, τότε ενεργοποιείται το ragdoll (kinematics = false) και ο χαρακτήρας πέφτει στο έδαφος ρεαλιστικά σαν να του ασκείται βαρύτητα. Παράλληλα, ασκείται μια μικρή δύναμη στο σημείο που χτυπήθηκε για να ‘πέσει’ προς αυτό το μέρος όπως θα γινόταν και στο πραγματικό κόσμο. Έτσι, πετυχαίνουμε ένα αρκετά όμορφο αποτέλεσμα δίνοντας την αίσθηση ενός σωστά προσεγμένου παιχνιδιού.



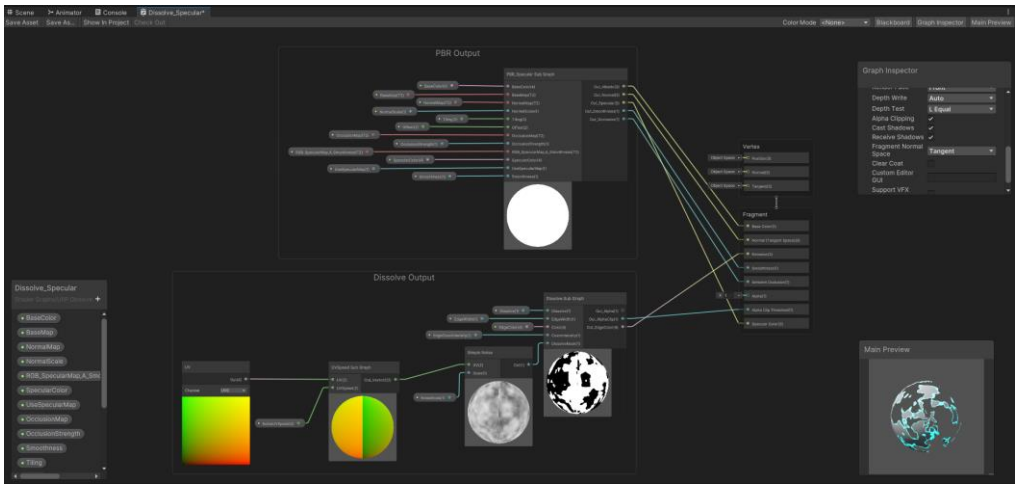
Εικόνα 95: Πρώτο Στάδιο Ragdoll - Άσκηση Δύναμης



Εικόνα 96: Δεύτερο Στάδιο Ragdoll - Επαφή Με Έδαφος

4.2.2.5 Εφέ διάλυσης (Dissolve Effect)

Όπως στα περισσότερα παιχνίδια, οι εχθροί που πεθαίνουν μετά από λίγο εξαφανίζονται για λόγους τόσο οπτικούς όσο και optimization. Για τον λόγο αυτόν, εφάρμοσα αυτήν την τεχνική και στο δικό μου παιχνίδι. Καθώς όμως δεν δίνει ωραία αίσθηση απλά να εξαφανίζεται ο εχθρός, προστέθηκε ένα εφέ διάλυσης το οποίο μέσω materials – shaders δίνει την αίσθηση ότι ο εχθρός 'διαλύεται'. Έπειτα αφού εξαφανιστεί εντελώς, καταστρέφουμε το gameobject ώστε να μην τρώει χώρο στην μνήμη. Για το εφέ αυτό, χρησιμοποιήθηκε ένας shader [asset 5] από το Unity Asset Store που είναι ιδανικός για URP. Αφού τον ορίσαμε ως shader για τα materials που έχουν οι εχθροί, παίρνουμε την τιμή Dissolve που έχουμε κάνει expose στο material και αλλάζουμε την τιμή ομαλά σε 1. Όταν γίνει αυτό, καταστρέφουμε το gameobject. Το αποτέλεσμα που πετυχαίνουμε είναι πολύ πιο εντυπωσιακό σε σχέση με τον προηγούμενο τρόπο που απλά καταστρέφαμε απευθείας τον εχθρό.



Εικόνα 97: Dissolve Shader



Εικόνα 98: Παράδειγμα Dissolve Στο Παιχνίδι

4.2.2.6 Οπτικό Εφέ Αίματος (Blood VFX)

Τα οπτικά εφέ στα παιχνίδια υπάρχουν για διάφορους λόγους όπως ρεαλισμό και δημιουργία ατμόσφαιρας. Καθώς στο παιχνίδι δόθηκε ιδιαίτερη προσοχή στον ρεαλισμό που προσδίδει, μια καλή προσθήκη υπήρξε το εφέ αίματος στους εχθρούς. Εδώ χρησιμοποίησα ένα μέρος από το δωρεάν asset που βρήκα στο Unity Asset Store [asset 6]. Καθώς έχει δημιουργηθεί στο Unity Particle System, όπου είναι το σύστημα στο οποίο η Unity σου επιτρέπει να φτιάξεις τα δικά σου εφέ μέσα στον Editor, το τροποποίησα ως προς τις ανάγκες του παιχνιδιού και το έθεσα ως prefab για να το εμφανίζω κάθε φορά που το χρειάζομαι. Στην εικόνα 63 βλέπουμε πως δημιουργείται και εμφανίζεται στον κώδικα το Blood VFX στο σημείο που δέχτηκε ζημιά ο εχθρός.



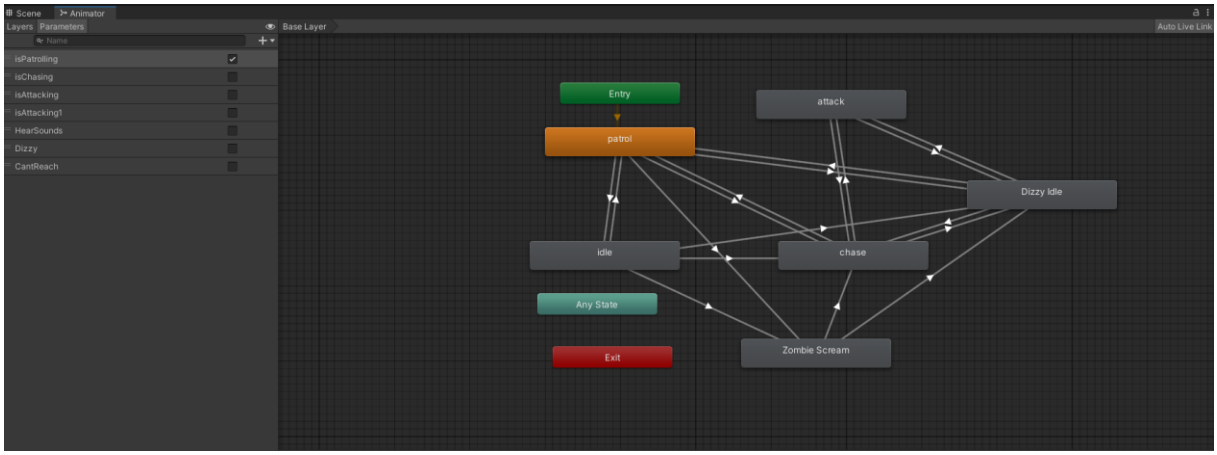
Εικόνα 99: Παράδειγμα Εφέ Αίματος

4.2.2.7 Ειδικός τύπος ζόμπι (Crawler Zombie)

Γενικά στο παιχνίδι υπάρχουν 4 είδη ζόμπι. Τα 3 από αυτά έχουν παρόμοιες λειτουργίες καθώς αλλάζει μόνο η ζημιά και το μέγεθός τους. Παρ' όλ' αυτά έχουν ίδια animation για επιθέσεις, τρέξιμο και περπάτημα. Για να υπάρχει ποικιλία, δημιουργήθηκε ένα ζόμπι το οποίο έχει διαφορετική μορφή τρεξίματος, κάτι που το κάνει πολύ πιο γρήγορο από τα άλλα, και έτσι μπορεί να αιφνιδιάσει τον παίκτη. Ακολουθεί παρόμοιο State Machine με μόνη διαφορά ότι έχει μόνο μια επίθεση η οποία γίνεται αρκετά γρήγορα ώστε να είναι πιο δύσκολο να την αποφύγει ο παίκτης.



Εικόνα 100: Παράδειγμα Crawler Zombie



Εικόνα 101: Μηχανή Καταστάσεων Crawler Zombie (State Machine)

5. Λοιπά Στοιχεία Παιχνιδιού

5.1 Διαχείριση δημιουργίας αντικειμένων – εχθρών (Spawn Manager)

Ο σκοπός του παιχνιδιού είναι ο παίκτης να επιβιώσει όσα περισσότερα κύματα εχθρών καταφέρει ώστε να πετύχει το καλύτερο σκορ. Για τον λόγο αυτό δημιουργήθηκε ένα script που υλοποιεί την δημιουργία, τη διαχείριση και την εμφάνιση κυμάτων ζόμπι αλλά και άλλων στοιχείων όπως όπλα, σφαίρες και ζωή που θα τον βοηθήσουν να επιβιώσει το περισσότερο δυνατόν. Τα αρχικά κύματα είναι αρκετά αδύναμα γι' αυτό και ο παίκτης ξεκινάει χωρίς όπλα ή κάποια επιπλέον βοήθεια. Όσο προχωράει στο παιχνίδι, περισσότερα και πιο δυνατά ζόμπι εμφανίζονται αλλά παράλληλα παίρνει όπλα που θα βοηθήσουν να τα αντιμετωπίσει. Στα πρώτα επίπεδα, ο αριθμός και η δύναμη των εχθρών είναι συγκεκριμένη ενώ μετά το 10ο κύμα δημιουργούνται τυχαίοι αριθμοί από ζόμπι. Επιπλέον, κάθε 5ο κύμα εμφανίζεται και ένας αρκετά πιο δυνατός εχθρός που κάνει το παιχνίδι πιο απαιτητικό. Παρακάτω γίνεται επεξήγηση του κώδικα που υλοποιήθηκε.

Στην αρχή, με το που κλείσουμε τις οδηγίες του παιχνιδιού, τρέχει η συνάρτηση StartFadeStartingText στην οποία εμφανίζεται για κάποια δευτερόλεπτα το κείμενο που ενημερώνει τον παίκτη για τον σκοπό του παιχνιδιού.



Εικόνα 102: Ξεκίνημα Παιχνιδιού (Στιγμιότυπο 1)

```
1  //τρεχει με το που κλεισουν τα instructions
2  public void StartFadeStartingText()
3  {
4      //μετα απο 4 δευτερολεπτα κανει fade το αρχικο κειμενο
5      StartCoroutine(FadeText(4.0f, startingWaveText, 2.0f));
6      //ξεκιναει τα waves απο το 1 που εχει αρχικοποιηθει
7      Invoke("ShowWaveText", 6.0f);
8  }
```

Εικόνα 103: Λογική Κώδικα SpawnManager (Στιγμιότυπο 1)

Έπειτα τρέχει η συνάρτηση ShowWaveText στην οποία εμφανίζεται κείμενο με το κύμα που είμαστε (στην περίπτωση μας 1 καθώς είμαστε στην αρχή), το οποίο μένει για τρία δευτερόλεπτα και μετά ξεκινάει το παιχνίδι καλώντας την συνάρτηση WaveHandling.



Εικόνα 104: Ξεκίνημα Παιχνιδιού (Στιγμιότυπο 2)

```
//δειχνει σε ποιο κυμα ειμαστε, το κανει fade out και ξεκιναει το waveHandling
void ShowWaveText()
{
    //επαναφερω το alpha(transparent) value για το επομενο wave
    waveNumberText.color = new Color(waveNumberText.color.r, waveNumberText.color.g, waveNumberText.color.b, 1f);
    waveNumberText.text = $"ΚΥΜΑ {currentWave}";
    waveNumberText.gameObject.SetActive(true);
    StartCoroutine(FadeText(3.0f, waveNumberText, 1.0f));
    Invoke("WaveHandling", 4.0f);
}
```

Εικόνα 105: Λογική Κώδικα SpawnManager (Στιγμιότυπο 2)

Στην WaveHandling είναι όλη η λογική για το πώς δημιουργούνται εχθροί και αντικείμενα ανάλογα το κύμα που βρισκόμαστε [παράρτημα 9]. Εκεί μέσα τρέχουν τέσσερις συναρτήσεις που είναι υπεύθυνες για την δημιουργία εχθρών, όπλων, σφαιρών και ζωής. Ας δούμε αναλυτικά την κάθε μια από αυτές:

SpawnZombie: Υπεύθυνη για την δημιουργία κάθε είδους εχθρού. Παίρνει ως όρισμα το τύπο εχθρού που θέλουμε να δημιουργήσει, το multiplier που ορίζει την δύναμη του εχθρού καθώς και έναν αριθμό που ορίζει τον αριθμό αυτού του είδους που θα δημιουργηθεί. Χρησιμοποιώ IEnumerator συνάρτηση ώστε να τα δημιουργήσω με μια μικρή καθυστέρηση κάθε φορά, κυρίως για λόγους optimization. Ο κάθε εχθρός δημιουργείται σε μια διαφορετική τυχαία θέση από αυτές που έχουμε ορίσει.

```
//χρησιμοποιω IEnumerator για να βαλω μια μικρη καθυστερηση 0.2 δευτερολεπτα πριν καθε spawn για να μην βγαινουν ολα μαζι
IEnumerator SpawnZombie(float multiplier, GameObject zombiePrefab, int numberOfSpawns)
{
    enemiesRemaining += numberOfSpawns;

    availableSpawnPoints.Clear();
    availableSpawnPoints.AddRange(spawnPoints);
    for (int i = 0; i < numberOfSpawns; i++)
    {
        if (availableSpawnPoints.Count > 0)
        {
            int randomIndex = Random.Range(0, availableSpawnPoints.Count);
            Transform spawnPoint = availableSpawnPoints[randomIndex];

            GameObject clone = Instantiate(zombiePrefab, spawnPoint.position, Quaternion.identity);
            clone.GetComponent<AttributesManager>().BuffZombies(multiplier);
            availableSpawnPoints.RemoveAt(randomIndex);
        }
        else
        {
            availableSpawnPoints.AddRange(spawnPoints);
        }
        yield return new WaitForSeconds(0.2f);
    }
}
```

Εικόνα 106: Λογική Κώδικα SpawnManager (Στιγμιότυπο 3)

