ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
UNIVERSITY OF PIRAEUS

Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Σχεδιασμός και Υλοποίηση Μηχανισμών Διαδραστικού Παιχνιδιού Η/Υ με τη Χρήση της Μηχανής Γραφικών Unity 3D** **Design and Implementation of Interactive Game Mechanisms using Unity Engine 3D**
Ονοματεπώνυμο Φοιτητή	**ΧΡΗΣΤΟΣ ΜΙΖΗΚΑΚΗΣ**
Πατρώνυμο	**ΙΩΑΝΝΗΣ**
Αριθμός Μητρώου	**ΜΠΠΛ/ 14050**
Επιβλέπων	**Μαρία Βίρβου, Καθηγήτρια**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Βίρβου Μαρία Καθηγήτρια	Τσιχριντζής Γεώργιος Καθηγητής	Αλέπης Ευθύμιος Επίκουρος Καθηγητής

Περίληψη

Το παιχνίδι μας υλοποιήθηκε στην πλατφόρμα Unity 2018.1.1f1. Στο **κεφάλαιο 1**,θα γίνει μια αναφορά στη σύντομη «Μεγάλη» ιστορία των βιντεοπαιχνιδιων. Στο **κεφάλαιο 2**, θα περιγράψουμε τη Λογική του παιχνιδιού καθώς και τους κανόνες – συστήματα που χρειάζονται για να φαίνεται ολοκληρώμενο το παιχνίδι μας. Στο **κεφάλαιο 3**, κάνουμε σχεδιασμό και ανάλυση με διαγράμματα Ροής .Στο **κεφάλαιο 4**, θα εξερευνήσουμε τους τροπους με τους οποίους μπορούμε να προγραμματίσουμε την αλληλεπίδραση του χρήστη με το παιχνίδι, πιο συγκεκριμένα αναφερόμαστε στα συστήματα χειρισμού της Uniy και τι κώδικας C# χρειάζεται να γραφεί για να υλοποιηθούν τα εν λόγω συστήματα. Στο **κεφάλαιο 5**, αφοσιωνόμαστε στην περιγραφή του σχεδιασμου του παιχνιδιού χρησμοποιώντας εικόνες και αναλύοντας τον κώδικα που χρησιμοποιήσαμε. Τέλος, στο **κεφάλαιο 6**, περιγράφουμε τις μελλοντικές προεκτάσεις του παιχνιδιού και παραθέτουμε τα συμπεράσματα μας.

Abstract

Our game was created on the Unity platform version 2018.1.1f1. In Chapter 1, there will be a quick reference to the history and evolution of Video Games.In Chapter 2, we are going to describe the logic and rules that hide behind the functionality of our game. In Chapter 3, we design and implement UML diagrams. In Chapter 4, we explore and study in depth how to program the interaction between the player and the game.In chapter 5, we focus on describing every part of the game using images and analyzing the code that we used. Finally in Chapter 6, we make suggestions that could improve the functionality of the game and come to concludions.

Ευχαριστίες

Αφιερώνω την παρούσα μεταπτυχιακή διατριβή στους υπέροχους γονείς μου Ιωάννη και Σοφία καθώς και στον πολυαγαπημένο μου αδερφό Τάσο που βρίσκεται εκεί ψηλα στον ουρανό. Σας ευχαριστώ γιατί δεν πάψατε λεπτό να είστε δίπλα μου και να με στηρίζετε με όλη σας την Ψυχή και την Αγάπη.

Πίνακας Περιεχομένων

ΚΕΦΑΛΑΙΟ 1. Η Ιστορία των Video Games

1.1 Ιστορία και Εξέλιξη

Η προέλευση των βιντεοπαιχνιδιών ή video games εντοπίζεται στα πρώτα καθοδικού σωλήνα συστήματα άμυνας που χρησιμοποιήθηκαν στα τέλη της δεκαετίας του 1940. Τα προγράμματα αυτά αργότερα προσαρμόστηκαν σε άλλα πιο απλά παιχνίδια κατά τη δεκαετία του ’50.

Από τα τέλη του ’50 και μέσα στη δεκαετία του ’60, τα περισσότερα παιχνίδια για υπολογιστές αναπτύχθηκαν κυρίως σε μεγάλα υπολογιστικά συστήματα, και σταδιακά άρχισαν να γίνονται πιο καινοτόμα και κυρίως πιο πολύπλοκα. Από την περίοδο αυτή και έπειτα τα βιντεοπαιχνίδια χωρίστηκαν σε διαφορετικές πλατφόρμες: arcade, mainframe, κονσόλες, προσωπικούς υπολογιστές ή κονσόλες χειρός (όπως π.χ. το GameBoy).

Το πρώτο εμπορικό βιντεοπαιχνίδι ήταν το Computer Space και κυκλοφόρησε το 1971, βάζοντας τα θεμέλια για τη δημιουργία και ανάπτυξη μιας νέας βιομηχανίας διασκέδασης στα τέλη της δεκαετίας του 1970 στις ΗΠΑ, την Ιαπωνία και την Ευρώπη. Αυτή η άνθιση επήλθε, όμως, μέσα από δύο μεγάλα τεχνολογικά κραχ.

