



ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή Εργασία

Τίτλος Πτυχιακής Εργασίας	Ο Κυνηγός Κλειδιών των Κρυμμένων Δωματίων The Keyhunter of Hidden Chambers
Ονοματεπώνυμο Φοιτητή	Μαφούτης Σωτήριος
Πατρώνυμο	Ηλίας
Αριθμός Μητρώου	Π22101
Επιβλέπων	Ευθύμιος Αλέπης, Καθηγητής

Copyright ©

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν αποκλειστικά τον συγγραφέα και δεν αντιπροσωπεύουν τις επίσημες θέσεις του Πανεπιστημίου Πειραιώς.

Ευχαριστίες

Με την ολοκλήρωση της παρούσας πτυχιακής εργασίας, θα ήθελα να ευχαριστήσω όλους εκείνους που με την υποστήριξή τους συνέβαλαν στην επιτυχή ολοκλήρωση αυτού του έργου.

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου, ο οποίος ανέλαβε την επίβλεψη της πτυχιακής μου εργασίας και με καθοδήγησε καθ' όλη τη διάρκεια της διαδικασίας. Η συνεχής υποστήριξη και οι διορατικές του συμβουλές σε κάθε στάδιο υπήρξαν εξαιρετικά σημαντικές.

Τέλος, αφιερώνω την εργασία αυτή στους γονείς μου, που με αγάπη στήριξαν κάθε βήμα της εκπαιδευτικής μου πορείας, στους φίλους μου, που περάσαμε μαζί αυτά τα χρόνια καθώς και στον σκύλο μου τον Ράστυ που με τον τρόπο του μου έδινε δύναμη κάθε στιγμή.

Περίληψη

Η εργασία αυτή αφορά την ανάπτυξη ενός διαδραστικού τρισδιάστατου βιντεοπαιχνιδιού πρώτου προσώπου (FPS) που υλοποιήθηκε στο περιβάλλον του Unity3D σε γλώσσα C# με τον τίτλο «The Keyhunter of Hidden Chambers» (Ο Κυνηγός Κλειδιών των Κρυμμένων Δωματίων). Το παιχνίδι είναι συμβατό με Windows και σχεδιάστηκε για να προσφέρει μια πλήρη εμπειρία εξερεύνησης, επιβίωσης και στρατηγικής σκέψης. Σε ένα εγκαταλελειμμένο βιομηχανικό συγκρότημα, ένα μυστικό εργαστήριο έχει δημιουργήσει πειράματα που έχουν ξεφύγει από κάθε έλεγχο. Το συγκρότημα, που κάποτε αποτελούσε κέντρο επιστημονικής έρευνας, έχει μετατραπεί σε ένα σκοτεινό λαβύρινθο γεμάτο επικίνδυνες οντότητες που φυλάσσουν τα μυστικά του. Ο ήρωας, ένας έμπειρος κυνηγός κλειδιών με ειδικευση στην εξερεύνηση επικίνδυνων χώρων, αναλαμβάνει την τελευταία αποστολή του: να εισέλθει στο συγκρότημα, να εξοντώσει τις απειλές και να ανακτήσει τα τρία κλειδιά που ξεκλειδώνουν την αλήθεια πίσω από τις κλειδωμένες πόρτες. Η μόνη διαφυγή βρίσκεται πίσω από την τελική πόρτα, αλλά εκεί τον περιμένει ο πιο επικίνδυνος εχθρός απ' όλους.

Ο παίκτης αναλαμβάνει τον ρόλο αυτού του κυνηγού κλειδιών, εξερευνά δύο επίπεδα, συλλέγει τρία κλειδιά συνολικά (2 κανονικά + 1 μυστικό), αντιμετωπίζει εχθρούς με τεχνητή νοημοσύνη, αναβαθμίζει τα όπλα μέσω συστήματος εμπειρίας (XP) και στο τέλος αντιμετωπίζει τον Final Boss.

Υλοποιήθηκαν τα εξής συστήματα: σύστημα 5 όπλων με εξέλιξη μέσω XP, σύστημα κλειδιών με Secret Room και 30-δευτερόλεπτο timer, inventory για εναλλαγή όπλων, ασπίδα (shield) που μειώνει το damage, medikit για αποκατάσταση υγείας, δυναμική μουσική που αλλάζει ανάλογα με την κατάσταση (εξερεύνηση/μάχη/Final Boss), ήχοι reload ανά όπλο, πλήρες UI (HealthBar, ShieldBar, AmmoDisplay, XP Bar, Key Counter) και ολοκληρωμένο Main Menu με Settings (Volume, Brightness, Sensitivity).

Λέξεις Κλειδιά: Unity3D, C#, FPS, NavMesh, State Machine, Weapon System, Key System, Inventory, Game Development, Artificial Intelligence

Abstract

This work involves the development of an interactive three-dimensional first-person shooter (FPS) video game implemented in the Unity3D environment using the C# programming language, titled "The Keyhunter of Hidden Chambers." The game is compatible with Windows and was designed to offer a complete experience of exploration, survival and strategic thinking. In an abandoned industrial complex, a secret laboratory has created experiments that have spiraled out of control. The facility, once a center of scientific research, has been transformed into a dark labyrinth filled with dangerous entities guarding its secrets. The hero, an experienced key hunter specializing in the exploration of hazardous locations, takes on his final mission: to enter the complex, eliminate the threats and recover the three keys that unlock the truth behind the locked doors. The only way out lies behind the final door but there, the most dangerous enemy of all awaits.

The player takes on the role of this key hunter, exploring two levels, collecting three keys in total (2 normal + 1 secret), facing enemies with artificial intelligence, upgrading weapons through an experience point (XP) system, and ultimately confronting the Final Boss.

The following systems were implemented: a 5-weapon system with XP-based progression, a key collection system with a Secret Room and 30-second timer, an inventory system for weapon switching, a shield system that reduces incoming damage, a medikit for health restoration, dynamic music that changes according to the game state (exploration/combat/Final Boss), per-weapon reload sounds, a complete UI (HealthBar, ShieldBar, AmmoDisplay, XP Bar, Key Counter) and a fully featured Main Menu with Settings (Volume, Brightness, Sensitivity).

Περιεχόμενα

Copyright ©.....	2
Ευχαριστίες.....	3
Περίληψη.....	4
Abstract.....	4
Εισαγωγή.....	9
1.1 Ιστορική Αναδρομή στο Game Development.....	9
1.2 Κίνητρο και Στόχοι.....	9
1.3 Δομή της Εργασίας.....	10
2. Έμπνευση.....	11
2.1 Παρόμοια Παιχνίδια.....	11
Doom Eternal (2020).....	11
Counter-Strike 1.6 (2000).....	11
Crash of the Titans (2007).....	11
Σύγκριση Παιχνιδιών.....	11
2.2 Εργαλεία Ανάπτυξης.....	12
Game Engine: Unity 2022.3.33f1.....	12
Γλώσσα Προγραμματισμού: C#.....	12
Σημαντικά Packages (Unity Package Manager).....	12
3. Overview Παιχνιδιού.....	12
3.1 Υπόθεση.....	12
Ιστορία.....	12
3.2 Μηχανισμοί Παίκτη.....	13
3.3 Σύστημα Υγείας και Ασπίδας.....	13
3.4 Σύστημα XP και Όπλων.....	14
3.5 Σύστημα Κλειδιών.....	16
3.6 Διαγράμματα.....	16
Use Case Diagram — Main Menu.....	16
Flow Diagram — Διαδικασία Πυροβολισμού.....	17
4. Scenes.....	19
4.1 Δομή Scenes.....	19
4.2 Main Menu.....	19
4.3 Level 1 (L1).....	20
4.4 Level 2 (L2).....	20
4.5 Level 3 - Final (L3).....	20

5. Animation / Animator	21
5.1 Βασικά Στοιχεία	21
5.2 Animator Εχθρού	21
5.3 Animator Final Boss	22
5.4 Animation Πόρτας	23
5.5 Animation Πιστολιού	24
6. Σημαντικά Scripts Παίκτη	25
Singleton και DontDestroyOnLoad	25
Νέο Input System	25
Κίνηση και Κάμερα	26
Σύστημα Pause	27
Μετάβαση στο Main Menu	28
Διαχείριση Scene Loading	28
6.2 PlayerMotor.cs	29
Αρχικοποίηση και Layer Collision	29
Σύστημα Κίνησης	30
Σύστημα Βαρύτητας και Άλματος	31
Σύστημα Sprint	31
Σύστημα Crouch	31
Διαχείριση Επαφής με Εχθρούς	32
6.3 PlayerLook.cs	33
Αρχικοποίηση Κάμερας	33
Real-time Sensitivity	33
Σύστημα Κίνησης Κάμερας	34
6.4 PlayerHealth.cs	34
Αρχικοποίηση και Scene Loading	35
Δυναμική Εύρεση UI References	35
Σύστημα Υγείας με Chip Damage Effect	36
Σύστημα Ασπίδας	38
Θάνατος Παίκτη	39
Αποκατάσταση Υγείας	40
6.5 PlayerShooting.cs	40
Αρχικοποίηση	40
Update — Λογική Πυροβολισμού	41
HandleHit — Σύστημα Ζημιάς	42
Shoot() — Πυροβολισμός Κανονικών Όπλων	43
ShootShotgun() — Πυροβολισμός Shotgun	44
SpawnBullet() — Οπτική Αναπαράσταση	45
ResetAmmo() και UpdateAmmoUI()	45
Reload() — Σύστημα Επαναγέμισης	46
RefillAmmo() - Αναπλήρωση Πυρομαχικών	47
OnSceneResetAmmo()	48

6.6 WeaponSystem.cs	48
Αρχικοποίηση	48
Σύστημα XP και Αναβάθμισης	49
EquipWeapon()	49
ResetXPToLevelStart() — Επαναφορά XP	50
Βοηθητικές Μέθοδοι	51
6.7 WeaponHandler.cs	51
[ExecuteAlways]	51
Οργάνωση με Headers	52
ApplyOffsets()	52
Σύστημα Ήχου	53
PlayShotgunReload()	53
Animations	54
6.8 InventorySystem.cs	54
Αρχικοποίηση και DontDestroyOnLoad	54
Δυναμική Εύρεση UI References	55
Άνοιγμα και Κλείσιμο Inventory	56
RefreshInventory()	56
7. Σύστημα Εχθρών	59
7.1 State Machine	59
7.2 PatrolState.cs	60
Enter()	60
Perform()	60
PatrolCycle()	60
FaceMovementDirection()	61
Exit() - Έξοδος από την Κατάσταση	61
7.3 AttackState.cs	61
Enter()	62
Perform()	62
Shoot()	63
7.4 Enemy.cs	63
Start() – InitializeEnemy()	63
Update()	64
CanSeePlayer()	65
OnHit()	66
UpdateAnimations()	66
7.5 EnemyHealth.cs	67
TakeDamage()	67
7.6 FinalBoss.cs	68
Awake()	68
Update()	68
PerformAttack()	69

EnterEnragedMode()	70
TakeDamage()	70
Die()	71
8. User Interface (UI)	72
8.1 Γενική Δομή	72
8.2 Health & Shield Bar	72
8.3 Ammo Display	73
8.4 Inventory Panel	73
8.5 Pause Menu	73
8.6 Settings	74
8.7 Game Over & Win Screen	74
9. Σύστημα Μουσικής και Ήχου	75
9.1 MusicManager.cs	75
Δυναμική Αλλαγή Μουσικής Εχθρών	76
Σύστημα Μουσικής Final Boss	76
FadeToNewMusic()	77
ResetToExploration()	77
9.2 Ήχοι Όπλων	77
9.3 Ήχοι Final Boss	78
10. Διαχείριση Επιπέδων	79
10.1 LevelManager.cs	79
Start()	79
Παρακολούθηση Εχθρών και Κλειδιών	79
CheckLevelComplete() και WaitBeforeLoadCoroutine()	79
Secret Room Timer	79
10.2 KeyPickup.cs	80
Τύποι Κλειδιών	80
Interact()	80
10.3 SecretRoomTrigger.cs	81
11. Μελλοντικά Implementations	82
11.1 Επέκταση Περιεχομένου	82
11.2 Τεχνικές Βελτιώσεις	82
11.3 Multiplayer	82
12. Ορολογία	84
12.1 Εργαλεία	84
12.2 Παιχνίδι	84
13. Συμπεράσματα	85
14. Βιβλιογραφία	86

Εισαγωγή

Η εκπόνηση μιας πτυχιακής εργασίας αποτελεί ένα σημαντικό στάδιο στην ολοκλήρωση των σπουδών ενός φοιτητή, καθώς του δίνει τη δυνατότητα να εφαρμόσει στην πράξη γνώσεις και δεξιότητες που απέκτησε κατά τη διάρκεια των σπουδών του. Παράλληλα, αποτελεί μια ευκαιρία για ενασχόληση με ένα αντικείμενο που παρουσιάζει ιδιαίτερο ενδιαφέρον και μπορεί να συμβάλει στην περαιτέρω ανάπτυξη των γνώσεων και των επαγγελματικών του δεξιοτήτων.

Στην παρούσα πτυχιακή εργασία, το αντικείμενο της μελέτης και της υλοποίησης αφορά την ανάπτυξη ενός ψηφιακού παιχνιδιού. Η επιλογή του συγκεκριμένου θέματος έγινε λόγω του ιδιαίτερου ενδιαφέροντος που παρουσιάζει ο τομέας του game development, καθώς συνδυάζει τον προγραμματισμό με στοιχεία σχεδιασμού, αλληλεπίδρασης και δημιουργίας ενός ολοκληρωμένου ψηφιακού περιβάλλοντος.

Κατά τη διάρκεια της ανάπτυξης του παιχνιδιού αξιοποιήθηκαν σύγχρονα εργαλεία και τεχνολογίες, με στόχο τη δημιουργία μιας λειτουργικής και ολοκληρωμένης εφαρμογής. Η διαδικασία περιλάμβανε τον σχεδιασμό του παιχνιδιού, την ανάπτυξη των βασικών μηχανισμών του, την υλοποίηση της αλληλεπίδρασης του παίκτη με το περιβάλλον και τους εχθρούς, καθώς και τη συνεχή δοκιμή και βελτίωση της εφαρμογής.

Μέσα από την υλοποίηση της συγκεκριμένης εργασίας, επιδιώκεται η καλύτερη κατανόηση της διαδικασίας ανάπτυξης ενός ψηφιακού παιχνιδιού, από την αρχική ιδέα μέχρι την τελική του μορφή. Παράλληλα, η εργασία αυτή αποτέλεσε μια ευκαιρία για πρακτική εφαρμογή γνώσεων προγραμματισμού και εξοικείωση με τις απαιτήσεις και τις προκλήσεις που μπορεί να προκύψουν κατά την ανάπτυξη ενός πραγματικού έργου.

1.1 Ιστορική Αναδρομή στο Game Development

Η ιστορία του game development ξεκίνησε τη δεκαετία του 1950. Το "Tennis for Two" (1958) του William Higinbotham θεωρείται ένα από τα πρώτα παιχνίδια. Ακολούθησε το "Spacewar!" (1962) και το "Pong" (1972) της Atari. Η δεκαετία του 1990 έφερε τα πρώτα 3D game engines — το Doom Engine (1993) και το Unreal Engine (1998).

Το Unity δημιουργήθηκε το 2005 από τους David Helgason, Nicholas Francis και Joachim Ante. Αρχικά μόνο για macOS, σύντομα έγινε cross-platform και σήμερα χρησιμοποιείται για πάνω από το 50% των παιχνιδιών παγκοσμίως.

1.2 Κίνητρο και Στόχοι

Το κύριο κίνητρο που με ώθησε στη δημιουργία αυτής της πτυχιακής ήταν η βαθιά αγάπη μου για τα video games και η επιθυμία να δημιουργήσω κάτι δικό μου — ένα πλήρες παιχνίδι που να συνδυάζει εξερεύνηση, μάχη και στρατηγική σκέψη. Θέλησα να μάθω πώς λειτουργούν από μέσα τα παιχνίδια που αγαπώ και να αποκτήσω πρακτική εμπειρία στον τομέα του game development.

Ο στόχος της πτυχιακής μου, εκτός από το τελικό αποτέλεσμα που ήθελα να επιτύχω, ήταν να αναπτύξω τις δεξιότητές μου σε όλα τα στάδια της δημιουργίας ενός ηλεκτρονικού παιχνιδιού. Αυτά τα στάδια περιλαμβάνουν πολλές και πολύπλοκες διαδικασίες, όπως το 3D

modelling, το animation, το level designing και το scripting, τα οποία απαιτούν εξειδίκευση. Οι στόχοι αυτοί ήταν βασικοί, και έπρεπε, σε κάποιο βαθμό, να ανταποκριθώ στις προκλήσεις που θα προέκυπταν. Άλλωστε, σε μια μεγάλη παραγωγή, κάθε ένας από αυτούς τους τομείς ανατίθεται σε εξειδικευμένες ομάδες με σημαντική εμπειρία και εξειδίκευση, καθώς θεωρούνται διακριτοί και αυτόνομοι ρόλοι.

1.3 Δομή της Εργασίας

Η δομή της εργασίας παρουσιάζεται παρακάτω:

- Στο πρώτο κεφάλαιο παρουσιάζεται η εισαγωγή της εργασίας, όπου αναλύονται με λίγα λόγια τόσο τα κίνητρα όσο και ο σκοπός της εργασίας, καθώς και μια σύντομη ιστορική αναδρομή στο game development και το Unity.
- Στο δεύτερο κεφάλαιο παρουσιάζεται η υπόθεση του παιχνιδιού, όπου αναλύεται το backstory του συγκροτήματος, η ιστορία του πρωταγωνιστή και η φύση των εχθρών που αντιμετωπίζει.
- Στο τρίτο κεφάλαιο γίνεται ένα overview του παιχνιδιού, όπου αναλύονται οι μηχανισμοί του παίκτη, το σύστημα υγείας και ασπίδας, το σύστημα XP και εξέλιξης όπλων, καθώς και το σύστημα κλειδιών με το Secret Room.
- Στο τέταρτο κεφάλαιο παρατίθενται τα scenes του παιχνιδιού — το Main Menu, τα δύο κανονικά επίπεδα και το τελικό επίπεδο με τον Final Boss.
- Στο πέμπτο κεφάλαιο γίνεται αναφορά στον τομέα των Animations και του Animator, αναλύοντας τα animation states των εχθρών, του Final Boss και των πορτών.
- Στο έκτο κεφάλαιο αναλύονται τα βασικά scripts του παιχνιδιού, από τον έλεγχο του παίκτη μέχρι τη διαχείριση εχθρών, όπλων και managers.
- Στο έβδομο κεφάλαιο παρουσιάζεται το σύστημα εχθρών με ανάλυση της τεχνητής νοημοσύνης μέσω State Machine και NavMesh, καθώς και τα χαρακτηριστικά του Final Boss.
- Στο όγδοο κεφάλαιο αναλύεται το User Interface του παιχνιδιού, περιλαμβάνοντας τις μπάρες υγείας, ασπίδας, XP, το inventory και το pause menu.
- Στο ένατο κεφάλαιο παρουσιάζεται το σύστημα μουσικής και ήχου, με ανάλυση της δυναμικής αλλαγής μουσικής και των ήχων των όπλων.
- Στο δέκατο κεφάλαιο παρουσιάζεται το σύστημα διαχείρισης επιπέδων.
- Στο ενδέκατο κεφάλαιο πραγματεύονται μελλοντικές εφαρμογές και implementations που μπορούν να προστεθούν στο παιχνίδι.
- Στο δωδέκατο κεφάλαιο περιέχεται η ορολογία που χρησιμοποιείται στην εργασία.
- Στο δέκατο τρίτο κεφάλαιο γίνεται αναφορά στα τελικά συμπεράσματα της εργασίας.
- Στο Δέκατο τέταρτο και τελευταίο κεφάλαιο περιέχεται η βιβλιογραφία.

2. Έμπνευση

2.1 Παρόμοια Παιχνίδια

Doom Eternal (2020)

Το Doom Eternal είναι ένα first-person shooter με εξαιρετικά γρήγορο ρυθμό gameplay. Ο παίκτης — ο Doom Slayer — αντιμετωπίζει ορδές δαιμόνων ανά επίπεδο, εξελίσσοντας τον οπλισμό του μέσω upgrades και XP. Ο βρόχος gameplay «σκοτώνω εχθρό έπειτα αποκτώ XP και αναβαθμίζομαι» ενέπνευσε άμεσα το σύστημα εξέλιξης 5 όπλων του Keyhunter. Επίσης, η σταδιακή αύξηση δυσκολίας ανά επίπεδο και η ποικιλία εχθρών με διαφορετικές αδυναμίες αντικατοπτρίζεται στον σχεδιασμό των κανονικών εχθρών και του Final Boss.

Counter-Strike 1.6 (2000)

Το Counter-Strike 1.6 (Valve, 2000) είναι ένα από τα πιο εμβληματικά multiplayer first-person shooters στην ιστορία του gaming. Παρά την ηλικία του, παραμένει σε ευρεία χρήση και αποτέλεσε την πρώτη επαφή πολλών παικτών, ανάμεσά τους και εμένα με το FPS είδος. Χαρακτηρίζεται από ακριβές recoil system ανά όπλο, εύχρηστο weapon switching και αισθητή αίσθηση βάρους στους πυροβολισμούς. Ο σχεδιασμός των όπλων του Keyhunter (ξεχωριστά damage, fire rate, magazine size και fire mode ανά όπλο) εμπνεύστηκε άμεσα από αυτή τη λογική διαφοροποίησης οπλισμού. Επιπλέον η επιλογή των όπλων έγινε με βάση όπλα που υπάρχουν στο παιχνίδι (pistol, uzi, shotgun, m4, ak-47).

Crash of the Titans (2007)

Το Crash of the Titans (Radical Entertainment / Sierra Entertainment, 2007) είναι ένα action-platformer με σαφή δομή προόδου ανά επίπεδο: ο παίκτης αντιμετωπίζει κανονικούς εχθρούς (Titans και Minions) κατά μήκος κάθε επιπέδου και κάθε ενότητα καταλήγει σε μία ισχυρότερη αναμέτρηση. Αυτή η δομή «σκοτώνω απλούς εχθρούς → δυνατότερος αντίπαλος στο τέλος» αντικατοπτρίζεται άμεσα στη δομή των τριών επιπέδων του Keyhunter: L1 και L2 με κανονικούς εχθρούς, και L3 που καταλήγει στον Final Boss. Επίσης, το σύστημα XP rewards μετά από κάθε kill ενέπνευσε το αντίστοιχο σύστημα XP (+25 ανά kill) του παιχνιδιού μου.

Σύγκριση Παιχνιδιών

Στον παρακάτω πίνακα συγκρίνονται τα τρία παιχνίδια με το The Keyhunter of Hidden Chambers:

Παιχνίδι	Doom Eternal	Counter-Strike 1.6	Crash of the Titans	The Keyhunter
Perspective	1st Person	1st Person	3rd Person	1st Person
Είδος	FPS / Action	FPS / Multiplayer	Action-Platform	FPS / Horror
Εχθροί & Boss	Δαίμονες+Boss	Αντίπαλη ομάδα	Titans+Boss Titans	Εχθροί+Final Boss
Σύστημα XP	Ναι (upgrades)	Όχι	Ναι (XP/kills)	Ναι (XP/kill)
Έμπνευση	XP/όπλο, δυσκολία ανά level	Μοντέλα όπλων, reload, weapon variety	Κανονικοί εχθροί→Final Boss	-----

2.2 Εργαλεία Ανάπτυξης

Game Engine: Unity 2022.3.33f1

Για την ανάπτυξη του παιχνιδιού επιλέχθηκε η Unity 2022.3.33f1, μία από τις πιο δημοφιλείς game engines που υπάρχουν σήμερα. Η Unity χρησιμοποιείται ευρέως τόσο από ανεξάρτητους προγραμματιστές όσο και από μεγάλες εταιρείες, κάτι που σημαίνει ότι υπάρχει πλούσια τεκμηρίωση, άφθονα tutorials και μια πολύ ενεργή κοινότητα που μπορεί να βοηθήσει όταν παρουσιαστεί πρόβλημα. Η έκδοση 2022.3.x επιλέχθηκε επειδή είναι η **Long-Term Support (LTS)** έκδοση, δηλαδή η πιο σταθερή διαθέσιμη επιλογή. Σε ένα project όπως αυτό, όπου η προτεραιότητα είναι η ολοκλήρωση και όχι η χρήση των πιο πρόσφατων χαρακτηριστικών, μια LTS έκδοση είναι η λογική επιλογή - αποφεύγονται έτσι απρόσμενα bugs και αλλαγές που μπορούν να δημιουργήσουν προβλήματα στη μέση της ανάπτυξης.

Γλώσσα Προγραμματισμού: C#

Η κύρια γλώσσα scripting της Unity είναι η C# (C-Sharp), αντικειμενοστραφής γλώσσα της Microsoft. Χρησιμοποιήθηκε για: κίνηση παίκτη (PlayerMovement.cs), πυροβολισμό (PlayerShooting.cs), AI εχθρών μέσω State Machine (PatrolState.cs, AttackState.cs), UI (UIManager.cs) και διαχείριση επιπέδων (LevelManager.cs).

Σημαντικά Packages (Unity Package Manager)

Πακέτα που εγκαταστάθηκαν:

New Input System (com.unity.inputsystem): Action-based σύστημα εισόδου. InputActions asset με actions: Move, Look, Jump, Sprint, Shoot, Reload, Interact, Inventory, Pause.

AI Navigation / NavMesh (com.unity.ai.navigation): Pathfinding εχθρών μέσω NavMeshAgent. PatrolState (waypoints) / AttackState (καταδίωξη). NavMesh Obstacle Carve=true στις πόρτες.

TextMeshPro (com.unity.textmeshpro): SDF text rendering για HP/Shield bars, ammo counter, XP display, key indicators και inventory tooltips

3. Overview Παιχνιδιού

3.1 Υπόθεση

Ιστορία

Το «The Keyhunter of Hidden Chambers» εκτυλίσσεται σε ένα εγκαταλελειμμένο βιομηχανικό συγκρότημα στα περίχωρα μιας άγνωστης πόλης. Πριν από χρόνια, το συγκρότημα αυτό στέγαζε ένα από τα πιο φιλόδοξα επιστημονικά εργαστήρια της εποχής του, ένα μέρος όπου οι επιστήμονες πίστευαν ότι μπορούσαν να σπρώξουν τα όρια της ανθρώπινης γνώσης. Τα πειράματα που διεξάγονταν εκεί ήταν απόρρητα, και κανείς εκτός των τειχών δεν γνώριζε την πραγματική φύση τους.

Οι επιστήμονες του εργαστηρίου είχαν αρχίσει να πειραματίζονται με τη βιολογική τροποποίηση του ανθρώπινου σώματος. Στόχος τους ήταν να δημιουργήσουν τον τέλει στρατιώτη, έναν άνθρωπο ταχύτερο, δυνατότερο και πιο ανθεκτικό από οτιδήποτε υπήρξε ποτέ. Τα πρώτα αποτελέσματα φάνηκαν ελπιδοφόρα, αλλά σύντομα τα πράγματα βγήκαν

εκτός ελέγχου. Τα πειραματικά υποκείμενα, άνθρωποι που είχαν εθελοντικά ή μη υποβληθεί στις διαδικασίες, άρχισαν να εκδηλώνουν ανεξέλεγκτη επιθετικότητα. Ο ορθολογισμός τους χάθηκε, αλλά η σωματική τους δύναμη και τα ένστικτά τους παρέμειναν πλήρως ανεπτυγμένα.