SpawnWeapon: Υπεύθυνη για την δημιουργία κάθε είδους όπλου. Παίρνει ως όρισμα τον τύπο όπλου που θέλουμε να δημιουργήσει και έναν float που ενημερώνει σε πόσο χρόνο πρέπει να καταστραφεί αν ο παίκτης δεν προλάβει να το πάρει. Εμφανίζεται παρομοίως σε τυχαίο σημείο στον χάρτη. Ένα πρόβλημα που υπήρχε εδώ ήταν όταν ο παίκτης πάταγε να πιάσει το όπλο και εκείνη την στιγμή καταστρεφόταν. Αυτό λύθηκε καλώντας μια άλλη συνάρτηση DestroyWeapon η οποία καταστρέφει το όπλο μόνο αν το Boolean ShouldDestroyWeapon είναι true. Έτσι, αν πατήσω να πιάσει το όπλο, αλλάζοντας την τιμή σε false διορθώθηκε εύκολα το πρόβλημα.

```
void SpawnWeapon(GameObject weapon, float destroyTime)
{
    shouldDestroyWeapon = true;
    int randomIndex = Random.Range(0, spawnPoints.Length);
    Transform spawnPoint = spawnPoints[randomIndex];
    GameObject weaponClone = Instantiate(weapon, spawnPoint.position, weapon.transform.rotation);
    //αν πατησω να το πιασει τοτε δεν θα διαγραφει μετα απο λιγο χρονο γιατι πλεον θα το εχω παρει
    StartCoroutine(DestroyWeapon(weaponClone, destroyTime));
}
IEnumerator DestroyWeapon(GameObject weapon, float timeToDestroy)
{
    yield return new WaitForSeconds(timeToDestroy);
    if (shouldDestroyWeapon)
    {
        Destroy(weapon);
    }
}
```

Εικόνα 107: Λογική Κώδικα SpawnManager (Στιγμιότυπο 4)

StartAmmoInterval: Υπεύθυνη για την δημιουργία σφαιρών. Η συγκεκριμένη είναι μια αναδρομική συνάρτηση, δηλαδή την καλούμε μόνο μια φορά και μετά καλεί τον εαυτό της ξανά σε τυχαίο χρόνο από 160 έως 200 δευτερόλεπτα. Για να γίνει αυτό χρησιμοποιήθηκε μια συνάρτηση που μας παρέχει η C#, η Invoke, στην οποία δίνουμε ως όρισμα την συνάρτηση που θέλουμε να καλεστεί, καθώς και ένα float που καθορίζει μετά από πόσο χρόνο θα καλεστεί. Υπήρχε το ίδιο σφάλμα με τα όπλα οπότε χρησιμοποιήθηκε η ίδια τεχνική για να το διορθώσουμε.

```
//καθε 160 με 200 δευτερολεπτα(μετα που καταστραφει ή παρω το αντικειμενο) θα κανει
// spawn σφαιρες που θα μενουν ενεργες για 120 δευτερολεπτα ωστε να παρει ο παιχτης
void StartAmmoInterval()
{
    shouldDestroyAmmo = true;
    float randomDelay = Random.Range(160f, 200f);
    int randomIndex = Random.Range(0, spawnPoints.Length);
    Transform spawnPoint = spawnPoints[randomIndex];
    GameObject ammoClone = Instantiate(ammoBox, spawnPoint.position, Quaternion.identity);
    //αν πατησω να το πιασει τοτε δεν θα διαγραφει μετα απο λιγο χρονο γιατι πλεον θα το εχω παρει
    StartCoroutine(DestroyAmmo(ammoClone, 120f));

    Invoke("StartAmmoInterval", randomDelay);
}

//bug-fix. Αν πατησω να παρει σφαιρες ή οπλο και οσο πηγαινε καταστραφουν υπηρχε bug οποτε με μια
//απλη bool που γινεται false αν πατησω να τα παρει τοτε θα τα παρει και δεν θα τα καταστρεψει
IEnumerator DestroyAmmo(GameObject ammoClone, float timeToDestroy)
{
    yield return new WaitForSeconds(timeToDestroy);

    if (shouldDestroyAmmo)
    {
        Destroy(ammoClone);
    }
}
```

Εικόνα 108: Λογική Κώδικα SpawnManager (Στιγμιότυπο 5)

StartHealthInterval: Ίδια ακριβώς λογική με τις σφαίρες, είναι μια αναδρομική συνάρτηση υπεύθυνη για την δημιουργία ζωής σε τακτά χρονικά διαστήματα. Εδώ, εφόσον ο παίκτης πρέπει να περάσει μέσα από την Gameobject και όχι να το πιάσει, δεν είχαμε το πρόβλημα οπότε καταστρέφεται αμέσως αν ο παίκτης δεν προλάβει να το πιάσει μέσα στο χρόνο που του δίνεται.

```
//καθε 160 με 200 δευτερολεπτα(μετα που καταστραφει ή παρω το αντικειμενο)
// θα κανει spawn ζωη που θα μενει ενεργη για 120 δευτερολεπτα ωστε να παρει ο παιχτης
void StartHealthInterval()
{
    float randomDelay = Random.Range(160f, 200f);
    int randomIndex = Random.Range(0, spawnPoints.Length);
    Vector3 spawnPoint = spawnPoints[randomIndex].position;
    //απλα για να κανει spawn λιγο πιο πανω
    spawnPoint += Vector3.up;
    GameObject healthClone = Instantiate(health, spawnPoint, Quaternion.identity);
    Destroy(healthClone, 120f);

    Invoke("StartHealthInterval", randomDelay);
}
```

Εικόνα 109: Λογική Κώδικα SpawnManager (Στιγμιότυπο 6)

Για να συνεχίσουμε στο επόμενο κύμα και να τρέξουν πάλι οι συναρτήσεις ώστε να εμφανιστούν οι επόμενοι εχθροί, πρέπει να υπάρχει μια μεταβλητή που θα αποθηκεύεται ο αριθμός των εχθρών που είναι ζωντανοί. Όταν ο αριθμός αυτός μηδενιστεί, προχωράμε στο επόμενο κύμα. Εδώ έχουμε μια πολύ απλή συνάρτηση που καλείται κάθε φορά που σκοτώνεται κάποιος εχθρός και τσεκάρει πότε ο αριθμός μηδενιστεί. Όταν γίνει αυτό, ξανατρέχει η ShowWaveText και συνεχίζει για όσο ο παίκτης είναι ακόμα ζωντανός.

```
1  public void KillOneEnemy()
2  {
3      enemiesRemaining--;
4      if (enemiesRemaining <= 0)
5      {
6          currentWave++;
7          ShowWaveText();
8      }
9  }
```

Εικόνα 110: Λογική Κώδικα SpawnManager (Στιγμιότυπο 7)

5.2 NavMesh

Το NavMesh (Navigation Mesh) είναι ένα σύστημα που χρησιμοποιείται συχνά σε παιχνίδια και εφαρμογές για τη δημιουργία μονοπατιών και την πλοήγηση των χαρακτήρων στον χώρο. Η βάση ενός NavMesh είναι ένα 3D mesh (πλέγμα), το οποίο καθορίζει τις περιοχές του χώρου που είναι προσβάσιμες ή μη προσβάσιμες από τους χαρακτήρες.

Το NavMeshAgent είναι ένα component του NavMesh που αντιπροσωπεύει έναν χαρακτήρα που μπορεί να κινείται σε ένα περιορισμένο χώρο, ακολουθώντας το μονοπάτι πάνω στο NavMesh. Το NavMeshAgent χρησιμοποιεί αλγόριθμους πλοήγησης για να υπολογίσει το καλύτερο μονοπάτι προς μια συγκεκριμένη θέση στον χώρο, λαμβάνοντας υπόψη τα εμπόδια που καθορίζονται από το NavMesh.

Η χρήση του NavMesh και του NavMeshAgent μπορεί να είναι ιδιαίτερα χρήσιμη σε παιχνίδια όπου οι χαρακτήρες πρέπει να ακολουθούν περίπλοκα μονοπάτια, να αποφεύγουν εμπόδια και να αντιδρούν έξυπνα στο περιβάλλον τους. Το συγκεκριμένο πακέτο της Unity ταίριαζε με τις ανάγκες του παιχνιδιού, καθώς θέλουμε οι εχθροί να κυνηγάνε τον παίκτη από την στιγμή που τον εντοπίσουν. Για να υλοποιηθεί, αρχικά δημιουργήθηκε το πλέγμα του NavMesh στον χάρτη. Για να γίνει αυτό, ορίζουμε τρία βασικά στοιχεία.

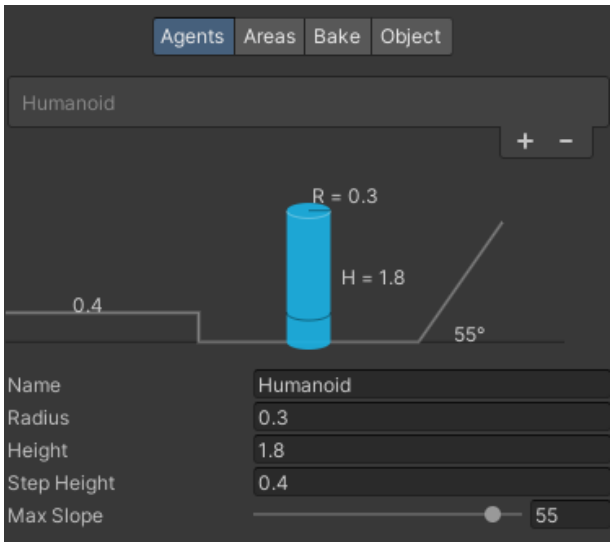
Αρχικά, δίνουμε τα χαρακτηριστικά του Agent που θα χρησιμοποιεί αυτό το πλέγμα. Για ευκολία χρησιμοποιήσαμε ίδιες ρυθμίσεις για όλους τους εχθρούς. Τα χαρακτηριστικά αυτά είναι η ακτίνα του χαρακτήρα, το ύψος του, το ύψος που μπορεί να θεωρήσει κάτι ως

σκαλοπάτι ώστε να το ανέβει και τέλος το max slope που αναφέρεται στη μέγιστη κλίση (σε μοίρες) που ένας ‘πράκτορας’ μπορεί να ανεβεί (εικόνα 115).

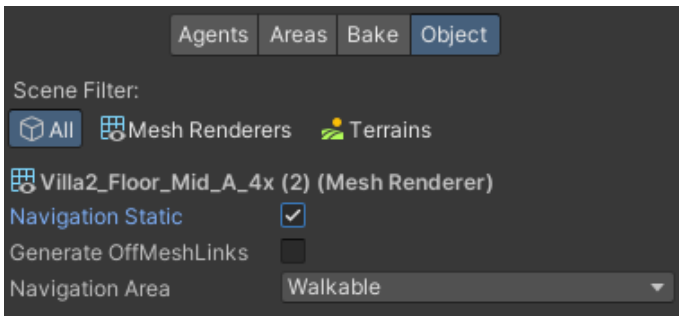
Το δεύτερο κομμάτι είναι να ορίσουμε τον ρόλο του κάθε αντικείμενου στον χώρο αυτό ως walkable ή not walkable ώστε να ξέρει ο ‘πράκτορας’ αν μπορεί να περπατήσει πάνω σε αυτό. Επίσης υπάρχει επιλογή ώστε να δημιουργηθούν αυτόματα συνδέσεις μεταξύ σημείων του NavMesh ώστε ο πράκτορας να μπορεί να κάνει άλμα μεταξύ δύο σημείων. Μπορούμε να βάλουμε τέτοια σημεία και χειροκίνητα αν η αυτοματοποιημένη λύση δεν μας καλύπτει. Τέλος υπάρχει επιλογή Navigation Static η οποία στην ουσία κάνει το Gameobject να συμμετέχει στην δημιουργία του πλέγματος αν είναι ενεργοποιημένο (εικόνα 116).

Το τρίτο κομμάτι αναφέρεται στις διαφορετικές περιοχές που μπορούμε να ορίσουμε για διαφορετική χρήση των agents. Από default επιλογή υπάρχουν τρεις περιοχές (walkable, not walkable, jump) οι οποίες μας ήταν ιδανικές για την χρήση του NavMesh στο συγκεκριμένο παιχνίδι (εικόνα 117).

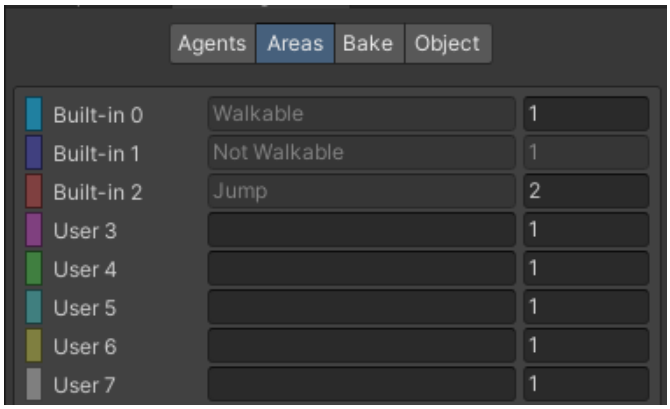
Αφού έχουμε ορίσει όλα τα προηγούμενα, μπορούμε πριν δημιουργήσουμε το πλέγμα (bake) να ορίσουμε και τις παραμέτρους drop height και jump distance που αναφέρονται στο μέγιστο ύψος πτώσης και στο μέγιστο μήκος άλματος που μπορεί να καλύψει ένας πράκτορας (εικόνα 118). Τέλος, κάνουμε bake και δημιουργείται το πλέγμα στον χάρτη που μπορούμε να χρησιμοποιήσουμε μέσω NavMeshAgents που θα αναλύσουμε παρακάτω.