Το πρώτο σημειώθηκε το 1977, όταν οι εταιρείες αναγκάστηκαν να πουλήσουν απαρχαιωμένα υπολογιστικά συστήματα, κατακλύζοντας την αγορά

Στις αρχές της δεκαετίας του 70 η αγορά των παιχνιδιών με κέρματα ευημερούσε με κυρίαρχο την Atari. Πολλές εταιρίες, μεγάλη παραγωγή παιχνιδιών. Αυτό σταμάτησε το 1978 με την πρώτη μεγάλη οικονομική κρίση της βιομηχανίας. Προκλήθηκε από την υπερπαραγωγή μηχανών χωρίς να υπάρχει αντίστοιχη ποικιλία από παιχνίδια. Η Atari έχασε την κυριαρχία της αγοράς και άνοιξε ο δρόμος για εταιρίες από την Ιαπωνία.

Έξι χρόνια αργότερα, το 1983, ένα δεύτερο, ακόμη μεγαλύτερο κραχ σημειώθηκε. Αιτία στάθηκε ένας «κατακλυσμός» βιντεοπαιχνιδιών που έπνιξε την αγορά και οδήγησε σε πλήρη κατάρρευση της βιομηχανίας παιχνιδοκονσολών σε παγκόσμιο επίπεδο. Αυτό οδήγησε σε μια γεωγραφική αλλαγή της παντοδυναμίας της αγοράς: η Βόρεια Αμερική έχασε τα πρωτεία στα βιντεοπαιχνίδια, με την Ιαπωνία να καθίσταται το απόλυτο αφεντικό.

Τότε, ακριβώς, ξεκίνησε η απόλυτη «έκρηξη» στα βιντεοπαιχνίδια...

Οι κύριες περίοδοι και η πορεία των ηλεκτρονικών παιχνιδιών:

1. Πρώτη γενιά (1972–1976)
2. Δεύτερη γενιά (1977–1982)
3. Η διάλυση της βιομηχανίας το 1983
4. Τρίτη γενιά (1984–1992)
5. Τέταρτη γενιά (1993–1996)
6. Πέμπτη γενιά (1997–2002)
7. Έκτη γενιά (2003–2006)
8. Έβδομη γενιά (2004–2009)

Όπως είναι φυσικό δεν ξύπνησε μια μέρα ο κόσμος και τα βρήκε όλα μπροστά του. Για να απολαμβάνουμε σήμερα τα ηλεκτρονικά παιχνίδια με τόσους πολλούς τρόπους υπήρξε μία "σειρά".

Η πορεία τους είχε ως εξής:

1. **Μεγάλους υπολογιστές** Ξεκινήσανε σε μεγάλους υπολογιστές πανεπιστημίων.
2. **Μίνι υπολογιστές (mini-computers)** Αρχικά και αυτοί από πανεπιστήμια.
3. **Καμπινες (Arcade cabinets)** Που ήταν και η πρώτη ουσιαστική επαφή του κόσμου με τα ηλεκτρονικά παιχνίδια.
4. **Κονσόλες** Με σχεδόν ταυτόχρονη κυκλοφορία με τα arcade cabinets απλά με διαφορά 6 μηνών.
5. **Φορητές κονσόλες** Μία χαρακτηριστική εταιρεία είναι η NINTENDO.
6. **Οικιακοί υπολογιστές** ξεκινώντας με παιχνίδια που έφτιαχναν οι ίδιοι ο χρήστες.
7. **Online gaming**

1.2 Ποια είναι τα video παιχνίδια με τα οποία ασχολούνται οι Έλληνες gamers;

Οι **Έλληνες gamers** προτιμούν τα παιχνίδια περιπέτειας, οι **Ελληνίδες gamers** τα puzzle games και τα smartphones γίνονται όλο και πιο δημοφιλή όσον αφορά το κομμάτι των video παιχνιδιών. Αυτά είναι ορισμένα από τα βασικά συμπεράσματα έρευνας της PayPal και της SuperData σχετικά με την καταναλωτική συμπεριφορά στη διεθνή αγορά των ψηφιακών αγαθών με τη συμμετοχή 25.000 ατόμων σε τουλάχιστον 25 αγορές, συμπεριλαμβανομένης και της Ελλάδας

Σύμφωνα με την έρευνα της PayPal και της SuperData το 2018, οι **άνδρες gamers** στην **Ελλάδα λατρεύουν τα παιχνίδια δράσης-περιπέτειας** – τουλάχιστον το 45% δήλωσε ότι έπαιξε τέτοιου είδους παιχνίδια κατά τους τελευταίους τρεις μήνες. Τα συγκεκριμένα αυτά παιχνίδια περιλαμβάνουν δοκιμασίες που ολοκληρώνεις καθώς προχωρά η αφήγηση της ιστορίας.

Οι περισσότερες **Ελληνίδες gamers** (55%) επέλεξαν την **ησυχία των puzzle games** (σε σχέση με το 27% των ανδρών) και το ένα τρίτο έπαιξε arcade games το τελευταίο τρίμηνο. Το στατιστικό αυτό στοιχείο συνάδει με τις ευρωπαϊκές τάσεις, με το 48% των gamers γένους

θηλυκού στην Γηραιά Ήπειρο να παίζει κυρίως puzzles. Σε διεθνές επίπεδο, το 49% των γυναικών gamers προτίμησε την ψυχαγωγία των παιχνιδιών δράσης.