Κάποια στιγμή, τα πειραματικά υποκείμενα εξεγέρθηκαν. Το προσωπικό σφαγιάστηκε ή εγκλωβίστηκε, και το συγκρότημα σφραγίστηκε από τις αρχές. Οι τροποποιημένοι άνθρωποι εξακολουθούν να ζουν μέσα στους διαδρόμους του κτηρίου, περιπολώντας με στρατιωτική πειθαρχία αλλά χωρίς καμία ανθρώπινη σκέψη. Μοιάζουν με ανθρώπους — κινούνται, πυροβολούν, παραμονεύουν - αλλά δεν είναι πλέον άνθρωποι με την ηθική έννοια του όρου.

Το πιο τρομακτικό από όλα βρίσκεται στο κέντρο του εργαστηρίου. Το κύριο πειραματικό υποκείμενο, ο πρώτος εθελοντής που υποβλήθηκε στη διαδικασία και το «επιτυχημένο» αποτέλεσμα των πειραμάτων, έχει μεταλλαχθεί σε κάτι που δεν μπορεί να χαρακτηριστεί πλέον ανθρώπινο. Το σώμα του έχει παραμορφωθεί ριζικά από τα πειράματα, αποκτώντας τεράστιο μέγεθος, εξωανθρώπινη δύναμη και απίστευτη αντοχή στις βλάβες.

Ο πρωταγωνιστής του παιχνιδιού είναι ένας έμπειρος κυνηγός κλειδιών που έχει κάνει επάγγελμα την εξερεύνηση επικίνδυνων εγκαταλελειμμένων χώρων. Η εντολή που λαμβάνει είναι να εισέλθει στο συγκρότημα, να εντοπίσει τρία κλειδιά και να ξεκλειδώσει την τελική πόρτα που οδηγεί στο κέντρο του εργαστηρίου. Δύο κλειδιά βρίσκονται στα πρώτα δύο επίπεδα, ενώ ένα μυστικό κλειδί φυλάσσεται σε κρυφό δωμάτιο.

Στο πρώτο επίπεδο, ο παίκτης βρίσκεται στη βιομηχανική πτέρυγα του συγκροτήματος, όπου τροποποιημένοι άνθρωποι-φρουροί περιπολούν αδιάκοπα. Στο δεύτερο επίπεδο, το συγκρότημα γίνεται πιο πολύπλοκο, οι εχθροί πιο επικίνδυνοι και η εξερεύνηση πιο απαιτητική. Σε κάθε επίπεδο υπάρχει και ένα μυστικό δωμάτιο με φρουρούς και ένα μυστικό κλειδί, αλλά ο παίκτης χρειάζεται μόνο ένα από αυτά για να ολοκληρώσει την αποστολή.

Στο τρίτο και τελευταίο επίπεδο βρίσκεται η καρδιά του εργαστηρίου. Πίσω από την κεντρική πόρτα, η οποία για να ανοιχτεί χρειάζεται και τα τρία κλειδιά βρίσκεται η δυνατότερη μετάλλαξη, ο πρώτος εθελοντής.

Η νίκη επί της μετάλλαξης σηματοδοτεί το τέλος της αποστολής. Ο κυνηγός κλειδιών ολοκληρώνει την πιο επικίνδυνη αποστολή της ζωής του, αποκαλύπτοντας την αλήθεια που κρυβόταν για τόσα χρόνια μέσα στους τοίχους του συγκροτήματος.

3.2 Μηχανισμοί Παίκτη

Ο παίκτης ελέγχεται ως εξής:

- WASD: κίνηση | Space: άλμα | Shift: sprint | Ctrl: crouch
- Mouse: περιστροφή κάμερας | Left Click: πυροβολισμός | R: reload
- E: interact | I: inventory | Escape: pause

Στατιστικά κίνησης: Walk 5 u/s, Sprint 9 u/s, Crouch 2.5 u/s, Jump Height 1.5, Gravity -9.8.

3.3 Σύστημα Υγείας και Ασπίδας

Ο παίκτης έχει 100 HP και μπορεί να αποκτήσει 100 Shield. Όταν η ασπίδα είναι ενεργή:

- Health Damage = damage $\times$ 0.5
- Shield Damage = damage $\times$ 2

Παράδειγμα: 25 damage $\rightarrow$ 12.5 HP loss + 50 Shield loss.



Εικόνα 1



Εικόνα 2

3.4 Σύστημα XP και Όπλων

Κάθε kill δίνει 25 XP. Πίνακας αναβαθμίσεων:

- Level 1 (0 XP): Πιστόλι | 25 dmg | 0.2s rate | 12/3 | Semi-auto
- Level 2 (100 XP): Uzi | 20 dmg | 0.08s rate | 30/4 | Auto
- Level 3 (250 XP): Shotgun | 50 dmg | 0.8s rate | 6/4 | 8 pellets
- Level 4 (650 XP): M4 | 30 dmg | 0.12s rate | 30/4 | Auto
- Level 5 (850XP): AK-47 | 35 dmg | 0.10s rate | 40/3 | Auto



Εικόνα 3



Εικόνα 4



Εικόνα 5



Εικόνα 6

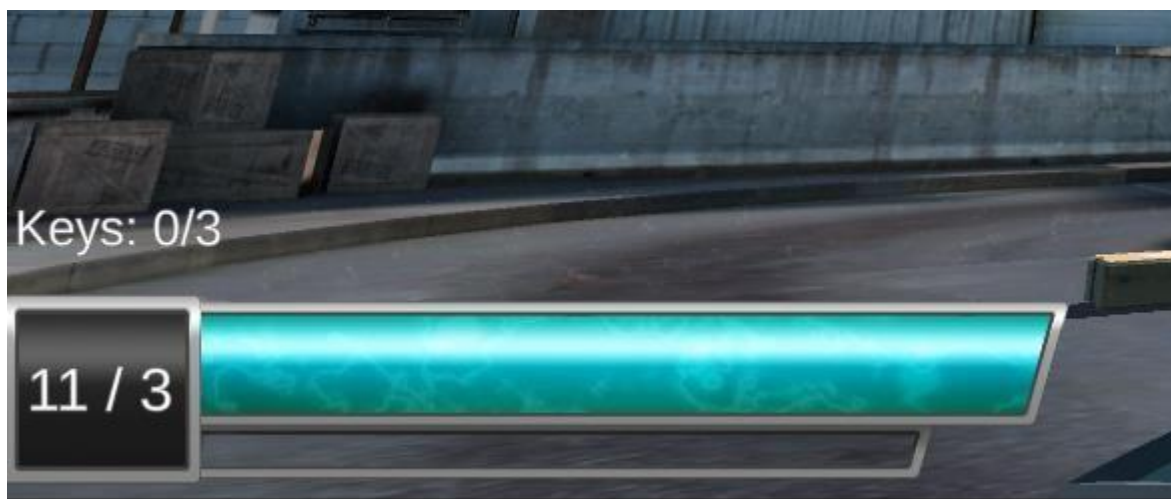


Εικόνα 7

3.5 Σύστημα Κλειδιών

Συνολικά υπάρχουν 3 κλειδιά: 2 κανονικά (L1, L2) + 1 μυστικό (Secret Room). Η τελική πόρτα απαιτεί και τα 3. Λογική μετάβασης επιπέδων:

- Σκοτώνεις όλους τους κανονικούς εχθρούς.
- Παίρνεις το κανονικό κλειδί (E).
- Προαιρετικά: μπαίνεις στο Secret Room (30-sec timer).
- Αν έχεις πάρει το μυστικό κλειδί σε προηγούμενο level, δεν εμφανίζεται ξανά.

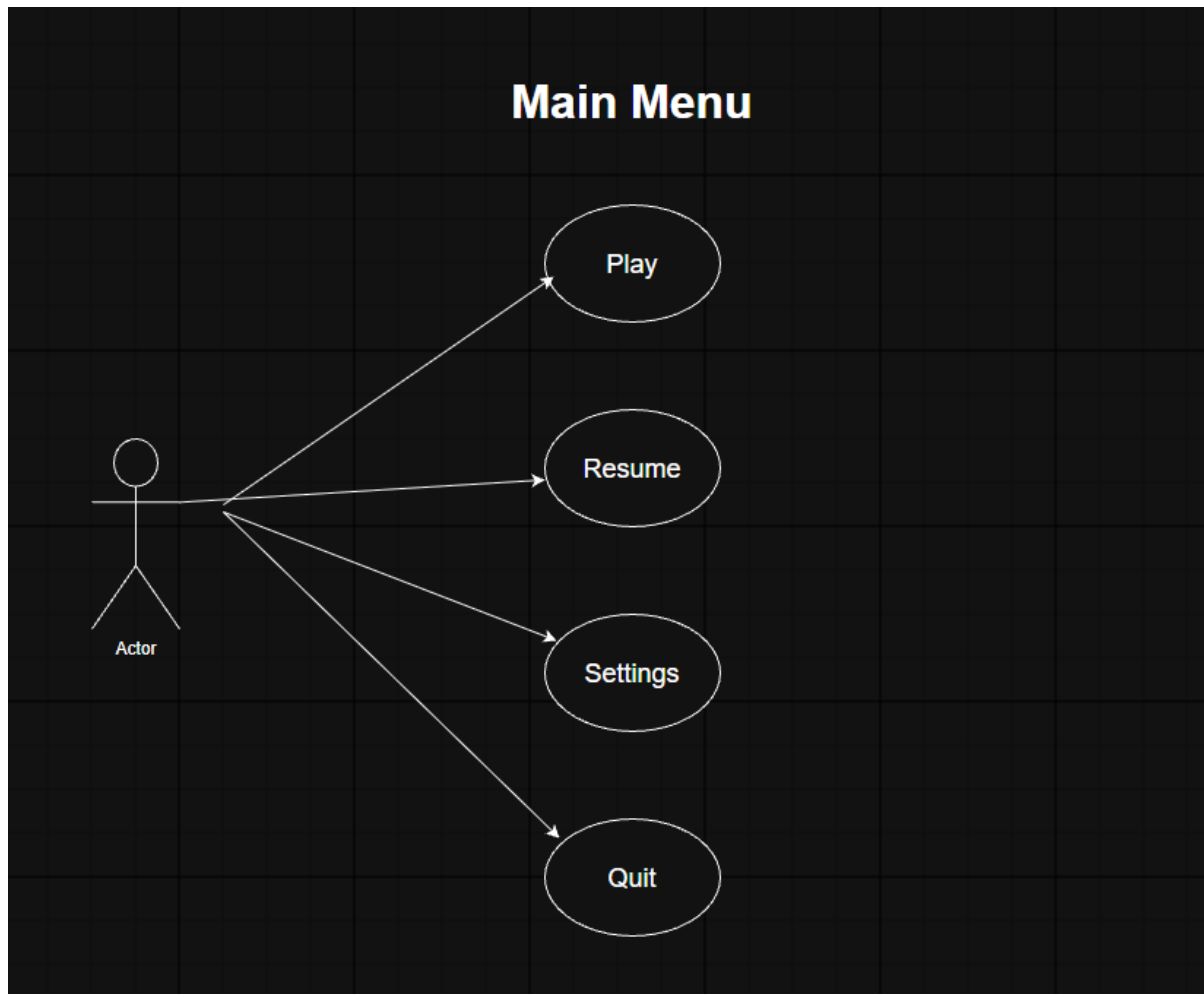


Εικόνα 8

3.6 Διαγράμματα

Use Case Diagram — Main Menu

Το παρακάτω Use Case Diagram απεικονίζει τις βασικές λειτουργίες που προσφέρει το Main Menu του παιχνιδιού. Ο Πάικτης (Actor) αλληλεπιδρά με τέσσερις διαθέσιμες επιλογές:

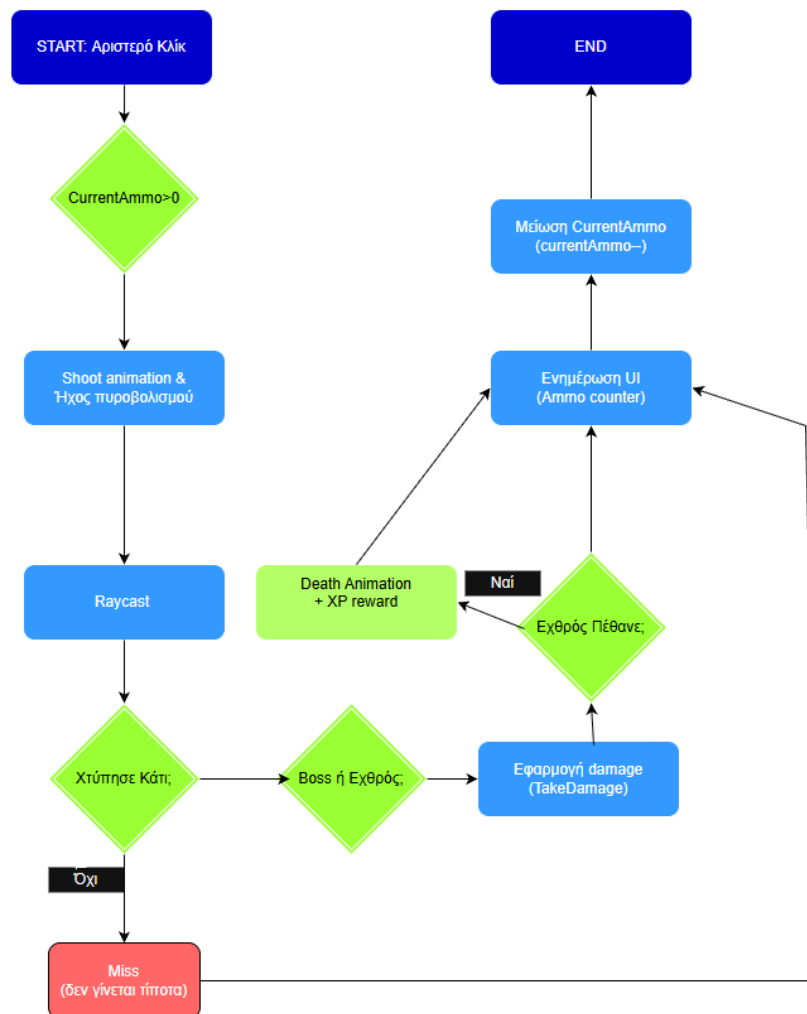


Εικόνα 9

Τα use cases του Main Menu: (1) Νέο Παιχνίδι (Play) - φόρτωση σκηνής L1 και αρχικοποίηση όλων των δεδομένων παίκτη. (2) Συνέχεια (Resume) - ενεργοποιείται μόνο αν υπάρχει αποθηκευμένο progress, διαφορετικά εμφανίζεται απενεργοποιημένο. (3) Ρυθμίσεις (Settings) - Volume, Brightness, Mouse Sensitivity. (4) Έξοδος (Exit) -Application.Quit().

Flow Diagram — Διαδικασία Πυροβολισμού

Το παρακάτω Flow Diagram αναλύει τη ροή εκτέλεσης κώδικα κατά τον πυροβολισμό (αριστερό κλικ), όπως υλοποιείται στο PlayerShooting.cs:



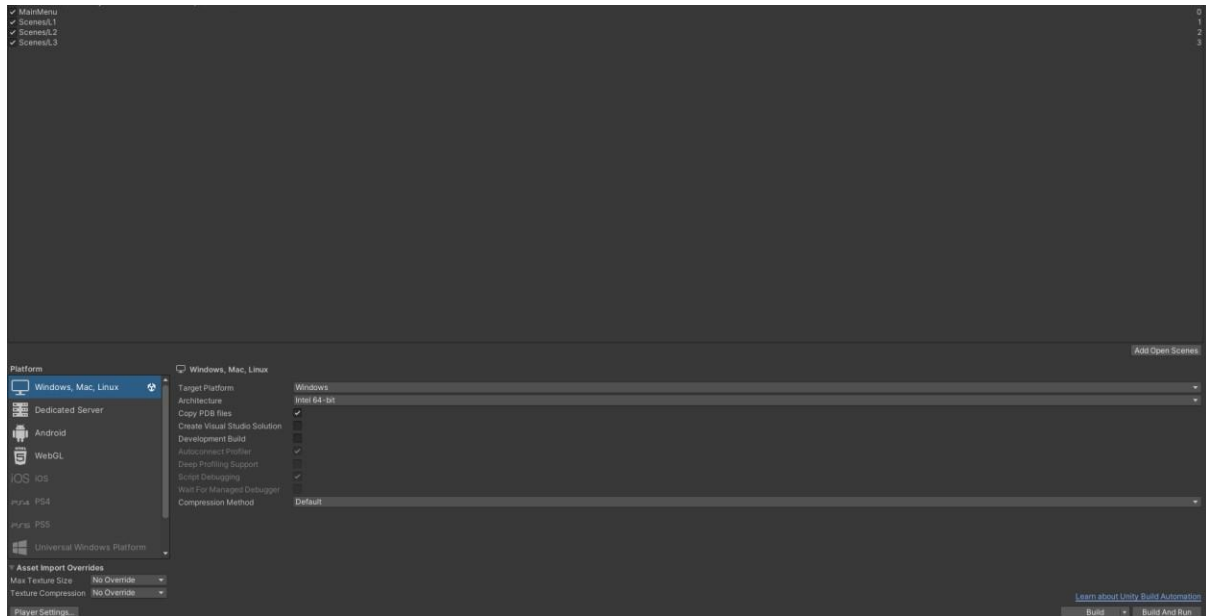
Εικόνα 10

Ανάλυση ροής: Αριστερό κλικ → έλεγχος $\text{currentAmmo} > 0$. Αν όχι δεν πυροβολεί. Αν Ναι: shoot animation + ήχος → `Physics.Raycast` → αν πετύχει εχθρό/Boss: `TakeDamage()` → αν πεθάνει: death animation + XP reward). Σε κάθε περίπτωση: $\text{currentAmmo}--$ και ενημέρωση UI ammo counter.

4. Scenes

4.1 Δομή Scenes

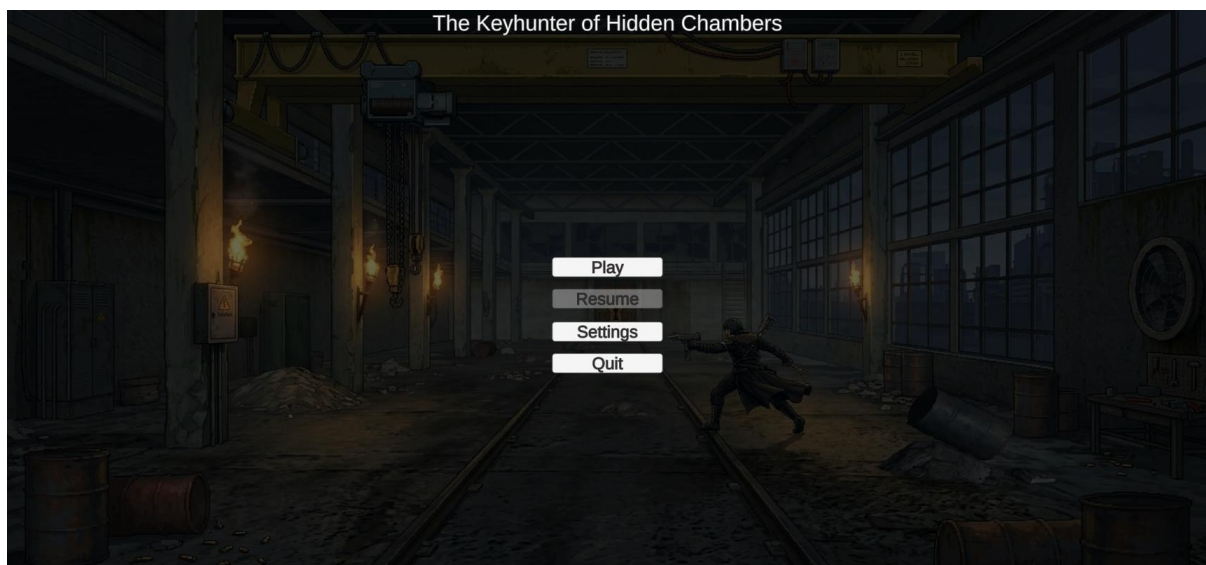
Το παιχνίδι αποτελείται από 4 scenes: MainMenu, L1, L2, L3 (Final). Η μετάβαση γίνεται με SceneManager.LoadScene(). Για persistence δεδομένων χρησιμοποιείται DontDestroyOnLoad στα: Player, KeyManager, GameManager, WeaponSystem, InputManager, InventorySystem, MusicManager.



Εικόνα 11

4.2 Main Menu

Το MainMenu περιέχει: Play (νέο παιχνίδι), Resume (συνέχεια - γκρι αν δεν παίζει), Settings (Volume/Brightness/Sensitivity), Quit. Το Play γκριζάρει όταν υπάρχει παιχνίδι σε εξέλιξη και το Resume γκριζάρει όταν δεν υπάρχει.



Εικόνα 12

4.3 Level 1 (L1)

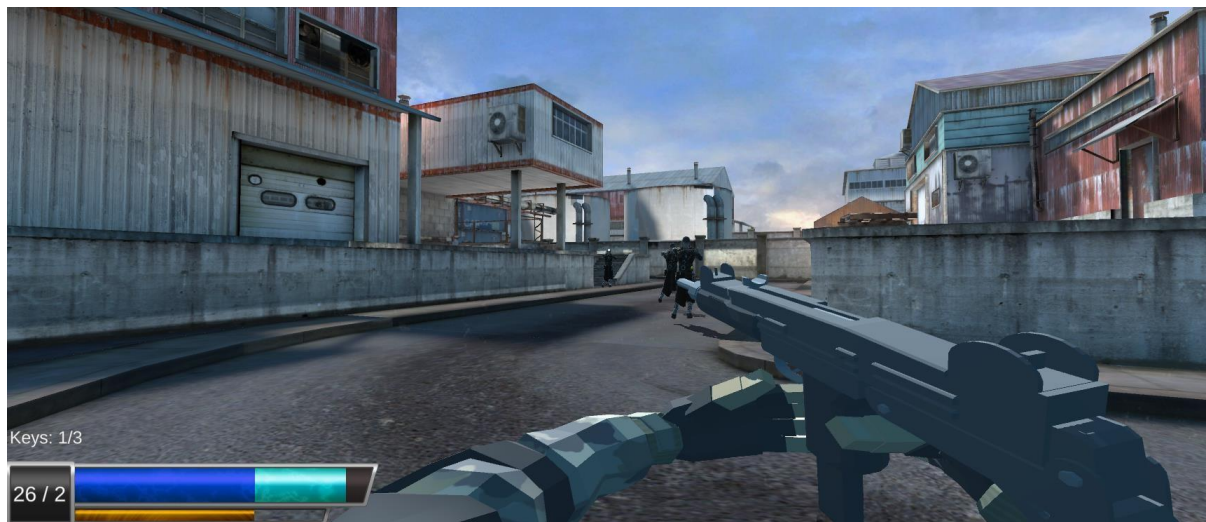
Factory περιβάλλον με κανονικούς εχθρούς και Secret Room. Λειτουργεί ως tutorial level για εξοικείωση με τους μηχανισμούς.



Εικόνα 13

4.4 Level 2 (L2)

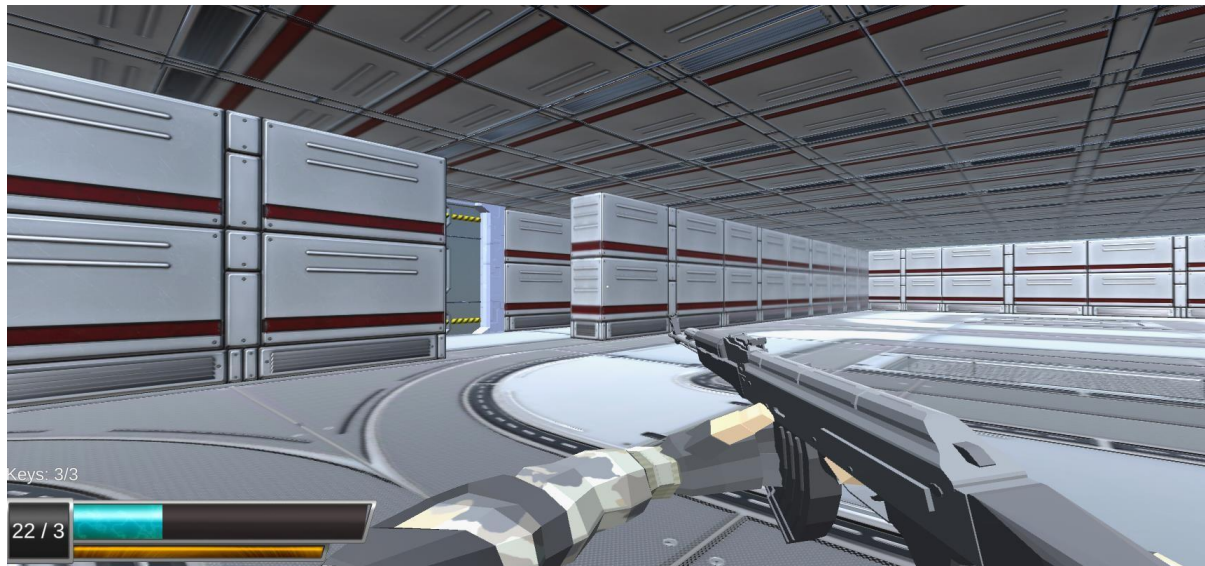
Αυξημένη δυσκολία με περισσότερους εχθρούς. Ο παίκτης φτάνει με τα δεδομένα (XP, κλειδιά, όπλο) από το L1.



Εικόνα 14

4.5 Level 3 - Final (L3)

Απαιτεί και τα 3 κλειδιά για είσοδο στο Boss Room. Περιέχει τον Final Boss με 1000 HP.



Εικόνα 15



Εικόνα 16

5. Animation / Animator

5.1 Βασικά Στοιχεία

Το Unity Animator διαχειρίζεται μεταβάσεις μεταξύ animations μέσω State Machine. Parameters: Bool (π.χ. IsRunning), Trigger (π.χ. Attack, Death), Float, Int. Για την υλοποίηση των animation χρησιμοποιήθηκε το Animator Controller της Unity, με στόχο τη διαχείριση των βασικών κινήσεων του χαρακτήρα και την ομαλή εναλλαγή μεταξύ των διαφορετικών animations ανάλογα με την κατάσταση που βρίσκεται ο εκάστοτε χαρακτήρας

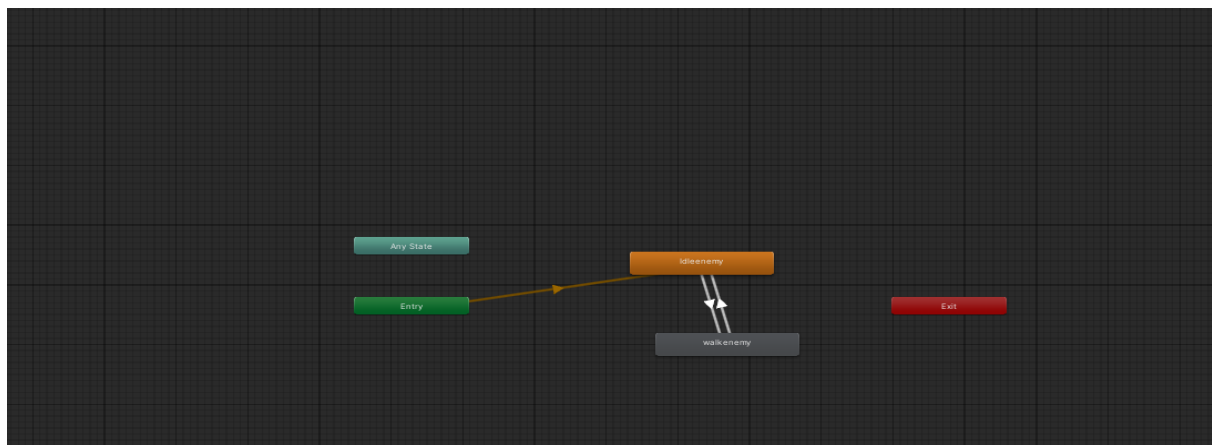
5.2 Animator Εχθρού

Για την υλοποίηση των κανονικών εχθρών χρησιμοποιήθηκε το Animator Controller της Unity, με στόχο τη διαχείριση των βασικών κινήσεων του χαρακτήρα και την ομαλή εναλλαγή μεταξύ των διαφορετικών animations ανάλογα με την κατάσταση που βρίσκεται ο εχθρός.