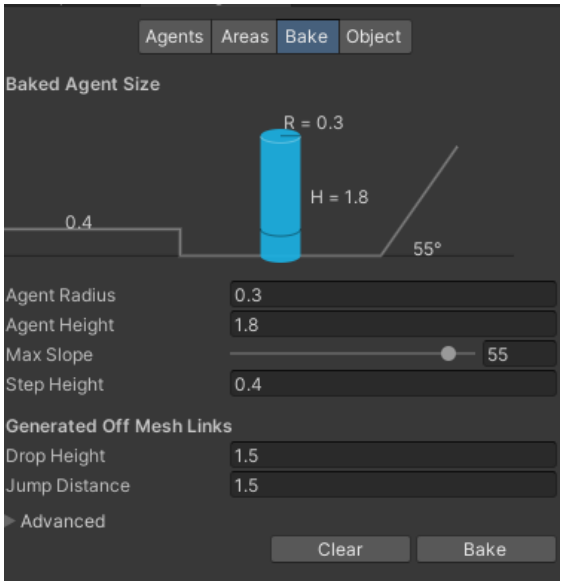
Εικόνα 111: NavMesh - Ρυθμίσεις Agents



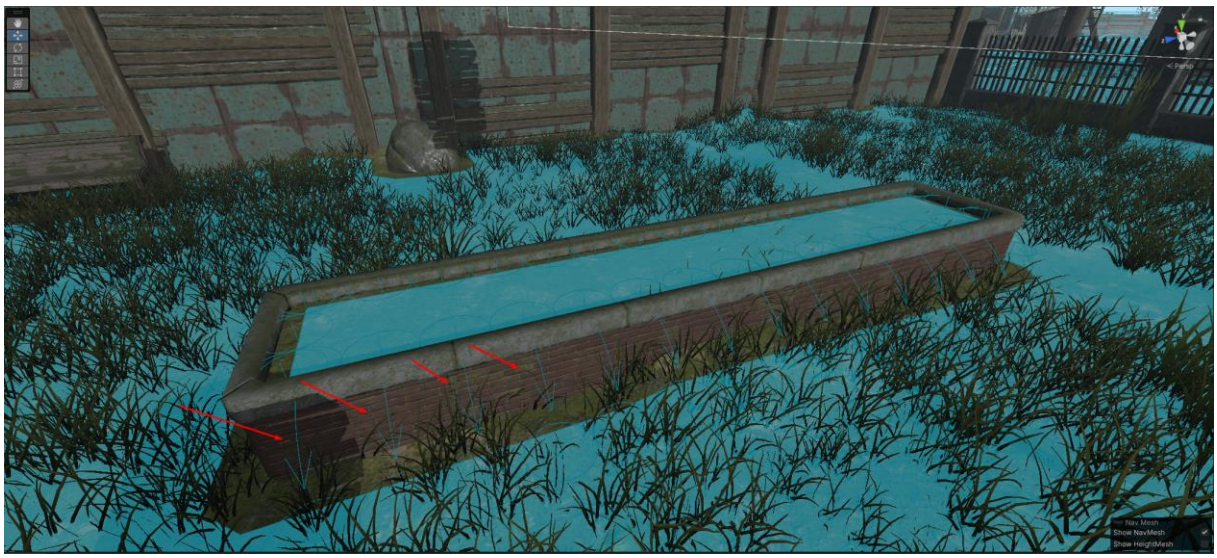
Εικόνα 112: NavMesh - Ρυθμίσεις Object



Εικόνα 113: NavMesh - Ρυθμίσεις Areas

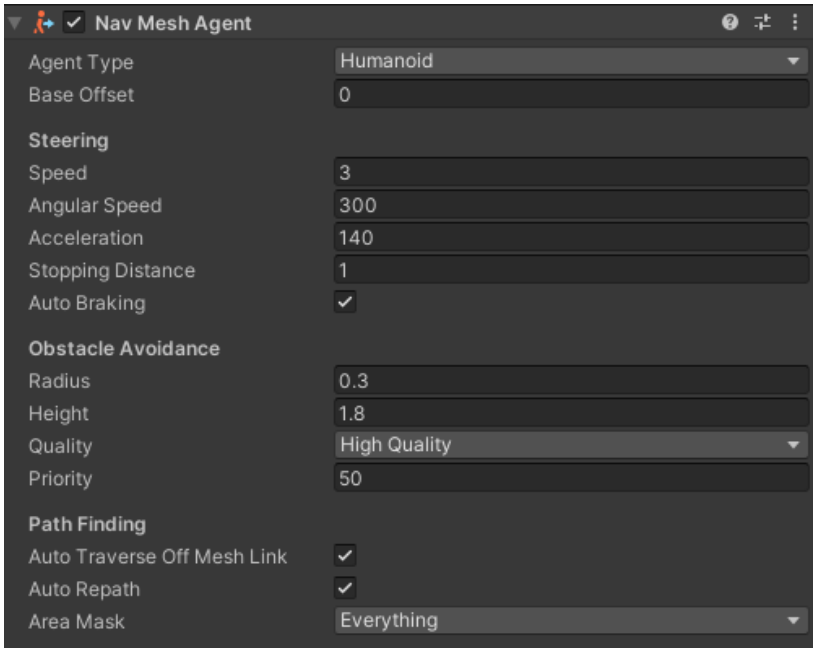


Εικόνα 114: NavMesh - Ρυθμίσεις Bake



Εικόνα 115: NavMesh - Παράδειγμα OffMeshLinks

Αφού έχουμε δημιουργήσει το πλέγμα, επόμενο βήμα είναι να προσθέσουμε το component NavMeshAgent σε κάθε είδος εχθρού που έχουμε. Αφού ορίσουμε για τον καθένα τις παραμέτρους που θέλουμε να έχει όπως ταχύτητα, περιστροφή, επιτάχυνση, απόσταση σταματήματος, προτεραιότητα κτλ. είμαστε έτοιμοι να το χρησιμοποιήσουμε μέσω κώδικα όπως δείξαμε σε παραπάνω ενότητα (βλ. State Machine).



Εικόνα 116: NavMeshAgent Component

5.3 Λεπτομέρειες στα όπλα

5.3.1 Κάλυκες

Στην πτυχιακή μου εργασία, ενσωμάτωσα μια δυναμική και ρεαλιστική προσέγγιση για τη ρίψη των καλύκων των όπλων. Αντί να χρησιμοποιήσω σταθερές ή προκαθορισμένες τροχιές, επέλεξα να χρησιμοποιήσω το στοιχείο της τυχαιότητας για να προσθέσω ρεαλισμό στην κίνηση των καλύκων.

Κάθε φορά που ο παίκτης πυροβολάει με το πιστολάκι ή το αυτόματο, δημιουργώ δυναμικά αντικείμενα-κάλυκες με χρήση της λειτουργίας Instantiate της Unity. Κατά τη συνέχεια, εφαρμόζω τυχαίες δυνάμεις και γωνίες σε αυτά τα αντικείμενα, εξομοιώνοντας την βαρύτητα, προκειμένου να αποδώσω ρεαλιστική συμπεριφορά κατά την πτώση τους. Για την υλοποίηση αυτή, έχω δημιουργήσει δύο μεθόδους που χρησιμοποιούνται σαν animation events ώστε να τρέξουν την κατάλληλη στιγμή που πρέπει να ‘βγει’ ο κάλυκας της σφαίρας από το όπλο. Αυτή η προσέγγιση προσφέρει μια εμπειρία gameplay πλούσια σε λεπτομέρειες, ενισχύοντας την αίσθηση της πραγματικότητας και της αυθεντικότητας για τον παίκτη.

```
//χρησιμοποιουνται για animation events ωστε να φευγουν οι καλυκες απο το οπλο οταν πυροβολαω
public void SpawnNewBullet()
{
    GameObject newBullet = Instantiate(bulletPrefab, transform.position, transform.rotation);
    rb = newBullet.GetComponent<Rigidbody>();
}
public void ExplodeBullet()
{
    float randomValue = Random.Range(-0.5f, 0.5f);
    rb.AddForce(rb.transform.TransformDirection(1 + randomValue, 1 + randomValue, randomValue), ForceMode.Impulse);
    rb.useGravity = true;
    Destroy(rb.gameObject, 5f);
}
```

Εικόνα 117: Λογική Κώδικα Για Κάλυκες Όπλων



Εικόνα 118: Παράδειγμα Παιχνιδιού Για Ρίψη Καλύκων

5.3.2 Γεμιστήρες

Όταν ο παίκτης οπλίζει, παίζει ένα animation στο οποίο βάζει στο όπλο ένα νέο γεμιστήρα αφού πρώτα πετάξει κάτω τον παλιό. Αντί απλά να εξαφανίζουμε το gameobject, την στιγμή που βγαίνει από το όπλο, το κάνουμε δυναμικό αντικείμενο το οποίο όπως, και οι σφαίρες, αντιδράει με την βαρύτητα και μένει στο πάτωμα ή όπου αλλού χτυπήσει για λίγο πριν το εξαφανίσουμε. Εδώ έχουμε τέσσερις διαφορετικές μεθόδους που λειτουργούν όλα σαν animation events. Η πρώτη απενεργοποιεί τον άδειο γεμιστήρα στο χέρι και δημιουργεί έναν ‘οπτικό’ ίδιο με αυτόν, η δεύτερη ενεργοποιεί την βαρύτητα στον ‘οπτικό’ γεμιστήρα και εφαρμόζει μια μικρή δύναμη προς τα πίσω ώστε να φαίνεται σαν να τον πετάει ο χαρακτήρας, η τρίτη δημιουργεί έναν γεμάτο ‘οπτικό’ γεμιστήρα στο χέρι του χαρακτήρα όταν φτάσει στην τσέπη του και η τελευταία όταν φτάσει στο σημείο που μπαίνει στο όπλο καταστρέφει τον ‘οπτικό’ γεμιστήρα και ενεργοποιεί ξανά τον πραγματικό γεμιστήρα του όπλου. Με αυτό το κόλπο πετυχαίνουμε ένα όμορφο και ρεαλιστικό αποτέλεσμα στην διαδικασία του reload. Η ίδια λογική ακολουθείται και στα 2 όπλα.

```
private void CatchRifleMagazine()
{
    actualMagazineRifle = GameObject.FindGameObjectWithTag("RifleMagazine");
    actualMagazineRifle.SetActive(false);
    emptyMagazineRifle = Instantiate(emptyRifleMagazine, parentEmptyRifleMagazine.transform);
}
private void DropRifleMagazine()
{
    Rigidbody rbRifleMagazine = emptyMagazineRifle.GetComponent<Rigidbody>();
    rbRifleMagazine.useGravity = true;
    rbRifleMagazine.transform.SetParent(null);

    rbRifleMagazine.AddForce(rbRifleMagazine.transform.TransformDirection(0.1f, -1, 0) * 2f, ForceMode.Impulse);
    Destroy(emptyMagazineRifle, 6.0f);
}
private void PutRifleMagazine()
{
    fullMagazineRifle = Instantiate(fullRifleMagazine, parentFullRifleMagazine.transform);
}
private void DestroyRifleMagazine()
{
    Destroy(fullMagazineRifle);
    actualMagazineRifle.SetActive(true);
}
```

Εικόνα 119: Λογική Κώδικα Για Γεμιστήρες Όπλων



Εικόνα 120: Παράδειγμα Γεμίσματος Όπλου (Στιγμιότυπο 1)



Εικόνα 121: Παράδειγμα Γεμίσματος Όπλου (Στιγμιότυπο 2)

5.3.3 Line Renderer και Particles για τα όπλα

Το Line Renderer είναι ένα συστατικό (component) στην Unity που χρησιμοποιείται για να απεικονίσει μια γραμμή ή μια σειρά συνεχόμενων γραμμών στο παιχνίδι. Συνήθως χρησιμοποιείται για γραφική αναπαράσταση διάφορων στοιχείων όπως πορείες, βολές, φωτιά κ.α. Ο κώδικας που χρησιμοποιώ είναι πολύ απλός καθώς τροποποιώ μόνο την alpha τιμή του χρώματος που έχω ορίσει ώστε να φανεί σαν να εξαφανίζεται με τον χρόνο. Η γραμμή αυτή υπάρχει στα περισσότερα shooter games και βοηθάει στην οπτικοποίηση της τροχιάς της σφαίρας.

```
private void SpawnBulletTrail(Vector3 hitPoint, Transform bulletStartPoint)
{
    GameObject bulletTrailEffect = Instantiate(bulletTrail.gameObject, bulletStartPoint.position, Quaternion.identity);
    LineRenderer lineR = bulletTrailEffect.GetComponent<LineRenderer>();
    lineR.SetPosition(0, bulletStartPoint.position);
    lineR.SetPosition(1, hitPoint);
    Destroy(bulletTrailEffect, 1f);
}
```

Εικόνα 122: Λογική Κώδικα για Μέθοδο SpawnBulletTrail

```
void Update()
{
    //για να κανει fade out
    color.a = Mathf.Lerp(color.a, 0, Time.deltaTime * speed);
    lineRenderer.startColor = color;
    lineRenderer.endColor = color;
}
```