Η δημοτικότητα των συσκευών gaming διαφέρει κατά πολύ ανά αγορά. Οι κονσόλες τα πάνε καλά στην Βόρεια Αμερική και τη Δυτική Ευρώπη, ενώ οι ασιατικές αγορές κλίνουν προς το mobile gaming. Όσον αφορά τις συνήθειες και τις διαθέσεις προς τα ψηφιακά παιχνίδια, σύμφωνα με τα αποτελέσματα της έρευνας ειδικά για την Ελλάδα, όταν οι Έλληνες ερωτήθηκαν σε τι είδους συσκευές έπαιξαν το τελευταίο τρίμηνο, διαθέτοντας την επιλογή πολλαπλών απαντήσεων, αποδείχθηκε ότι προτίμησαν κυρίως τις κινητές συσκευές – το 71% των συμμετεχόντων έκανε χρήση smartphones, ενώ το 49% διάλεξε τα tablets.

1.3 Το Gaming ως επάγγελμα

Οι Έλληνες gamers όχι μόνο παρακολουθούν περιεχόμενο gaming video (είναι ενδιαφέρον το γεγονός ότι οι άνδρες είναι πολύ πιο δραστήριοι στον τομέα αυτόν – το 45% σε σχέση με το 29% των γυναικών gamers), αλλά και δημιουργούν μόνοι το δικό τους – το 9% των ερωτηθέντων, το οποίο βέβαια ως ποσοστό είναι ούτως ή άλλως δύο φορές χαμηλότερο από τον παγκόσμιο μέσο όρο.

Την ίδια στιγμή, το ένα τρίτο δεν πληρώνεται καν για τη δημιουργία του. Παραδόξως, οι γυναίκες streamers έχουν ακόμα λιγότερες πιθανότητες να λάβουν οποιαδήποτε πληρωμή σε σχέση με τον μέσο άνδρα Έλληνα streamer – 28% vs 39%. Εντούτοις, εάν πληρωθούν τελικά, ισχύουν οι ίδιοι βασικοί τρόποι πληρωμής με τους υπόλοιπους – διαφημίσεις και χορηγίες.

ΚΕΦΑΛΑΙΟ 2. Η Λογική του παιχνιδιού – Κανόνες – Συστήματα

2.1 Σενάριο

Το game μας είναι ένα διασκεδαστικό action First Person Shooter που εξελίσσεται σε ένα απομακρυσμένο χωριό όπου το ξέσπασμα ενός παράξενου ιου μετατρέπει τους ανθρώπους σε αιμοσταγή επικίνδυνα ζόμπι.Η βασική φιλοσοφία του παιχνιδιού είναι να προμηθευτούμε όπλα γρήγορα και να σώσουμε το χωριό από τα ζόμπι.

2.2 Οι Οντότητες – Ο Κόσμος στο παιχνίδι μας

Εδώ καλούμστε να αναλύσουμε τις οντότητες που θα υπάρχουν στο παιχνλιδι μας.

- Ο Player – Χαρακτήρας Ήρωας μας (first person Character)
- Ο Vendor – Προμηθευτής όπλων

Εικόνα 2.2 Vendor – Προμηθευτής όπλων

- Στρατιώτης

Εικόνα 2.2.1 Στρατιώτης

- Τα εχθρικά ζόμπι

Εικόνα 2.2.2 Ζόμπι

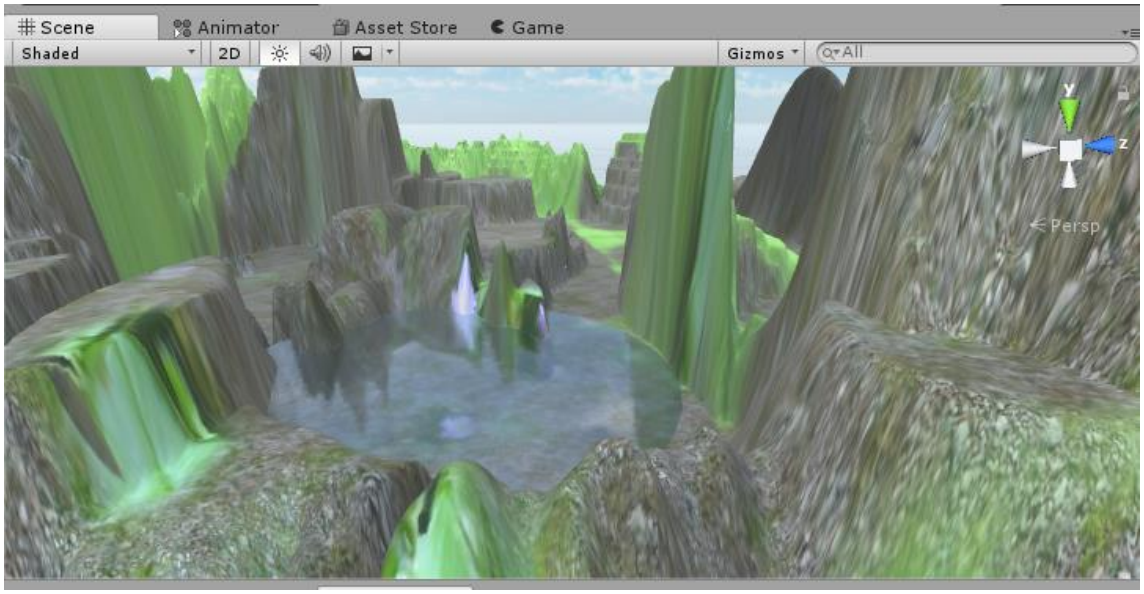
Το παιχνίδι μας περιλαμβάνει 3 διαφορετικές περιοχές (Territories).