Με την έναρξη του παιχνιδιού, η ροή του Animator ξεκινά από το Entry και οδηγείται αυτόματα στην κατάσταση **Idle_enemy**, η οποία αντιστοιχεί στην κατάσταση αναμονής του εχθρού. Όταν ο εχθρός δεν εκτελεί κάποια ενέργεια, για παράδειγμα όταν στέκεται σε ένα waypoint κατά τη διάρκεια της περιπολίας του, παραμένει σε αυτή την κατάσταση.

Όταν ο εχθρός χρειάζεται να κινηθεί μέσα στο περιβάλλον, είτε κατά τη διάρκεια της περιπολίας είτε όταν κυνηγά τον παίκτη, μεταβαίνει στην κατάσταση **Walk_enemy**. Η μετάβαση αυτή ελέγχεται μέσω του Float parameter **Speed**, το οποίο παίρνει την τιμή του από το μέτρο της ταχύτητας του NavMesh Agent (agent.velocity.magnitude). Έτσι, το animation κίνησης ενεργοποιείται αυτόματα όταν ο εχθρός αρχίζει να κινείται και απενεργοποιείται όταν σταματά, δημιουργώντας μια φυσική και ρεαλιστική συμπεριφορά.

Με την ολοκλήρωση της κίνησης, ο Animator επιστρέφει στην κατάσταση **Idle_enemy**, δημιουργώντας έναν συνεχή κύκλο μεταξύ αναμονής και κίνησης που προσδίδει στον εχθρό πιο φυσική παρουσία στο επίπεδο. Στον Animator υπάρχει επίσης η **Any State**, η οποία επιτρέπει τη μετάβαση σε οποιαδήποτε κατάσταση ανεξάρτητα από την τρέχουσα, χρησιμοποιείται για άμεσες ενέργειες όπως το animation θανάτου (**Death**), το οποίο πρέπει να ενεργοποιείται άμεσα ανεξάρτητα από το τι έκανε ο εχθρός εκείνη τη στιγμή.



Εικόνα 17

5.3 Animator Final Boss

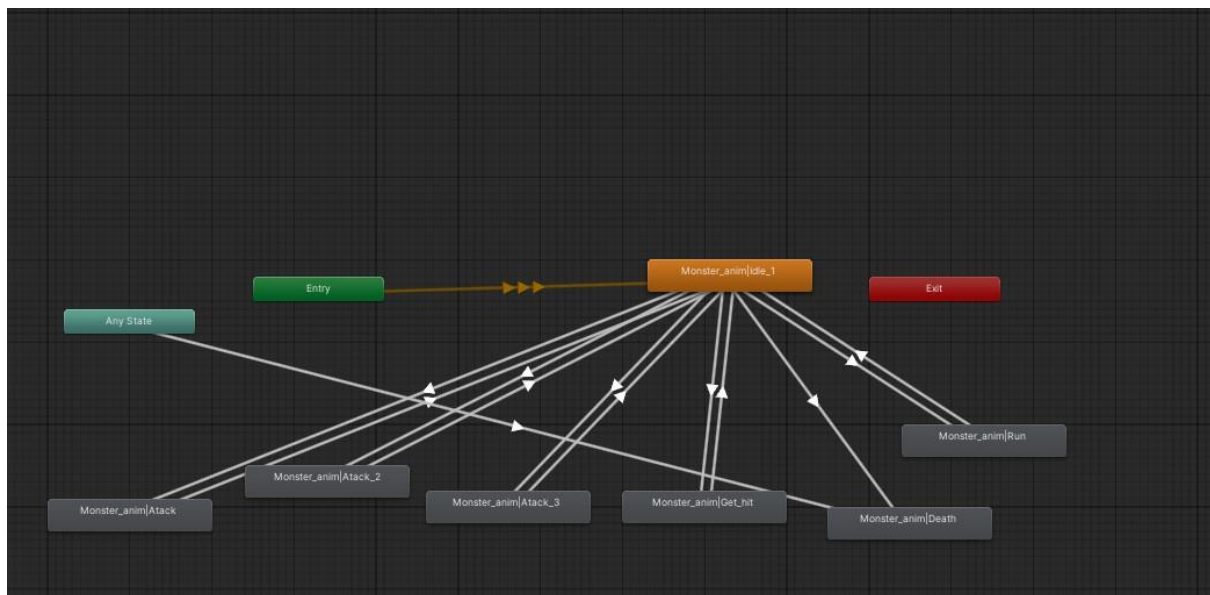
Με την έναρξη του παιχνιδιού, η ροή του Animator ξεκινά από το Entry και οδηγείται αυτόματα στην κατάσταση **Monster_anim|Idle_1**, η οποία αντιστοιχεί στην κατάσταση αδράνειας του Boss τη στιγμή δηλαδή που ο χαρακτήρας βρίσκεται σε αναμονή πριν εντοπίσει τον παίκτη. Από εκεί, ανάλογα με τις ενέργειες που πραγματοποιούνται κατά τη διάρκεια της μάχης, ο χαρακτήρας μεταβαίνει δυναμικά σε διαφορετικές καταστάσεις.

Η κατάσταση **Monster_anim|Run** ενεργοποιείται όταν ο Boss κινείται στο περιβάλλον, για παράδειγμα όταν πλησιάζει ή καταδιώκει τον παίκτη μέσω του NavMesh Agent. Για τις επιθέσεις έχουν δημιουργηθεί τρεις διαφορετικές καταστάσεις: **Monster_anim|Attack**, **Monster_anim|Attack_2** και **Monster_anim|Attack_3**. Αυτές οι καταστάσεις αντιστοιχούν σε τρεις διαφορετικούς τύπους επίθεσης και επιτρέπουν μεγαλύτερη ποικιλία και αντίστοιχα

μεγαλύτερη δυσκολία στις κινήσεις του τελικού αντιπάλου, ιδιαίτερα κατά το Enraged Mode. Επιπλέον, η κατάσταση **Monster_anim|Get_hit** ενεργοποιείται κάθε φορά που ο Boss δέχεται επίθεση από τον παίκτη, αποτυπώνοντας οπτικά την αντίδρασή του στη ζημιά που υφίσταται.

Ιδιαίτερα σημαντική είναι η χρήση της **Any State**, καθώς επιτρέπει τη μετάβαση σε συγκεκριμένες καταστάσεις ανεξάρτητα από το animation που παίζει εκείνη τη στιγμή. Με αυτόν τον τρόπο, ενέργειες όπως η επίθεση, η αντίδραση σε χτύπημα ή ο θάνατος μπορούν να ενεργοποιούνται άμεσα χωρίς να απαιτείται η δημιουργία ξεχωριστής μετάβασης από κάθε πιθανή προηγούμενη κατάσταση, κάτι που απλοποιεί σημαντικά τη δομή του Animator.

Τέλος, όταν η υγεία του Boss μηδενιστεί, ενεργοποιείται η κατάσταση **Monster_anim|Death**, η οποία αναπαριστά το animation θανάτου και αποτελεί το τελικό στάδιο της συμπεριφοράς του χαρακτήρα. Μετά την ολοκλήρωση αυτού του animation, η μάχη τελειώνει και εμφανίζεται το Win Screen.



Εικόνα 18

5.4 Animation Πόρτας

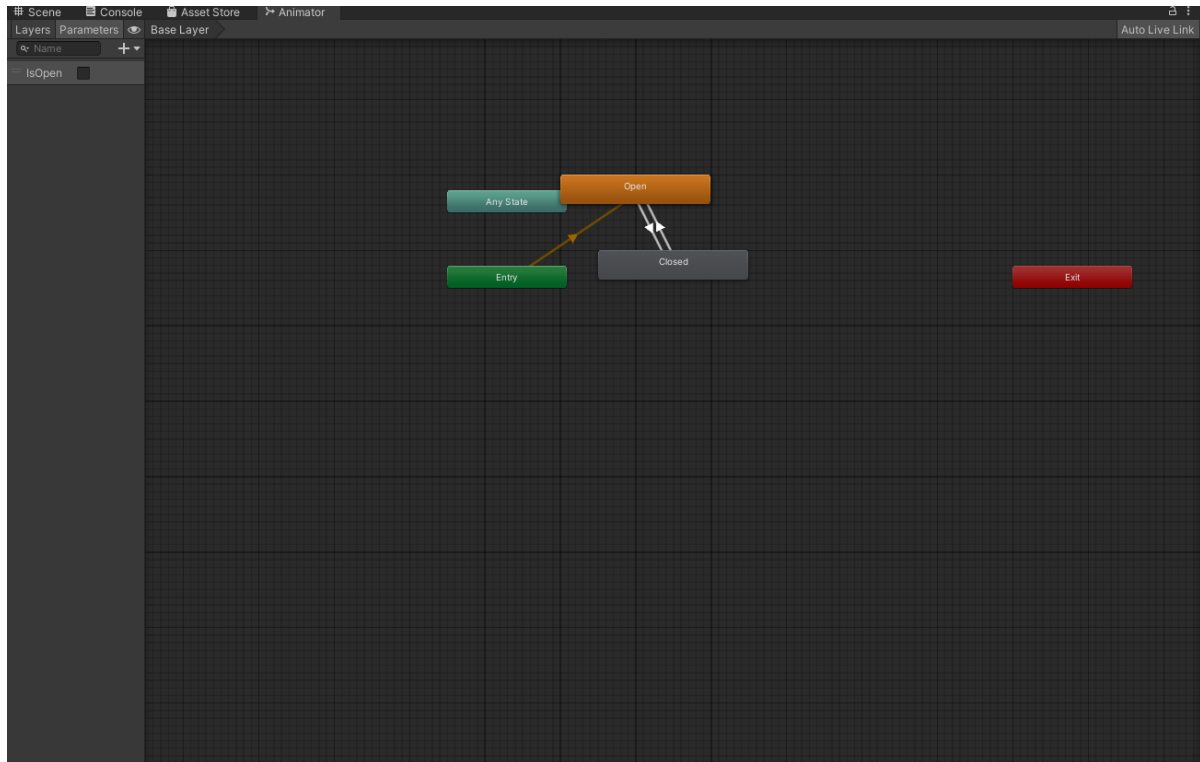
Για την υλοποίηση των πορτών του παιχνιδιού χρησιμοποιήθηκε το Animator Controller της Unity, το οποίο διαχειρίζεται την εναλλαγή μεταξύ της ανοιχτής και κλειστής κατάστασης της πόρτας μέσω μιας απλής αλλά αποτελεσματικής State Machine.

Η δομή του Animator αποτελείται από δύο βασικές καταστάσεις: την **Open** και την **Closed**. Η κατάσταση **Closed** ορίζεται ως Default State, που σημαίνει ότι κατά την έναρξη του παιχνιδιού η πόρτα βρίσκεται πάντα στην κλειστή της θέση, κάτι που εξυπηρετεί τη λογική του παιχνιδιού, καθώς οι πόρτες πρέπει να ανοίγουν μόνο μετά από αλληλεπίδραση του παίκτη με το Keypad.

Οι μεταβάσεις μεταξύ των δύο καταστάσεων ελέγχονται μέσω του Bool parameter **IsOpen**, το οποίο τίθεται σε true ή false από το Keypad.cs script ανάλογα με την ενέργεια του παίκτη. Όταν ο παίκτης αλληλεπιδρά με το Keypad και πληρούνται οι απαραίτητες προϋποθέσεις, όπως η κατοχή των απαραίτητων κλειδιών, το script καλεί animator.SetBool("IsOpen", true) και η πόρτα μεταβαίνει από την κατάσταση **Closed** στην κατάσταση **Open**, αναπαράγοντας το animation ανοίγματος. Αντίστροφα, αν η πόρτα έχει ρυθμιστεί να κλείνει αυτόματα μετά

από ορισμένα δευτερόλεπτα, μέσω του `AutoClose Coroutine`, το `parameter` τίθεται ξανά σε `false` και η πόρτα επιστρέφει στην κλειστή της κατάσταση.

Ιδιαίτερη σημασία έχει η ρύθμιση **Has Exit Time** στις μεταβάσεις. Αυτή είναι απενεργοποιημένη, ώστε η αλλαγή κατάστασης να γίνεται άμεσα μόλις αλλάξει η τιμή του `parameter`, χωρίς να χρειάζεται να ολοκληρωθεί πρώτα το τρέχον `animation`. Αυτό εξασφαλίζει ομαλή και άμεση απόκριση στις ενέργειες του παίκτη.



Εικόνα 19

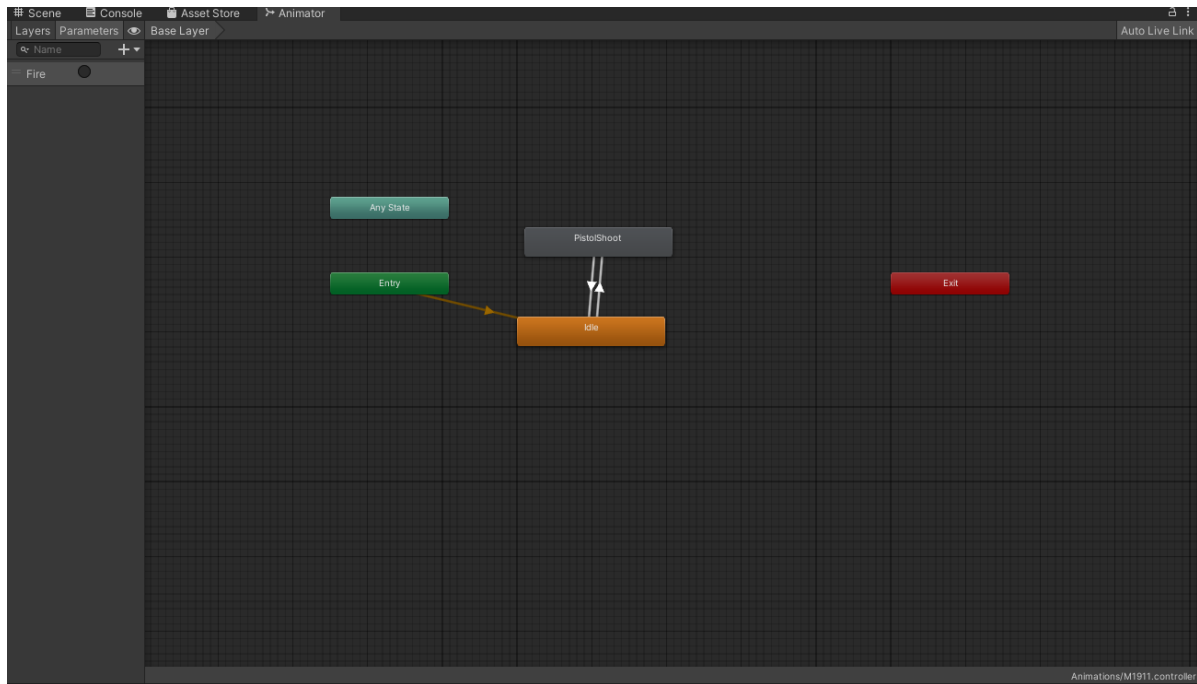
5.5 Animation Πιστολιού

Για την οπτική αναπαράσταση του πιστολιού M1911 χρησιμοποιήθηκε `Animator Controller`, με στόχο την ομαλή εναλλαγή μεταξύ της κατάστασης αναμονής και του `animation πυροβολισμού`.

Η δομή του `Animator` αποτελείται από δύο βασικές καταστάσεις. Η κατάσταση **Idle** ορίζεται ως `Default State`, που σημαίνει ότι κατά την έναρξη το πιστόλι βρίσκεται πάντα σε κατάσταση αναμονής — ο χαρακτήρας κρατά το όπλο χωρίς να εκτελεί κάποια ενέργεια. Η δεύτερη κατάσταση, **PistolShoot**, αναπαράγει το `animation πυροβολισμού`, συμπεριλαμβανομένης της ανάκρουσης (`recoil`) που προσδίδει ρεαλισμό στη βολή.

Η μετάβαση από το **Idle** στο **PistolShoot** ελέγχεται μέσω του `Trigger parameter Fire`, το οποίο ενεργοποιείται από το `WeaponHandler.cs` script κάθε φορά που ο παίκτης πυροβολεί, μέσω της κλήσης `animator.SetTrigger("Fire")`. Η χρήση `Trigger` εξασφαλίζει ότι το `animation πυροβολισμού` παίζει ακριβώς μια φορά ανά `πυροβολισμό`.

Μόλις ολοκληρωθεί το `animation`, η `State Machine` επιστρέφει αυτόματα στην κατάσταση **Idle** μέσω της ρύθμισης **Has Exit Time**, περιμένοντας τον επόμενο `πυροβολισμό`.



Εικόνα 20

6. Σημαντικά Scripts Παίκτη

Το InputManager.cs αποτελεί ένα από τα πιο κεντρικά scripts του παιχνιδιού, καθώς είναι υπεύθυνο για τη διαχείριση όλης της εισόδου του χρήστη από την κίνηση και τη σκόπευση μέχρι το pause menu και τη μετάβαση μεταξύ scenes. Υλοποιεί το **Singleton pattern** με DontDestroyOnLoad, ώστε να επιβιώνει σε όλη τη διάρκεια του παιχνιδιού χωρίς να καταστρέφεται κατά τη μετάβαση από scene σε scene.

Singleton και DontDestroyOnLoad

Στο Awake() εφαρμόζεται το Singleton pattern: αν δεν υπάρχει ήδη instance του InputManager, το τρέχον object ορίζεται ως το μοναδικό instance και επιβιώνει μεταξύ scenes μέσω DontDestroyOnLoad(gameObject). Αν υπάρχει ήδη ένα instance κάτι που μπορεί να συμβεί αν φορτωθεί ξανά μια scene που περιέχει τον Player το νέο object καταστρέφεται αμέσως, διασφαλίζοντας ότι υπάρχει πάντα ακριβώς ένας InputManager.

Νέο Input System

Το script χρησιμοποιεί το νέο **Unity Input System** μέσω της κλάσης PlayerInput και των OnFootActions. Κατά την αρχικοποίηση, εγγράφονται callbacks για τις ενέργειες άλματος, sprint και crouch:

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

public class InputManager : MonoBehaviour
{
    public static InputManager Instance;
    private PlayerInput playerInput;
    public PlayerInput.OnFootActions onFoot;
    private PlayerMotor motor;
    private PlayerLook look;
    private GameObject pausePanel;
    public bool isPaused = false;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
            return;
        }

        playerInput = new PlayerInput();
        onFoot = playerInput.OnFoot;
        motor = GetComponent<PlayerMotor>();
        look = GetComponent<PlayerLook>();
        onFoot.Jump.performed += ctx => motor.Jump();
        onFoot.Sprint.performed += ctx => motor.Sprint(true);
        onFoot.Sprint.canceled += ctx => motor.Sprint(false);
        onFoot.Crouch.performed += ctx => motor.Crouch(true);
        onFoot.Crouch.canceled += ctx => motor.Crouch(false);
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
}
```

Εικόνα 21

Αυτός ο τρόπος εγγραφής επιτρέπει άμεση και αποδοτική αντίδραση στις ενέργειες του χρήστη χωρίς να χρειάζεται polling κάθε frame.

Κίνηση και Κάμερα

Η κίνηση του παίκτη διαχειρίζεται στο `FixedUpdate()`, το οποίο τρέχει σε σταθερό ρυθμό ανεξάρτητα από το framerate ιδανικό για φυσική κίνηση. Διαβάζει το διανυσματικό input κίνησης και το προωθεί στο `PlayerMotor.ProcessMove()`. Αντίστοιχα, η κάμερα διαχειρίζεται στο `LateUpdate()`, το οποίο τρέχει μετά το `Update()` εξασφαλίζοντας ότι η κάμερα ακολουθεί την τελική θέση του παίκτη κάθε frame. Και οι δύο μέθοδοι ελέγχουν αν το `isPaused` είναι `false` πριν εκτελεστούν, ώστε η κίνηση να σταματά κατά τη διάρκεια του pause.

Επιπλέον, εφαρμόζεται αυτόματη ανεύρεση των components `PlayerMotor` και `PlayerLook` αν αυτά είναι `null` ή `disabled` κάτι που είναι απαραίτητο μετά από restart ή αλλαγή scene:

```

void FixedUpdate()
{
    if (!isPaused)
    {
        if (motor == null || !motor.enabled)
        {
            GameObject player = GameObject.FindGameObjectWithTag("Player");
            if (player != null)
                motor = player.GetComponent<PlayerMotor>();
        }
        if (motor != null && motor.enabled)
            motor.ProcessMove(onFoot.Movement.ReadValue<Vector2>());
    }
}

private void LateUpdate()
{
    if (!isPaused)
    {
        if (look == null || !look.enabled)
        {
            GameObject player = GameObject.FindGameObjectWithTag("Player");
            if (player != null)
                look = player.GetComponent<PlayerLook>();
        }
        if (look != null && look.enabled)
            look.ProcessLook(onFoot.Look.ReadValue<Vector2>());
    }
}

void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if (isPaused) Resume();
        else Pause();
    }

    if (Input.GetMouseButtonDown(0) &&
        Cursor.lockState == CursorLockMode.None &&
        !isPaused &&
        Time.timeScale > 0f)
    {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
}

```

Εικόνα 22

Σύστημα Pause

Στο Update() εντοπίζεται το πάτημα του πλήκτρου Escape. Αν το παιχνίδι είναι σε pause, καλείται το Resume(), αλλιώς καλείται το Pause(). Η μέθοδος Pause() παγώνει τον χρόνο (Time.timeScale = 0f), απελευθερώνει τον cursor και εμφανίζει το PausePanel. Το PausePanel βρίσκεται δυναμικά μέσω της FindPausePanel(), η οποία χρησιμοποιεί Resources.FindObjectsOfTypeAll<Canvas>() για να εντοπίσει το panel ακόμα και αν είναι inactive. Αντίστροφα, το Resume() επαναφέρει τον χρόνο, κλειδώνει τον cursor και κλείνει τόσο το PausePanel όσο και το InGameSettingsPanel αν είναι ανοιχτό.

```

private void Pause()
{
    isPaused = true;
    pausePanel = FindPausePanel();
    if (pausePanel != null) pausePanel.SetActive(true);
    Time.timeScale = 0f;
    Cursor.lockState = CursorLockMode.None;
    Cursor.visible = true;
}

public void Resume()
{
    isPaused = false;
    if (pausePanel != null) pausePanel.SetActive(false);

    // Κλείνει και το InGameSettingsPanel αν είναι ανοιχτό
    GameObject settingsPanel = GameObject.Find("InGameSettingsPanel");
    if (settingsPanel != null) settingsPanel.SetActive(false);

    Time.timeScale = 1f;
    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;
}

```

Εικόνα 23

Μετάβαση στο Main Menu

Η μέθοδος GoToMainMenu() αποθηκεύει την τρέχουσα scene στο PlayerPrefs με το key "LastScene", ώστε το MainMenuManager να γνωρίζει πού να επιστρέψει ο παίκτης αν επιλέξει Resume. Στη συνέχεια επαναφέρει τον χρόνο και φορτώνει το MainMenu scene.

```

public void GoToMainMenu()
{
    string currentScene = SceneManager.GetActiveScene().name;
    PlayerPrefs.SetString("LastScene", currentScene);
    PlayerPrefs.Save();
    Time.timeScale = 1f;
    isPaused = false;
    if (pausePanel != null) pausePanel.SetActive(false);
    SceneManager.LoadScene("MainMenu");
}

```

Εικόνα 24

Διαχείριση Scene Loading

Η μέθοδος OnSceneLoaded() εγγράφεται στο event SceneManager.sceneLoaded και εκτελείται αυτόματα κάθε φορά που φορτώνεται νέα scene. Διακρίνει δύο περιπτώσεις:

Αν η νέα scene **δεν είναι το MainMenu**, επανενεργοποιεί όλα τα scripts του παίκτη (PlayerLook, PlayerMotor, PlayerShooting), ενεργοποιεί την Camera και κλειδώνει τον cursor για κανονικό gameplay. Αν η νέα scene **είναι το MainMenu**, απενεργοποιεί το input system και απελευθερώνει τον cursor ώστε ο παίκτης να μπορεί να αλληλεπιδράσει με τα κουμπιά του menu.

```
private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    pausePanel = null;
    isPaused = false;
    Time.timeScale = 1f;

    if (scene.name != "MainMenu")
    {
        GameObject player = GameObject.FindGameObjectWithTag("Player");
        if (player != null)
        {
            PlayerLook pl = player.GetComponent<PlayerLook>();
            PlayerMotor pm = player.GetComponent<PlayerMotor>();
            PlayerShooting ps = player.GetComponent<PlayerShooting>();
            if (pl != null) { pl.enabled = true; look = pl; }
            if (pm != null) { pm.enabled = true; motor = pm; }
            if (ps != null) ps.enabled = true;

            // Ενεργοποιεί την κάμερα
            Camera cam = player.GetComponentInChildren<Camera>(true);
            if (cam != null) cam.enabled = true;
        }

        onFoot.Enable();
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
    else
    {
        onFoot.Disable();
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }
}
```

Εικόνα 25

6.2 PlayerMotor.cs

Το PlayerMotor.cs είναι υπεύθυνο για τη φυσική κίνηση του παίκτη στον τρισδιάστατο χώρο. Διαχειρίζεται όλες τις μορφές κίνησης περπάτημα, τρέξιμο, σκύψιμο και άλμα χρησιμοποιώντας το **CharacterController** component της Unity αντί για Rigidbody, κάτι που προσφέρει μεγαλύτερο έλεγχο και αποφεύγει ανεπιθύμητες φυσικές αλληλεπιδράσεις.

Αρχικοποίηση και Layer Collision

Στο Awake() απενεργοποιείται η σύγκρουση μεταξύ του layer του παίκτη (layer 11) και του layer των εχθρών (layer 19) μέσω της Physics.IgnoreLayerCollision(). Αυτό εμποδίζει τον παίκτη να σπρώχνει ή να αλληλεπιδρά φυσικά με τους εχθρούς κατά την επαφή:

```

public void Awake()
{
    Physics.IgnoreLayerCollision(11, 19, true);
    Physics.IgnoreLayerCollision(19, 11, true);
}

```

Εικόνα 26

Στο Start() γίνεται αρχικοποίηση του CharacterController και εντοπισμός της Camera μέσω GetComponentInChildren<Camera>(true) το true εξασφαλίζει ότι βρίσκει την κάμερα ακόμα και αν είναι disabled, κάτι που είναι απαραίτητο στην αρχή του παιχνιδιού από το MainMenu scene. Αποθηκεύεται επίσης το αρχικό ύψος της κάμερας (standCamHeight) για χρήση στο crouch σύστημα.