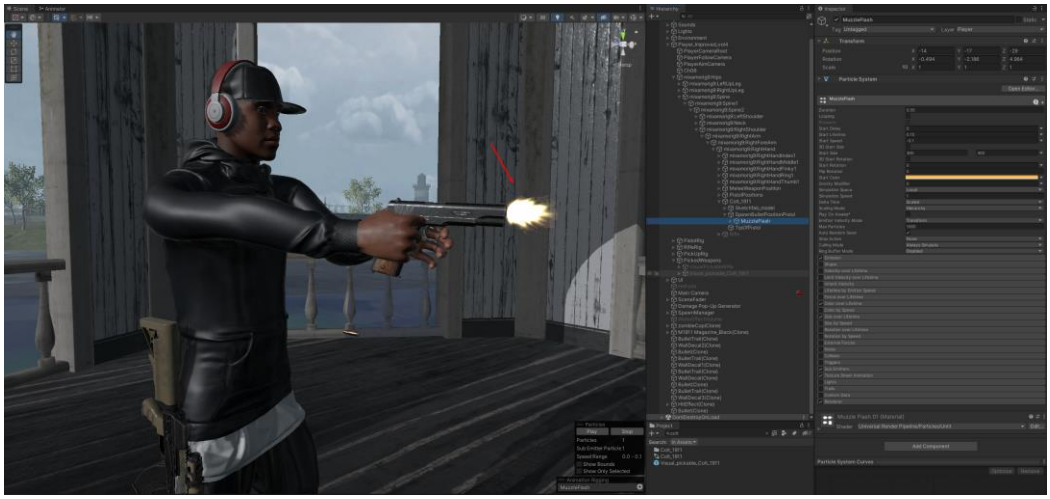
Εικόνα 123: Λογική Κώδικα LineFade



Εικόνα 124: Παράδειγμα LineRenderer

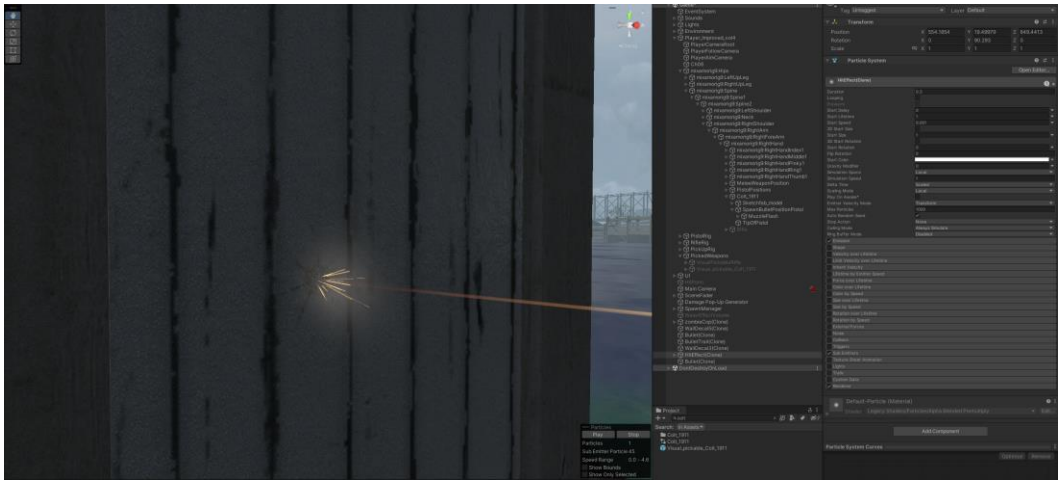
Το muzzle flash αναφέρεται στην λάμψη που προκύπτει από την κάννη του όπλου, όταν αυτό πυροβολεί. Είναι η έντονη, σύντομη εκπομπή φωτός που συνοδεύει την απελευθέρωση της ενέργειας από την εκρηκτική φόρτωση του πυροβόλου και την εκτόξευση του σφαίρας.

Το muzzle flash είναι σημαντικό γραφικό στοιχείο στα shooter games, καθώς προσθέτει ρεαλισμό. Συνήθως, αναπαρίσταται ως ένα έντονο φωτεινό φλας στην περιοχή της κάννης του όπλου κατά τη στιγμή της πυροδότησης. Τα χαρακτηριστικά του muzzle flash μπορούν να ποικίλλουν, ανάλογα με το είδος του όπλου, τον τύπο των πυρομαχικών, και το γενικό στυλ του παιχνιδιού. Για την πτυχιακή αυτή χρησιμοποιήθηκε ένα έτοιμο muzzle flash effect [asset 7] από το Unity Asset Store που δημιουργήθηκε μέσω του particle system της Unity.



Εικόνα 125: Παράδειγμα Muzzle Flash

Από το ίδιο asset, πήρα και ένα hit effect που περιείχε στο οποίο βγαίνουν σπίθες στο σημείο που χτύπησε η σφαίρα, πέρα από το decal που δημιουργείται στον τοίχο (βλ. 4.1.11).

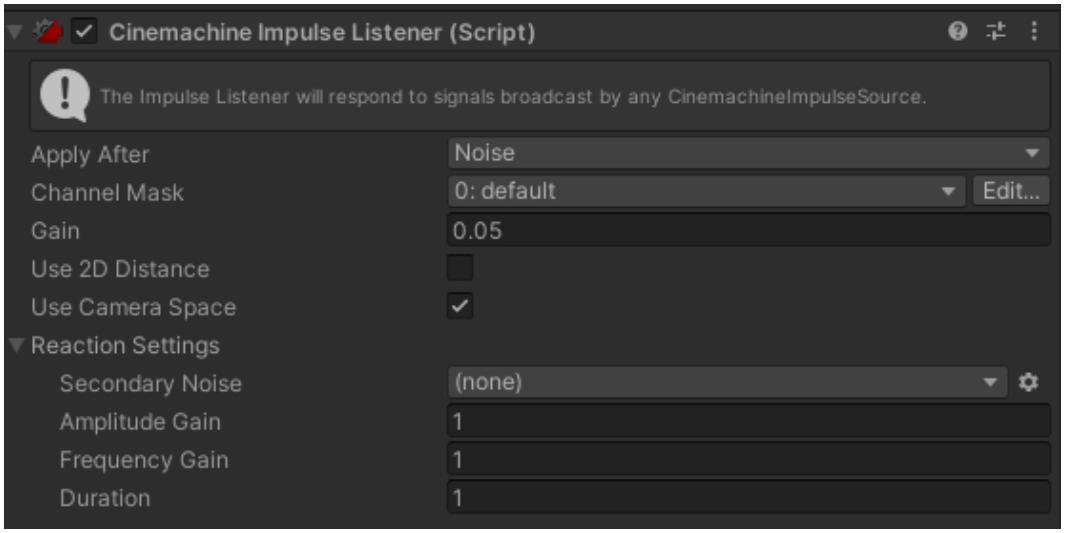


Εικόνα 126: Παράδειγμα Hit Effect

Έτσι, με όλα τα παραπάνω, αφιερώνοντας σημαντικό χρόνο στις λεπτομέρειες δημιουργήθηκε ένα ολοκληρωμένο και εντυπωσιακό περιβάλλον για τα όπλα του παιχνιδιού. Με εστιασμένη προσοχή στη λεπτομέρεια και την οπτική αισθητική, έχω ενσωματώσει εφέ που προσθέτουν ρεαλισμό και ομορφιά σε κάθε πτυχή της εμπειρίας του παίκτη.

5.4 Cinemachine Impulse Listener

Στο Cinemachine, ο όρος ‘impulse’ αναφέρεται συνήθως σε μια ξαφνική αλλαγή στη θέση ή την κίνηση μιας εικονικής κάμερας. Το σύστημα Cinemachine Impulse επιτρέπει την εφαρμογή δυνάμεων στην εικονική κάμερα για τη δημιουργία δυναμικών κινήσεων όπως κραδασμούς και δονήσεις. Είναι συνήθης τεχνική σε παιχνίδια καθώς προσδίδει μια όμορφη λεπτομέρεια στο τελικό αποτέλεσμα. Για τον λόγο αυτό, πρόσθεσα impulse σε δύο σημεία του κώδικα, πρώτον όταν ο παίκτης δέχεται ζημιά, ώστε να γίνεται πιο αισθητό και δεύτερον όταν ο παίκτης πυροβολάει αφού και το όπλο εκπέμπει ενέργεια προς τον χαρακτήρα. Η υλοποίηση του ήταν σχετικά απλή καθώς μετά την παραμετροποίηση του component που υπάρχει στον χαρακτήρα μας ενεργοποιείται με μια γραμμή κώδικα.



Εικόνα 127: Cinemachine Impulse Listener Component

```
1  //δημιουργει ενα impulse στην καμερα
2  impulse.GenerateImpulse(1);
```

Εικόνα 128: Λογική Κώδικα Cinemachine Impulse

5.5 Διάφοροι ήχοι

Πέρα από τους ήχους που προαναφέρθηκαν που αφορούν το περιβάλλον του παιχνιδιού, έχουν προστεθεί και σε άλλα σημεία όπως στα όπλα, στους εχθρούς και στο περπάτημα του χαρακτήρα. Για τους ήχους χρησιμοποιήθηκε το Unity Asset Store καθώς και ένας δωρεάν ιστότοπος με πολλές δωρεάν επιλογές [asset 8]. Ας δούμε πιο αναλυτικά για το καθένα.

Ξεκινώντας από τα όπλα, έχουμε τρεις διαφορετικές περιπτώσεις: πυροβολισμός, άδειο όπλο, και γέμισμα όπλου. Για να παίξουν οι ήχοι μέσω κώδικα αρκεί να ορίσουμε τα AudioClips και μετά για να ακουστούν μια φορά αρκεί να καλέσουμε την PlayOneShot συνάρτηση. Ο κώδικας για κάθε περίπτωση φαίνεται στις εικόνες 133 και 134.

```
//αν εχω σφαιρες παιζω τον ηχο μειωνω κατα 1 και ανανεωνω το UI
public void ConsumeAmmo()
{
    if (currentPistolAmmo > 0)
    {
        audioSource.PlayOneShot(pistolShoot);
        currentPistolAmmo--;
    }
    if (pistolTotalAmmo > 0)
    {
        canReload = true;
    }
    UpdateText(true);
}
```

Εικόνα 129: Λογική Κώδικα Για Ήχους Όπλων (Στιγμιότυπο 1)

```
//αν εχω υπολειπομενες σφαιρες τοτε μπορω να κανω reload
    public void ReloadAmmo()
    {
        int missingRounds = pistolMaxMagazineAmmo - currentPistolAmmo;

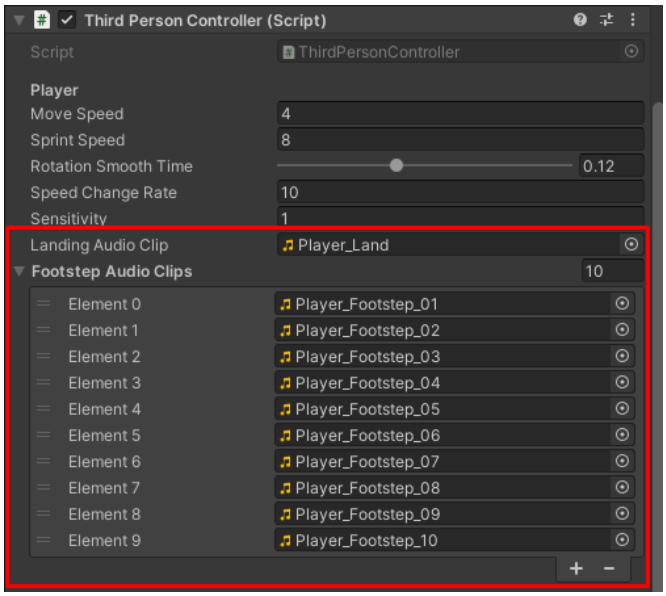
        if (pistolTotalAmmo >= missingRounds)
        {
            currentPistolAmmo = pistolMaxMagazineAmmo;
            pistolTotalAmmo -= missingRounds;
            canReload = false;
        }
        else if (pistolTotalAmmo < missingRounds && pistolTotalAmmo > 0)
        {
            currentPistolAmmo += pistolTotalAmmo;
            pistolTotalAmmo = 0;
            canReload = false;
        }
        //αν δεν εχω για reload τοτε παιζω τον ηχο για αδειο γεμιστηρα
        else if (pistolTotalAmmo == 0)
        {
            PlayEmptySound();
            canReload = false;
        }
        UpdateText(true);
    }

    public void PlayEmptySound()
    {
        audioSource.PlayOneShot(pistolShootEmpty);
    }
    public void PlayReloadSound()
    {
        audioSource.PlayOneShot(pistolReload);
    }
```

Εικόνα 130: Λογική Κώδικα Για Ήχους Όπλων (Στιγμιότυπο 2)

Για καλύτερη αίσθηση στο παιχνίδι, έβαλα και στους εχθρούς διάφορους ήχους, οι οποίοι, είναι ανάλογοι την κατάσταση που βρίσκονται την δεδομένη στιγμή. Συγκεκριμένα, έχουμε ποικίλους ήχους για την κατάσταση Patrol, Angry, Damaged, AttackLight, AttackStrong και Die. Είναι οι καταστάσεις από το state machine των εχθρών και οι συναρτήσεις που φαίνονται στον κώδικα [παράρτημα 10] χρησιμοποιούνται ως animation events. Είναι άξιο σημείωσης να προσέξουμε ότι η ένταση του ήχου αλλάζει ανάλογα την απόσταση του εχθρού από το παίκτη ώστε να έχουμε πιο ρεαλιστικά αποτελέσματα. Αυτό επιτυγχάνεται βρίσκοντας την απόσταση με τον παίκτη και περνώντας την τιμή ως όρισμα στην συνάρτηση PlayOneShot.

Η ίδια λογική του κώδικα χρησιμοποιείται και στον χαρακτήρα ο οποίος έχει ήχους για τα βήματα καθώς και όταν προσγειώνεται στην επιφάνεια μετά από κάποιο άλμα. Για τα βήματα έχουμε διάφορους ήχους που παίζουν κάθε φορά τυχαία ενώ για την προσγείωση υπάρχει ο ίδιος ήχος κάθε φορά.



Εικόνα 131: Εισαγωγή AudioClips Στο Component

```
private void OnFootstep(AnimationEvent animationEvent)
{
    if (animationEvent.animatorClipInfo.weight > 0.5f)
    {
        if (FootstepAudioClips.Length > 0)
        {
            var index = Random.Range(0, FootstepAudioClips.Length);
            _audioSource.PlayOneShot(FootstepAudioClips[index]);
        }
    }
}

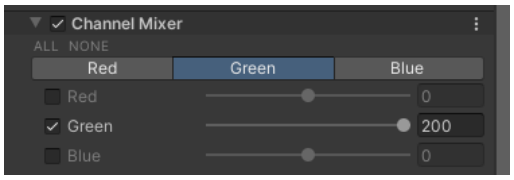
private void OnLand(AnimationEvent animationEvent)
{
    if (animationEvent.animatorClipInfo.weight > 0.5f)
    {
        _audioSource.PlayOneShot(LandingAudioClip);
    }
}
```

Εικόνα 132: Λογική Κώδικα Για Ήχους Χαρακτήρα Παίκτη

5.6 Post-Processing εφέ για το νερό

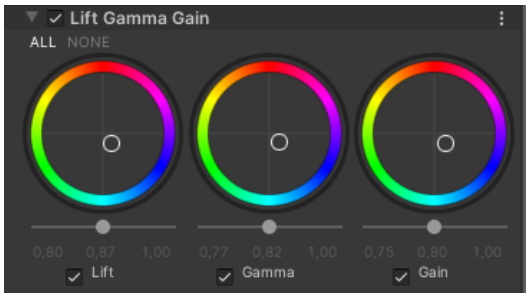
Το Post-Processing στην Unity αναφέρεται σε μια σειρά εφέ που εφαρμόζονται στη σκηνή μετά την απεικόνιση (post-rendering). Αυτά τα εφέ μπορούν να βελτιώσουν την αισθητική και την ποιότητα της γραφικής απεικόνισης στο παιχνίδι. Καθώς ο συγκεκριμένος χάρτης που χρησιμοποίησα περιλαμβάνει νερό, ήθελα να δημιουργήσω ένα εφέ που ενεργοποιείται όταν ο παίκτης μπαίνει μέσα στο νερό (και συγκεκριμένα η κάμερα του παιχνιδιού). Αυτό το υλοποίησα με Post-Processing. Πιο αναλυτικά, δημιούργησα ένα νέο Gameobject στο οποίο πρόσθεσα ένα volume component που σχετίζεται με το Post-Processing σύστημα. Συγκεκριμένα, τα Volume Components περιέχουν τις παραμέτρους για τα διάφορα Post-Processing εφέ που εφαρμόζονται στη σκηνή. Αυτά τα στοιχεία περιλαμβάνουν ρυθμίσεις όπως Bloom, Depth of Field, Color Grading, και πολλά άλλα. Για το εφέ στο νερό χρησιμοποίησα τα παρακάτω:

Channel Mixer: Επιτρέπει τον έλεγχο των καναλιών χρωμάτων (κόκκινο, πράσινο, μπλε), δίνοντας τη δυνατότητα για δημιουργία εξατομικευμένων χρωματικών εφέ.