- Village Territory

Εικόνα 2.2.3 Village – Χωριό

- Κοιλάδα – Lake Valley

Εικόνα 2.2.4 Κοιλάδα



- Ηφαιστειγενής περιοχή – Lava Territory
  - ***Εικόνα 2.2.5 Ηφαιστειγενής περιοχή***

2.3 Κανόνες – Συστήματα

Τέλος θα αναφέρουμε ενδεικτικά τους μηχανισμούς εκείνους, που χρειάζεται το παιχνιδι μας να φαίνεται ολοκληρωμένο. Θα χρειαστούμε να κράτάμε τις εξής τιμές που θα παίζουμε:

- Ενέργεια – Player HealthBar
- Πόντοι
- Πυρομαχικά – Ammo
- Σκορ
- Χρήματα - Money
- Ζόμπι που σκοτώσαμε – Zombies Killed

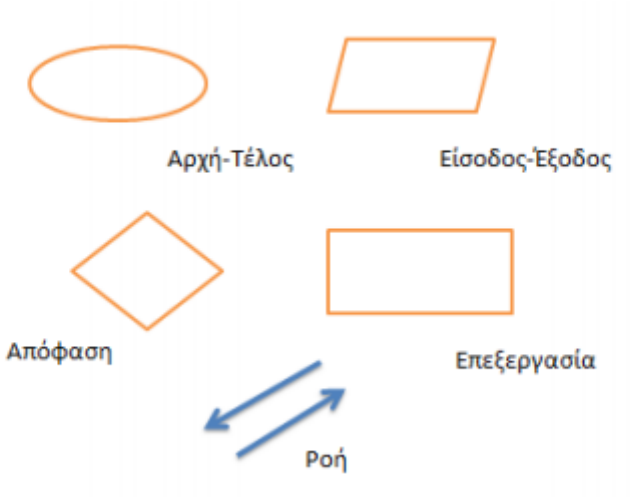
2.3 .1 Κανόνες – Συστήματα

1. Όσον αφορά την ενέργεια θα έχουμε 100 πόντους. Κάθε φορά που θα μας επιτίθεται ένα ζόμπι θα μας αφαιρεί 10 πόντους. Σε δέκα χτυπήματα έχουμε καταστραφελι.
2. Όσον αφορά τους πόντους, θα κερδίζουμε 5 πόντους κάθε φορά που τους πετυχαίνει η σφαίρα μας.
3. Μπορούμε να πυροβολήσουμε κάθε Ν δευτερόλεπτα (Fire rate)
4. Οι εχθροί θα μας αφαιρούν ζωή κάθε φορά που υπάρχει σύγκρουση (collision) με τον χαρακτήρα μας
5. Οι εχθροία θα βγαίνουν από καθορισμένα WayPoints με τυχαίο εύρος που έχουμε δημιουργήσει κάθε 2 δευτερόλεπτα.
6. Αν πέσουμε στη λάβα ο ήρωας χάνει 10 πόντους από την ενέργεια του.
7. Όταν έχουμε χάσει ενέργεια μπορούμε να την αναπληρώσουμε με μαγικό κόκκινο μπουκαλάκι
8. Ο χαρακτήρας μας έχει τη δυνατότητα να συλλέγει αντικείμενα από το έδαφος
9. Τα αντικείμενα τα αποθηκέυιε στο Inventory απ’οπου μπορεί και να τα χρησιμοποιήσει.
10. Ο χαρακτήρας μας μπορεί να αλλάζει όπλα (weapon switching)
11. Έχει τη δυνατότητα να γεμίζει με πυρομαχικά τα όπλα του.

ΚΕΦΑΛΑΙΟ 3. Υλοποίηση με Διαγράμματα Ροής Εργασίας (Workflow Diagrams)

1.1 Διάγραμμα Ροής για την Έναρξη του Παιχνιδιού

Το διάγραμμα ροής προγράμματος (flowchart), σε συντομογραφία ΔΡΠ, μπορεί να το συναντήσουμε σπανιότερα και ως λογικό διάγραμμα. Πρόκειται για μία σχηματική παράσταση του αλγορίθμου. Η έννοια του αλγορίθμου είναι πολύ βασική στην Πληροφορική, ενώ το διάγραμμα ροής ένα πολύ χρήσιμο περιγραφικό εργαλείο. Ένα διάγραμμα ροής αποτελείται από σύμβολα και λέξεις οι οποίες είναι σε θέση να περιγράψουν με λεπτομέρεια κάθε αλγόριθμο, κάθε διαδικασία δηλαδή επίλυσης οποιουδήποτε προβλήματος.



Θα ξεκινήσουμε να αναλύσουμε την γενική επαφή του χρήστη με το παιχνίδι μας,

Όπως βλέπουμε στο παρακάτω διάγραμμα, ο χρήστης αρχικά έρχεται σε επαφή με το αρχικό μενού του παιχνιδιού. Του παρουσιάζονται 2 επιλογές:

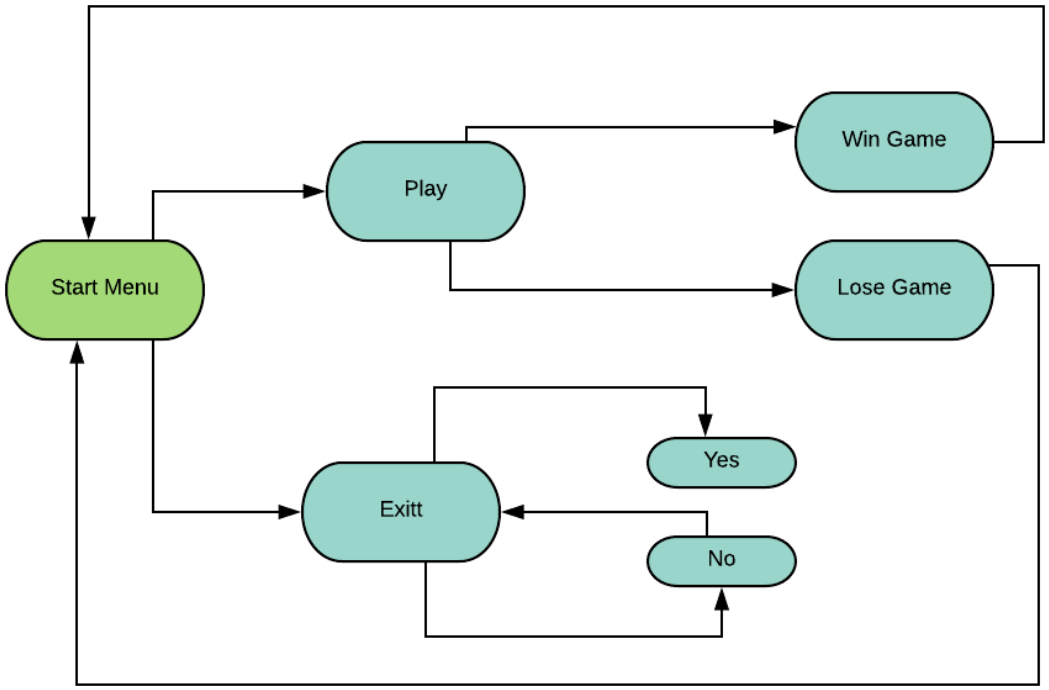
I. Να ξεκινήσει το παιχνίδι πατώντας το κουμπί ¨Play¨,
II. Να αποχωρήσει από το παιχνίδι, πατώντας στο κουμπί ¨EXIT¨ και στην συνέχεια ¨YES¨ στο υπομενού που του εμφανίζεται.

Εφόσον επιλέξει να εκκινήσει το παιχνίδι δύο είανια τα σενάρια που παρουσιάζονται.

I. Να κερδίσει, τερματίζοντας το παιχνίδι,
II. Να χάσει.

Όποιο από τα δύο αυτά σενάρια πραγματοποιηθεί, ο χρήστης επανέρχεται στο αρχικό μενού.

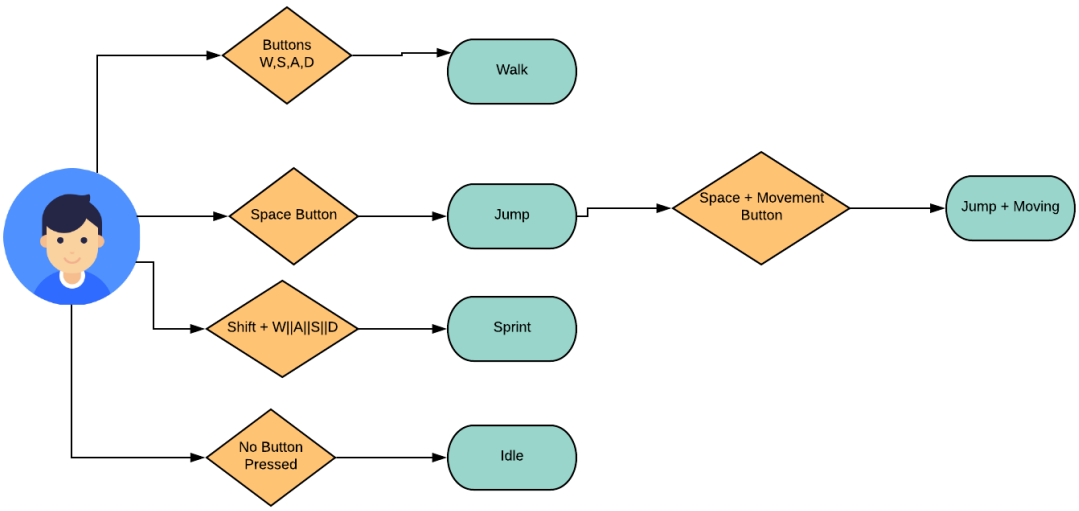
Εφόσον η επιλογή του χρήστη είναι να ξεκινήσει το παιχνίδι, ο χρήστης είναι έτοιμος να πλοηγηθεί στο παιχνίδ με τον Player που έχουμε φτιάξει από την αρχή from Scratch.