```

void Start()
{
    controller = GetComponent<CharacterController>();
    int playerLayer = gameObject.layer;
    int enemyLayer = LayerMask.NameToLayer("Enemy");
    Debug.Log($"Player layer: {playerLayer} ({LayerMask.LayerToName(playerLayer)})");
    Debug.Log($"Enemy layer: {enemyLayer}");
    Physics.IgnoreLayerCollision(playerLayer, enemyLayer, true);

    cam = GetComponentInChildren<Camera>(true); // true = βρίσκει και disabled
    if (cam != null)
        standCamHeight = cam.transform.localPosition.y;

    speed = walkSpeed;
}

```

Εικόνα 27

Σύστημα Κίνησης

Η κίνηση του παίκτη πραγματοποιείται στη μέθοδο ProcessMove(), η οποία καλείται από το InputManager στο FixedUpdate(). Λαμβάνει ως είσοδο ένα Vector2 με τις τιμές των πλήκτρων WASD και το μετατρέπει σε τρισδιάστατη κατεύθυνση κίνησης:

```

public void ProcessMove(Vector2 input)
{
    Vector3 moveDirection = new Vector3(input.x, 0, input.y);
    controller.Move(transform.TransformDirection(moveDirection) * speed * Time.deltaTime);
    playerVelocity.y += gravity * Time.deltaTime;
    if (isGrounded && playerVelocity.y < 0)
        playerVelocity.y = -2f;
    controller.Move(playerVelocity * Time.deltaTime);
}

```

Εικόνα 28

Η TransformDirection() μετατρέπει την κατεύθυνση από το τοπικό σύστημα συντεταγμένων του παίκτη στο παγκόσμιο, ώστε η κίνηση να είναι πάντα σχετική με την κατεύθυνση που κοιτά ο παίκτης.

Σύστημα Βαρύτητας και Άλματος

Η βαρύτητα εφαρμόζεται χειροκίνητα κάθε frame μέσω της μεταβλητής `playerVelocity.y`:

Όταν ο παίκτης βρίσκεται στο έδαφος (`isGrounded`), η κατακόρυφη ταχύτητα ορίζεται σε `-2f` αντί για `0f` για να εξασφαλιστεί ότι ο `CharacterController` παραμένει σταθερά στο έδαφος και δεν αιωρείται. Το άλμα υπολογίζεται με βάση τον τύπο της κινηματικής:

```
public void Jump()
{
    if (isGrounded && !isCrouching)
        playerVelocity.y = Mathf.Sqrt(jumpHeight * -3.0f * gravity);
}
```

Εικόνα 29

Αυτός ο τύπος εξασφαλίζει ότι το ύψος του άλματος είναι πάντα σταθερό ανεξάρτητα από τη βαρύτητα, και μπορεί να ρυθμιστεί εύκολα από τον Inspector.

Σύστημα Sprint

Η μέθοδος `Sprint()` αλλάζει την ταχύτητα του παίκτη μεταξύ `walkSpeed` και `sprintSpeed`. Περιλαμβάνει έλεγχο που αποτρέπει το sprint κατά τη διάρκεια του crouch, διασφαλίζοντας ότι ο παίκτης δεν μπορεί να τρέχει και να είναι σκυμμένος ταυτόχρονα.

```
public void Sprint(bool sprinting)
{
    if (isCrouching) return;
    isSprinting = sprinting;
    speed = isSprinting ? sprintSpeed : walkSpeed;
}
```

Εικόνα 30

Σύστημα Crouch

Το σύστημα crouch τροποποιεί τόσο το ύψος του `CharacterController` όσο και τη θέση της κάμερας για να αποδώσει ρεαλιστικά το σκύψιμο. Όταν ο παίκτης σκύβει, το ύψος του controller μειώνεται από `standHeight` σε `crouchHeight`, και η κάμερα κατεβαίνει κατά `crouchCamHeight` μονάδες.

```

public void Crouch(bool crouching)
{
    isCrouching = crouching;
    if (cam == null) cam = GetComponentInChildren<Camera>(true);

    if (isCrouching)
    {
        controller.height = crouchHeight;
        speed = crouchSpeed;
        if (cam != null)
            cam.transform.localPosition = new Vector3(
                cam.transform.localPosition.x,
                standCamHeight - crouchCamHeight,
                cam.transform.localPosition.z);
    }
    else
    {
        controller.height = standHeight;
        speed = walkSpeed;
        if (cam != null)
            cam.transform.localPosition = new Vector3(
                cam.transform.localPosition.x,
                standCamHeight,
                cam.transform.localPosition.z);
    }
}

```

Εικόνα 31

Όταν ο παίκτης σηκωθεί, όλες οι τιμές επιστρέφουν στις αρχικές τους.

Διαχείριση Επαφής με Εχθρούς

Η μέθοδος OnControllerColliderHit() καλείται αυτόματα από τον CharacterController κάθε φορά που ο παίκτης έρχεται σε επαφή με άλλο collider. Αν το object ανήκει στο layer των εχθρών και έχει Rigidbody, μηδενίζεται η ταχύτητά του για να αποτραπεί το φαινόμενο οι εχθροί να σπρώχνονται ή να πέφτουν από τον παίκτη:

```

void OnControllerColliderHit(ControllerColliderHit hit)
{
    if (hit.gameObject.layer == LayerMask.NameToLayer("Enemy"))
    {
        Rigidbody rb = hit.collider.attachedRigidbody;
        if (rb != null)
        {
            rb.velocity = Vector3.zero;
            rb.angularVelocity = Vector3.zero;
        }
    }
}

```

Εικόνα 32

6.3 PlayerLook.cs

Το PlayerLook.cs είναι υπεύθυνο για τη διαχείριση της κάμερας του παίκτη και την αντίδραση στην κίνηση του ποντικιού. Υλοποιεί το κλασικό σύστημα σκόπευσης πρώτου προσώπου (First Person Camera) που χρησιμοποιείται στα FPS παιχνίδια, διαχωρίζοντας την οριζόντια από την κατακόρυφη κίνηση της κάμερας.

Αρχικοποίηση Κάμερας

Στο Awake() και στο Start() εντοπίζεται δυναμικά η κάμερα μέσω GetComponentInChildren<Camera>(true). Η παράμετρος true είναι κρίσιμη, καθώς επιτρέπει την εύρεση της κάμερας ακόμα και αν είναι disabled κάτι που συμβαίνει στο MainMenu scene όπου η κάμερα του παίκτη είναι απενεργοποιημένη για να αποφευχθεί σύγκρουση με την κάμερα του menu. Μόλις εντοπιστεί, ενεργοποιείται αμέσως.

```
public class PlayerLook : MonoBehaviour
{
    public Camera cam;
    private float xRotation = 0f;
    public float xSensitivity = 30f;
    private float ySensitivity = 30f;

    void Awake()
    {
        if (void PlayerLook.Awake()
            cam = GetComponentInChildren<Camera>(true);
            if (cam != null) cam.enabled = true;
        }

    void Start()
    {
        Debug.Log("PlayerLook Awake!");
        if (cam == null)
            cam = GetComponentInChildren<Camera>(true);
        if (cam != null) cam.enabled = true;

        Debug.Log($"PlayerLook Start! cam={cam?.name} enabled={cam?.enabled}");

        float savedSensitivity = PlayerPrefs.GetFloat("Sensitivity", 30f);
        if (savedSensitivity <= 0f) savedSensitivity = 30f;
        xSensitivity = savedSensitivity;
        ySensitivity = savedSensitivity;
    }
}
```

Εικόνα 33

Real-time Sensitivity

Στο Update() διαβάζεται η τιμή της ευαισθησίας ποντικιού από τα PlayerPrefs κάθε frame.

```

void Update()
{
    float savedSensitivity = PlayerPrefs.GetFloat("Sensitivity", 30f);
    if (savedSensitivity <= 0f) savedSensitivity = 30f;
    xSensitivity = savedSensitivity;
    ySensitivity = savedSensitivity;
}

```

Εικόνα 34

Αυτή η προσέγγιση εξασφαλίζει ότι αλλαγές στις ρυθμίσεις ευαισθησίας από το Settings menu εφαρμόζονται άμεσα χωρίς να χρειάζεται επανεκκίνηση του παιχνιδιού. Επίσης, ο έλεγχος `if (savedSensitivity <= 0f)` αποτρέπει την περίπτωση η ευαισθησία να είναι μηδενική κάτι που θα σήμαινε ότι η κάμερα δεν θα κινούνταν καθόλου.

Σύστημα Κίνησης Κάμερας

Η βασική λογική υλοποιείται στη μέθοδο `ProcessLook()`, η οποία καλείται από το `InputManager` στο `LateUpdate()` κάθε frame. Λαμβάνει ως είσοδο ένα `Vector2` με τις τιμές κίνησης του ποντικιού και εφαρμόζει δύο διαφορετικές περιστροφές:

Η κατακόρυφη κίνηση (`mouseY`) περιστρέφει μόνο την κάμερα γύρω από τον τοπικό άξονα X. Η γωνία `xRotation` αυξάνεται ή μειώνεται ανάλογα με την κίνηση του ποντικιού και περιορίζεται μεταξύ -80° και $+80^\circ$ μέσω `Mathf.Clamp()` για να αποτραπεί η ανεξέλεγκτη περιστροφή της κάμερας

Η οριζόντια κίνηση (`mouseX`) περιστρέφει ολόκληρο το σώμα του παίκτη γύρω από τον κατακόρυφο άξονα Y, ώστε ο χαρακτήρας να στρίβει αριστερά και δεξιά:

Αυτός ο διαχωρισμός κάμερα για κατακόρυφη κίνηση, σώμα για οριζόντια — είναι η κλασική υλοποίηση του FPS camera system και εξασφαλίζει φυσική και ομαλή εμπειρία σκόπευσης.

```

public void ProcessLook(Vector2 input)
{
    if (cam == null) return;
    float mouseX = input.x;
    float mouseY = input.y;
    xRotation -= (mouseY * Time.deltaTime) * ySensitivity;
    xRotation = Mathf.Clamp(xRotation, -80f, 80f);
    cam.transform.localRotation = Quaternion.Euler(xRotation, 0, 0);
    transform.Rotate(Vector3.up * (mouseX * Time.deltaTime) * xSensitivity);
}

```

Εικόνα 35

6.4 PlayerHealth.cs

Το `PlayerHealth.cs` διαχειρίζεται το σύστημα υγείας και ασπίδας του παίκτη. Είναι υπεύθυνο για την εφαρμογή ζημιάς, την αποκατάσταση υγείας, την οπτική απόδοση των μπαρών υγείας

και ασπίδας, καθώς και την ενεργοποίηση του Game Over όταν η υγεία του παίκτη μηδενιστεί.

Αρχικοποίηση και Scene Loading

Το script εγγράφεται στο event SceneManager.sceneLoaded μέσω OnEnable() και OnDisable(), ώστε να εκτελείται αυτόματα η μέθοδος OnSceneLoaded() κάθε φορά που φορτώνεται νέα scene. Αυτό είναι απαραίτητο επειδή ο παίκτης είναι DontDestroyOnLoad και επιβιώνει μεταξύ scenes. Στο OnSceneLoaded() επαναφέρονται οι τιμές υγείας και ασπίδας στις αρχικές τους τιμές, επανενεργοποιούνται όλα τα scripts του παίκτη και εντοπίζονται δυναμικά τα UI references της νέας scene:

```
// ΑΥΤΗ Η ΣΥΝΑΡΤΗΣΗ ΤΡΕΧΕΙ ΟΠΩΣΔΗΠΟΤΕ ΣΕ ΚΑΘΕ RESTART
private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    health = maxHealth;
    shield = 0f;
    isDead = false;

    PlayerMotor motor = GetComponent<PlayerMotor>();
    PlayerLook look = GetComponent<PlayerLook>();
    PlayerShooting shooting = GetComponent<PlayerShooting>();
    PlayerInteract interact = GetComponent<PlayerInteract>();

    if (motor != null) motor.enabled = true;
    if (look != null) look.enabled = true;
    if (shooting != null) shooting.enabled = true;
    if (interact != null) interact.enabled = true;

    FindUIReferences();

    if (frontShieldBar != null)
        frontShieldBar.fillAmount = 0f;
}
```

Εικόνα 36

Δυναμική Εύρεση UI References

Η μέθοδος FindUIReferences() εντοπίζει δυναμικά τα UI elements της τρέχουσας scene μέσω GameObject.Find(). Αυτό είναι απαραίτητο επειδή τα UI elements ανήκουν στη scene και καταστρέφονται κατά τη μετάβαση, οπότε ο παίκτης πρέπει να τα βρίσκει ξανά σε κάθε νέα scene:

```
private void FindUIReferences()
{
    frontHealthBar = GameObject.Find("FrontHealthBar")?.GetComponent<Image>();
    backHealthBar = GameObject.Find("BackHealthBar")?.GetComponent<Image>();
    frontShieldBar = GameObject.Find("Shieldbar")?.GetComponent<Image>();
    gameOverManager = FindObjectOfType<GameOverManager>();
}
```

Εικόνα 37

Σύστημα Υγείας με Chip Damage Effect

Στο Update() καλείται κάθε frame η UpdateHealthUI(), η οποία υλοποιεί το κλασικό chip damage effect που βλέπουμε σε πολλά σύγχρονα παιχνίδια. Η μπάρα υγείας αποτελείται από δύο στρώματα: το FrontHealthBar που αλλάζει αμέσως όταν ο παίκτης χάνει ή αποκτά υγεία, και το BackHealthBar που ακολουθεί με μια ομαλή καθυστέρηση μέσω Mathf.Lerp():

```
void Update()
{
    health = Mathf.Clamp(health, 0, maxHealth);
    shield = Mathf.Clamp(shield, 0, maxShield);

    UpdateHealthUI();
    UpdateShieldUI();
}
```

Εικόνα 38

Η χρήση του τετραγώνου του percentComplete ($\text{percentComplete} * \text{percentComplete}$) δημιουργεί ένα easing effect η μπάρα κινείται αρχικά γρήγορα και στη συνέχεια επιβραδύνει καθώς πλησιάζει την τελική τιμή, δίνοντας πιο φυσική αίσθηση. Όταν ο παίκτης αποκτά υγεία, η πίσω μπάρα γίνεται πράσινη για οπτική ανατροφοδότηση.

```

public void UpdateHealthUI()
{
    if (frontHealthBar == null || backHealthBar == null) return;

    float fillF = frontHealthBar.fillAmount;
    float fillB = backHealthBar.fillAmount;
    float hFraction = health / maxHealth;

    if (fillB > hFraction)
    {
        frontHealthBar.fillAmount = hFraction;
        backHealthBar.color = Color.red;
        lerpTimer += Time.deltaTime;
        float percentComplete = lerpTimer / chipSpeed;
        percentComplete = percentComplete * percentComplete;
        backHealthBar.fillAmount = Mathf.Lerp(fillB, hFraction, percentComplete);
    }
    if (fillF < hFraction)
    {
        backHealthBar.color = Color.green;
        backHealthBar.fillAmount = hFraction;
        lerpTimer += Time.deltaTime;
        float percentComplete = lerpTimer / chipSpeed;
        percentComplete = percentComplete * percentComplete;
        frontHealthBar.fillAmount = Mathf.Lerp(fillF, backHealthBar.fillAmount, percentComplete);
    }
}

```

Εικόνα 39

Η μέθοδος TakeDamage() καλείται κάθε φορά που ο παίκτης δέχεται ζημιά — είτε από σφαίρα εχθρού είτε από επίθεση του Final Boss. Πριν εφαρμοστεί οποιαδήποτε ζημιά, ελέγχεται αν ο παίκτης είναι ήδη νεκρός μέσω του flag isDead, ώστε να αποτραπεί οποιαδήποτε επιπλέον επεξεργασία

Στη συνέχεια ελέγχεται αν η ασπίδα του παίκτη είναι ενεργή. Αν η τιμή της shield είναι μεγαλύτερη από μηδέν, η ζημιά κατανέμεται μεταξύ υγείας και ασπίδας με βάση τους πολλαπλασιαστές 0.5f και 2f αντίστοιχα. Αν η ασπίδα έχει εξαντληθεί, ολόκληρη η ζημιά πηγαίνει απευθείας στην υγεία

Μετά την εφαρμογή της ζημιάς, μηδενίζονται οι timers lerpTimer και shieldLerpTimer ώστε το chip damage effect να ξεκινήσει από την αρχή και να αποδοθεί σωστά οπτικά. Τέλος, ελέγχεται αν η υγεία έφτασε στο μηδέν και αν ο παίκτης δεν είναι ήδη νεκρός, οπότε ενεργοποιείται το Game Over.

```

public void TakeDamage(float damage)
{
    if (isDead) return;

    if (shield > 0)
    {
        float healthDamage = damage * 0.5f;
        float shieldDamage = damage * 2f;
        health -= healthDamage;
        shield -= shieldDamage;
        if (shield < 0f) shield = 0f;
    }
    else
    {
        health -= damage;
    }
    lerpTimer = 0f;
    shieldLerpTimer = 0f;

    if (frontShieldBar != null)
    {
        frontShieldBar.fillAmount = shield / maxShield;
    }

    if (health <= 0 && !isDead)
    {
        isDead = true;
        if (gameOverManager != null)
        {
            gameOverManager.ShowGameOver();
        }
    }
}

```

Εικόνα 40

Σύστημα Ασπίδας

Η ασπίδα λειτουργεί ως ένα επιπλέον στρώμα προστασίας. Όταν είναι ενεργή, η ζημιά κατανέμεται ως εξής: το μισό damage πηγαίνει στην υγεία και το διπλάσιο στην ασπίδα

```

public void TakeDamage(float damage)
{
    if (isDead) return;

    if (shield > 0)
    {
        float healthDamage = damage * 0.5f;
        float shieldDamage = damage * 2f;
        health -= healthDamage;
        shield -= shieldDamage;
    }
}

```

Εικόνα 41

Παράδειγμα: για 25 damage, ο παίκτης χάνει 12.5 HP και 50 Shield. Αυτό σημαίνει ότι η ασπίδα αντέχει περίπου 2 πλήρη χτυπήματα πριν εξαντληθεί. Η ασπίδα εμφανίζεται μέσω του frontShieldBar και ενημερώνεται με Lerp για ομαλή κίνηση.

Επιπλέον, το σύστημα ασπίδας περιλαμβάνει δύο βοηθητικές μεθόδους που διαχειρίζονται την πρόσθεση ασπίδας και την οπτική της απόδοση.

Η μέθοδος `AddShield()` καλείται όταν ο παίκτης αποκτά ασπίδα μέσω ειδικών αντικειμένων στο περιβάλλον. Προσθέτει την καθορισμένη ποσότητα ασπίδας και διασφαλίζει ότι η τιμή δεν ξεπερνά το `maxShield` μέσω απλού ελέγχου:

```
public void AddShield(float amount)
{
    shield += amount;
    if (shield > maxShield) shield = maxShield;

    if (frontShieldBar != null)
    {
        frontShieldBar.fillAmount = shield / maxShield;
    }
}
```

Εικόνα 42

Στη συνέχεια ενημερώνεται άμεσα το `frontShieldBar` με την νέα τιμή, ώστε ο παίκτης να βλέπει αμέσως την αλλαγή στο UI χωρίς καθυστέρηση.

Η μέθοδος `UpdateShieldUI()` καλείται κάθε frame από το `Update()` και διαχειρίζεται την ομαλή κίνηση της μπάρας ασπίδας μέσω `Mathf.Lerp()`:

Σε αντίθεση με το `AddShield()` που ενημερώνει άμεσα την μπάρα, το `UpdateShieldUI()` παρέχει μια ομαλή μετάβαση μεταξύ της τρέχουσας και της επιθυμητής τιμής, δίνοντας πιο φυσική αίσθηση όταν η ασπίδα μειώνεται σταδιακά από τις επιθέσεις των εχθρών.

Θάνατος Παίκτη

Όταν η υγεία φτάσει στο μηδέν, ενεργοποιείται το flag `isDead` και καλείται το `gameOverManager.ShowGameOver()`. Το flag `isDead` αποτρέπει την πολλαπλή ενεργοποίηση του Game Over σε περίπτωση που ο παίκτης δεχτεί ταυτόχρονα πολλαπλά χτυπήματα.

```
if (health <= 0 && !isDead)
{
    is[ (field) float PlayerHealth.health
    if (gameOverManager != null)
    {
        gameOverManager.ShowGameOver();
    }
}
```

Εικόνα 43

Αποκατάσταση Υγείας

Η μέθοδος `RestoreHealth()` αυξάνει την υγεία κατά το καθορισμένο ποσό και μηδενίζει τον `lerpTimer` ώστε το `chip damage effect` να ξεκινήσει από την αρχή. Η τιμή περιορίζεται αυτόματα από το `Mathf.Clamp()` στο `Update()` για να μην ξεπεράσει το `maxHealth`.

```
public void RestoreHealth(float healAmount)
{
    Debug.Log($"RestoreHealth called! health before: {health} healAmount: {healAmount}");
    health += healAmount;
    lerpTimer = 0f;
    Debug.Log($"health after: {health}");
}
```

Εικόνα 44

6.5 PlayerShooting.cs

Το `PlayerShooting.cs` είναι το πιο σύνθετο script του παίκτη, υπεύθυνο για ολόκληρο το σύστημα πυροβολισμού. Διαχειρίζεται τους τύπους πυροβολισμού, το σύστημα γεμιστήρων, το `reload`, τον εντοπισμό εχθρών μέσω `Raycast` και την εφαρμογή ζημιάς με διαφορετικούς πολλαπλασιαστές ανάλογα με το σημείο που χτυπήθηκε ο εχθρός.

Αρχικοποίηση

Στο `Awake()` εντοπίζεται το `PlayerUI` component για την ενημέρωση του `ammo display`. Στο `Start()` εντοπίζεται η κάμερα μέσω του `PlayerLook` component και αρχικοποιείται το σύστημα γεμιστήρων μέσω `ResetAmmo()`. Το script εγγράφεται επίσης στο `SceneManager.sceneLoaded` event ώστε να γίνεται `reset` των πυρομαχικών σε κάθε νέα `scene`.

```

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

public class PlayerShooting : MonoBehaviour
{
    private Camera cam;
    private InputManager inputManager;
    private PlayerUI playerUI;

    public GameObject bulletPrefab;
    public Transform gunBarrel;
    public float bulletSpeed = 150f;
    public float fireRate = 0.2f;
    public float reloadTime = 1.5f;
    public int damage = 25;
    public float shootDistance = 100f;
    public bool isAutomatic = false;
    public bool isShotgun = false;
    public int pelletsCount = 8;
    public float spreadAngle = 15f;

    [Header("Magazine System")]
    public int magazineSize = 12;
    public int maxMagazines = 3;

    private void OnEnable() => SceneManager.sceneLoaded += OnSceneResetAmmo;
    private void OnDisable() => SceneManager.sceneLoaded -= OnSceneResetAmmo;

    private int currentAmmo;
    private int currentMagazines;
    private float nextFireTime = 0f;
    private bool isReloading = false;
    private WeaponHandler currentWeaponHandler;

    void Awake()
    {
        playerUI = GetComponent<PlayerUI>();
    }

    void Start()
    {
        cam = GetComponent<PlayerLook>().cam;
        inputManager = GetComponent<InputManager>();
        ResetAmmo();
    }
}

```

Εικόνα 45

Η μέθοδος SetWeaponHandler() καλείται από το WeaponSystem κάθε φορά που αλλάζει όπλο, ενημερώνοντας το reference στον τρέχοντα WeaponHandler για πρόσβαση στους ήχους και τα animations του όπλου.

```

public void SetWeaponHandler(WeaponHandler wh)
{
    currentWeaponHandler = wh;
}

```

Εικόνα 46

Update — Λογική Πυροβολισμού

Στο Update() υλοποιείται ολόκληρη η λογική πυροβολισμού. Πρώτα ελέγχεται αν το παιχνίδι είναι σε pause για να αποτραπεί οποιαδήποτε ενέργεια. Στη συνέχεια καθορίζεται αν ο παίκτης πιέζει το κουμπί πυροβολισμού, διαφορετικά ανάλογα με τον τύπο του όπλου

```

void Update()
{
    if (inputManager.isPaused) return;

    bool firePressed = isAutomatic
        ? inputManager.onFoot.Fire.ReadValue<float>() > 0.5f
        : inputManager.onFoot.Fire.triggered;

    // Interrupt reload MONO για shotgun
    if (isReloading && isShotgun && firePressed && currentAmmo > 0)
    {
        StopAllCoroutines();
        isReloading = false;
        if (currentWeaponHandler != null)
            currentWeaponHandler.StopReloadSound();
    }

    if (isReloading) return;

    if (inputManager.onFoot.Reload.triggered)
    {
        if (currentAmmo < magazineSize && currentMagazines > 0)
            StartCoroutine(Reload());
        else if (currentMagazines <= 0)
            Debug.Log("No Ammo!");
        return;
    }

    if (currentAmmo <= 0)
    {
        if (currentMagazines > 0)
            StartCoroutine(Reload());
        else
            Debug.Log("No Ammo!");
        return;
    }

    if (firePressed && Time.time >= nextFireTime)
    {
        if (isShotgun) ShootShotgun();
        else Shoot();
    }
}

```

Εικόνα 47

Για τα αυτόματα όπλα (Uzi, M4, AK-47) διαβάζεται η συνεχής τιμή του πλήκτρου (ReadValue), ενώ για τα semi-auto (Πιστόλι) χρησιμοποιείται το triggered που ενεργοποιείται μόνο τη στιγμή που πατιέται το πλήκτρο.

Ιδιαίτερα για το Shotgun υλοποιείται η δυνατότητα interrupt reload: αν ο παίκτης πατήσει fire κατά τη διάρκεια του reload και έχει έστω μία σφαίρα, το reload διακόπτεται άμεσα

Αυτό είναι ρεαλιστικό χαρακτηριστικό των Shotgun στα FPS παιχνίδια, όπου ο παίκτης μπορεί να διακόψει το γέμισμα αν χρειαστεί να πυροβολήσει άμεσα. Στη συνέχεια ελέγχεται η ανάγκη για reload — είτε αυτόματα όταν τελειώσουν οι σφαίρες, είτε χειροκίνητα με το πλήκτρο R.

HandleHit — Σύστημα Ζημιάς

Η μέθοδος HandleHit() είναι υπεύθυνη για τον εντοπισμό του στόχου που χτυπήθηκε και την εφαρμογή της κατάλληλης ζημιάς. Διακρίνει τις παρακάτω περιπτώσεις:

Πρώτα ελέγχεται αν χτυπήθηκε ο Final Boss. Αναζητείται το FinalBoss component τόσο στο χτυπημένο object όσο και στο root του για ασφάλεια. Αν βρεθεί BossHealth component στο hitbox, λαμβάνεται ο αντίστοιχος πολλαπλασιαστής ζημιάς:

```

BossHealth bossHitBox = hitInfo.collider.GetComponent<BossHealth>();
FinalBoss targetBoss = hitInfo.collider.GetComponentInParent<FinalBoss>();

// Fail-safe αν το FinalBoss είναι στο root
if (targetBoss == null)
{
    targetBoss = hitInfo.collider.transform.root.GetComponentInChildren<FinalBoss>();
}

if (targetBoss != null)
{
    float multiplier = 1f;
    string partName = "Main Body";

    // Αν χτυπήσαμε συγκεκριμένο μέλος που έχει BossHealth, παίρνουμε το multi του
    if (bossHitBox != null)
    {
        multiplier = bossHitBox.damageMultiplier;
        partName = bossHitBox.bodyPart.ToString();
    }

    float totalDamage = damage * multiplier;

    Debug.Log($"[HandleHit -> BOSS] Found Boss! Part: {partName}, Multiplier: {multiplier}x, Final Damage: {totalDamage}");

    // Στέλνουμε τη ζημιά στο Boss
    targetBoss.TakeDamage(totalDamage);
    return;
}

```

Εικόνα 48

Αν δεν χτυπήθηκε ο Boss, ελέγχεται αν χτυπήθηκε κανονικός εχθρός μέσω των components HitBox και EnemyHealth. Το HitBox component επιτρέπει διαφορετικούς πολλαπλασιαστές ανάλογα με το σημείο που χτυπήθηκε ο εχθρός — κεφάλι, σώμα ή πόδια:

```

HitBox regularHitBox = hitInfo.collider.GetComponent<HitBox>();
EnemyHealth regularEnemy = hitInfo.collider.GetComponentInParent<EnemyHealth>();

if (regularEnemy != null || regularHitBox != null)
{
    if (regularEnemy == null && regularHitBox != null)
    {
        regularEnemy = regularHitBox.GetComponentInParent<EnemyHealth>();
    }

    if (regularEnemy != null)
    {
        float multiplier = (regularHitBox != null) ? regularHitBox.GetMultiplier() : 1f;
        int finalDamage = Mathf.RoundToInt(damage * multiplier);
        string partName = (regularHitBox != null) ? regularHitBox.bodyPart.ToString() : "Body";

        regularEnemy.TakeDamage(finalDamage);
        Debug.Log($"[HandleHit -> ENEMY] Hit Enemy {partName}! Damage: {finalDamage}");
        return;
    }
}

```

Εικόνα 49

Shoot() — Πυροβολισμός Όπλων

Η μέθοδος Shoot() υλοποιεί τον πυροβολισμό για όλα τα όπλα εκτός από το shotgun (Πιστόλι, Uzi, M4, AK-47). Πρώτα ενημερώνεται ο χρόνος επόμενου πυροβολισμού και μειώνεται ο αριθμός σφαιρών:

Στη συνέχεια ενεργοποιούνται το animation και ο ήχος πυροβολισμού μέσω του WeaponHandler. Ο εντοπισμός στόχου γίνεται μέσω **Raycast** από τη θέση της κάμερας προς την κατεύθυνση που κοιτά.

Η χρήση του ~LayerMask.GetMask("Player") εξασφαλίζει ότι η ακτίνα αγνοεί τον ίδιο τον παίκτη και δεν χτυπά τον εαυτό του. Αν η ακτίνα βρει στόχο, καλείται η HandleHit() για εφαρμογή ζημιάς. Τέλος καλείται η SpawnBullet() για οπτική αναπαράσταση.