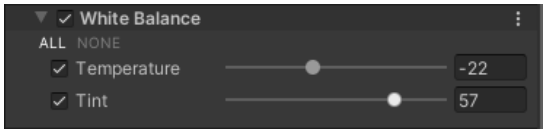
Εικόνα 133: Volume Component - Channel Mixer

Lift Gamma Gain: Προσφέρει ακριβή έλεγχο των σκοτεινών, μεσαίων και φωτεινών περιοχών της εικόνας για τη βελτίωση της φωτεινότητας και της αντίθεσης.



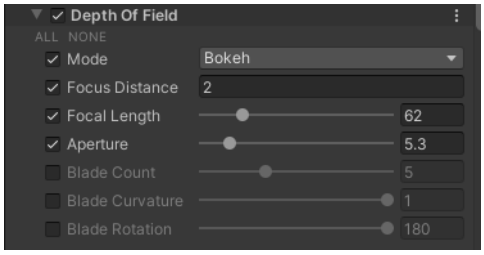
Εικόνα 134: Volume Component - Lift Gamma Gain

White Balance: Επιτρέπει τον έλεγχο της χρωματικής ισορροπίας, ρυθμίζοντας τα χρώματα της εικόνας για να επιτευχθεί η επιθυμητή αισθητική.



Εικόνα 135: Volume Component - White Balance

Depth of Field: Ρυθμίζει το βάθος εστίασης για να δημιουργήσει εφέ ασάφειας γύρω από τα αντικείμενα που δεν είναι στο επίκεντρο.



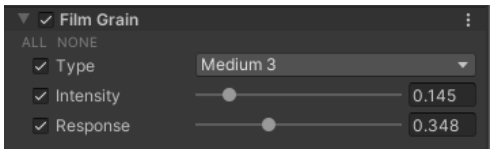
Εικόνα 136: Volume Component - Depth Of Field

Vignette: Προσθέτει σκοτεινότερες άκρες στην οθόνη για να εστιάσει την προσοχή στο κέντρο της.



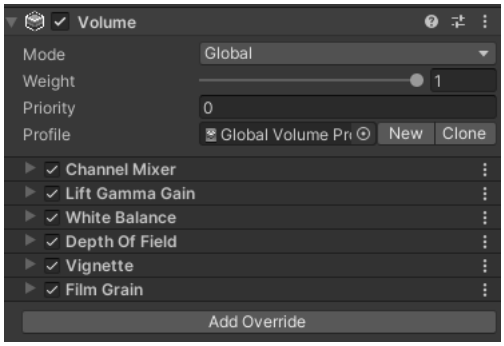
Εικόνα 137: Volume Component – Vignette

Film Grain: Προσθέτει εφέ κόκκων φιλμ για να μιμηθεί την υφή και την αίσθηση παλιών φιλμ.



Εικόνα 138: Film Grain

Όλα τα παραπάνω προστέθηκαν σε ένα νέο προφίλ που δημιουργήθηκε στο Volume Component στο οποίο ορίσαμε τον τύπο του ως Global για να επηρεάζει όλη την σκηνή, το βάρος του στο μέγιστο και την προτεραιότητα στο μηδέν ώστε να εμφανίζεται πάνω από άλλα τυχόν post-processing εφέ. Τα τελικά αποτελέσματα από το πριν και το μετά φαίνονται στις εικόνες 144-145.



Εικόνα 139: Volume Component



Εικόνα 140: Παράδειγμα Παιχνιδιού Χωρίς Εφέ Νερού



Εικόνα 141: Παράδειγμα Παιχνιδιού Με Εφέ Νερού

5.7 PlayerPrefs

Το PlayerPrefs είναι μια διεπαφή στο Unity που παρέχει απλό και ευέλικτο τρόπο αποθήκευσης και ανάκτησης δεδομένων που αντιπροσωπεύουν την κατάσταση του παιχνιδιού ή τις επιλογές του χρήστη. Χρησιμοποιείται κυρίως για την αποθήκευση απλών τιμών, όπως ακέραιοι, δεκαδικοί αριθμοί, ή συμβολοσειρές, μεταξύ διαφόρων συνεδριών του παιχνιδιού.

Οι τιμές που αποθηκεύονται με το PlayerPrefs παραμένουν μεταξύ των διαφόρων εκτελέσεων της εφαρμογής και μπορούν να χρησιμοποιηθούν για την διατήρηση πληροφοριών όπως ρυθμίσεις παιχνιδιού, επίπεδα προόδου ή παραμετροποίηση προτιμήσεων του χρήστη. Ωστόσο, πρέπει να χρησιμοποιείται με προσοχή, καθώς δεν είναι κατάλληλο για μεγάλα ή ευαίσθητα δεδομένα.

Για την εργασία αυτήν, χρησιμοποιήθηκε ως μια γρήγορη λύση για την αποθήκευση του υψηλότερου κύματος που έχει φτάσει ο παίκτης. Είναι μια απλή τιμή ακέραιου αριθμού που ορίζεται όταν πεθαίνει ο χαρακτήρας και διαβάζεται στο μενού ώστε να αλλάξει το κείμενο με τον κατάλληλο αριθμό.

```
int waveNumber = SpawnManager.current.GetCurrentWave();
//αν εκανα καλυτερο ρεκορ απο το προηγουμενο, το αποθηκευει
if (waveNumber > PlayerPrefs.GetInt("BestWave"))
{
    PlayerPrefs.SetInt("BestWave", waveNumber);
}
gameOverText.text = $"You made it to <color=red>Wave {waveNumber}</color> !";
```

Εικόνα 142: Λογική Κώδικα PlayerPrefs (Στιγμιότυπο 1)

```
public class ChangeBestWaveText : MonoBehaviour
{
    //τρεχει οταν μπαινω στο μενου ωστε να ανανεωσει το best Score
    void Start()
    {
        TextMeshProUGUI text = GetComponent<TextMeshProUGUI>();
        int waveNumber = PlayerPrefs.GetInt("BestWave");
        text.text = $"Play (Best Wave: {waveNumber})";
    }
}
```

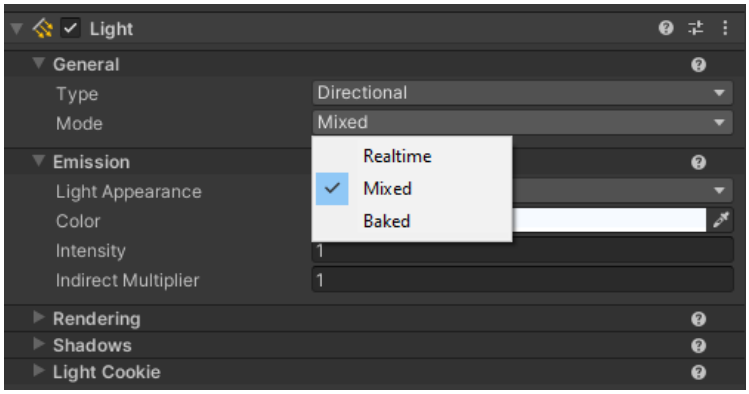
Εικόνα 143: Λογική Κώδικα PlayerPrefs (Στιγμιότυπο 2)

5.8 Βελτιστοποίηση Απόδοσης (Performance Optimization)

Η βελτιστοποίηση στα παιχνίδια αποτελεί κρίσιμο στάδιο της ανάπτυξης, επηρεάζοντας την απόδοση, την εμπειρία του χρήστη και την επιτυχία του παιχνιδιού. Είναι κεντρικής σημασίας για τη δημιουργία ομαλών και αποτελεσματικών παιχνιδιών, μειώνοντας τον χρόνο φόρτωσης, το lag και τις τεχνικές δυσκολίες. Οι παίκτες προτιμούν παιχνίδια που προσφέρουν αξιοπρεπή απόδοση και αποφεύγουν τεχνικά θέματα. Επιπλέον, η βελτιστοποίηση συμβάλλει στη διαχείριση των πόρων, την επαναχρησιμοποίηση του κώδικα και την οικονομία ενέργειας, καθιστώντας την κρίσιμη για την επιτυχία των παιχνιδιών σε κινητές πλατφόρμες και μη. Σε κάθε παιχνίδι ένας ικανοποιητικός αριθμός καρέ ανά δευτερόλεπτο (FPS) είναι 60 και πάνω καθώς συνδέεται με το γεγονός ότι το ανθρώπινο μάτι μπορεί να δει από 30 έως 60 καρέ ανά δευτερόλεπτο. Για να έχει ο παίκτης μια ομαλή και ευχάριστη εμπειρία έπρεπε ο αριθμός FPS στο παιχνίδι να βρίσκεται εκεί κοντά. Καθώς το παιχνίδι περιλαμβάνει ένα τεράστιο χάρτη με χιλιάδες αντικείμενα και υπάρχουν πολλές λειτουργίες που τρέχουν κάθε frame, στην αρχική φάση το παιχνίδι λειτουργούσε με πολύ χαμηλά καρέ ανά δευτερόλεπτο (περίπου 20-30 FPS), δημιουργώντας μια αρνητική εμπειρία για τον χρήστη. Για τον λόγο αυτό, εφάρμοσα τρεις διαφορετικές τεχνικές βελτιστοποίησης που αύξησαν αρκετά τον αριθμό FPS. Παρακάτω αναλύονται ξεχωριστά:

5.8.1 Baked Φωτισμός

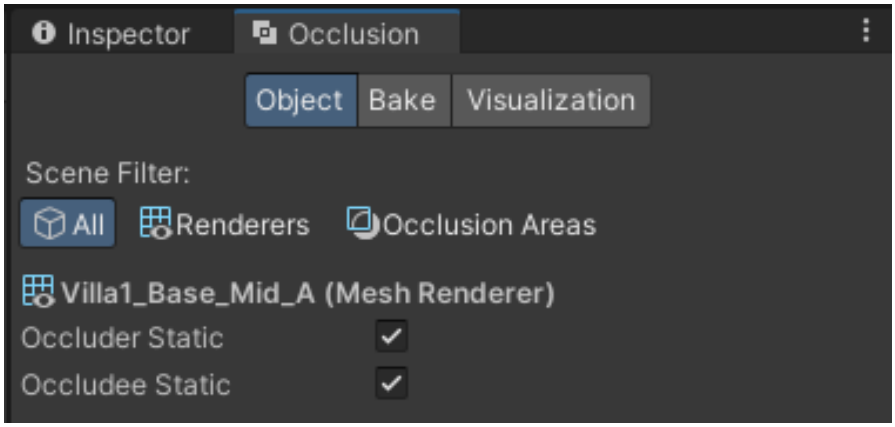
Στην Unity, υπάρχουν δύο κατηγορίες φωτισμού, baked και realtime. Ο baked φωτισμός στο πλαίσιο της δημιουργίας παιχνιδιών αναφέρεται σε μια διαδικασία όπου τα φωτεινά και σκιώδη χαρακτηριστικά του περιβάλλοντος ενσωματώνονται στα textures του παιχνιδιού αντί να υπολογίζονται κατά τη διάρκεια της εκτέλεσης του παιχνιδιού. Ο σταθερός και προσδιορισμένος φωτισμός αυξάνει την απόδοση και βελτιστοποιεί τον τρόπο που το παιχνίδι χειρίζεται τον φωτισμό. Αντίθετα στον realtime, ο φωτισμός του περιβάλλοντος υπολογίζεται κατά την εκτέλεση του παιχνιδιού, καταναλώνοντας πολύτιμους πόρους της GPU και CPU. Καθώς στο παιχνίδι έχουμε τον χαρακτήρα και τους εχθρούς που χρειάζονται δυναμικό φωτισμό, χρησιμοποιήσαμε την επιλογή mixed lighting στην οποία υπολογίζεται ο baked φωτισμός στα στατικά αντικείμενα και ο realtime φωτισμός στα δυναμικά αντικείμενα. Έτσι ολοκληρώθηκε το πρώτο στάδιο βελτιστοποίησης.