Ας συνεχίσουμε να δούμε το διάγραμμα Ροής για την πλοήγηση του χρήστη στο παιχνίδι **Player'sMovements Flow Chart**

Όπως βλέπουμε στο παρακάτω διάγραμμα, ο παίχτης μπορεί να κάνει τις εξής κινήσεις:

1. Move forward, πατώντας στο πληκτρολόγιο ένα από τα κουμπιά ('W','S','A','D')
2. Jump, πατώντας στο πληκτρολόγιο το κουμπί ¨Space¨
3. Jump + Moving, πατώντας στο πληκτρολόγιο το κουμπί 'Space' μαζί με οποιοδήποτε κουμπί κίνησης
4. Sprint, πατώντας στο πληκτρολόγιο το κουμπί 'Shift' μαζί με οποιοδήποτε κουμπί κίνησης



ΚΕΦΑΛΑΙΟ 4. Βασικές Έννοιες Της Unity3d Game Engine

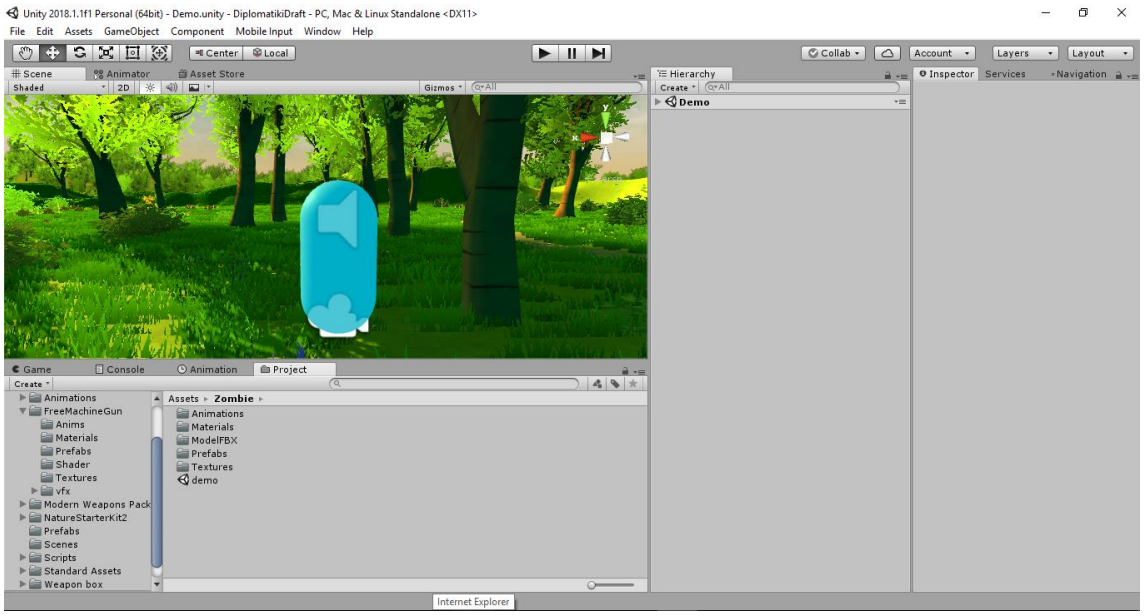
4.1 Εισαγωγή Στο User Interface Της Uinty3d Game Engine

Πριν ξεκινήσουμε την ανάπτυξη του παιχνιδιού μας και τη παρουσίαση των λειτουργιών τηςUnity3D, ας δούμε από ποια παράθυρα απαρτίζεται το εργαλείο αυτό. Η πρώτη μας επαφή με οποιουδήποτε είδους λογισμικού λοιπόν έχει να κάνει με την διεπαφή χρήστη (User Interface) της εφαρμογής.

Εάν καθίσουμε και «σπάσουμε» αυτό που βλέπουμε σε διαφορετικά υπό-παράθυρα, τα οποία απαρτίζουν αυτήν την «επιβλητική» εικόνα που έχει, θα παρατηρήσουμε πως όλα χρειάζονται και όλα έχουν κάποιο σκοπό και τρόπο χρήσης από τον χρήστη.

4.1.1 Παράθυρο 1: Μια πρώτη ματιά στο User Interface

Εικόνα 4.2.1. Το User Interface της Unity3D

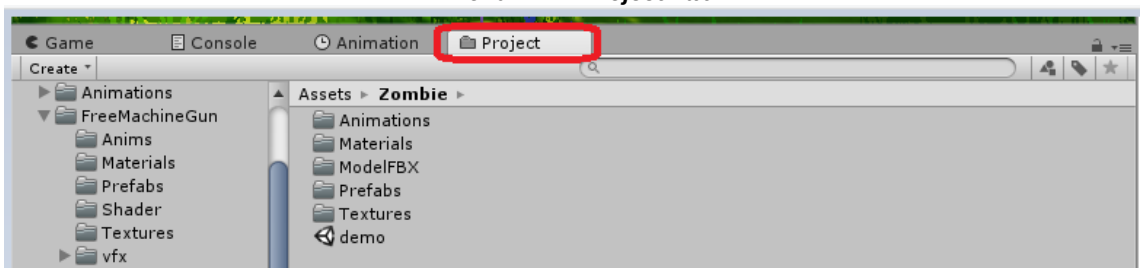


4.1.2 Παράθυρο 2: Project Tab

Στο φάκελο αυτό, όπως λέει και το όνομά του, υπάρχουν όλα τα Assets του παιχνιδιού μας.

Καλό είναι να υπάρχουν μέσα μόνο τα Assets που χρησιμοποιούμε στο παιχνίδι, καθώς κάθε τι που υπάρχει σε αυτόν τον φάκελο αυξάνει το μέγεθος του τελικού παιχνιδιού.