```
private void Shoot()
{
    nextFireTime = Time.time + fireRate;
    currentAmmo--;
    UpdateAmmoUI();

    if (currentWeaponHandler != null)
    {
        currentWeaponHandler.PlayFireAnimation();
        currentWeaponHandler.PlayFireSound();
    }

    int layerMask = ~LayerMask.GetMask("Player");
    Ray ray = new Ray(cam.transform.position, cam.transform.forward);
    RaycastHit hitInfo;

    Vector3 targetPoint;

    if (Physics.Raycast(ray, out hitInfo, shootDistance, layerMask))
    {
        Debug.Log("HIT: " + hitInfo.collider.gameObject.name);
        targetPoint = hitInfo.point;
        HandleHit(hitInfo);
    }
    else
    {
        Debug.Log("MISS - nothing hit");
        targetPoint = ray.origin + ray.direction * shootDistance;
    }

    SpawnBullet(targetPoint);
}
```

Εικόνα 50

ShootShotgun() — Πυροβολισμός Shotgun

Η μέθοδος ShootShotgun() υλοποιεί τον πυροβολισμό του Shotgun με **8 pellets** σε κυκλική διασπορά. Για κάθε pellet υπολογίζεται τυχαία απόκλιση από την κεντρική κατεύθυνση. Κάθε pellet εκτελεί ξεχωριστό Raycast στην κατεύθυνσή του και εφαρμόζει ζημιά ανεξάρτητα. Αυτό σημαίνει ότι ένα πλήρες χτύπημα Shotgun μπορεί να κάνει έως και 8 φορές τη βασική ζημιά αν όλα τα pellets βρουν στόχο κάνοντάς το ιδιαίτερα αποτελεσματικό σε κοντινές αποστάσεις αλλά λιγότερο αξιόπιστο από μακριά.

```

private void ShootShotgun()
{
    nextFireTime = Time.time + fireRate;
    currentAmmo--;
    UpdateAmmoUI();

    if (currentWeaponHandler != null)
    {
        currentWeaponHandler.PlayFireAnimation();
        currentWeaponHandler.PlayFireSound();
    }

    int layerMask = ~LayerMask.GetMask("Player");

    for (int i = 0; i < pelletsCount; i++)
    {
        Vector3 spread = new Vector3(
            Random.Range(-spreadAngle, spreadAngle),
            Random.Range(-spreadAngle, spreadAngle),
            0
        );
        Vector3 spreadDir = Quaternion.Euler(spread) * cam.transform.forward;
        Ray ray = new Ray(cam.transform.position, spreadDir);
        RaycastHit hitInfo;

        Vector3 targetPoint;
        if (Physics.Raycast(ray, out hitInfo, shootDistance, layerMask))
        {
            targetPoint = hitInfo.point;
            HandleHit(hitInfo);
        }
        else
        {
            targetPoint = ray.origin + ray.direction * shootDistance;
        }

        SpawnBullet(targetPoint);
    }
}

```

Εικόνα 51

SpawnBullet() — Οπτική Αναπαράσταση

Η μέθοδος SpawnBullet() δημιουργεί ένα οπτικό bullet prefab που κινείται προς τον στόχο, παρέχοντας οπτική ανατροφοδότηση στον παίκτη. Σημαντικό: ο πραγματικός εντοπισμός στόχου γίνεται μέσω Raycast (instant-hit), ενώ το bullet είναι μόνο για αισθητικούς λόγους. Το bullet καταστρέφεται αυτόματα μετά από 3 δευτερόλεπτα για να αποφευχθεί συσσώρευση objects στη σκηνή.

```

private void SpawnBullet(Vector3 targetPoint)
{
    if (bulletPrefab == null || gunBarrel == null) return;
    Vector3 dir = (targetPoint - gunBarrel.position).normalized;
    GameObject bullet = Instantiate(bulletPrefab, gunBarrel.position, Quaternion.FromToRotation(Vector3.up, dir));
    Rigidbody rb = bullet.GetComponent<Rigidbody>();
    if (rb != null) rb.velocity = dir * bulletSpeed;
    Destroy(bullet, 3f);
}

```

Εικόνα 52

ResetAmmo() και UpdateAmmoUI()

Η `ResetAmmo()` καλείται κατά την αλλαγή όπλου και κατά τη φόρτωση νέας scene, επαναφέροντας τις σφαίρες στις αρχικές τιμές του τρέχοντος όπλου. Η `UpdateAmmoUI()` ενημερώνει το ammo display στο UI κάθε φορά που αλλάζει ο αριθμός σφαιρών μετά από πυροβολισμό, reload ή αλλαγή όπλου. Αν το `playerUI` είναι null, το εντοπίζει δυναμικά μέσω `GetComponent()`.

```
public void UpdateAmmoUI()
{
    if (playerUI == null) playerUI = GetComponent<PlayerUI>();
    if (playerUI != null)
    {
        playerUI.UpdateAmmo(currentAmmo, magazineSize, currentMagazines);
    }
}

public void ResetAmmo()
{
    currentAmmo = magazineSize;
    currentMagazines = maxMagazines;
    isReloading = false;
    UpdateAmmoUI();
}
```

Εικόνα 53

Reload() — Σύστημα Επαναγέμισης

Η μέθοδος `Reload()` είναι Coroutine και υλοποιεί δύο εντελώς διαφορετικές λογικές επαναγέμισης ανάλογα με τον τύπο του όπλου.

Για τα **κανονικά όπλα**, η επαναγέμιση είναι απλή: παίζει ο ήχος reload, περιμένει το `reloadTime` και γεμίζει αμέσως ολόκληρος ο γεμιστήρας.

Για το **Shotgun** η λογική είναι εντελώς διαφορετική και πιο ρεαλιστική. Αντί να γεμίζει ολόκληρος ο γεμιστήρας ταυτόχρονα, οι σφαίρες εισάγονται **μία-μία** με ξεχωριστό ήχο για κάθε μία. Ο χρόνος ανά σφαίρα υπολογίζεται διαιρώντας τη διάρκεια του audio clip με τον αριθμό των σφαιρών.

Αυτή η υλοποίηση έχει δύο σημαντικά πλεονεκτήματα. Πρώτον, ο ήχος συγχρονίζεται απόλυτα με τον αριθμό σφαιρών που εισάγονται. Δεύτερον, το UI ενημερώνεται σε πραγματικό χρόνο για κάθε σφαίρα που προστίθεται, ώστε ο παίκτης να βλέπει τον αριθμό σφαιρών να αυξάνεται σταδιακά. Μετά την ολοκλήρωση του γεμίσματος παίζει ο τελικός ήχος.

Σε συνδυασμό με το `interrupt reload` που υλοποιείται στο `Update()`, ο παίκτης μπορεί να διακόψει το γέμισμα του Shotgun οποιαδήποτε στιγμή έχει έστω μία σφαίρα, ακριβώς όπως συμβαίνει στα πραγματικά FPS παιχνίδια.

```

private IEnumerator Reload()
{
    isReloading = true;
    if (playerUI != null) playerUI.UpdateText("Reloading...");

    if (currentWeaponHandler != null)
    {
        if (isShotgun)
        {
            float timePerBullet = currentWeaponHandler.shotgunInsertSound != null
                ? currentWeaponHandler.shotgunInsertSound.length / 8f
                : 0.5f;

            // Βάζει μια σφαίρα κάθε φορά
            while (currentAmmo < magazineSize && currentMagazines > 0)
            {
                // Παιζει ηχο για μια σφαίρα
                if (currentWeaponHandler.audioSource != null && currentWeaponHandler.shotgunInsertSound != null)
                {
                    currentWeaponHandler.audioSource.clip = currentWeaponHandler.shotgunInsertSound;
                    currentWeaponHandler.audioSource.time = (currentAmmo) * timePerBullet;
                    currentWeaponHandler.audioSource.Play();
                }

                yield return new WaitForSeconds(timePerBullet);

                // Προσθετει τη σφαίρα
                currentAmmo++;
                UpdateAmmoUI();

                // Αν γεμισε ο γεμιστήρας, σταματά
                if (currentAmmo >= magazineSize) break;
            }

            currentMagazines--;

            // Ηχος pump στο τέλος
            if (currentWeaponHandler.shotgunEndSound != null)
                currentWeaponHandler.audioSource.PlayOneShot(currentWeaponHandler.shotgunEndSound);
        }
        else
        {
            currentWeaponHandler.PlayReloadSound();
            yield return new WaitForSeconds(reloadTime);
            currentAmmo = magazineSize;
            currentMagazines--;
        }
    }
}

```

Εικόνα 54

```

        currentMagazines--;
    }
}
else
{
    yield return new WaitForSeconds(reloadTime);
    currentAmmo = magazineSize;
    currentMagazines--;
}

isReloading = false;
UpdateAmmoUI();
}

```

Εικόνα 55

RefillAmmo() - Αναπλήρωση Πυρομαχικών

Η μέθοδος RefillAmmo() καλείται από το AmmoPickup script όταν ο παίκτης μαζεύει ammo box. Διακόπτει οποιοδήποτε τρέχον reload Coroutine μέσω StopAllCoroutines(), προσθέτει τους νέους γεμιστήρες και γεμίζει αμέσως τον τρέχοντα γεμιστήρα:

Στη συνέχεια καλείται το VerifyAmmo() Coroutine που επαληθεύει μετά από 0.1 δευτερόλεπτα ότι οι τιμές αποθηκεύτηκαν σωστά και ενημερώνει το UI.

```

public void RefillAmmo(int magazines)
{
    StopAllCoroutines();
    currentMagazines += magazines;
    currentAmmo = magazineSize;
    isReloading = false;

    Debug.Log($"[RefillAmmo] currentMagazines={currentMagazines} maxMagazines={maxMagazines}");

    // Verify αμεσως μετα
    StartCoroutine(VerifyAmmo());
}

```

Εικόνα 56

OnSceneResetAmmo()

Η μέθοδος αυτή καλείται αυτόματα κάθε φορά που φορτώνεται νέα scene μέσω του event `SceneManager.sceneLoaded`. Καλεί απλώς την `ResetAmmo()` για να επαναφέρει τα πυρομαχικά στις αρχικές τιμές του τρέχοντος όπλου, εξασφαλίζοντας ότι ο παίκτης ξεκινά κάθε επίπεδο με πλήρη γεμιστήρα.

```
private void OnSceneResetAmmo(Scene scene, LoadSceneMode mode)
{
    ResetAmmo();
}
```

Εικόνα 57

6.6 WeaponSystem.cs

Το `WeaponSystem.cs` είναι το κεντρικό script διαχείρισης του συστήματος όπλων. Είναι υπεύθυνο για την εξέλιξη των όπλων μέσω XP, την εναλλαγή μεταξύ τους και τη μεταφορά των stats κάθε όπλου στο `PlayerShooting`.

Αρχικοποίηση

Στο `Start()` αρχικοποιείται το σύστημα: καταστρέφονται τυχόν υπάρχοντα children του `weaponHolder` για να αποφευχθεί διπλασιασμός όπλων, και εξοπλίζεται το πρώτο όπλο (Πιστόλι) μέσω `EquipWeapon(0)`. Επίσης αποθηκεύεται το τρέχον XP ως `xpAtLevelStart` για χρήση στο restart.

```
void Start()
{
    xpAtLevelStart = currentXP;
    playerShooting = GetComponent<PlayerShooting>();
    playerUI = GetComponent<PlayerUI>();

    foreach (Transform child in weaponHolder)
        Destroy(child.gameObject);

    EquipWeapon(0);
    playerUI.UpdateXP(0, GetXPForNextLevel());

    if (armsHolder != null)
    {
        armsHolder.localPosition = currentHandler.armsPositionOffset;
        armsHolder.localRotation = Quaternion.Euler(currentHandler.armsRotationOffset);
    }
    else
    {
        Debug.LogWarning("armsHolder is NULL!");
    }
}
```

Εικόνα 58

Το script εγγράφεται επίσης στο SceneManager.sceneLoaded event ώστε να ενημερώνεται το xpAtLevelStart κάθε φορά που φορτώνεται νέα scene — εξασφαλίζοντας ότι σε περίπτωση restart ο παίκτης επιστρέφει στο XP που είχε κατά την είσοδο στο τρέχον επίπεδο.

Σύστημα XP και Αναβάθμισης

Η μέθοδος AddXP() καλείται από το EnemyHealth κάθε φορά που σκοτώνεται εχθρός, προσθέτοντας 25 XP. Μετά από κάθε προσθήκη ελέγχεται αν το συσσωρευμένο XP ξεπερνά το κατώφλι του επόμενου επιπέδου.

```
public void AddXP(int amount)
{
    currentXP += amount;
    playerUI.UpdateXP(currentXP, GetXPForNextLevel());

    if (currentLevel < xpThresholds.Length)
    {
        if (currentXP >= xpThresholds[currentLevel])
            LevelUp(currentLevel + 1);
    }
}
```

Εικόνα 59

Τα thresholds των XP ορίζονται στον πίνακα xpThresholds = { 0, 100, 250, 400, 550 }, αντιστοιχώντας στα πέντε επίπεδα όπλων. Η μέθοδος LevelUp() αυξάνει το επίπεδο και καλεί αμέσως την EquipWeapon() για αυτόματη αναβάθμιση του όπλου, ενώ ενημερώνεται και το XP Bar στο UI.

```
private void LevelUp(int newLevel)
{
    currentLevel = newLevel;
    Debug.Log($"Level Up! Level {currentLevel}");
    EquipWeapon(currentLevel - 1);
    playerUI.UpdateXP(currentXP, GetXPForNextLevel());
}
```

Εικόνα 60

EquipWeapon()

Η μέθοδος EquipWeapon() είναι η πιο σύνθετη του script και εκτελείται κάθε φορά που αλλάζει όπλο — είτε από αναβάθμιση XP είτε από επιλογή στο inventory. Πρώτα καταστρέφονται όλα τα υπάρχοντα children του weaponHolder για να αφαιρεθεί το προηγούμενο όπλο, και στη συνέχεια δημιουργείται το νέο από το αντίστοιχο prefab. Αφού εντοπιστεί το WeaponHandler component του νέου όπλου, μεταφέρονται όλα τα stats του στο PlayerShooting. Αυτή η προσέγγιση επιτρέπει κάθε όπλο να έχει εντελώς διαφορετικά χαρακτηριστικά χωρίς να χρειάζεται αλλαγή κώδικα. Επίσης ρυθμίζεται η θέση, περιστροφή και κλίμακα του όπλου στον χώρο μέσω των offsets που ορίζονται στον WeaponHandler.

Τέλος, ρυθμίζεται η θέση των χεριών (armsHolder) και καλείται η ResetAmmo() για να γεμίσει αυτόματα ο γεμιστήρας του νέου όπλου.

```
private void EquipWeapon(int index)
{
    if (index >= weaponPrefabs.Length) return;

    foreach (Transform child in weaponHolder)
        Destroy(child.gameObject);

    currentWeapon = Instantiate(weaponPrefabs[index], weaponHolder);
    currentHandler = currentWeapon.GetComponent<WeaponHandler>();

    if (currentHandler != null)
    {
        currentHandler.armsHolder = armsHolder;

        currentWeapon.transform.localPosition = currentHandler.positionOffset;
        currentWeapon.transform.localRotation = Quaternion.Euler(currentHandler.rotationOffset);
        currentWeapon.transform.localScale = currentHandler.scaleOverride;

        playerShooting.SetWeaponHandler(currentHandler);
        playerShooting.damage = currentHandler.damage;
        playerShooting.fireRate = currentHandler.fireRate;
        playerShooting.reloadTime = currentHandler.reloadTime;
        playerShooting.shootDistance = currentHandler.shootDistance;
        playerShooting.isAutomatic = currentHandler.isAutomatic;
        playerShooting.isShotgun = currentHandler.isShotgun;
        playerShooting.pelletsCount = currentHandler.pelletsCount;
        playerShooting.spreadAngle = currentHandler.spreadAngle;
        playerShooting.bulletPrefab = currentHandler.bulletPrefab;
        playerShooting.bulletSpeed = currentHandler.bulletSpeed;
        playerShooting.magazineSize = currentHandler.magazineSize;
        playerShooting.maxMagazines = currentHandler.maxMagazines;
        playerShooting.gunBarrel = currentHandler.gunBarrel != null
            ? currentHandler.gunBarrel
            : currentWeapon.transform;

        if (armsHolder != null)
        {
            armsHolder.localPosition = currentHandler.armsPositionOffset;
            armsHolder.localRotation = Quaternion.Euler(currentHandler.armsRotationOffset);
        }

        Debug.Log($"Equipped: {currentHandler.weaponName} | Damage: {currentHandler.damage} | Level: {currentLevel}");
    }

    playerShooting.ResetAmmo();
}
```

ResetXPToLevelStart() — Επαναφορά XP

Η μέθοδος ResetXPToLevelStart() καλείται από το GameOverManager κατά το restart. Επαναφέρει το XP στην τιμή που είχε ο παίκτης κατά την είσοδο στο τρέχον επίπεδο, αποτρέποντας την κατάχρηση του συστήματος XP μέσω επαναλαμβανόμενων προσπαθειών. Ο βρόχος διατρέχει τα κατώφλια από το υψηλότερο προς το χαμηλότερο για να βρει το σωστό επίπεδο που αντιστοιχεί στο αποκατεστημένο XP, και εξοπλίζει αυτόματα το κατάλληλο όπλο.

```
public void ResetXPToLevelStart()
{
    currentXP = xpAtLevelStart;
    for (int i = xpThresholds.Length - 1; i >= 0; i--)
    {
        if (currentXP >= xpThresholds[i])
        {
            currentLevel = i + 1;
            EquipWeapon(currentLevel - 1);
            break;
        }
    }

    if (playerUI != null)
        playerUI.UpdateXP(currentXP, GetXPForNextLevel());
    Debug.Log($"XP reset to: {currentXP} Level: {currentLevel}");
}
```

Εικόνα 61

Βοηθητικές Μέθοδοι

Το script παρέχει μια σειρά από public μεθόδους που χρησιμοποιούνται από άλλα scripts: Η `GetCurrentLevel()` και `GetCurrentXP()` επιστρέφουν το τρέχον επίπεδο και XP για χρήση από το UI και το inventory. Η `GetXPForNextLevel()` επιστρέφει το κατώφλι του επόμενου επιπέδου ή -1 αν ο παίκτης έχει φτάσει στο μέγιστο επίπεδο. Η `GetCurrentWeaponName()` και `GetCurrentWeaponIndex()` χρησιμοποιούνται από το `InventorySystem` για την εμφάνιση του ονόματος και του εικονιδίου κάθε όπλου στο inventory panel. Τέλος, η `EquipWeaponByIndex()` επιτρέπει στο inventory να εξοπλίζει οποιοδήποτε διαθέσιμο όπλο, ελέγχοντας πρώτα αν ο παίκτης έχει φτάσει στο απαιτούμενο επίπεδο XP.

```
public int GetCurrentLevel() => currentLevel;
public int GetCurrentXP() => currentXP;
public int GetXPForNextLevel() => currentLevel < xpThresholds.Length ? xpThresholds[currentLevel] : -1;
public string GetCurrentWeaponName() => currentHandler != null ? currentHandler.weaponName : "None";
public int GetCurrentWeaponIndex() => currentLevel - 1;

public string GetWeaponName(int index)
{
    if (index < weaponPrefabs.Length)
    {
        WeaponHandler wh = weaponPrefabs[index].GetComponent<WeaponHandler>();
        if (wh != null) return wh.weaponName;
        return weaponPrefabs[index].name;
    }
    return "Unknown";
}

public void EquipWeaponByIndex(int index)
{
    if (index < currentLevel)
        EquipWeapon(index);
}

public Sprite GetWeaponIcon(int index)
{
    if (index < weaponPrefabs.Length)
    {
        WeaponHandler wh = weaponPrefabs[index].GetComponent<WeaponHandler>();
        if (wh != null) return wh.weaponIcon;
    }
    return null;
}
```

Εικόνα 62

6.7 WeaponHandler.cs

Το `WeaponHandler.cs` είναι το script που αποθηκεύει όλα τα χαρακτηριστικά και τις λειτουργίες κάθε όπλου ξεχωριστά. Λειτουργεί ως data container και controller για κάθε weapon prefab, παρέχοντας στο `WeaponSystem` και το `PlayerShooting` πρόσβαση σε όλα τα stats και τις μεθόδους του όπλου.

[ExecuteAlways]

Η κλάση διακοσμείται με το attribute `[ExecuteAlways]`, το οποίο επιτρέπει στο script να εκτελείται και εντός του Unity Editor χωρίς να χρειάζεται να τρέχει το παιχνίδι. Αυτό χρησιμοποιείται για την άμεση προεπισκόπηση των αλλαγών στη θέση και περιστροφή του όπλου κατά τη διάρκεια του level design.

```

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

[ExecuteAlways]
public class WeaponHandler : MonoBehaviour
{

```

Εικόνα 63

Οργάνωση με Headers

Το script χρησιμοποιεί εκτενώς [Header] attributes για την οργάνωση των πεδίων στον Inspector σε λογικές ομάδες: Weapon Info, Magazine Settings, Shotgun Settings, Visual, Arms, Audio και Inventory. Αυτό διευκολύνει σημαντικά τη ρύθμιση κάθε όπλου από τον Inspector χωρίς να χρειάζεται αλλαγή κώδικα.

```

[Header("Weapon Info")]
public string weaponName;
public int damage = 25;
public float fireRate = 0.2f;
public float reloadTime = 1.5f;
public float shootDistance = 100f;
public bool isAutomatic = false;
public bool isShotgun = false;

[Header("Magazine Settings")]
public int magazineSize = 12;
public int maxMagazines = 3;

[Header("Shotgun Settings")]
public int pelletsCount = 8;
public float spreadAngle = 15f;

[Header("Shotgun Reload")]
public AudioClip shotgunInsertSound;
public AudioClip shotgunEndSound;

public float insertInterval = 0.5f;

[Header("Visual")]
public Transform gunBarrel;
public GameObject bulletPrefab;
public float bulletSpeed = 150f;
public Vector3 positionOffset;
public Vector3 rotationOffset;
public Vector3 scaleOverride = Vector3.one;

[Header("Arms")]
public Vector3 armsPositionOffset;
public Vector3 armsRotationOffset;
[HideInInspector] public Transform armsHolder;

[HideInInspector] public Animator weaponAnimator;

[Header("Audio")]
public AudioClip fireSound;
public float fireSoundDuration = 0.1f;
public AudioSource audioSource;
private AudioSource reloadAudioSource;
public AudioClip reloadSound;
private Coroutine stopSoundCoroutine;

[Header("Inventory")]

```

Εικόνα 64

ApplyOffsets()

Η μέθοδος ApplyOffsets() καλείται τόσο στο Start() όσο και στο Update(), εφαρμόζοντας τα offsets θέσης, περιστροφής και κλίμακας του όπλου καθώς και τη θέση των χειριών (armsHolder). Η κλήση στο Update() σε συνδυασμό με το [ExecuteAlways] επιτρέπει real-time προεπισκόπηση στον Editor.

```

void Start()
{
    weaponAnimator = GetComponent<Animator>();
    audioSource = GetComponent<AudioSource>();
    if (audioSource == null)
        audioSource = gameObject.AddComponent<AudioSource>();

    reloadAudioSource = gameObject.AddComponent<AudioSource>();
    ApplyOffsets();
}

void Update()
{
    ApplyOffsets();
}

public void ApplyOffsets()
{
    transform.localPosition = positionOffset;
    transform.localRotation = Quaternion.Euler(rotationOffset);
    transform.localScale = scaleOverride;

    if (armsHolder != null)
    {
        armsHolder.localPosition = armsPositionOffset;
        armsHolder.localRotation = Quaternion.Euler(armsRotationOffset);
    }
}

```

Εικόνα 65

Σύστημα Ήχου

Το script χρησιμοποιεί δύο ξεχωριστά AudioSources για να αποφευχθεί αλληλεπικάλυψη ήχων. Το audioSource χρησιμοποιείται για τον ήχο πυροβολισμού και το Shotgun reload, ενώ το reloadAudioSource δημιουργείται δυναμικά στο Start() και χρησιμοποιείται αποκλειστικά για τον ήχο reload των κανονικών όπλων. Η μέθοδος PlayFireSound() παίζει τον ήχο πυροβολισμού και ξεκινά ένα Coroutine που σταματά τον ήχο μετά από fireSoundDuration δευτερόλεπτα. Αυτό αποτρέπει την επικάλυψη ήχων σε αυτόματα όπλα που πυροβολούν με υψηλό ρυθμό.

```

public void PlayFireSound()
{
    if (audioSource == null || fireSound == null) return;
    if (stopSoundCoroutine != null)
        StopCoroutine(stopSoundCoroutine);
    audioSource.PlayOneShot(fireSound);
    stopSoundCoroutine = StartCoroutine(StopSoundAfter(fireSoundDuration));
}

```

Εικόνα 66

PlayShotgunReload()

Η πιο σύνθετη μέθοδος ήχου είναι το PlayShotgunReload() Coroutine, το οποίο συγχρονίζει τον ήχο reload με τον αριθμό σφαιρών που εισάγονται. Το audio clip διαιρείται σε ίσα τμήματα ένα ανά σφαίρα και παίζει από τη σωστή χρονική θέση για κάθε σφαίρα:

Έτσι αν χρειάζονται 3 σφαίρες, ακούγεται μόνο το αντίστοιχο τμήμα του clip και όχι ολόκληρος ο ήχος δίνοντας πλήρη ρεαλισμό στο σύστημα επαναγέμισης.