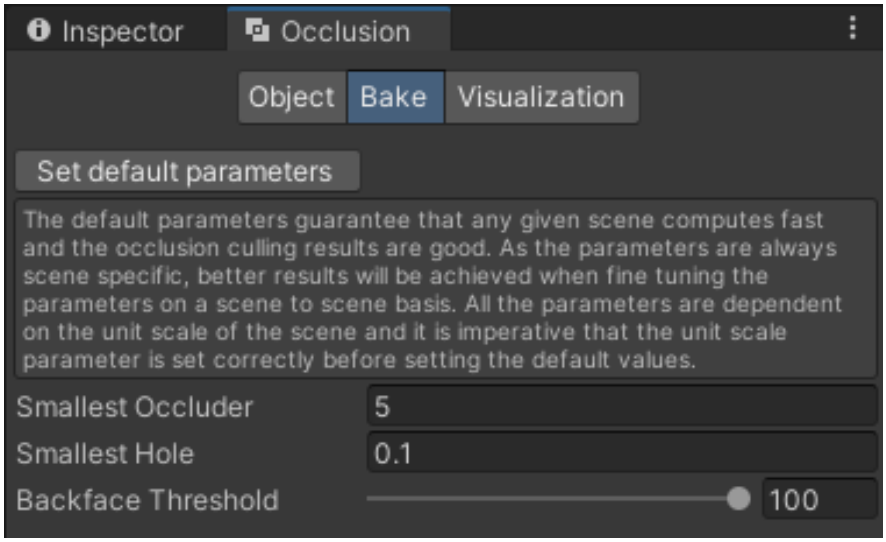
Εικόνα 144: Mixed Lightning

5.8.2 Occlusion Culling

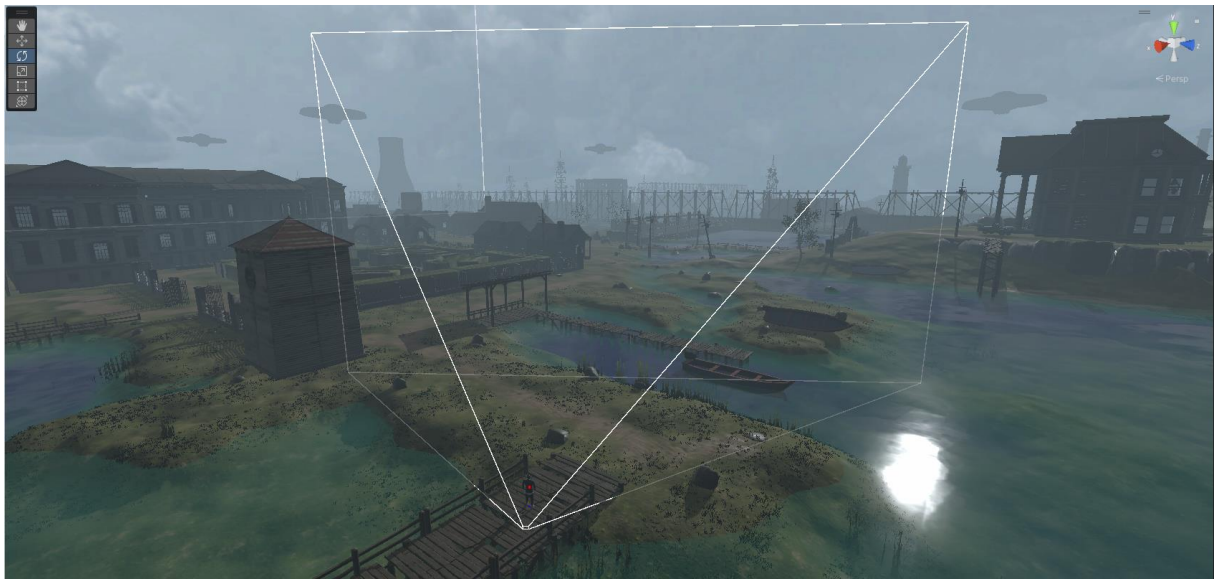
Το occlusion culling είναι μια τεχνική που χρησιμοποιείται στον τομέα των υπολογιστικών γραφικών και παιχνιδιών για να βελτιώσει την απόδοση. Στο πλαίσιο της Unity, το occlusion culling είναι μια λειτουργία που εμποδίζει να εκτελεστούν rendering υπολογισμοί για Gameobjects που είναι εντελώς κρυμμένα από το οπτικό πεδίο της κάμερας. Για να χρησιμοποιήσουμε αυτή την τεχνική αρχικά ενεργοποιούμε την επιλογή Occlusion Culling στην κάμερα. Έπειτα, στην καρτέλα occlusion culling του Unity Editor, έχουμε κάποιες δυνατότητες. Αφού επιλέξουμε όλα τα Gameobjects που είναι στατικά, δηλαδή δεν έχουν κάποια κίνηση, στην υπο-καρτέλα Object (εικόνα 145) τα ορίζουμε ως Occluder και Occludee static. Στην ουσία, λέμε στην Unity ότι αυτά τα αντικείμενα είναι πιθανό να καλύψουν ή καλυφθούν από άλλα αντικείμενα. Στην επόμενη υπο-καρτέλα Bake (εικόνα 149), μπορούμε να αλλάξουμε κάποιες έξτρα ρυθμίσεις. Εγώ άφησα τις default τιμές και έκανα Bake ώστε να δημιουργηθούν τα δεδομένα. Στην τελευταία υπο-καρτέλα Visualization μπορούμε να δούμε τα αποτελέσματα πριν και μετά. (εικόνες 150-151)



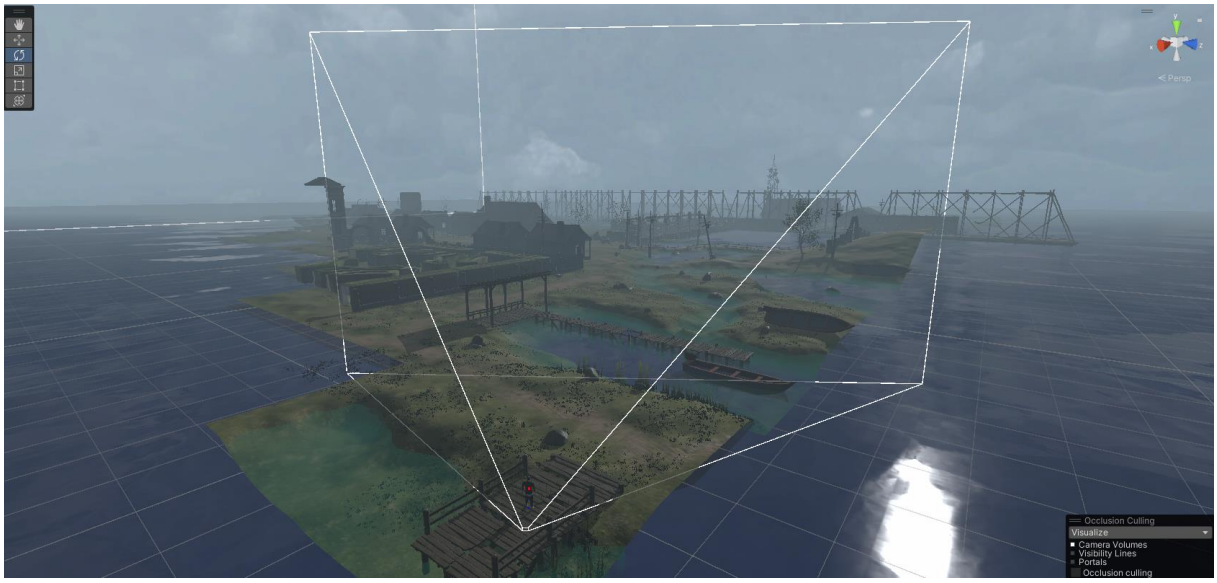
Εικόνα 145: Occlusion Culling - Καρτέλα Object



Εικόνα 146: Occlusion Culling - Καρτέλα Bake



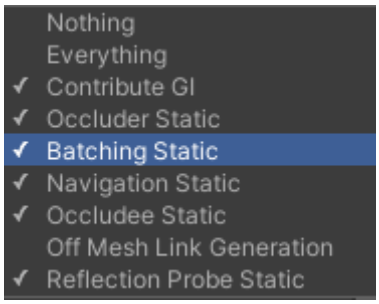
Εικόνα 147: Παράδειγμα Σκηνής Χωρίς Occlusion Culling



Εικόνα 148: Παράδειγμα Σκηνής Με Occlusion Culling

5.8.3 Static Batching

Το "Static Batching" στην Unity αναφέρεται στη διαδικασία συγχώνευσης (batching) γραφικών αντικειμένων (meshes) κατά τη διάρκεια του χρόνου μεταγλώττισης (compile time) αντί του χρόνου εκτέλεσης (runtime) για να βελτιώσει την απόδοση στα παιχνίδια. Αυτό είναι χρήσιμο ειδικά για αντικείμενα που παραμένουν σταθερά και δεν αλλάζουν κατά τη διάρκεια της εκτέλεσης του παιχνιδιού. Για να το χρησιμοποιήσω, επέλεξα όλα τα στοιχεία του περιβάλλοντος που δεν μετακινούνται ούτε περιστρέφονται και τα όρισα ως batching static όπως φαίνεται στην εικόνα 153. Η εφαρμογή του είναι απλή καθώς τα υπόλοιπα τα χειρίζεται η Unity, δεν έχουμε έλεγχο πάνω στο πως θα γίνει η διαδικασία, παρ' όλ' αυτά, έχουμε καλά αποτελέσματα αν χρησιμοποιηθεί σωστά.



Εικόνα 149: Batching Static

Συμπερασματικά, η βελτιστοποίηση ενός παιχνιδιού παίζει σημαντικό ρόλο στη δημιουργία μιας ομαλής και απολαυστικής εμπειρίας για τους παίκτες. Η εξειδικευμένη βελτιστοποίηση, όπως η χρήση του Static Batching στην Unity, μπορεί να έχει μεγάλη επίδραση στην απόδοση του παιχνιδιού, καθώς μειώνει τον φόρτο εργασίας της GPU και βελτιστοποιεί τη ροή γραφικών. Εκτός από τις παραπάνω τεχνικές βελτιστοποίησης, υπάρχουν και άλλες, όπως η χρήση LOD (Level of Detail) για τα μοντέλα, η αποτελεσματική διαχείριση μνήμης, η χρήση συμπίεσης γραφικών, και η ελαχιστοποίηση των κλήσεων στη μονάδα επεξεργασίας γραφικών. Ενισχύοντας την απόδοση του παιχνιδιού, οι προγραμματιστές μπορούν να εξασφαλίσουν πιο υψηλές και σταθερές τιμές καρέ ανά δευτερόλεπτο (FPS), οι οποίες είναι κρίσιμες για την αίσθηση ρεαλιστικότητας και απόκρισης στον κόσμο του παιχνιδιού. Πολλές φορές βέβαια η διαδικασία γίνεται περίπλοκη και δύσκολη αφού σε κάθε παιχνίδι διαφέρει. Για τον λόγο αυτό, υλοποίησα τις πιο απλές μεθόδους όπου η εφαρμογή τους ήταν ευκολότερη σε σχέση με άλλες. Τα τελικά αποτελέσματα φαίνονται παρακάτω.



Εικόνα 150: Στατιστικά Πριν Το Optimization



Εικόνα 151: Στατιστικά Μετά Το Optimization

6. Επίλογος

6.1 Συμπεράσματα

Στην υλοποίηση του παιχνιδιού “Zombie Uprising Last Stand!” κατανοήσαμε σε μεγάλο βαθμό τις δυνατότητες που προσφέρει η παιχνιδομηχανή της Unity. Κατανοήθηκαν βασικές αλλά και πιο σύνθετες τεχνικές κατά τη διάρκεια της διαδικασίας ανάπτυξης, προτού φτάσουμε στο τελικό αποτέλεσμα. Καθώς το παιχνίδι δημιουργήθηκε ολοκληρωτικά από την αρχή και υπήρξα το μόνο άτομο στην δημιουργία του, έπρεπε να ασχοληθώ με κομμάτια τα οποία δεν είχα καμία εμπειρία και να τα ενσωματώσω στο παιχνίδι. Τα περισσότερα επιτυχημένα παιχνίδια συνήθως υλοποιούνται από ομάδες που αναλαμβάνουν ξεχωριστά κομμάτια όπως ο προγραμματισμός, η σχεδίαση των επιπέδων (Level Design), τα γραφικά, ο ήχος, η ιστορία, η βελτιστοποίηση και πολλά ακόμα. Όμως, με τον κατάλληλο σχεδιασμό, την βοήθεια από τα ισχυρά εργαλεία της Unity και κάποια έτοιμα στοιχεία από το Unity Asset Store, το τελικό αποτέλεσμα για το πρώτο μου παιχνίδι θεωρώ ότι είναι αρκετά ικανοποιητικό.

6.2 Δυσκολίες

Κατά την υλοποίηση του παιχνιδιού ήρθα αντιμέτωπος με δεκάδες προβλήματα και bugs που χρειάστηκαν πολύ χρόνο και έρευνα σε forums και documentation της Unity. Καθώς ήθελα να δημιουργήσω ένα παιχνίδι το οποίο θα έδινε μια ευχάριστη εμπειρία στον χρήστη, προσπάθησα να λύσω κάθε πιθανό λογικό λάθος που σκεφτόμουν ή ερχόταν στην επιφάνεια

κατά το playtesting. Πράγματα που δεν είχα ασχοληθεί ξανά όπως το animation rigging και το navigation της Unity ήταν τα πιο χρονοβόρα προβλήματα που έλυσα καθώς χρειάζονται προσοχή και μελέτη για να λειτουργήσουν σωστά. Επιπλέον, κάποιες έξτρα λεπτομέρειες που προστέθηκαν για ρεαλισμό στο παιχνίδι φάνηκαν αρκετά προκλητικές, κυρίως λόγω της απειρίας πάνω στο κομμάτι αυτό. Εκτός από αυτό, δυσκολεύτηκα και με την γλώσσα C# στην αρχή καθώς δεν είχα προγραμματίσει ποτέ ξανά πριν ασχοληθώ με την Unity, με αποτέλεσμα να σβήνω και να ξαναγράφω κώδικα από την αρχή ώστε να είναι πιο ευανάγνωστος, σωστός και να ακολουθεί design patterns.