Εικόνα 4.1.2. Project Tab

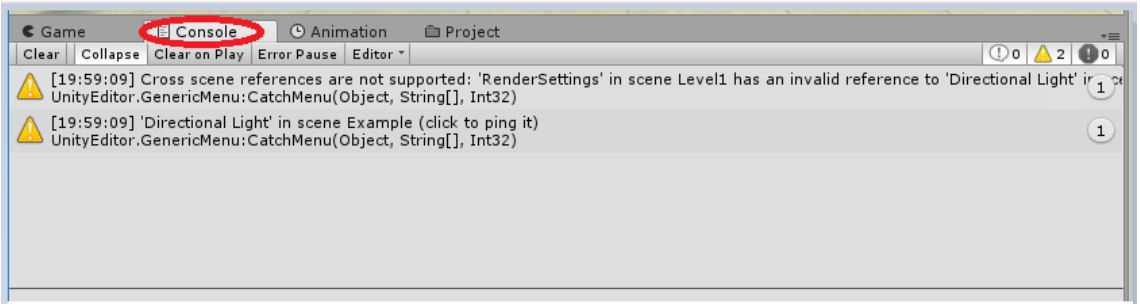


4.1.3 Παράθυρο 3: Console Tab

Εδώ θα ανατρέχουμε συχνά όταν γράφουμε κώδικα. Στο Console Tab υπάρχει η κονσόλα της Unity, η οποία μας δίνει τη δυνατότητα να κάνουμε debugging όταν έχουμε μπει στο προγραμματιστικό κομμάτι της παραγωγής (production).

Στην κονσόλα καταγράφονται errors που αφορούν εσωτερικά την Unity και τον Compiler της, compile/syntax errors που αφορούν τα scripts τα οποία γράφουμε, warnings τα οποία μας ενημερώνουν για τυχόν ρυθμίσεις οι οποίες ενδέχεται να προκαλέσουν errors σε runtime ή μεταβλητές κ.ά. που καταλαμβάνουν χώρο στη μνήμη, ενώ ουσιαστικά δεν έχουν καμία χρησιμότητα, και επιπλέον μας παρέχει generic messages, τα οποία αφορούν διάφορες ενέργειες της Unity, όπως πχ κάποιο Update κάποιου plugin ή ότι δεν έχουμε προετοιμάσει σωστά το scene μας.

Εικόνα 4.1.3 ConsoleTab

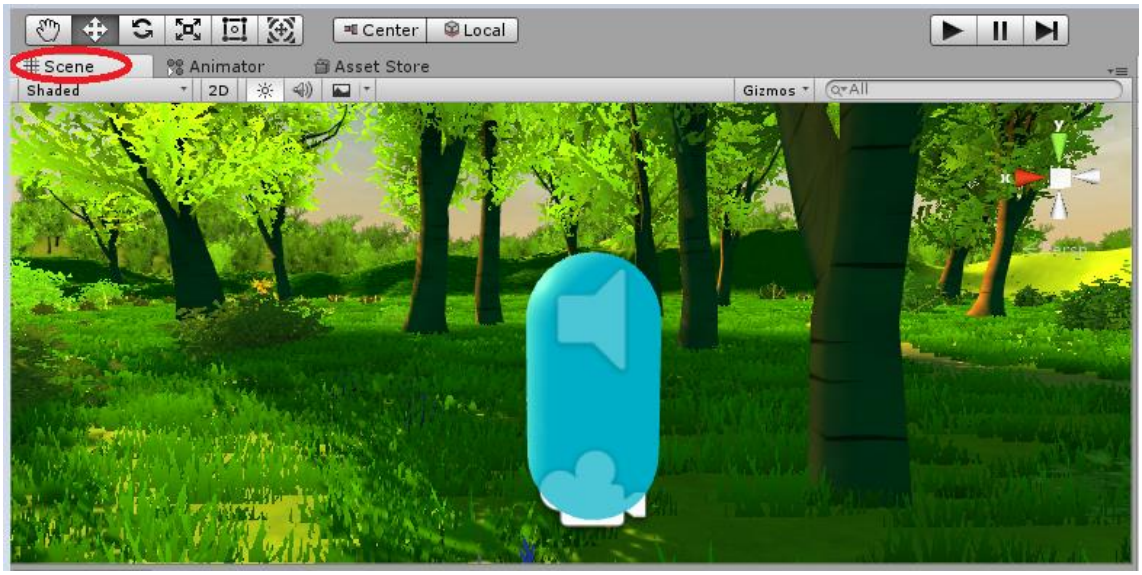


4.1.4 Παράθυρο 4: Scene View

Στο Scene view πρακτικά βλέπουμε όλο μας το Scene το οποίο σχεδιάζουμε αυτή τη στιγμή. Scene σαν να λέμε Map ή Level για ευκολία. Με το Scene view κάνω εύκολα και γρήγορα περιήγηση (navigation) μέσα στο Scene μου σε πραγματικό χρόνο (Realtime), χωρίς να τρέχει το παιχνίδι.

Με το ποντίκι και το πληκτρολόγιο μπορώ εύκολα να δω όλο το level από την αρχή μέχρι το τέλος και να το σχεδιάσω όπως με βολεύει και μετά να δω το αποτέλεσμα σε runtime.