```

public IEnumerator PlayShotgunReload(int bulletsToReload)
{
    if (shotgunInsertSound == null) yield break;

    float timePerBullet = shotgunInsertSound.length / 8f; // 11sec / 8 σφαίρες

    for (int i = 0; i < bulletsToReload; i++)
    {
        // Παίζει από τη σωστή θέση του clip
        audioSource.clip = shotgunInsertSound;
        audioSource.time = i * timePerBullet;
        audioSource.Play();

        yield return new WaitForSeconds(timePerBullet);

        audioSource.Stop();
    }

    // Ηχος rump στο τέλος
    if (shotgunEndSound != null)
        audioSource.PlayOneShot(shotgunEndSound);
}

```

Εικόνα 67

Animations

Οι μέθοδοι PlayFireAnimation() και PlayReloadAnimation() ενεργοποιούν τα αντίστοιχα Triggers στον Animator του όπλου. Στο project μας μόνο το πιστόλι έχει FireAnimation γεγονός το οποίο αφήνει αυτές τις μεθόδους για μελλοντικό implementation. Το StopReloadSound() σταματά και τα δύο AudioSources ταυτόχρονα, χρησιμοποιείται κυρίως κατά το interrupt reload του Shotgun.

```

public void PlayFireAnimation()
{
    if (weaponAnimator != null)
        weaponAnimator.SetTrigger("Fire");
}

```

Εικόνα 68

```

public void PlayReloadAnimation()
{
    if (weaponAnimator != null)
        weaponAnimator.SetTrigger("Reload");
}

```

Εικόνα 69

6.8 InventorySystem.cs

Το InventorySystem.cs διαχειρίζεται το σύστημα inventory του παιχνιδιού, επιτρέποντας στον παίκτη να βλέπει και να επιλέγει οποιοδήποτε όπλο έχει ξεκλειδώσει μέχρι εκείνη τη στιγμή. Είναι DontDestroyOnLoad και επιβιώνει σε όλα τα επίπεδα.

Αρχικοποίηση και DontDestroyOnLoad

Στο Awake() το script ορίζεται ως persistent object και εγγράφεται στο SceneManager.sceneLoaded event.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using UnityEngine.SceneManagement;

public class InventorySystem : MonoBehaviour
{
    [Header("UI")]
    [SerializeField] private GameObject inventoryPanel;
    [SerializeField] private Transform weaponSlotContainer;
    [SerializeField] private GameObject weaponSlotPrefab;

    [Header("References")]
    [SerializeField] private WeaponSystem weaponSystem;
    [SerializeField] private InputManager inputManager;

    private bool isOpen = false;

    void Awake()
    {
        DontDestroyOnLoad(gameObject);
        SceneManager.sceneLoaded += OnSceneLoaded;
    }
}

```

Εικόνα 70

Αυτό εξασφαλίζει ότι το inventory λειτουργεί σωστά σε όλα τα επίπεδα χωρίς να χρειάζεται επανεκκίνηση.

Δυναμική Εύρεση UI References

Το πιο σημαντικό χαρακτηριστικό του script είναι η δυναμική εύρεση των UI panels μετά από κάθε αλλαγή scene. Στο OnSceneLoaded() μηδενίζονται τα references και αναζητούνται εκ νέου μέσω Resources.FindObjectsOfTypeAll<Canvas>().

```

private void OnSceneLoaded(Scene scene, LoadSceneMode mode)
{
    inventoryPanel = null;
    weaponSlotContainer = null;

    Canvas[] canvases = Resources.FindObjectsOfTypeAll<Canvas>();
    foreach (Canvas c in canvases)
    {
        if (c == null) continue;

        Transform ip = c.transform.Find("InventoryPanel");
        if (ip != null) inventoryPanel = ip.gameObject;

        Transform wsc = ip?.Find("WeaponSlotContainer");
        if (wsc != null) weaponSlotContainer = wsc;
    }

    GameObject player = GameObject.FindGameObjectWithTag("Player");
    if (player != null)
    {
        weaponSystem = player.GetComponent<WeaponSystem>();
        inputManager = player.GetComponent<InputManager>();
    }

    isOpen = false;
    if (inventoryPanel != null)
        inventoryPanel.SetActive(false);
}

```

Εικόνα 71

Η χρήση `Resources.FindObjectsOfTypeAll<Canvas>()` αντί για `FindObjectOfType()` είναι κρίσιμη, καθώς επιτρέπει την εύρεση objects ακόμα και αν είναι inactive. Επίσης εντοπίζονται δυναμικά τα references σε `WeaponSystem` και `InputManager` από τον `Player`.

Άνοιγμα και Κλείσιμο Inventory

Το inventory ανοίγει και κλείνει με το πλήκτρο **I**, το οποίο ανιχνεύεται στο `Update()` μέσω του νέου Input System. Κατά το άνοιγμα (`OpenInventory()`), πρώτα ελέγχεται αν τα UI references είναι έγκυρα και αν όχι αναζητούνται εκ νέου. Στη συνέχεια εμφανίζεται το panel, απελευθερώνεται ο cursor και τίθεται το `isPaused` flag του `InputManager` σε `true` για να σταματήσει η κίνηση της κάμερας.

Σημαντικό: το `isPaused = true` **δεν παγώνει τον χρόνο** (`Time.timeScale` παραμένει 1). Αυτό σημαίνει ότι το παιχνίδι συνεχίζεται κανονικά ενώ το inventory είναι ανοιχτό — ο παίκτης μπορεί να κινείται αλλά δεν μπορεί να κουνήσει την κάμερα. Κατά το κλείσιμο (`CloseInventory()`), επιστρέφεται ο cursor στην κλειδωμένη κατάσταση και το `isPaused` τίθεται σε `false`.

```
private void OpenInventory()
{
    // Ξαναψαχνει τα panels αν εχουν χαθει
    if (inventoryPanel == null || weaponSlotContainer == null)
    {
        Canvas[] canvases = Resources.FindObjectsOfTypeAll<Canvas>();
        foreach (Canvas c in canvases)
        {
            if (inventoryPanel == null)
            {
                Transform ip = c.transform.Find("InventoryPanel");
                if (ip != null) inventoryPanel = ip.gameObject;
            }
            if (weaponSlotContainer == null)
            {
                Transform wsc = c.transform.Find("WeaponSlotContainer");
                if (wsc != null) weaponSlotContainer = wsc;
            }
        }
    }

    if (inventoryPanel == null || weaponSlotContainer == null) return;

    isOpen = true;
    inventoryPanel.SetActive(true);
    Cursor.lockState = CursorLockMode.None;
    Cursor.visible = true;
    if (InputManager.Instance != null)
        InputManager.Instance.isPaused = true;
    RefreshInventory();
}

private void CloseInventory()
{
    isOpen = false;
    if (inventoryPanel != null) inventoryPanel.SetActive(false);
    Cursor.lockState = CursorLockMode.Locked;
    Cursor.visible = false;
    if (InputManager.Instance != null)
        InputManager.Instance.isPaused = false;
}

private void RefreshInventory()
```

Εικόνα 72

RefreshInventory()

Η μέθοδος `RefreshInventory()` δημιουργεί δυναμικά τα weapon slots κάθε φορά που ανοίγει το inventory. Πρώτα διαγράφονται όλα τα υπάρχοντα slots για να αποφευχθεί διπλασιασμός, και στη συνέχεια δημιουργείται ένα slot για κάθε ξεκλειδωμένο όπλο.

```
private void RefreshInventory()
{
    if (weaponSlotContainer == null) return;

    // Διαγραφή παλιων slots
    List<GameObject> toDelete = new List<GameObject>();
    foreach (Transform child in weaponSlotContainer)
        toDelete.Add(child.gameObject);
    foreach (GameObject obj in toDelete)
        Destroy(obj);

    int currentLevel = weaponSystem.GetCurrentLevel();

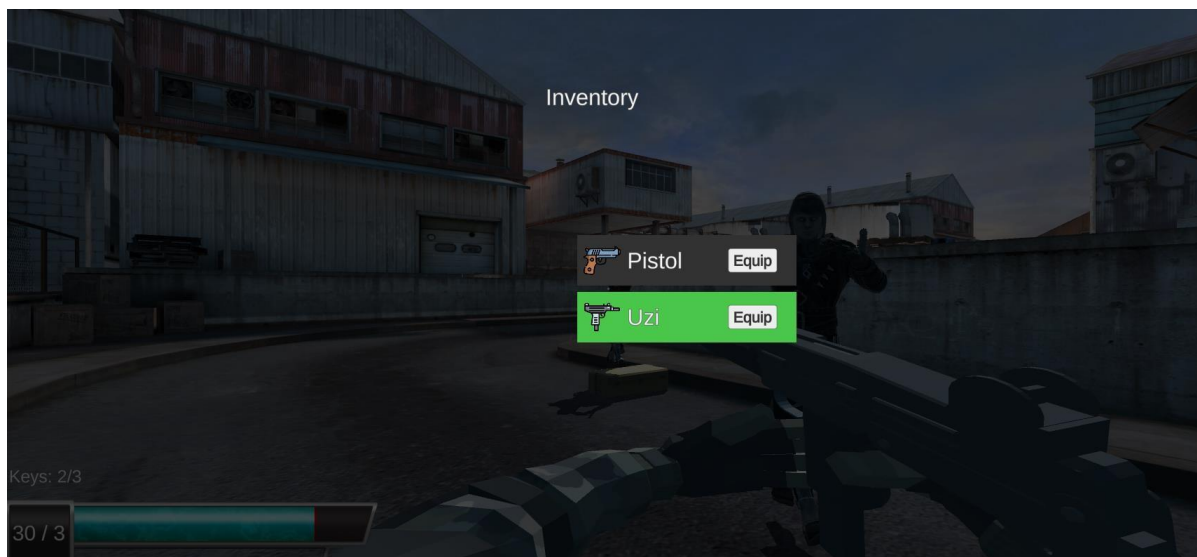
    for (int i = 0; i < currentLevel; i++)
    {
        GameObject slot = Instantiate(weaponSlotPrefab, weaponSlotContainer);
        WeaponSlotUI slotUI = slot.GetComponent<WeaponSlotUI>();

        if (slotUI != null)
        {
            int weaponIndex = i;
            Sprite icon = weaponSystem.GetWeaponIcon(i);
            slotUI.Setup(
                weaponSystem.GetWeaponName(i),
                i == weaponSystem.GetCurrentWeaponIndex(),
                icon,
                () =>
                {
                    weaponSystem.EquipWeaponByIndex(weaponIndex);
                    CloseInventory();
                }
            );
        }
    }
}
```

Εικόνα 73

Κάθε slot εμφανίζει το όνομα και το εικονίδιο του όπλου, ενώ το τρέχον εξοπλισμένο όπλο χρωματίζεται διαφορετικά (πράσινο background). Η λογική επιλογής όπλου υλοποιείται μέσω **lambda expression** που καλεί `EquipWeaponByIndex()` και κλείνει αυτόματα το inventory.

Ιδιαίτερης σημασίας είναι η χρήση της τοπικής μεταβλητής `weaponIndex` μέσα στον βρόχο. Χωρίς αυτή, όλα τα lambda expressions θα αναφέρονταν στην τελική τιμή του `i` λόγω του **closure mechanism** της C# ένα κλασικό bug που αποφεύγεται με αυτόν τον τρόπο.



Εικόνα 74

7. Σύστημα Εχθρών

7.1 State Machine

Το StateMachine.cs υλοποιεί το State Machine pattern για τη διαχείριση της συμπεριφοράς των εχθρών. Είναι ένα από τα πιο σημαντικά patterns στο game development, καθώς επιτρέπει την οργάνωση σύνθετης συμπεριφοράς σε ξεχωριστές, διαχειρίσιμες καταστάσεις. Η ιδέα πίσω από το State Machine είναι απλή: ο εχθρός βρίσκεται πάντα σε μία και μόνο κατάσταση (activeState), και κάθε κατάσταση γνωρίζει πότε και πώς να μεταβεί σε άλλη. Αυτό αποφεύγει την ανάγκη για πολύπλοκα if/else chains στον κώδικα του εχθρού.

Η μέθοδος Initialise() καλείται από το Enemy.cs μετά από ένα frame καθυστέρηση μέσω Coroutine, ώστε να εξασφαλιστεί ότι το NavMesh έχει φορτωθεί πλήρως πριν ξεκινήσει η State Machine. Ορίζει ως αρχική κατάσταση το PatrolState. Στο Update() καλείται η Perform() της ενεργής κατάστασης κάθε frame, εκτελώντας τη λογική που αντιστοιχεί στην τρέχουσα συμπεριφορά του εχθρού. Ο έλεγχος if (activeState != null) αποτρέπει NullReferenceException κατά την αρχικοποίηση, πριν οριστεί η πρώτη κατάσταση. Η μέθοδος ChangeState() διαχειρίζεται τη μετάβαση μεταξύ καταστάσεων σε τρία βήματα. Πρώτα καλεί Exit() στην τρέχουσα κατάσταση για καθαρισμό, στη συνέχεια ορίζει τη νέα κατάσταση και τέλος την αρχικοποιεί με Enter(). Σημαντικό στοιχείο είναι ότι κάθε νέα κατάσταση λαμβάνει reference στο StateMachine και στον Enemy κατά την είσοδό της. Αυτό επιτρέπει σε κάθε κατάσταση να έχει πρόσβαση σε όλα τα δεδομένα του εχθρού και να μπορεί να ζητήσει αλλαγή κατάστασης όταν το κρίνει απαραίτητο.

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

public class StateMachine : MonoBehaviour
{
    public BaseState activeState;

    public void Initialise()
    {
        ChangeState(new PatrolState());
    }

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

    // Update is called once per frame
    void Update()
    {
        if (activeState != null)
        {
            activeState.Perform();
        }
    }

    public void ChangeState(BaseState newState)
    {
        if (activeState != null)
        {
            activeState.Exit();
        }
        activeState = newState;

        if (activeState != null)
        {
            activeState.stateMachine = this;
            activeState.enemy = GetComponent<Enemy>();
            activeState.Enter();
        }
    }
}
```

Εικόνα 75

7.2 PatrolState.cs

Το PatrolState.cs υλοποιεί την κατάσταση περιπολίας του εχθρού. Κατά την περιπολία, ο εχθρός κινείται μεταξύ προκαθορισμένων σημείων (waypoints) και παρακολουθεί συνεχώς αν εντοπίσει τον παίκτη.

Enter()

Η μέθοδος Enter() καλείται μια φορά κατά την είσοδο στην κατάσταση περιπολίας. Απενεργοποιεί την αυτόματη περιστροφή του NavMeshAgent (updateRotation = false) γιατί η περιστροφή διαχειρίζεται χειροκίνητα από την FaceMovementDirection(), και ορίζει ως πρώτο στόχο το τρέχον waypoint.

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

public class PatrolState : BaseState
{
    public int waypointIndex;
    private float rotationSpeed = 5f;

    public override void Enter()
    {
        if (enemy == null || enemy.path == null) return;
        enemy.Agent.updateRotation = false;
        enemy.Agent.SetDestination(enemy.path.waypoints[waypointIndex].position);
    }
}
```

Εικόνα 76

Perform()

Στο Perform() που εκτελείται κάθε frame, ελέγχεται **πρώτα** αν ο εχθρός εντοπίζει τον παίκτη μέσω CanSeePlayer(). Αν ναι, αλλάζει αμέσως σε AttackState και σταματά την εκτέλεση με return για να αποφευχθεί οποιαδήποτε επιπλέον επεξεργασία.

Αν ο εχθρός δεν εντοπίσει τον παίκτη, συνεχίζεται κανονικά η περιπολία μέσω PatrolCycle() και FaceMovementDirection().

```
public override void Perform()
{
    if (enemy == null) return;

    // Τσέκαρε ΠΡΩΤΑ αν βλέπει τον player
    if (enemy.CanSeePlayer())
    {
        stateMachine.ChangeState(new AttackState());
        return; // ← σταμάτα αμέσως
    }

    PatrolCycle();
    FaceMovementDirection();
}
```

Εικόνα 77

PatrolCycle()

Η μέθοδος `PatrolCycle()` ελέγχει αν ο εχθρός έχει φτάσει κοντά στο τρέχον waypoint (`remainingDistance < 0.2f`) και αν ναι, προχωρά στο επόμενο. Όταν φτάσει στο τελευταίο waypoint, επιστρέφει στο πρώτο δημιουργώντας έτσι έναν ατέρμονο κύκλο περιπολίας.

```
public void PatrolCycle()
{
    if (enemy == null || enemy.path == null) return;
    if (enemy.Agent.remainingDistance < 0.2f)
    {
        if (waypointIndex < enemy.path.waypoints.Count - 1)
            waypointIndex++;
        else
            waypointIndex = 0;

        enemy.Agent.SetDestination(enemy.path.waypoints[waypointIndex].position);
    }
}
```

Εικόνα 78

FaceMovementDirection()

Η μέθοδος `FaceMovementDirection()` περιστρέφει ομαλά τον εχθρό προς την κατεύθυνση κίνησής του χρησιμοποιώντας `Quaternion.Slerp()` για φυσική και ομαλή περιστροφή: Ο έλεγχος `sqrMagnitude > 0.1f` εξασφαλίζει ότι η περιστροφή εφαρμόζεται μόνο όταν ο εχθρός κινείται πραγματικά, αποτρέποντας τυχαίες περιστροφές όταν ο εχθρός στέκεται σε waypoint.

```
private void FaceMovementDirection()
{
    if (enemy == null) return;
    if (enemy.Agent.velocity.sqrMagnitude > 0.1f)
    {
        Quaternion targetRotation = Quaternion.LookRotation(enemy.Agent.velocity.normalized);
        enemy.transform.rotation = Quaternion.Slerp(
            enemy.transform.rotation,
            targetRotation,
            Time.deltaTime * rotationSpeed
        );
    }
}
```

Εικόνα 79

Exit() - Έξοδος από την Κατάσταση

Κατά την έξοδο από το `PatrolState`, επανενεργοποιείται η αυτόματη περιστροφή του `NavMeshAgent` (`updateRotation = true`), ώστε στο `AttackState` ο εχθρός να στρέφεται αυτόματα προς τον παίκτη.

```
public override void Exit()
{
    if (enemy == null) return;
    enemy.Agent.updateRotation = true;
}
```

Εικόνα 80

7.3 AttackState.cs

Το `AttackState.cs` υλοποιεί την κατάσταση επίθεσης του εχθρού. Όταν ο εχθρός εντοπίσει τον παίκτη, μεταβαίνει σε αυτή την κατάσταση όπου κυνηγά τον παίκτη, στρέφεται προς αυτόν και πυροβολεί σε τακτά χρονικά διαστήματα.

Enter()

Κατά την είσοδο στο AttackState, ο εχθρός ορίζει αμέσως ως στόχο τη θέση του παίκτη μέσω του NavMeshAgent. Αυτό εξασφαλίζει ότι ο εχθρός αρχίζει αμέσως να κινείται προς τον παίκτη μόλις τον εντοπίσει, χωρίς καμία καθυστέρηση.

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

public class AttackState : BaseState
{
    private float moveTimer;
    private float losePlayerTimer;
    private float shotTimer;

    public override void Enter()
    {
        enemy.agent.SetDestination(enemy.Player.transform.position);
    }
}
```

Εικόνα 81

Perform()

Στο Perform() διακρίνονται δύο περιπτώσεις ανάλογα με το αν ο εχθρός βλέπει τον παίκτη: Αν ο εχθρός **βλέπει** τον παίκτη, μηδενίζεται ο losePlayerTimer, ενημερώνονται οι timers και ο εχθρός στρέφεται άμεσα προς τον παίκτη:

Ο shotTimer ελέγχει αν έχει περάσει αρκετός χρόνος από τον τελευταίο πυροβολισμό και αν ναι καλεί τη Shoot(). Ο moveTimer υλοποιεί τυχαία πλευρική κίνηση: κάθε 3-7 δευτερόλεπτα ο εχθρός μετακινείται σε τυχαία θέση εντός 5 μονάδων, κάνοντας την κίνησή του λιγότερο προβλέψιμη:

Αν ο εχθρός **δεν βλέπει** τον παίκτη, αυξάνεται ο losePlayerTimer. Αν ο παίκτης παραμείνει εκτός οπτικής επαφής για 8 δευτερόλεπτα, ο εχθρός εγκαταλείπει την επίθεση και επιστρέφει σε PatrolState. Η διπλή εξακρίβωση με enemy.CanSeePlayer() αποτρέπει τυχαίες μεταβάσεις σε περίπτωση που ο παίκτης κρυφτεί και ξαναεμφανιστεί γρήγορα.

```
public override void Perform()
{
    if (enemy.CanSeePlayer())
    {
        losePlayerTimer = 0;
        moveTimer += Time.deltaTime;
        shotTimer += Time.deltaTime;
        enemy.transform.LookAt(enemy.Player.transform);
        if (shotTimer > enemy.fireRate)
        {
            Shoot();
        }

        if (moveTimer > Random.Range(3, 7))
        {
            enemy.Agent.SetDestination(enemy.transform.position + (Random.insideUnitSphere * 5));
            moveTimer = 0;
        }
    }
    else
    {
        losePlayerTimer += Time.deltaTime;

        // Γυρνάει Patrol MONO αν ο παίκτης είναι εκτός οπτικής γωνίας για 8 δευτερόλεπτα
        if (losePlayerTimer > 8 && !enemy.CanSeePlayer())
        {
            stateMachine.ChangeState(new PatrolState());
        }
    }
}
```

Εικόνα 82

Shoot()

Η μέθοδος Shoot() υλοποιεί τον πυροβολισμό του εχθρού. Υπολογίζει την κατεύθυνση προς τον παίκτη από τη θέση του κάνη του όπλου (gunBarrel) και δημιουργεί ένα bullet από τα Resources.

Στο bullet αποδίδονται το damage του εχθρού και το ownerTag = "Enemy" ώστε το Bullet.cs να γνωρίζει ότι προέρχεται από εχθρό και να εφαρμόζει ζημιά μόνο στον παίκτη και όχι σε άλλους εχθρούς. Μετά τον πυροβολισμό μηδενίζεται ο shotTimer για την έναρξη νέας μέτρησης.

```
public void Shoot()
{
    Transform gunbarrel = enemy.gunBarrel;

    Vector3 shootDirection = (enemy.Player.transform.position - gunbarrel.position).normalized;

    GameObject bullet = GameObject.Instantiate(
        Resources.Load("Prefabs/Bullet") as GameObject,
        gunbarrel.position,
        Quaternion.FromToRotation(Vector3.up, shootDirection) // ← NEO
    );

    bullet.GetComponent<Rigidbody>().velocity = shootDirection * 40;

    Bullet b = bullet.GetComponent<Bullet>();
    if (b != null)
    {
        b.damage = enemy.bulletDamage; // παίρνει το damage από τον Enemy
        b.ownerTag = "Enemy";
    }

    Debug.Log("Shoot");
    shotTimer = 0;
}
```

Εικόνα 83

7.4 Enemy.cs

Το Enemy.cs είναι το κεντρικό script κάθε εχθρού πέρα του Final Boss, που συνδέει όλα τα επιμέρους συστήματα — State Machine, NavMesh, animations, όπλα και μουσική. Λειτουργεί ως το «μυαλό» του εχθρού, παρέχοντας στις καταστάσεις (PatrolState, AttackState) πρόσβαση σε όλα τα απαραίτητα δεδομένα.

Start() – InitializeEnemy()

Στο Start() αρχικοποιούνται τα βασικά components, StateMachine, NavMeshAgent, Animator, και εντοπίζεται ο παίκτης μέσω tag. Στη συνέχεια καλείται το InitializeEnemy() Coroutine που περιμένει **2 frames** πριν αρχικοποιήσει τη State Machine.

Η καθυστέρηση δύο frames είναι κρίσιμη γιατί το NavMesh χρειάζεται λίγο χρόνο για να φορτωθεί πλήρως κατά την έναρξη του επιπέδου. Χωρίς αυτή την καθυστέρηση, το SetDestination() στο PatrolState.Enter() θα αποτύγχανε με error γιατί ο εχθρός δεν θα ήταν ακόμα τοποθετημένος στο NavMesh.

```

void Start()
{
    stateMachine = GetComponent<StateMachine>();
    agent = GetComponent<NavMeshAgent>();
    player = GameObject.FindGameObjectWithTag("Player");
    animator = GetComponentInChildren<Animator>();
    StartCoroutine(InitializeEnemy());
}

```

Εικόνα 84

```

private IEnumerator InitializeEnemy()
{
    yield return null;
    yield return null; // 2 frames καθυστέρηση

    agent = GetComponent<NavMeshAgent>();
    if (agent.isOnNavMesh)
    {
        stateMachine.Initialise();
    }
    else
    {
        Debug.LogWarning($"{gameObject.name} is not on NavMesh!");
    }
}

```

Εικόνα 85

Update()

Στο Update() εφαρμόζονται δύο σημαντικοί έλεγχοι ασφαλείας πριν εκτελεστεί οποιαδήποτε λογική. Αυτοί οι έλεγχοι αποτρέπουν NullReferenceExceptions κατά την αρχικοποίηση, πριν η State Machine και το NavMesh είναι έτοιμα. Στη συνέχεια ενημερώνονται τα animations, ελέγχεται η ορατότητα του παίκτη και διαχειρίζεται η μουσική μέσω του MusicManager.

Ιδιαίτερης σημασίας είναι η διαχείριση της μουσικής: παρακολουθείται αν ο εχθρός μεταβαίνει από PatrolState σε AttackState και αντίστροφα, και ειδοποιείται ο MusicManager αναλόγως. Το flag wasAwareOfPlayer εξασφαλίζει ότι τα events μουσικής ενεργοποιούνται μόνο τη στιγμή αλλαγής κατάστασης και όχι κάθε frame.

Τέλος, εφαρμόζεται fix για την κλίση του εχθρού: κλειδώνονται οι άξονες X και Z περιστροφής ώστε ο εχθρός να παραμένει πάντα όρθιος.

```

void Update()
{
    if (stateMachine == null || stateMachine.activeState == null) return;
    if (agent == null || !agent.isOnNavMesh) return;

    UpdateAnimations();
    CanSeePlayer();
    currentState = stateMachine.activeState.ToString();

    bool isAware = stateMachine.activeState is AttackState;
    if (isAware && !wasAwareOfPlayer)
    {
        if (MusicManager.Instance != null)
            MusicManager.Instance.EnemySpottedPlayer();
    }
    else if (!isAware && wasAwareOfPlayer)
    {
        if (MusicManager.Instance != null)
            MusicManager.Instance.EnemyLostPlayer();
    }
    wasAwareOfPlayer = isAware;

    transform.rotation = Quaternion.Euler(0, transform.rotation.eulerAngles.y, 0);
}

```

Εικόνα 86

CanSeePlayer()

Η μέθοδος CanSeePlayer() υλοποιεί ένα ρεαλιστικό σύστημα όρασης που ελέγχει τρεις συνθήκες διαδοχικά. Πρώτα ελέγχεται το flag isAggro — αν ο εχθρός έχει χτυπηθεί, βλέπει πάντα τον παίκτη ανεξάρτητα από απόσταση ή γωνία. Στη συνέχεια ελέγχεται η **απόσταση** μεταξύ εχθρού και παίκτη, και αν αυτή είναι εντός του sightDistance ελέγχεται η **γωνία** μεταξύ της κατεύθυνσης βλέμματος και του παίκτη.

Αν ο παίκτης βρίσκεται εντός πεδίου ορατότητας 85°, εκτελείται **Raycast** από το eyeTransform για να ελεγχθεί αν υπάρχουν εμπόδια μεταξύ εχθρού και παίκτη. Για οπτική αποσφαλμάτωση, η ακτίνα σχεδιάζεται κόκκινη όταν δεν βλέπει τον παίκτη και πράσινη όταν τον βλέπει. Η χρήση IsChildOf() εξασφαλίζει ότι το Raycast αναγνωρίζει τον παίκτη ακόμα και αν χτυπά child object του (π.χ. hitbox ή collider παιδί).