6.3 Μελλοντικές Βελτιώσεις και Επεκτάσεις

Στο παιχνίδι, μπορούν να γίνουν πολλές βελτιώσεις αλλά και επεκτάσεις οι οποίες θα το φτάσουν σε ένα επίπεδο που θα μπορεί να βγει πλέον και στην αγορά. Αρχικά, μια ενδιαφέρουσα προσθήκη στο παιχνίδι θα ήταν η ενσωμάτωση αποστολών. Αυτές οι αποστολές θα μπορούσαν να προσφέρουν πρόσθετη ποικιλία και προκλήσεις στους παίκτες, ενισχύοντας την διάρκεια ζωής του παιχνιδιού και προσφέροντας επιπλέον ανταμοιβές. Η προσθήκη λειτουργίας multiplayer για co-op θα επιτρέπει στους παίκτες να συνεργαστούν για την επιβίωση στον κόσμο των ζόμπι. Αυτό θα προσφέρει μια νέα διάσταση στην εμπειρία, καθώς θα μπορούν να ανταλλάσσουν τεχνικές, να προστατεύονται μεταξύ τους και να επιλύουν αποστολές ως ομάδα. Ένα κομμάτι που δεν είχα το χρόνο να υλοποιήσω καθώς είναι αρκετά περίπλοκο είναι η δυνατότητα του χαρακτήρα να κολυμπάει. Ο χάρτης περιλαμβάνει σημεία που έχει νερό και θα ήταν αρκετά ενδιαφέρον να μπορεί να κολυμπάει από την μία άκρη στην άλλη. Μελλοντικά, θα μπορούσε να επεκταθεί η ποικιλία των όπλων για περισσότερες επιλογές ώστε ο καθένας να προσαρμόζει τον χαρακτήρα του ανάλογα το στυλ του παιχνιδιού που προτιμάει. Τέλος, ένα δυναμικό σύστημα καιρού με αλλαγές σε βροχή, χιόνι, ή ακόμα και καταιγίδες, μπορεί να προσθέσει περισσότερη ατμόσφαιρα και προκλήσεις στο παιχνίδι. Με αυτές τις προσθήκες, η εμπειρία του "Zombie Uprising Last Stand!" θα εμπλουτιστεί και θα προσφέρει στους παίκτες νέες και συναρπαστικές δυνατότητες.

7. Βιβλιογραφία και Πηγές

Για τη συγγραφή της πτυχιακής εργασίας:

- https://en.wikipedia.org/wiki/Computer_graphics
- https://en.wikipedia.org/wiki/Unity_(game_engine)
- https://en.wikipedia.org/wiki/Mixamo
- https://en.wikipedia.org/wiki/Sketchfab
- https://medium.com/@niitwork0921/what-is-unity-asset-store-4f010402a5c6
- https://docs.unity3d.com/ScriptReference/PlayerPrefs.html
- https://docs.unity3d.com/Manual/OcclusionCulling.html
- https://docs.unity3d.com/Manual/static-batching.html
- https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.3/manual
- https://docs.unity3d.com/ScriptReference/AI.NavMesh.html
- https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@12.0/manual/renderer-feature-decal.html

Για την υλοποίηση του παιχνιδιού:

- https://assetstore.unity.com/packages/essentials/starter-assets-third-person-character-controller-urp-196526
- https://assetstore.unity.com/packages/tools/utilities/health-system-for-dummies-215755
- https://assetstore.unity.com/packages/tools/game-toolkits/game-core-radar-system-lite-226229
- https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488
- https://assetstore.unity.com/packages/vfx/shaders/urp-dissolve-2020-191256
- https://assetstore.unity.com/packages/vfx/particles/blood-gush-73426
- https://assetstore.unity.com/packages/vfx/particles/particle-pack-127325
- https://pixabay.com/sound-effects/

8. Παραρτήματα

```
[SerializeField] AudioMixer audioMixer;
    [SerializeField] Slider musicSlider,sfxSlider;
    [SerializeField] TMP_Dropdown resolutionDropdown;
    [SerializeField] Toggle fullscreenToggle;
    Resolution[] resolutions;
    List<Resolution> filteredResolutions;
    float currentRefreshRate;
    int currentResolutionIndex = 0;

    private void Start()
    {
        musicSlider.onValueChanged.AddListener(SetMusicVolume);
        sfxSlider.onValueChanged.AddListener(SetSFXVolume);

        //για τα settings βρισκω τα screen resolution που υποστηριζει ο υπολογιστης και τα περναω στην λιστα του dropdown
        resolutions = Screen.resolutions;
        filteredResolutions = new();

        resolutionDropdown.ClearOptions();
        currentRefreshRate = Screen.currentResolution.refreshRate;

        //κραταω μονο τα refresh rate που τρεχει ο υπολογιστης εκεινη την στιγμη
        for (int i = 0; i < resolutions.Length; i++)
        {
            if (resolutions[i].refreshRate == currentRefreshRate)
            {
                filteredResolutions.Add(resolutions[i]);
            }
        }

        List<string> options = new();
        for (int i = 0; i < filteredResolutions.Count; i++)
        {
            string resolutionOption = filteredResolutions[i].width + "x" + filteredResolutions[i].height;
            options.Add(resolutionOption);
            if (filteredResolutions[i].width == Screen.width && filteredResolutions[i].height == Screen.height)
            {
                currentResolutionIndex = i;
            }
        }

        resolutionDropdown.AddOptions(options);
        resolutionDropdown.value = currentResolutionIndex;
        resolutionDropdown.RefreshShownValue();
    }
    public void SetResolution(int resolutionIndex)
    {
        Resolution resolution = filteredResolutions[resolutionIndex];
        Screen.SetResolution(resolution.width, resolution.height, fullscreenToggle.isOn);
    }

    public void SetQuality(int qualityIndex)
    {
        QualitySettings.SetQualityLevel(qualityIndex);
    }

    public void SetFullscreen(bool isFullscreen)
    {
        Screen.fullScreen = isFullscreen;
    }

    public void SetMusicVolume(float value)
    {
        audioMixer.SetFloat("MusicVolume", value);
    }
    public void SetSFXVolume(float value)
    {
        audioMixer.SetFloat("SFXVolume", value);
        audioMixer.SetFloat("TerrainVolume", value);
    }
```

Παράρτημα 1

```
public void FadeInGameOrMenu()
{
    StartCoroutine(FadeIn());
}
public void FadeToGame(string scene)
{
    instructions.SetActive(true);
    //καλω την συναρτηση μετα απο 0.5 ωστε να σιγουρευτω οτι ανοιξε
    //το instructions και δεν κολλησε πριν αρχισει να κανει load
    StartCoroutine(FadeOutToGame(scene, 0.5f));
}

public void FadeToRetryGame(string scene)
{
    FadeOutToRetryGame(scene);
}
public void FadeToMenu(string scene)
{
    StartCoroutine(FadeOutToMenu(scene));
}

IEnumerator FadeIn()
{
    float t = 1f;

    while (t > 0f)
    {
        t -= Time.deltaTime;
        float a = curve.Evaluate(t);
        img.color = new Color(0f, 0f, 0f, a);
        yield return 0;
    }
}
IEnumerator FadeOutToMenu(string scene)
{
    float t = 0f;

    while (t < 1f)
    {
        t += Time.deltaTime;
        float a = curve.Evaluate(t);
        img.color = new Color(0f, 0f, 0f, a);
        yield return 0;
    }
    SceneManager.LoadScene(scene);
}

void FadeOutToRetryGame(string scene)
{
    instructions.SetActive(false);
    instructionsWithBar.SetActive(true);
    StartCoroutine(ReLoadLevelAsync(scene));
}
IEnumerator FadeOutToGame(string scene,float delay)
{
    yield return new WaitForSeconds(delay);
    StartCoroutine(LoadLevelAsync(scene));
}


IEnumerator ReLoadLevelAsync(string levelToLoad)
{
    progressSlider.value = 0;
    AsyncOperation load = SceneManager.LoadSceneAsync(levelToLoad);
    load.allowSceneActivation = false;
    float progress = 0;
    while (!load.isDone)
    {
        progress = Mathf.MoveTowards(progress, load.progress, Time.deltaTime);
        progressSlider.value = progress;
        progressText.text = $"Φορτώνει.. ({Mathf.RoundToInt(progress * 100)}%)";
        if (progress >= 0.9f)
        {
            progressSlider.value = 1;
            progressText.text = "Φορτώνει.. (100%)";
            load.allowSceneActivation = true;
        }
        yield return null;
    }
    instructions.SetActive(true);
    instructionsWithBar.SetActive(false);
}

//συναρτηση που φορτωνει το loading bar αναλογα ποσο % εχει φορτωσει η σκηνη του παιχνιδιου
IEnumerator LoadLevelAsync(string levelToLoad)
{
    progressSlider.value = 0;
    AsyncOperation load = SceneManager.LoadSceneAsync(levelToLoad);
    load.allowSceneActivation = false;
    float progress = 0;
    while (!load.isDone)
    {
        progress = Mathf.MoveTowards(progress, load.progress, Time.deltaTime);
        progressSlider.value = progress;
        progressText.text = $"Φορτώνει.. ({Mathf.RoundToInt(progress * 100)}%)";
        if (progress >= 0.9f)
        {
            progressSlider.value = 1;
            progressText.text = "Φορτώνει.. (100%)";
            load.allowSceneActivation = true;
        }
        yield return null;
    }
}
```

Παράρτημα 2

```
ThirdPersonController controller;
StarterAssetsInputs _input;
public float playerStamina = 100.0f;
public bool hasRegenerated = true;
public bool weAreSprinting = false;
[SerializeField] float maxStamina = 100.0f;
[SerializeField] float jumpCost = 20.0f;
[SerializeField] float staminaDrain = 0.5f;
[SerializeField] float staminaRegen = 0.5f;
[SerializeField] float slowedRunSpeed = 3.0f;
[SerializeField] float normalRunSpeed = 5.335f;
[SerializeField] Image staminaProgressUI = null;
[SerializeField] CanvasGroup sliderCanvasGroup = null;

void Awake()
{
    controller = GetComponent<ThirdPersonController>();
    _input = GetComponent<StarterAssetsInputs>();
}


private void Update()
{
    //αν δεν τρεχουμε
    if (!weAreSprinting)
    {
        //και δεν εχουμε max stamina
        if (playerStamina <= maxStamina)
        {
            //τοτε γεμιζει η μπαρα με το rate που του εχω δωσει
            playerStamina = Mathf.Clamp(playerStamina + staminaRegen * Time.deltaTime, 0f, maxStamina);
            //ανανεωνω το stamina bar
            UpdateStamina(1);
            //αν γινει max stamina τοτε αλλαζω την ταχυτητα πισω σε normal
            if (playerStamina >= maxStamina)
            {
                controller.SetRunSpeed(normalRunSpeed);
                sliderCanvasGroup.alpha = 0;
                hasRegenerated = true;
            }
        }
    }
}

public void Sprinting()
{
    //αν τρεχω και δεν μου ειχε τελειωσει η stamina νωριτερα
    if (hasRegenerated)
    {
        //μειωνω την stamina με το rate που εχω δωσει
        weAreSprinting = true;
        playerStamina -= staminaDrain * Time.deltaTime;
        UpdateStamina(1);

        //και αν τελειωσει τοτε οριζουμε την ταχυτητα του παιχτη σε χαμηλοτερη μεχρι να γεμισει παλι η μπαρα και να γινει φυσιολογικη
        if (playerStamina <= 0)
        {
            hasRegenerated = false;
            controller.SetRunSpeed(slowedRunSpeed);
            if (!_input.sprint)
                sliderCanvasGroup.alpha = 0;
        }
    }
}

//αν πηδηξει ο παιχτης χαλαει επισης stamina με τιμη που εχω δωσει
public void StaminaJump()
{
    if (playerStamina >= jumpCost)
    {
        playerStamina -= jumpCost;
        controller.PlayerJump();
        UpdateStamina(1);
        _input.jump = false;
    }
    else
    {
        _input.jump = false;
    }
}

//η συναρτηση αυτη απλα ανανεωνει την μπαρα στο UI
public void UpdateStamina(int value)
{
    staminaProgressUI.fillAmount = playerStamina / maxStamina;
    if (value == 0)
    {
        sliderCanvasGroup.alpha = 0;
    }
    else
    {
        sliderCanvasGroup.alpha = 1;
    }
}
```

Παράρτημα 3

```
private void JobForItemGrabbed()
    {
        switch (moveToObjectTag)
        {
            case "MeleeWeapons":
                WeaponManager.current.EnableCarryWeapon("Melee");
                visualWeapons[0].SetActive(true);
                moveToHit.transform.SetParent(WeaponsSlot.transform);
                moveToHit.transform.gameObject.SetActive(false);
                break;
            case "Pistol":
                WeaponManager.current.EnableCarryWeapon("Pistol");
                visualWeapons[1].SetActive(true);
                moveToHit.transform.SetParent(WeaponsSlot.transform);
                weapons[0].SetActive(true);
                PistolHandler.current.SetCurrentAmmo(moveToHit.transform.GetComponent<AmmoInside>().GetPistolCurrentAmmo());
                PistolHandler.current.SetTotalAmmo(moveToHit.transform.GetComponent<AmmoInside>().GetPistolTotalAmmo());
                PistolHandler.current.UpdateText(true);
                moveToHit.transform.gameObject.SetActive(false);
                weapons[0].SetActive(false);
                break;
            case "Rifle":
                WeaponManager.current.EnableCarryWeapon("Rifle");
                visualWeapons[2].SetActive(true);
                moveToHit.transform.SetParent(WeaponsSlot.transform);
                weapons[1].SetActive(true);
                RifleHandler.current.SetCurrentAmmo(moveToHit.transform.GetComponent<AmmoInside>().GetRifleCurrentAmmo());
                RifleHandler.current.SetTotalAmmo(moveToHit.transform.GetComponent<AmmoInside>().GetRifleTotalAmmo());
                RifleHandler.current.UpdateText(true);
                moveToHit.transform.gameObject.SetActive(false);
                weapons[1].SetActive(false);
                break;
            case "Ammo":
                if (WeaponManager.current.CheckCarryWeaponState("Pistol") && WeaponManager.current.CheckCarryWeaponState("Rifle"))
                {
                    Destroy(moveToObject.gameObject);
                    EnableWeapons();
                    PistolHandler.current.RefillAmmo();
                    RifleHandler.current.RefillAmmo();
                    DisableWeapons();
                    break;
                }
                else if (WeaponManager.current.CheckCarryWeaponState("Pistol") && !WeaponManager.current.CheckCarryWeaponState("Rifle"))
                {
                    Destroy(moveToObject.gameObject);
                    EnableWeapons();
                    PistolHandler.current.RefillAmmo();
                    DisableWeapons();
                    break;
                }
                else if (!WeaponManager.current.CheckCarryWeaponState("Pistol") && WeaponManager.current.CheckCarryWeaponState("Rifle"))
                {
                    EnableWeapons();
                    RifleHandler.current.RefillAmmo();
                    DisableWeapons();
                    break;
                }
                else
                {
                    Destroy(moveToObject.gameObject);
                    break;
                }
        }
        moveToObject = null;
    }
```