```

public bool CanSeePlayer()
{
    if (isAggro) return true;
    if (player != null)
    {
        if (Vector3.Distance(transform.position, player.transform.position) < sightDistance)
        {
            Vector3 targetDirection = player.transform.position - eyeTransform.position;
            float angleToPlayer = Vector3.Angle(eyeTransform.forward, targetDirection);
            if (angleToPlayer <= fieldOfView)
            {
                Ray ray = new Ray(eyeTransform.position, targetDirection.normalized);
                RaycastHit hitInfo;
                Debug.DrawRay(ray.origin, ray.direction * sightDistance, Color.red);
                if (Physics.Raycast(ray, out hitInfo, sightDistance))
                {
                    if (hitInfo.transform.IsChildOf(player.transform) ||
                        hitInfo.transform.gameObject == player)
                    {
                        Debug.DrawRay(ray.origin, ray.direction * sightDistance, Color.green);
                        return true;
                    }
                }
            }
        }
    }
    return false;
}

```

Εικόνα 87

OnHit()

Η μέθοδος OnHit() καλείται από το EnemyHealth όταν ο εχθρός δεχτεί ζημιά. Ενεργοποιεί το flag isAggro και αυξάνει το sightDistance σε 1000 μονάδες, ουσιαστικά κάνοντας τον εχθρό να «βλέπει» τον παίκτη από παντού. Αυτό εξασφαλίζει ότι αν ο παίκτης επιτεθεί από πίσω, ο εχθρός αντιδρά άμεσα και μεταβαίνει σε AttackState.

```

public void OnHit()
{
    isAggro = true;
    sightDistance = 1000f;
}

```

Εικόνα 88

UpdateAnimations()

Η μέθοδος UpdateAnimations() ενημερώνει το Float parameter **Speed** του Animator με το μέτρο της ταχύτητας του NavMeshAgent. Έτσι το animation κίνησης ενεργοποιείται αυτόματα όταν ο εχθρός κινείται και απενεργοποιείται όταν σταματά, χωρίς να χρειάζεται χειροκίνητη διαχείριση transitions.

```

private void UpdateAnimations()
{
    if (animator == null) return;
    float speed = agent.velocity.magnitude;
    animator.SetFloat("Speed", speed);
}

```

Εικόνα 89

7.5 EnemyHealth.cs

Το EnemyHealth.cs διαχειρίζεται την υγεία κάθε εχθρού και συντονίζει όλα τα συστήματα που επηρεάζονται όταν ένας εχθρός χτυπιέται ή πεθαίνει. Χρησιμοποιεί State Machine, μουσική, LevelManager και σύστημα XP.

TakeDamage()

Η μέθοδος TakeDamage() είναι η μοναδική μέθοδος του script και εκτελεί διαφορετική λογική ανάλογα με το αν ο εχθρός επιβιώνει ή πεθαίνει μετά τη ζημιά.

Πρώτα αφαιρείται η ζημιά από την υγεία. Στη συνέχεια, αν ο εχθρός **επιβιώνει** (health > 0), ελέγχεται αν βρίσκεται ήδη σε AttackState. Αν όχι, μεταβαίνει αμέσως σε επίθεση ανεξάρτητα από το αν βλέπει τον παίκτη — αυτό υλοποιεί το **Aggro system**: ο εχθρός αντιδρά άμεσα όταν χτυπηθεί, ακόμα και από πίσω.

```
if (e.stateMachine != null && !(e.stateMachine.activeState is AttackState))
{
    e.stateMachine.ChangeState(new AttackState());

    // Ενημερώνει μουσική
    if (!e.wasAwareOfPlayer && MusicManager.Instance != null)
    {
        e.wasAwareOfPlayer = true;
        MusicManager.Instance.EnemySpottedPlayer();
    }
}
```

Εικόνα 90

Ταυτόχρονα ειδοποιείται ο MusicManager να αλλάξει σε combat music, αλλά μόνο αν ο εχθρός δεν ήταν ήδη aware — αποτρέποντας πολλαπλές κλήσεις EnemySpottedPlayer() για τον ίδιο εχθρό.

Αν ο εχθρός **πεθαίνει** (health <= 0), εκτελείται μια σειρά από ενέργειες συντονισμού. Πρώτα ελέγχεται αν ο εχθρός ήταν aware του παίκτη και αν ναι, καλείται EnemyLostPlayer() στον MusicManager για να μειωθεί ο μετρητής εχθρών που παρακολουθούν τον παίκτη.

```
// Ενημερώνει μουσική
if (!e.wasAwareOfPlayer && MusicManager.Instance != null)
{
    e.wasAwareOfPlayer = true;
    MusicManager.Instance.EnemySpottedPlayer();
}
```

Εικόνα 91

Στη συνέχεια ειδοποιείται ο LevelManager για τον θάνατο του εχθρού, διακρίνοντας αν είναι κανονικός ή Secret Room εχθρός.

```
if (lm != null && e != null)
{
    if (e.isSecretRoomEnemy)
        lm.SecretEnemyKilled();
    else
        lm.NormalEnemyKilled();
}
```

Εικόνα 92

Αυτή η διάκριση είναι κρίσιμη για τη λογική ολοκλήρωσης επιπέδου — το LevelManager χρειάζεται να γνωρίζει ξεχωριστά πόσοι κανονικοί και πόσοι Secret Room εχθροί έχουν απομείνει. Τέλος, δίνεται XP στον παίκτη και το object καταστρέφεται.

```
WeaponSystem ws = GameObject.FindGameObjectWithTag("Player")?.GetComponent<WeaponSystem>();
if (ws != null) ws.AddXP(xpReward);

Destroy(gameObject);
```

7.6 FinalBoss.cs

Το FinalBoss.cs διαχειρίζεται τη συμπεριφορά του τελικού αντιπάλου του παιχνιδιού. Σε αντίθεση με τους κανονικούς εχθρούς που χρησιμοποιούν State Machine, ο Boss διαχειρίζεται τη συμπεριφορά του απευθείας στο Update() με έλεγχο αποστάσεων και timers.

Awake()

Στο Awake() εφαρμόζεται ένα ιδιαίτερα έξυπνο trick για την τοποθέτηση του Boss στο NavMesh. Πρώτα απενεργοποιείται ο NavMeshAgent, εκτελείται Raycast προς τα κάτω για να βρεθεί το ακριβές σημείο του εδάφους, τοποθετείται ο Boss εκεί και στη συνέχεια επανενεργοποιείται ο Agent. Αυτή η τεχνική αποτρέπει το κλασικό πρόβλημα όπου ο Agent δεν μπορεί να τοποθετηθεί στο NavMesh αν το object δεν βρίσκεται ακριβώς πάνω στην επιφάνεια.

```
void Awake()
{
    currentHealth = maxHealth;

    NavMeshAgent ag = GetComponent<NavMeshAgent>();
    if (ag != null) ag.enabled = false;

    if (Physics.Raycast(transform.position + Vector3.up, Vector3.down, out RaycastHit hit, 10f))
    {
        transform.position = hit.point;
    }

    if (ag != null) ag.enabled = true;
}
```

Εικόνα 93

Update()

Στο Update() υλοποιείται η κύρια λογική του Boss με βάση την απόσταση από τον παίκτη. Πρώτα ελέγχονται δύο συνθήκες που αλλάζουν μόνο μία φορά καθ' όλη τη διάρκεια της μάχης. Η μετάβαση σε **Enraged Mode** όταν η υγεία πέσει κάτω από 50%. Όταν ο παίκτης βρίσκεται εντός detectionRange, ο Boss στρέφεται πάντα προς τον παίκτη κλειδώνοντας τον Y άξονα. Η κλήση με transform.position.y αντί για player.transform.position.y εξασφαλίζει ότι ο Boss δεν γέρνει προς τα πάνω ή κάτω, κοιτά πάντα οριζόντια. Στη συνέχεια διακρίνονται δύο υπο-περιπτώσεις: αν ο παίκτης βρίσκεται εντός attackRange, ο Boss σταματά και επιτίθεται. Αν βρίσκεται εκτός εμβέλειας, ο Boss κινηγά τον παίκτη και ενεργοποιείται το running animation. Αν ο παίκτης είναι εκτός detectionRange, ο Boss σταματά και περιμένει.

```

void Update()
{
    if (isDead || player == null) return;

    float distanceToPlayer = Vector3.Distance(transform.position, player.transform.position);

    if (!isEnraged && currentHealth <= maxHealth * 0.5f)
    {
        EnterEnragedMode();
    }
    if (distanceToPlayer <= detectionRange && !hasStartedBossFight)
    {
        hasStartedBossFight = true;
        if (MusicManager.Instance != null)
            MusicManager.Instance.StartBossFight();
    }
    if (distanceToPlayer <= detectionRange)
    {
        transform.LookAt(new Vector3(player.transform.position.x, transform.position.y, player.transform.position.z));

        if (distanceToPlayer <= attackRange)
        {
            agent.SetDestination(transform.position);
            animator.SetBool("IsRunning", false);

            if (Time.time >= nextAttackTime && !isAttacking)
            {
                StartCoroutine(PerformAttack());
            }
        }
        else
        {
            agent.SetDestination(player.transform.position);
            animator.SetBool("IsRunning", true);
        }
    }
    else
    {
        agent.SetDestination(transform.position);
        animator.SetBool("IsRunning", false);
    }
}

```

Εικόνα 94

PerformAttack()

Η μέθοδος PerformAttack() είναι Coroutine και υλοποιεί ένα σύστημα επίθεσης με διαφορετικούς τύπους ανάλογα με την κατάσταση του Boss. Χρησιμοποιεί το flag isAttacking για να αποτρέψει πολλαπλές ταυτόχρονες επιθέσεις. Η επιλογή τύπου επίθεσης γίνεται τυχαία, με τον αριθμό διαθέσιμων επιθέσεων να εξαρτάται από την κατάσταση. Στην κανονική κατάσταση ο Boss επιλέγει τυχαία μεταξύ δύο τύπων επίθεσης (Attack, Attack_2), ενώ στο Enraged Mode έχει στη διάθεσή του και τον τρίτο τύπο (Attack_3), κάνοντάς τον πιο απρόβλεπτο και επικίνδυνο.

Μετά από 0.5 δευτερόλεπτα, ώστε να συγχρονιστεί η ζημιά με το animation, ελέγχεται αν ο παίκτης βρίσκεται ακόμα εντός εμβέλειας και εφαρμόζεται η ζημιά. Το attackRange + 0.5f προσθέτει μικρό buffer για να αποφευχθεί η περίπτωση η επίθεση να αστοχεί λόγω ελαφράς κίνησης του παίκτη. Μετά την επίθεση ορίζεται το cooldown ανάλογα με την κατάσταση. Στο Enraged Mode το cooldown μειώνεται από 2 σε 1 δευτερόλεπτο, κάνοντας τον Boss να επιτίθεται δύο φορές γρηγορότερα.

```

private IEnumerator PerformAttack()
{
    isAttacking = true;

    int attackType = isEnraged ? Random.Range(0, 3) : Random.Range(0, 2);

    if (attackType == 0)
        animator.SetTrigger("Attack");
    else if (attackType == 1)
        animator.SetTrigger("Attack_2");
    else
        animator.SetTrigger("Attack_3");

    yield return new WaitForSeconds(0.5f);

    if (player != null)
    {
        float dist = Vector3.Distance(transform.position, player.transform.position);
        if (dist <= attackRange + 0.5f && playerHealth != null)
        {
            playerHealth.TakeDamage(attackDamage);
            Debug.Log($"Boss attacked! Damage: {attackDamage}");
        }
    }

    float cooldown = isEnraged ? enragedAttackCooldown : attackCooldown;
    nextAttackTime = Time.time + cooldown;

    yield return new WaitForSeconds(0.5f);
    isAttacking = false;
}

```

Εικόνα 95

EnterEnragedMode()

Η μέθοδος EnterEnragedMode() ενεργοποιείται μία μόνο φορά όταν η υγεία πέσει κάτω από 50%. Αυξάνει την ταχύτητα από walkSpeed σε runSpeed, ενεργοποιεί το Enraged Bool στον Animator και ειδοποιεί τον MusicManager να παίξει τον βρυχηθμό.

```

private void EnterEnragedMode()
{
    isEnraged = true;
    agent.speed = runSpeed;
    animator.SetBool("Enraged", true);
    if (MusicManager.Instance != null)
        MusicManager.Instance.PlayBossEnragedRoar();
}

public void TakeDamage(float damage)

```

Εικόνα 96

TakeDamage()

Η μέθοδος TakeDamage() καλείται από το HandleHit() του PlayerShooting κάθε φορά που ο παίκτης χτυπά τον Boss. Πρώτα ελέγχεται αν ο Boss είναι ήδη νεκρός για να αποφευχθεί οποιαδήποτε επιπλέον επεξεργασία. Στη συνέχεια αφαιρείται η ζημιά και ενεργοποιείται το GetHit animation για οπτική ανατροφοδότηση. Αν η υγεία φτάσει στο μηδέν, καλείται αμέσως η Die().

```

public void TakeDamage(float damage)
{
    Debug.Log($"[FinalBoss] TakeDamage called! Damage incoming: {damage}. Current HP before hit: {currentHealth}");
    if (isDead)
    {
        Debug.LogWarning("[FinalBoss] Damage blocked because isDead is TRUE!");
        return;
    }

    currentHealth -= damage;

    if (animator != null)
    {
        animator.SetTrigger("GetHit");
    }

    Debug.Log($"[FinalBoss] Damage applied! New HP: {currentHealth}");
    if (currentHealth <= 0)
    {
        Die();
    }
}

```

Εικόνα 97

Die()

Η μέθοδος Die() ορίζει το flag isDead, σταματά τον Agent, ενεργοποιεί το Death animation και συντονίζει τον MusicManager για ήχο θανάτου και επιστροφή στη μουσική εξερεύνησης.

```

private void Die()
{
    isDead = true;
    agent.SetDestination(transform.position);
    agent.speed = 0;
    animator.SetTrigger("Death");
    if (MusicManager.Instance != null)
    {
        MusicManager.Instance.PlayBossDeathSound();
        MusicManager.Instance.StopBossFight();
    }
    StartCoroutine>ShowWinScreen());
}

```

Εικόνα 98

ShowWinScreen() — Εμφάνιση Win Screen

Το ShowWinScreen() Coroutine περιμένει 3 δευτερόλεπτα ώστε να ολοκληρωθεί το Death animation και στη συνέχεια καλεί το winScreen.ShowWinScreen(). Η καθυστέρηση δίνει στον παίκτη χρόνο να απολαύσει τη νίκη του πριν εμφανιστεί το Win Screen.

```

private IEnumerator ShowWinScreen()
{
    yield return new WaitForSeconds(3f);
    if (winScreen != null)
        winScreen.ShowWinScreen();
}

```

Εικόνα 99

8. User Interface (UI)

8.1 Γενική Δομή

Το User Interface αποτελεί ένα από τα πιο κρίσιμα κομμάτια οποιουδήποτε παιχνιδιού, καθώς είναι η γέφυρα επικοινωνίας μεταξύ του παιχνιδιού και του παίκτη. Ένα καλά σχεδιασμένο UI δίνει στον παίκτη όλες τις πληροφορίες που χρειάζεται για να λάβει αποφάσεις, χωρίς να τον αποσπά από την εμπειρία του παιχνιδιού.

Το UI σχεδιάστηκε με γνώμονα την απλότητα και τη λειτουργικότητα. Ο παίκτης χρειάζεται να γνωρίζει ανά πάσα στιγμή την υγεία του, την κατάσταση της ασπίδας του, τα πυρομαχικά που του απομένουν, την πρόοδό του σε XP και πόσα κλειδιά έχει συλλέξει. Όλες αυτές οι πληροφορίες εμφανίζονται ταυτόχρονα στην οθόνη με τρόπο που δεν εμποδίζει την ορατότητα του παίκτη.

Για την υλοποίηση του UI χρησιμοποιήθηκε το **Canvas** system της Unity, το οποίο επιτρέπει την τοποθέτηση στοιχείων UI πάνω στην οθόνη σε σταθερές θέσεις ανεξάρτητα από την κίνηση της κάμερας. Για την εμφάνιση κειμένων χρησιμοποιήθηκε το **TextMeshPro** αντί για το παραδοσιακό Unity Text, καθώς προσφέρει σαφώς καλύτερη ποιότητα γραμματοσειράς και υποστήριξη για ελληνικούς χαρακτήρες. Για τις μπάρες υγείας, ασπίδας και XP χρησιμοποιήθηκαν **Image components** με τύπο Fill, τα οποία γεμίζουν από αριστερά προς τα δεξιά ανάλογα με την τρέχουσα τιμή.

Τα κύρια στοιχεία του UI είναι τα εξής: η **Health Bar** που εμφανίζει την τρέχουσα υγεία του παίκτη με το χαρακτηριστικό chip damage effect, η **Shield Bar** που εμφανίζεται πάνω από την health bar και δείχνει την κατάσταση της ασπίδας, το **Ammo Display** που εμφανίζει τις διαθέσιμες σφαίρες σε μορφή «τρέχουσες/γεμιστήρες», η **XP Bar** που δείχνει την πρόοδο προς την επόμενη αναβάθμιση όπλου, το **Key Counter** που ενημερώνει τον παίκτη για τα κλειδιά που έχει συλλέξει, το **Crosshair** στο κέντρο της οθόνης για τη σκόπευση, και η **Boss Health Bar** που εμφανίζεται στο πάνω μέρος της οθόνης μόνο κατά τη μάχη με τον Final Boss.

Ένα από τα βασικά προβλήματα που αντιμετωπίστηκαν κατά τη σχεδίαση του UI ήταν η διατήρηση των references μεταξύ των σκηνών. Επειδή ο παίκτης είναι DontDestroyOnLoad και επιβιώνει σε όλες τις σκηνές, ενώ τα Canvas objects ανήκουν στη σκηνή και καταστρέφονται κατά τη μετάβαση, χρειάστηκε να αναπτυχθεί μηχανισμός δυναμικής εύρεσης των UI elements σε κάθε νέα σκηνή. Αυτό υλοποιήθηκε μέσω του OnSceneLoaded event σε συνδυασμό με GameObject.Find() και μικρή καθυστέρηση frames για να εξασφαλιστεί ότι το Canvas έχει φορτωθεί πλήρως πριν αναζητηθούν τα elements

Το UI χρησιμοποιεί Canvas με TextMeshPro και Image components. Κύρια στοιχεία: HealthBar (chip damage effect), ShieldBar, AmmoDisplay (X/Y format), XP Bar, Key Counter, BossHealthBar, Crosshair.

8.2 Health & Shield Bar

Η μπάρα υγείας αποτελεί ένα από τα πιο άμεσα αναγνωρίσιμα στοιχεία κάθε παιχνιδιού, καθώς είναι συνεχώς ορατή και ενημερώνει τον παίκτη για την κατάστασή του σε πραγματικό χρόνο. Δόθηκε ιδιαίτερη προσοχή στο οπτικό αποτέλεσμα, με στόχο να δοθεί στον παίκτη μια πιο ζωντανή εμπειρία από μια απλή μπάρα που αδειάζει στατικά.

Για τον λόγο αυτό υλοποιήθηκε σύστημα διπλής μπάρας, γνωστό ως **chip damage effect**. Η μπάρα αποτελείται από δύο επάλληλα Image components: το **FrontHealthBar** και το **BackHealthBar**. Το FrontHealthBar αντιπροσωπεύει την τρέχουσα υγεία και αλλάζει αμέσως μόλις ο παίκτης δεχτεί ζημιά ή θεραπευτεί. Το BackHealthBar βρίσκεται από κάτω

και ακολουθεί με καθυστέρηση, δημιουργώντας ένα εφέ που αποδίδει οπτικά την «απώλεια» υγείας.

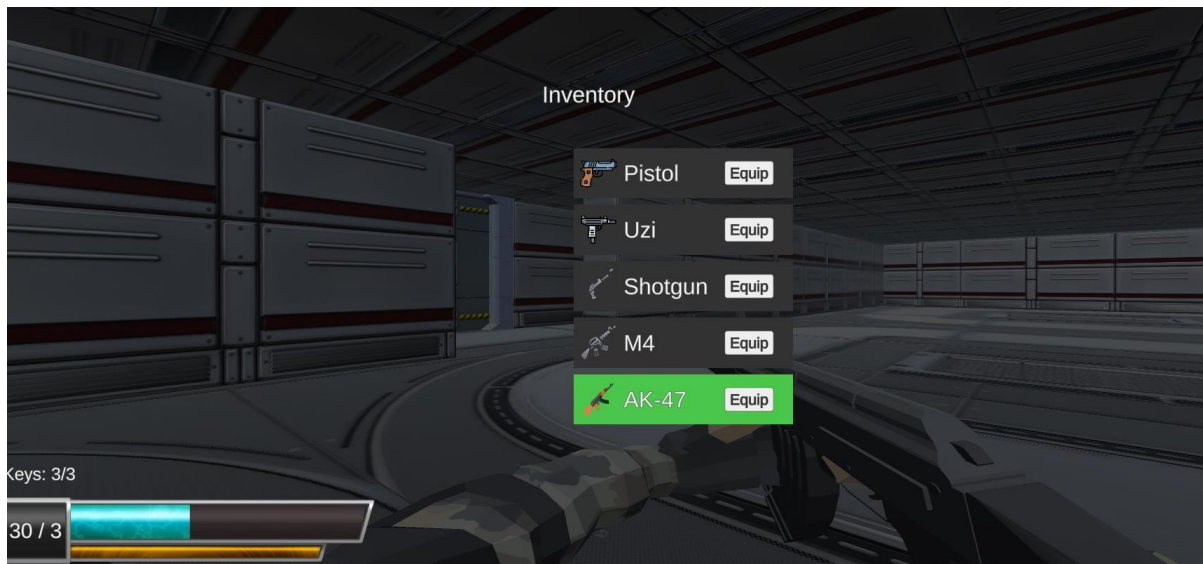
Η καθυστέρηση υλοποιείται μέσω της `Mathf.Lerp()`, η οποία παρεμβάλλει ομαλά μεταξύ της τρέχουσας και της επιθυμητής τιμής. Για πιο φυσική αίσθηση χρησιμοποιείται τετραγωνική παρεμβολή αντί για γραμμική το `percentComplete * percentComplete` που κάνει τη μπάρα να κινείται αρχικά γρήγορα και να επιβραδύνει καθώς πλησιάζει την τελική τιμή.

8.3 Ammo Display

Εμφανίζει "currentAmmo / currentMagazines" σε real-time

8.4 Inventory Panel

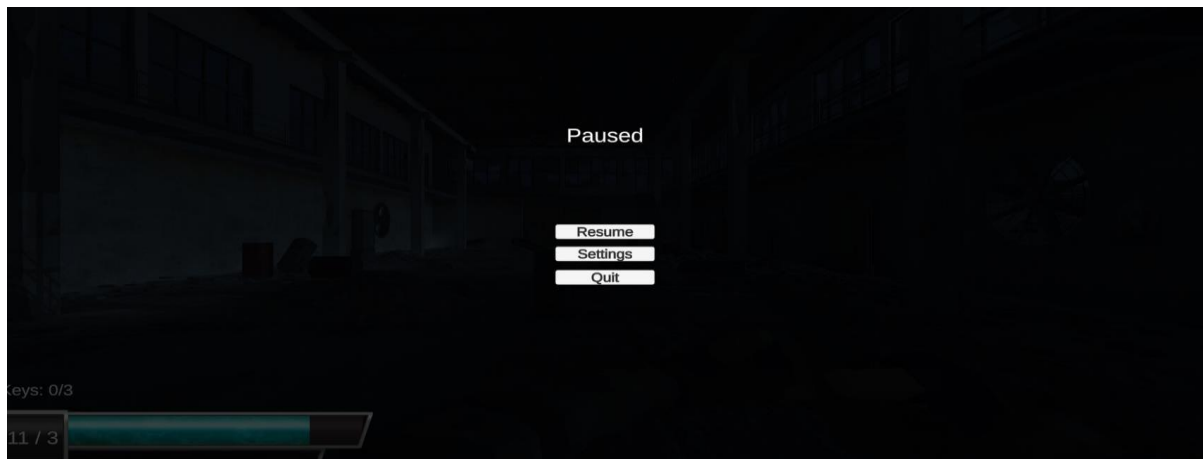
Ανοίγει με I. Εμφανίζει τα διαθέσιμα όπλα δυναμικά. Το τρέχον όπλο έχει πράσινο background.



Εικόνα 100

8.5 Pause Menu

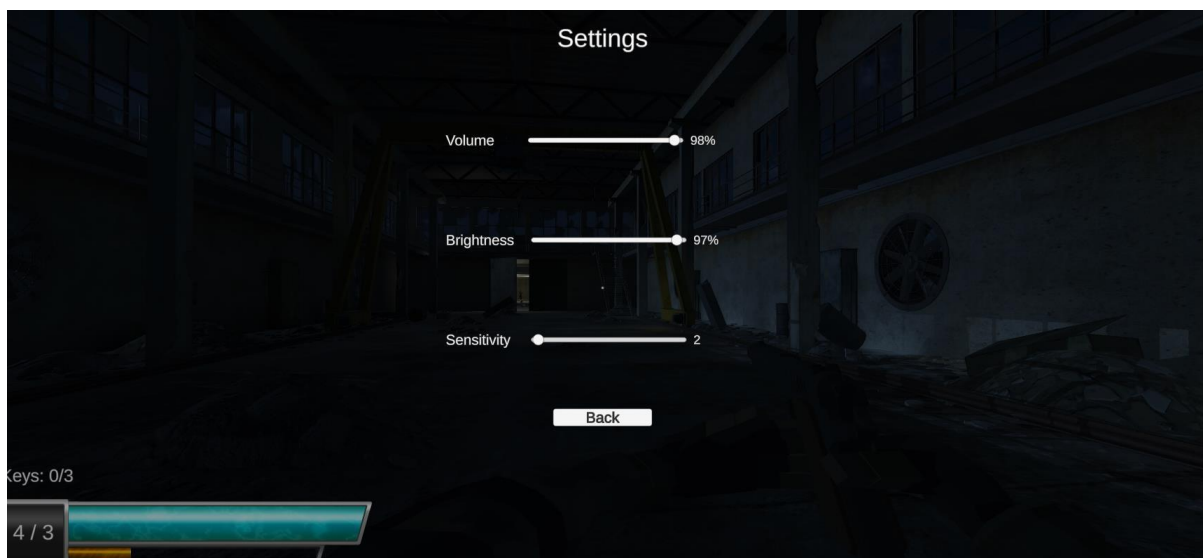
Ανοίγει με Escape. `Time.timeScale = 0`. Περιέχει Resume, Settings, Quit.



Εικόνα 101

8.6 Settings

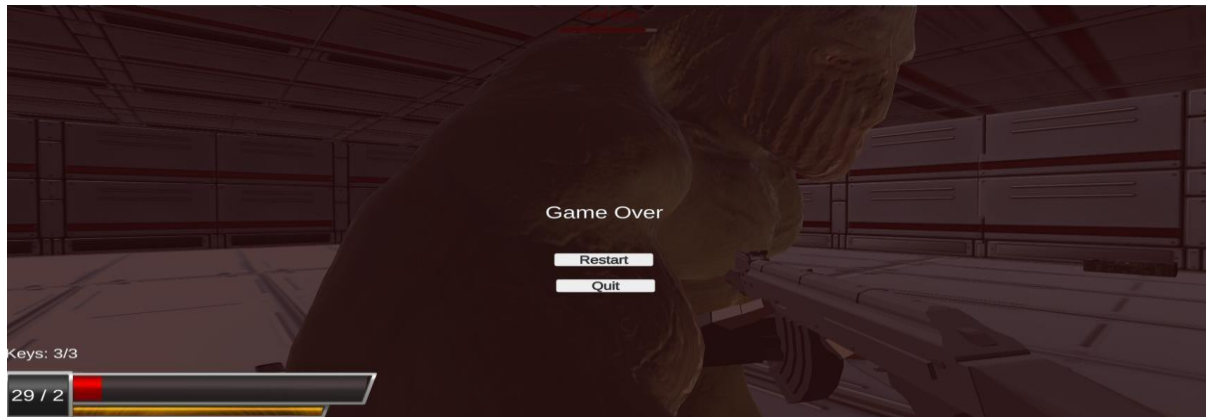
Volume (AudioListener.volume), Brightness (alpha overlay), Sensitivity (PlayerPrefs real-time). Αποθηκεύονται με PlayerPrefs.Save().



Εικόνα 102

8.7 Game Over & Win Screen

Game Over: fade-in animation, Restart (επαναφορά HP + XP στο level start) + Quit. Win Screen: εμφανίζεται μετά τη νίκη επί του Boss, Restart (από L1, reset κλειδιά) + Quit.



Εικόνα 103



Εικόνα 104

9. Σύστημα Μουσικής και Ήχου

9.1 MusicManager.cs

Το MusicManager.cs διαχειρίζεται ολόκληρο το σύστημα μουσικής του παιχνιδιού. Είναι υπεύθυνο για την αναπαραγωγή της κατάλληλης μουσικής ανάλογα με την κατάσταση του παιχνιδιού, εξερεύνηση, μάχη ή Final Boss, καθώς και για τους ήχους του Boss. Υλοποιεί το **Singleton pattern** με DontDestroyOnLoad, ώστε να επιβιώνει σε όλες τις σκηνές και να παρέχει συνεχή μουσική εμπειρία χωρίς διακοπές κατά τις μεταβάσεις.

Στο Awake() εφαρμόζεται το Singleton pattern: αν δεν υπάρχει ήδη instance, το τρέχον object ορίζεται ως το μοναδικό και επιβιώνει μεταξύ scenes. Αν υπάρχει ήδη κάτι που μπορεί να συμβεί αν η σκηνή φορτωθεί ξανά το νέο object καταστρέφεται αμέσως. Στη συνέχεια δημιουργούνται δυναμικά **δύο ξεχωριστά AudioSource**.