Παράρτημα 4

```
public class StarterAssetsInputs : MonoBehaviour
    {
        [Header("Character Input Values")]
        public Vector2 move;
        public Vector2 look;
        public bool jump;
        public bool sprint;
        public bool meleeAttack;
        public bool aim;
        public bool shoot;
        public bool reload;
        public bool dragFists;
        public bool dragMeleeWeapon;
        public bool dragPistol;
        public bool dragRifle;
        public bool grabItem;
        public bool throwItem;
        public bool pauseGame;
        [SerializeField] GameObject pauseMenu;
        [SerializeField] GameObject gameOverMenu;

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

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

#if ENABLE_INPUT_SYSTEM && STARTER_ASSETS_PACKAGES_CHECKED
        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);
        }
        public void OnMeleeAttack(InputValue value)
        {
            MeleeAttackInput(value.isPressed);
        }
        public void OnAim(InputValue value)
        {
            AimInput(value.isPressed);
        }
        public void OnShoot(InputValue value)
        {
            ShootInput(value.isPressed);
        }
        public void OnReload(InputValue value)
        {
            ReloadInput(value.isPressed);
        }
        public void OnDragFists(InputValue value)
        {
            DragFistsInput(value.isPressed);
        }
        public void OnDragMeleeWeapon(InputValue value)
        {
            DragMeleeWeaponInput(value.isPressed);
        }
        public void OnDragPistol(InputValue value)
        {
            DragPistolInput(value.isPressed);
        }
        public void OnDragRifle(InputValue value)
        {
            DragRifleInput(value.isPressed);
        }
        public void OnGrabItem(InputValue value)
        {
            GrabItemInput(value.isPressed);
        }
        public void OnThrowItem(InputValue value)
        {
            ThrowItemInput(value.isPressed);
        }
        public void OnPauseItem(InputValue value)
        {
            PauseItemInput(value.isPressed);
        }
#endif
```

Παράρτημα 5

```
    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;
    }
    public void MeleeAttackInput(bool newMeleeAttackState)
    {
        meleeAttack = newMeleeAttackState;
    }
    public void AimInput(bool newAimState)
    {
        aim = newAimState;
    }
    public void ShootInput(bool newShootState)
    {
        shoot = newShootState;
    }
    public void ReloadInput(bool newReloadState)
    {
        reload = newReloadState;
    }
    public void DragFistsInput(bool newDragFistsState)
    {
        dragFists = newDragFistsState;
    }
    public void DragMeleeWeaponInput(bool newDragMeleeWeaponState)
    {
        dragMeleeWeapon = newDragMeleeWeaponState;
    }
    public void DragPistolInput(bool newDragPistolState)
    {
        dragPistol = newDragPistolState;
    }
    public void DragRifleInput(bool newDragRifleState)
    {
        dragRifle = newDragRifleState;
    }
    public void GrabItemInput(bool newGrabItemState)
    {
        grabItem = newGrabItemState;
    }
    public void ThrowItemInput(bool newThrowItemState)
    {
        throwItem = newThrowItemState;
    }
    public void PauseItemInput(bool newPauseItemState)
    {
        pauseGame = newPauseItemState;
    }

    public bool fistsInputOnly()
    {
        return dragFists && !dragMeleeWeapon && !dragPistol && !dragRifle;
    }
    public bool meleeInputOnly()
    {
        return !dragFists && dragMeleeWeapon && !dragPistol && !dragRifle;
    }
    public bool pistolInputOnly()
    {
        return !dragFists && !dragMeleeWeapon && dragPistol && !dragRifle;
    }
    public bool rifleInputOnly()
    {
        return !dragFists && !dragMeleeWeapon && !dragPistol && dragRifle;
    }

    private void OnApplicationFocus(bool hasFocus)
    {
        if(pauseMenu.activeSelf || gameOverMenu.activeSelf)
        {
        SetCursorState(!cursorLocked);
        }
        else
        {
        SetCursorState(cursorLocked);
        }
    }

    public void DisableActiveInputs()
    {
        jump = sprint = meleeAttack = aim = shoot = reload = dragFists = dragMeleeWeapon = dragPistol = dragRifle = grabItem = throwItem = pauseGame = false;
    }

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

Παράρτημα 6

```
public void DealDamage(GameObject target, string tag, Vector3 contactPoint)
{
    //απλα κραταω απο το string μονο το κομματι που λεει ποιο μερος του σωματος δεχτηκε ζημια ωστε να εφαρμοσω αναλογη ζημια
    string prefix = "mixamorig:";
    string bodyPart = target.name;

    if (bodyPart.StartsWith(prefix))
    {
        bodyPart = bodyPart[prefix.Length..];
    }

    //παιρνω τον συντελεστη αναλογα το σημειο σωματος που χτυπησε
    float multiplier = GetMultiplier(bodyPart);
    target = target.transform.root.gameObject;
    var atm = target.GetComponent<AttributesManager>();
    if (atm != null)
    {
        float totalFistDamage = fistAttack;
        float totalMeleeDamage = MeleeAttack;
        float totalPistolDamage = pistolAttack;
        float totalRifleDamage = rifleAttack;
        bool crit = CheckCrit();
        //Αν ειναι crit attack προσαρμοζω το damage
        if (crit)
        {
            totalFistDamage *= critFistAttack;
            totalMeleeDamage *= critMeleeAttack;
            totalPistolDamage *= critPistolAttack;
            totalRifleDamage *= critRifleAttack;
        }
        switch (tag)
        {
            case "LeftFist":
                atm.TakeDamage((int)(totalFistDamage * multiplier), crit, tag, contactPoint, target);
                break;
            case "RightFist":
                atm.TakeDamage((int)(totalFistDamage * multiplier), crit, tag, contactPoint, target);
                break;
            case "MeleeWeapons":
                atm.TakeDamage((int)(totalMeleeDamage * multiplier), crit, tag, contactPoint, target);
                break;
            case "Pistol":
                atm.TakeDamage((int)(totalPistolDamage * multiplier), crit, tag, contactPoint, target);
                break;
            case "Rifle":
                atm.TakeDamage((int)(totalRifleDamage * multiplier), crit, tag, contactPoint, target);
                break;
        }
    }
}
```

Παράρτημα 7

```
public void TakeDamage(int amount, bool isCrit, string weaponTag, Vector3 contactPoint, GameObject target = null)
{
    health -= amount;
    Vector3 randomness = new(Random.Range(0f, 0.25f), Random.Range(0f, 0.25f), Random.Range(0f, 0.25f));

    if (health <= 0)
    {
        //αν πεθανει εχθρος
        if (target.tag == "Enemy" && !isDeadZombie)
        {
            SpawnManager.current.KillOneEnemy();
            sounds.OnDie();
            GetComponent<Ragdoll>().EnableRagdoll(contactPoint);
            isDeadZombie = true;
            Destroy(healthSystem);
            Destroy(healthBarUI);
            Destroy(this.GetComponent<Locatable>());
            return;
        }
        //αν πεθανει ο παιχτης
        else if (target.tag == "Player" && !isDeadPlayer)
        {
            GetComponent<Ragdoll>().EnableRagdoll(contactPoint);
            isDeadPlayer = true;
            StartCoroutine(StopGameAfterDie());
        }
    }
    //αλλιως στους εχθρους βγαζει ποσο ζημια δεχτηκε
    else if (target.tag == "Enemy" && !isDeadZombie)
    {
        CheckForDizz(weaponTag);
        healthSystem.AddToCurrentHealth(-amount);
        sounds.OnDamaged();
        if (isCrit)
        {
            DamagePopUpGenerator.current.CreatePopUp(transform.position + randomness, amount.ToString(), Color.red);
        }
        else
        {
            DamagePopUpGenerator.current.CreatePopUp(transform.position + randomness, amount.ToString(), Color.yellow);
        }
    }
}
```

Παράρτημα 8

```
void WaveHandling()
{
    //τα πρωτα 10 κυματα ειναι fixed, μετα ειναι στανταρ που απλα δυναμωνουν οι εχθροι
    if (currentWave <= 10)
    {
        switch (currentWave)
        {
            case 1:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 1));
                break;
            case 2:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 2));
                break;
            case 3:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 3));
                SpawnWeapon(axe, 120f);
                StartHealthInterval();
                break;
            case 4:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 6));
                break;
            case 5:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 2));
                StartCoroutine(SpawnZombie(baseEnemyStrength, crawlerZombie, 2));
                StartCoroutine(SpawnZombie(baseEnemyStrength, bossZombie, 1));
                SpawnWeapon(pistol, 120f);
                StartAmmoInterval();
                break;
            case 6:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 8));
                StartCoroutine(SpawnZombie(baseEnemyStrength, crawlerZombie, 2));
                break;
            case 7:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 6));
                StartCoroutine(SpawnZombie(baseEnemyStrength, crawlerZombie, 1));
                StartCoroutine(SpawnZombie(baseEnemyStrength, bigZombie, 1));
                SpawnWeapon(rifle, 120f);
                break;
            case 8:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 8));
                StartCoroutine(SpawnZombie(baseEnemyStrength, crawlerZombie, 3));
                StartCoroutine(SpawnZombie(baseEnemyStrength, bigZombie, 2));
                break;
            case 9:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 10));
                StartCoroutine(SpawnZombie(baseEnemyStrength, crawlerZombie, 5));
                StartCoroutine(SpawnZombie(baseEnemyStrength, bigZombie, 3));
                break;
            case 10:
                StartCoroutine(SpawnZombie(baseEnemyStrength, copZombie, 10));
                StartCoroutine(SpawnZombie(baseEnemyStrength, crawlerZombie, 10));
                StartCoroutine(SpawnZombie(baseEnemyStrength, bigZombie, 5));
                StartCoroutine(SpawnZombie(baseEnemyStrength, bossZombie, 1));
                break;
        }
    }
    else
    {
        //μετα το wave 10 βγαινουν και τα 3 ειδη εχθρων με random αριθμο(8-13) και σε καθε wave αυξανετε η δυναμη τους(ζωη και ζημια) κατα 0.1
        int numEnemies = Random.Range(8, 13);
        float enemyStrength = (float)(baseEnemyStrength + (currentWave - 10) * 0.1);
        StartCoroutine(SpawnZombie(enemyStrength, copZombie, numEnemies));
        StartCoroutine(SpawnZombie(enemyStrength, crawlerZombie, numEnemies));
        StartCoroutine(SpawnZombie(enemyStrength, bigZombie, numEnemies - 5));
        //καθε 5 wave συνεχιζει να βγαινει και ενας αρχηγος ζομπιε
        if (currentWave % 5 == 0)
        {
            StartCoroutine(SpawnZombie(enemyStrength, bossZombie, 1));
            //υπαρχει και 30% πιθανοτητα να κανει spawn πιστολακι ή 20% rifle
            if (Random.Range(0, 100) < 30)
            {
                SpawnWeapon(pistol, 60f);
            }
            if (Random.Range(0, 100) < 20)
            {
                SpawnWeapon(rifle, 60f);
            }
        }
    }
}
```

Παράρτημα 9

```
private void Awake()
{
    TryGetComponent<AudioSource>(out audioSource);
}

private void Update()
{
    //το patrol sound δεν πρεπει να ξαναπαιξει πριν περασουν 5 δευτερολεπτα
    if (isPlayingPatrolSound)
    {
        timerToPatrolSound += Time.deltaTime;
        if (timerToPatrolSound >= 5f)
        {
            isPlayingPatrolSound = false;
            timerToPatrolSound = 0f;
        }
    }
}

//οι παρακατω συναρτησεις ειναι για χρηση animation event
//παιζουν με ενταση αναλογα την αποσταση απο τον παιχτη
//καποια διαλεγουν τυχαια ενα clip που θα παιχτει
private void OnPatrolling(AnimationEvent animationEvent)
{
    if (!isPlayingPatrolSound)
    {
        if (animationEvent.animatorClipInfo.weight > 0.5f)
        {
            float distance = Vector3.Distance(GameObject.FindWithTag("Player").transform.position, transform.position);
            float volume = Mathf.Clamp01(1.0f - (distance / maxDistance));
            volume = Mathf.Max(volume, minVolume);
            int randomClipIndex = Random.Range(0, zombiePatrolling.Length);
            audioSource.PlayOneShot(zombiePatrolling[randomClipIndex], volume);
            isPlayingPatrolSound = true;
        }
    }
}
private void OnAngry(AnimationEvent animationEvent)
{
    if (animationEvent.animatorClipInfo.weight > 0.5f)
    {
        float distance = Vector3.Distance(GameObject.FindWithTag("Player").transform.position, transform.position);
        float volume = 1.0f - Mathf.Clamp01(distance / maxDistance);
        volume = Mathf.Max(volume, minVolume);
        audioSource.PlayOneShot(zombieAngry, volume);
    }
}
public void OnDamaged()
{
    float distance = Vector3.Distance(GameObject.FindWithTag("Player").transform.position, transform.position);
    float volume = 1.0f - Mathf.Clamp01(distance / maxDistance);
    volume = Mathf.Max(volume, minVolume);
    int randomClipIndex = Random.Range(0, zombieDamaged.Length);
    audioSource.PlayOneShot(zombieDamaged[randomClipIndex], volume);
}
private void OnAttackLight(AnimationEvent animationEvent)
{
    if (animationEvent.animatorClipInfo.weight > 0.5f && !audioSource.isPlaying)
    {
        {
            audioSource.PlayOneShot(zombieAttackLight);
        }
    }
}

private void OnAttackStrong(AnimationEvent animationEvent)
{

    if (animationEvent.animatorClipInfo.weight > 0.5f && !audioSource.isPlaying)
    {
        {
            audioSource.PlayOneShot(zombieAttackStrong);
        }
    }
}

public void OnDie()
{
    float distance = Vector3.Distance(GameObject.FindWithTag("Player").transform.position, transform.position);
    float volume = 1.0f - Mathf.Clamp01(distance / maxDistance);
    volume = Mathf.Max(volume, minVolume);
    audioSource.PlayOneShot(zombieDie, volume);
}
```

Παράρτημα 10