```
// AudioSource για μουσική
musicSource = gameObject.AddComponent<AudioSource>();
musicSource.loop = true;
musicSource.volume = 0.5f;
musicSource.spatialBlend = 0f;

// AudioSource για sound effects (roar, death)
sfxSource = gameObject.AddComponent<AudioSource>();
sfxSource.loop = false;
sfxSource.volume = 1f;
sfxSource.spatialBlend = 0f;
```

Εικόνα 105

Το musicSource χρησιμοποιείται αποκλειστικά για τη μουσική υπόκρουση (exploration, combat, boss) με loop = true, ενώ το sfxSource χρησιμοποιείται για τους ήχους του Boss (roar, enraged roar, death sound) που παίζουν μία φορά. Το spatialBlend = 0f εξασφαλίζει ότι και τα δύο ακούγονται ως 2D ήχος ομοιόμορφα από όλα τα ηχεία ανεξάρτητα από τη θέση του παίκτη στον χώρο.

```
public void EnemySpottedPlayer()
{
    enemiesAwareOfPlayer++;
    if (!isInCombat)
    {
        isInCombat = true;
        SwitchMusic(combatMusic);
    }
}
```

Εικόνα 106

Δυναμική Αλλαγή Μουσικής Εχθρών

Το σύστημα μουσικής εχθρών βασίζεται σε έναν μετρητή enemiesAwareOfPlayer που παρακολουθεί πόσοι εχθροί έχουν εντοπίσει τον παίκτη. Η μέθοδος EnemySpottedPlayer() καλείται από το Enemy.cs κάθε φορά που εχθρός μεταβαίνει σε AttackState, αυξάνει τον μετρητή και αν δεν παίζει ήδη combat music, ενεργοποιεί τη μετάβαση.

Σύστημα Μουσικής Final Boss

Η StartBossFight() καλείται από το FinalBoss.cs μόλις ο παίκτης εισέλθει στην εμβέλεια ανίχνευσης. Παίζει αμέσως τον εναρκτήριο βρυχηθμό του Boss μέσω PlayOneShot(), ώστε να ακουστεί ολόκληρος χωρίς να διακοπεί και στη συνέχεια ενεργοποιεί τη μετάβαση στη boss music.

```
public void StartBossFight()
{
    if (bossRoarSound != null)
        sfxSource.PlayOneShot(bossRoarSound);
    SwitchMusic(bossMusic);
}
```

Εικόνα 107

Οι μέθοδοι `PlayBossEnragedRoar()` και `PlayBossDeathSound()` καλούνται από το `FinalBoss.cs` στις αντίστοιχες στιγμές όταν η υγεία πέσει κάτω από 50% και όταν ο Boss πεθάνει. Η `StopBossFight()` καλείται επίσης κατά τον θάνατο και επιστρέφει στην `exploration music`.

FadeToNewMusic()

Η πιο σημαντική βοηθητική μέθοδος είναι το `FadeToNewMusic()` Coroutine, το οποίο υλοποιεί ομαλή μετάβαση μεταξύ μουσικών. Πρώτα κάνει **fade out** στην τρέχουσα μουσική μειώνοντας σταδιακά την ένταση, αλλάζει το clip και κάνει **fade in** στη νέα.

```
private IEnumerator FadeToNewMusic(AudioClip newClip)
{
    while (musicSource.volume > 0f)
    {
        musicSource.volume -= Time.deltaTime * fadeSpeed;
        yield return null;
    }

    musicSource.clip = newClip;
    musicSource.Play();

    while (musicSource.volume < 0.5f)
    {
        musicSource.volume += Time.deltaTime * fadeSpeed;
        yield return null;
    }

    musicSource.volume = 0.5f;
}
```

Εικόνα 108

Ιδιαίτερης σημασίας είναι η διαχείριση του `fadeCoroutine`. Αν κληθεί νέα μετάβαση ενώ μία τρέχει ήδη, η παλιά διακόπτεται αμέσως και ξεκινά η νέα, αποτρέποντας συγκρούσεις μεταξύ ταυτόχρονων fade effects.

ResetToExploration()

Η μέθοδος `ResetToExploration()` καλείται από το `GameOverManager` κατά το restart. Μηδενίζει τον μετρητή εχθρών και το flag `isInCombat`, και επιστρέφει στην `exploration music` εξασφαλίζοντας ότι μετά από κάθε restart η μουσική ξεκινά από μηδενική βάση ανεξάρτητα από την κατάσταση που ήταν πριν τον θάνατο του παίκτη.

9.2 Ήχοι Όπλων

Κάθε όπλο έχει δικό του fire sound και reload sound στο `WeaponHandler`. Ξεχωριστά `AudioSources` για fire και reload αποτρέπουν αλληλεπικάλυψη.

Shotgun reload sound: Αναπαράγεται ανάλογα με τον αριθμό σφαιρών που χρειάζεται. Clip 11 δευτερολέπτων για 8 σφαίρες → 1.375 sec/σφαίρα.

```

public IEnumerator PlayShotgunReload(int bulletsToReload)
{
    if (shotgunInsertSound == null) yield break;

    float timePerBullet = shotgunInsertSound.length / 8f; // 11sec / 8 σφαίρες

    for (int i = 0; i < bulletsToReload; i++)
    {
        // Παιζει απο τη σωστη θεση του clip
        audioSource.clip = shotgunInsertSound;
        audioSource.time = i * timePerBullet;
        audioSource.Play();

        yield return new WaitForSeconds(timePerBullet);

        audioSource.Stop();
    }

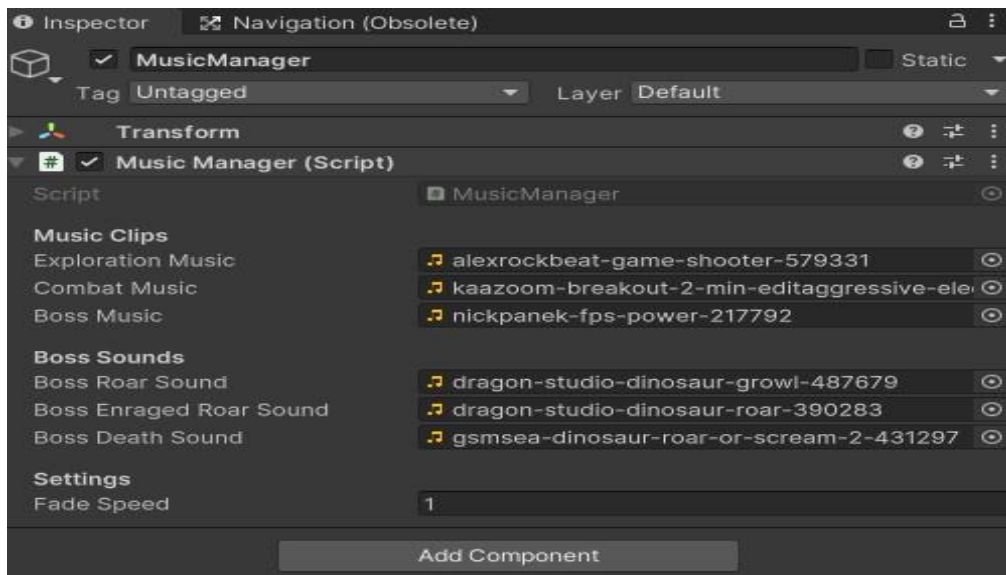
    // Ηχος rump στο τελος
    if (shotgunEndSound != null)
        audioSource.PlayOneShot(shotgunEndSound);
}

```

Εικόνα 109

9.3 Ήχοι Final Boss

- Εμφάνιση Boss: Roar sound έπειτα switch σε Boss music.
- 50% HP (Enraged): Enraged roar sound.
- Θάνατος Boss: Death sound και επιστροφή σε Exploration music.



Εικόνα 110

10. Διαχείριση Επιπέδων

10.1 LevelManager.cs

Το LevelManager.cs είναι το script που διαχειρίζεται τη λογική ολοκλήρωσης κάθε επιπέδου. Είναι υπεύθυνο για την παρακολούθηση των εχθρών, τη συλλογή κλειδιών, τη διαχείριση του Secret Room και την απόφαση για το πότε φορτώνεται το επόμενο επίπεδο. Σε αντίθεση με τα DontDestroyOnLoad scripts, το LevelManager ανήκει σε κάθε σκηνή ξεχωριστά και καταστρέφεται κατά τη μετάβαση.

Start()

Στο Start() μετρούνται οι εχθροί της σκηνής διακρίνοντας κανονικούς από Secret Room εχθρούς μέσω του isSecretRoomEnemy flag. Αν ο παίκτης έχει ήδη μυστικό κλειδί από προηγούμενο επίπεδο, καλείται η HideSecretKey() που αποκρύπτει το κλειδί και απενεργοποιεί το trigger του Secret Room.

```
void Start()
{
    keysCollectedThisLevel = 0;
    levelCompleteStarted = false;

    Enemy[] allEnemies = FindObjectsOfType<Enemy>();
    foreach (Enemy e in allEnemies)
    {
        EnemyHealth eh = e.GetComponent<EnemyHealth>();
        if (eh != null)
        {
            if (e.isSecretRoomEnemy) secretEnemies++;
            else normalEnemies++;
        }
    }

    Debug.Log($"Normal enemies: {normalEnemies} Secret enemies: {secretEnemies}");

    // Αν έχει πάρει secret key από ΟΠΟΙΟΔΗΠΟΤΕ level
    if (GameManager.Instance != null && GameManager.Instance.HasAnySecretKey())
        HideSecretKey();
}
```

Εικόνα 111

Παρακολούθηση Εχθρών και Κλειδιών

Οι μέθοδοι NormalEnemyKilled(), SecretEnemyKilled(), NormalKeyCollected() και SecretKeyCollected() καλούνται από τα αντίστοιχα scripts και ενημερώνουν τους μετρητές. Η SecretKeyCollected() επιπλέον ειδοποιεί τον GameManager να αποθηκεύσει ότι το μυστικό κλειδί έχει ληφθεί, ώστε να μην εμφανιστεί ξανά σε επόμενο επίπεδο.

CheckLevelComplete() και WaitBeforeLoadCoroutine()

Η CheckLevelComplete() ελέγχει διαδοχικά αν έχουν εξοντωθεί όλοι οι κανονικοί εχθροί και αν έχει ληφθεί το κανονικό κλειδί. Αν ναι, ξεκινά το WaitBeforeLoadCoroutine() που δίνει στον παίκτη waitBeforeLoad δευτερόλεπτα για να εισέλθει στο Secret Room. Αν ο παίκτης έχει ήδη μυστικό κλειδί, το επόμενο επίπεδο φορτώνεται αμέσως χωρίς αναμονή.

Secret Room Timer

Όταν ο παίκτης εισέλθει στο Secret Room, η `PlayerEnteredSecretRoom()` ενεργοποιεί τον 30-δευτερόλεπτο timer και ακυρώνει τυχόν τρέχον Coroutine φόρτωσης μέσω `StopAllCoroutines()`. Το επόμενο επίπεδο φορτώνεται μόνο αφού εξοντωθούν οι Secret Room εχθροί και ληφθεί το μυστικό κλειδί εντός του χρονικού ορίου

10.2 KeyPickup.cs

Το `KeyPickup.cs` υλοποιεί τη λογική συλλογής κλειδιών. Κληρονομεί από την αφηρημένη κλάση `Interactable`, οπότε ο παίκτης το ενεργοποιεί πατώντας E όταν πλησιάσει.

Τύποι Κλειδιών

Το script χρησιμοποιεί enum για να διακρίνει δύο τύπους κλειδιών.

```
public enum KeyType { Normal, Secret }

[Header("Key Settings")]
public KeyType keyType = KeyType.Normal;

private LevelManager levelManager;

private void Awake()
{
    promptMessage = keyType == KeyType.Secret
        ? "Pick up Secret Key [E]"
        : "Pick up Key [E]";
}
```

Εικόνα 112

Στο `Awake()` ορίζεται αυτόματα το κατάλληλο prompt message ανάλογα με τον τύπο, ώστε ο παίκτης να γνωρίζει τι πρόκειται να πάρει πριν το συλλέξει.

Interact()

Κατά την αλληλεπίδραση εκτελούνται τρεις ενέργειες διαδοχικά. Πρώτα ειδοποιείται ο `KeyManager` να καταγράψει τη συλλογή και ενημερώνεται το UI με τον νέο αριθμό κλειδιών. Στη συνέχεια ειδοποιείται ο `LevelManager` για τον τύπο του κλειδιού που συλλέχθηκε.

```
protected override void Interact()
{
    Debug.Log("Key picked up!");

    if (KeyManager.Instance != null)
    {
        KeyManager.Instance.CollectKey();
        PlayerUI playerUI = FindObjectOfType<PlayerUI>();
        if (playerUI != null)
            playerUI.UpdateKeys(KeyManager.Instance.GetKeysCollected(), KeyManager.Instance.totalKeysRequired);
    }

    if (levelManager != null)
    {
        if (keyType == KeyType.Normal)
            levelManager.NormalKeyCollected();
        else
            levelManager.SecretKeyCollected();
    }

    Destroy(gameObject);
}
```

Εικόνα 113

10.3 SecretRoomTrigger.cs

Το SecretRoomTrigger.cs είναι ένα απλό αλλά σημαντικό script που ανιχνεύει την είσοδο του παίκτη στο Secret Room και ειδοποιεί τον LevelManager να ενεργοποιήσει τον 30-δευτερόλεπτο timer.

```
using UnityEngine;

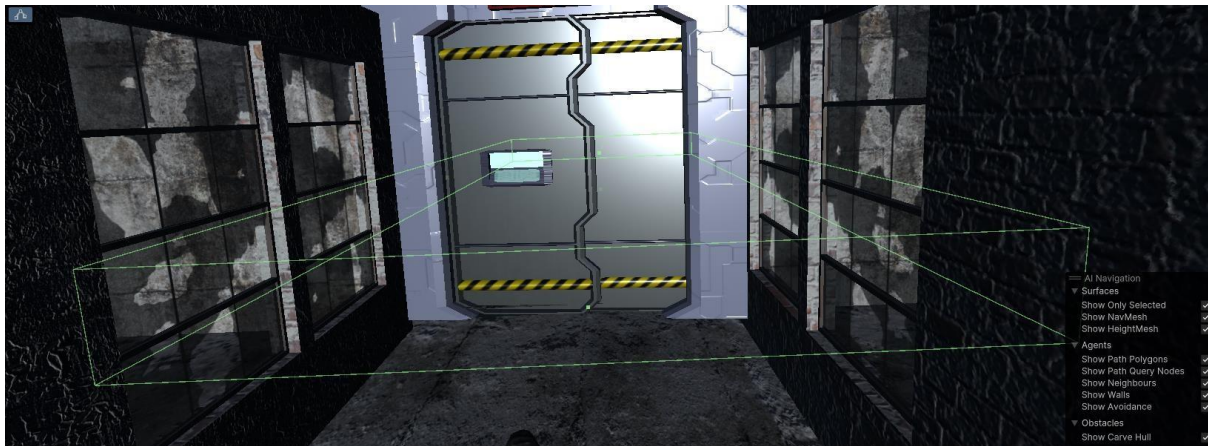
public class SecretRoomTrigger : MonoBehaviour
{
    private LevelManager levelManager;

    void Start()
    {
        levelManager = FindObjectOfType<LevelManager>();
    }

    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            if (levelManager != null)
                levelManager.PlayerEnteredSecretRoom();
        }
    }
}
```

Εικόνα 114

Το script τοποθετείται σε ένα invisible trigger collider στην είσοδο του Secret Room. Στο OnTriggerEnter() ελέγχεται αν το object που μπήκε στον trigger είναι ο παίκτης μέσω του tag, και αν ναι καλείται η PlayerEnteredSecretRoom() του LevelManager



Εικόνα 115

11. Μελλοντικά Implementations

11.1 Επέκταση Περιεχομένου

Το παιχνίδι στην τρέχουσα μορφή του αποτελεί μια ολοκληρωμένη εμπειρία, ωστόσο υπάρχουν πολλές δυνατότητες επέκτασης του περιεχομένου που θα μπορούσαν να εμπλουτίσουν σημαντικά το gameplay. Πρώτα και κύρια, η προσθήκη επιπλέον επιπέδων με νέα και διαφορετικά περιβάλλοντα θα έδινε στον παίκτη μεγαλύτερη ποικιλία και θα επεκτείνει τη διάρκεια ζωής του παιχνιδιού. Κάθε νέο επίπεδο θα μπορούσε να εισάγει νέους τύπους εχθρών με ξεχωριστές ικανότητες — για παράδειγμα εχθρούς που θεραπεύουν τους γύρω τους ή εχθρούς που επιτίθενται από απόσταση με διαφορετικά μοτίβα. Επιπλέον, η προσθήκη νέων όπλων όπως sniper rifle ή rocket launcher θα εμπλούτιζε το σύστημα XP και θα έδινε στον παίκτη περισσότερες επιλογές στρατηγικής. Τέλος, τα μυστικά δωμάτια θα μπορούσαν να εξελιχθούν από απλούς χώρους μάχης σε πλήρη puzzles που απαιτούν σκέψη και εξερεύνηση για να λυθούν.

11.2 Τεχνικές Βελτιώσεις

Από τεχνικής άποψης, η σημαντικότερη βελτίωση που θα μπορούσε να υλοποιηθεί είναι ένα σύστημα αποθήκευσης και φόρτωσης (Save/Load), το οποίο θα επέτρεπε στον παίκτη να συνεχίσει από εκεί που σταμάτησε χωρίς να χρειάζεται να ξεκινά από την αρχή. Παράλληλα, η τεχνητή νοημοσύνη των εχθρών θα μπορούσε να βελτιωθεί με πιο εξελιγμένους αλγόριθμους pathfinding, κάνοντας τους εχθρούς πιο έξυπνους και δύσκολους να αντιμετωπιστούν. Στον οπτικό τομέα, η προσθήκη particle effects για πυροβολισμούς και εκρήξεις καθώς και post-processing effects όπως bloom και depth of field θα αναβάθμιζαν σημαντικά την αισθητική του παιχνιδιού. Τέλος, τα animations των όπλων θα μπορούσαν να εμπλουτιστούν με πιο λεπτομερείς κινήσεις κατά τον πυροβολισμό και το reload.

11.3 Multiplayer

Ένα από τα πιο φιλόδοξα μελλοντικά σχέδια είναι η προσθήκη multiplayer λειτουργικότητας. Ένα co-op mode θα επέτρεπε σε δύο ή περισσότερους παίκτες να συνεργαστούν για να ολοκληρώσουν τα επίπεδα μαζί, μοιράζοντας τα κλειδιά και καλύπτοντας ο ένας τον άλλον κατά τη μάχη. Ένα online leaderboard θα πρόσθετε ανταγωνιστικό στοιχείο, επιτρέποντας στους παίκτες να συγκρίνουν τα αποτελέσματά τους με άλλους σε όλο τον κόσμο. Τέλος, ένα

PvP mode θα έφερνε έναν εντελώς νέο τρόπο παιχνιδιού, βάζοντας τους παίκτες να αντιμετωπίζουν ο ένας τον άλλον στα περιβάλλοντα του παιχνιδιού.

12. Ορολογία

12.1 Εργαλεία

- Unity: Game engine που χρησιμοποιήθηκε.
- C#: Γλώσσα προγραμματισμού.
- NavMesh: Σύστημα AI πλοήγησης.
- Raycast: Ανίχνευση σύγκρουσης μέσω ακτίνας.
- CharacterController: Unity component για κίνηση.
- DontDestroyOnLoad: Επιβίωση object μεταξύ scenes.
- Singleton: Pattern με μοναδικό instance.
- State Machine: Μηχανή καταστάσεων AI.
- PlayerPrefs: Αποθήκευση ρυθμίσεων τοπικά.

12.2 Παιχνίδι

- FPS: First Person Shooter.
- XP: Experience Points - πόντοι εμπειρίας.
- HP: Health Points - ζωή.
- Shield: Ασπίδα που μειώνει damage.
- Magazine: Γεμιστήρας όπλου.
- Pellets: Σφαιρίδια Shotgun.
- Final Boss: Τελευταίος εχθρός.
- Secret Room: Κρυφό δωμάτιο με μυστικό κλειδί.
- Enraged Mode: Κατάσταση Boss με αυξημένη επιθετικότητα.
- Inventory: Σύστημα εναλλαγής όπλων.
- Chip Damage: Οπτικό εφέ σταδιακής μείωσης HP bar.
- Waypoint: Σημείο πλοήγησης για περιπολία.
- Hitbox: Αόρατος collider για damage detection.
- Spawn: Δημιουργία αντικειμένου στη σκηνή.

13. Συμπεράσματα

Η δημιουργία του "The Keyhunter of Hidden Chambers" αποτέλεσε μια συναρπαστική και απαιτητική εμπειρία. Κατά τη διάρκεια της πτυχιακής εργασίας εμβάθυνα σε πολλούς τομείς του game development: φυσική κίνηση, τεχνητή νοημοσύνη, σύστημα όπλων, διαχείριση UI, persistence δεδομένων και δυναμική μουσική.

Η μεγαλύτερη τεχνική πρόκληση ήταν η διαχείριση persistence μεταξύ σκηνών. Το να διατηρούνται XP, κλειδιά, όπλα και υγεία κατά τη μετάβαση απαιτήσε προσεκτικό σχεδιασμό με Singleton pattern και DontDestroyOnLoad. Επίσης, η υλοποίηση του Shotgun reload sound (ανάλογα με τον αριθμό σφαιρών) και του δυναμικού music system ήταν ιδιαίτερα δημιουργικές λύσεις.

Το σύστημα κλειδιών με τη λογική του Secret Room (30-sec timer, persistence μεταξύ levels) αποτέλεσε τον κεντρικό μηχανισμό του παιχνιδιού και έδωσε στον παίκτη ελευθερία επιλογής, να αναζητήσει το μυστικό κλειδί ή να το αφήσει για αργότερα.

Συνοψίζοντας, το "The Keyhunter of Hidden Chambers" είναι ένα πλήρες παιχνίδι που αποδεικνύει τις δυνατότητες της Unity3D και της C# για δημιουργία σύνθετων game mechanics. Η εμπειρία αυτή αποτελεί εξαιρετική βάση για μελλοντικά projects στον τομέα του game development.

14. Βιβλιογραφία

- [1] Unity game engine <https://unity.com/>
- [2] Unity Asset Store <https://assetstore.unity.com/>
- [3] Unity Learn <https://learn.unity.com/>
- [4] Unity Documentation <https://docs.unity3d.com/>
- [5] Unity Input System <https://docs.unity3d.com/Packages/com.unity.inputsystem@latest>
- [6] Unity NavMesh <https://docs.unity3d.com/Manual/nav-NavigationSystem.html>
- [7] Mixamo <https://www.mixamo.com/>
- [8] Nystrom, Robert. Game Programming Patterns. 2014.
<https://gameprogrammingpatterns.com>
- [9] Sung, Kelvin και Smith, Gregory. Basic Math for Game Development with Unity 3D. CRC Press, 2019.
- [10] Low Poly Weapons VOL.1 - Unity Asset Store
- [11] Kamitsios, M., Papadimitriou, S., Chrysafiadi, K., & Virvou, M. (2018). Incorporating quizzes into a 3D educational game calling web services. In EdMedia + Innovate Learning 2018 (pp. 1170-1179). Association for the Advancement of Computing in Education (AACE).
- [12] Kamitsios, M., Chrysafiadi, K., Virvou, M., & Sakkopoulos, E. (2018). A Stereotype User Model for an Educational Game: Overcome the Difficulties in Game Playing and Focus on the Educational Goal. 2018 9th International Conference on Information, Intelligence, Systems and Applications (IISA).
IEEE. <https://doi.org/10.1109/IISA.2018.8633655>
- [13] Papadimitriou, S., Chrysafiadi, K., Kamitsios, M., & Virvou, M. (2018). FSP Creator: A Novel Web Service API Creator of Fuzzy Students Progress Profile. 2018 9th International Conference on Information, Intelligence, Systems and Applications (IISA), 1-7. IEEE.
<https://doi.org/10.1109/IISA.2018.8633662>
- [14] Papadimitriou, S., Kamitsios, M., Chrysafiadi, K., & Virvou, M. (2021). Learn-and-play personalised reasoning from point-and-click to virtual reality mobile educational games. Intelligent Decision Technologies, 15(2), 321-332. <https://doi.org/10.3233/IDT-200204>
- [15] Chrysafiadi, K., Kamitsios, M., & Virvou, M. (2023). Fuzzy-based dynamic difficulty adjustment of an educational 3D-game. Multimedia Tools and Applications, 82, 27525-27549. <https://doi.org/10.1007/s11042-023-14515-w>
- [16] Chrysafiadi, K., Papadimitriou, S., Kamitsios, M., & Virvou, M. (2026; first online 2025). User Experience in Serious Games: Adaptivity in the Plot and Environment of the Game. In M. Virvou et al. (Eds.), Artificial Intelligence-Empowered Software Engineering 2024, Lecture Notes in Artificial Intelligence and Soft Computing, Vol. 52, pp. 215-230. Springer. https://doi.org/10.1007/978-3-031-98410-5_13
- [17] Monster Orc - Unity Asset Store
- [18] RPG_FPS_game_assets_industrial - Unity Asset Store

[19] Future Pad Door - FREE - Unity Asset Store

[20] Hunter_1 - Unity Asset Store

[21] _Creepy_Cat - Unity Asset Store