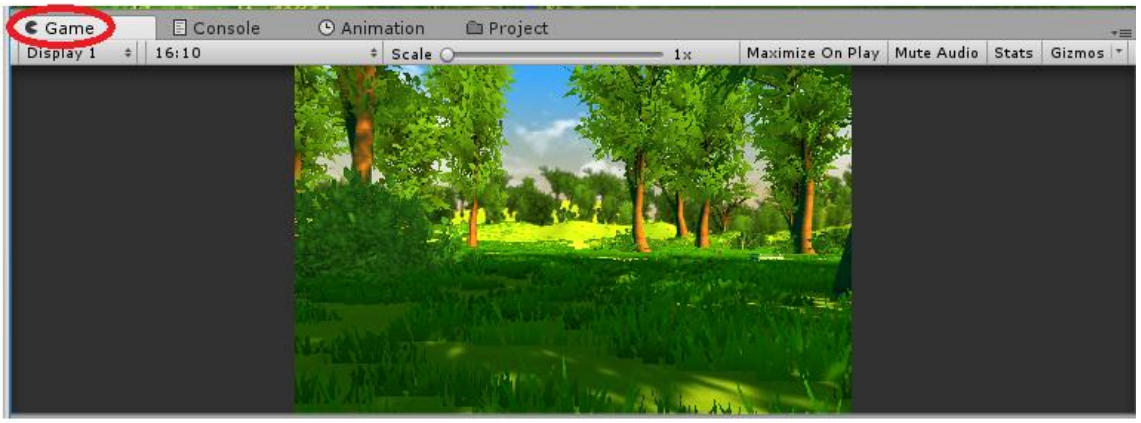
Εικόνα 4.1.4.Scene View



4.1.5 Παράθυρο 5: Game View

Σε αντίθεση με το Scene View, το παράθυρο Game View πρακτικά δείχνει αυτό που θα φαίνεται και μέσα στο παιχνίδι. Ό,τι βλέπω δηλαδή στο Game View είναι αυτό που θα δει και ο χρήστης, και συνήθως το game view είναι το rendered αποτέλεσμα6 από όλες τις in-game κάμερες που έχουμε στήσει στο scene μας. Για παράδειγμα, η εικόνα 3 δείχνει τις διαφορές ανάμεσα στο Scene και Game View.

Εικόνα 4.1.5.Game View



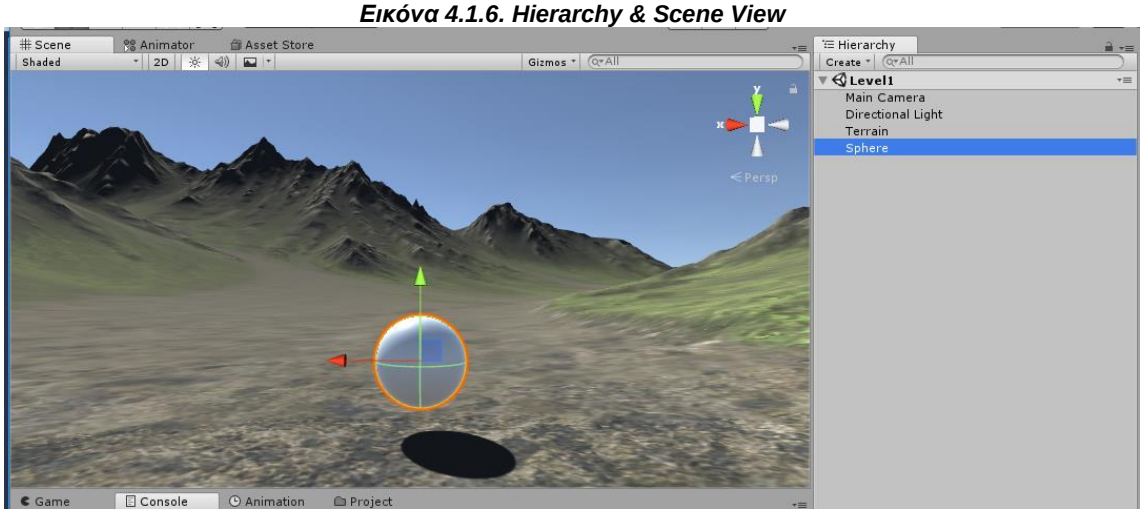
Στο **game** view παρατηρούμε ότι όντως βλέπουμε αυτό που κάνει render η κάμερα! Στο **scene** κάνω ελεύθερο navigation και στο **game** βλέπω αυτό που θα βλέπει και ο παίχτης όταν παίζει!

4.1.6 Παράθυρο 6: Hierarchy

Στο Hierarchy βλέπω όλα τα game objects που υπάρχουν αυτή τη στιγμή στο Scene, σε μια ιεραρχική δομή.

Για παράδειγμα, μπορούμε να δούμε ξεκάθαρα πως στην Εικόνα 6 έχουμε μια **Main camera** ένα **Directional Light** ένα **Terrain** και ένα **Sphere**.

Εικόνα 4.1.6. Hierarchy & Scene View



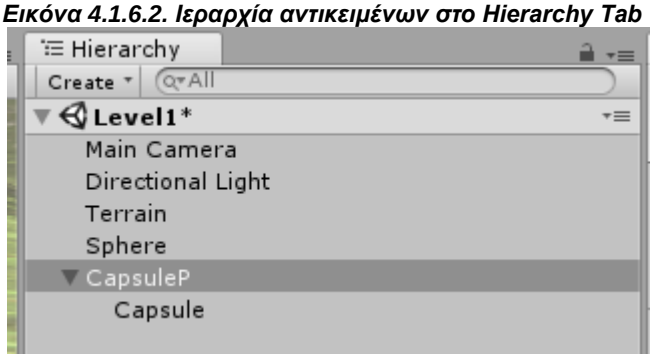
Άρα στο **Hierarchy** βλέπουμε όλα τα **gameobjects** που έχουν φορτωθεί στο

scene μας αυτή τη στιγμή!

Επίσης, αναφέραμε πως στο Hierarchy βλέπουμε όλα τα gameobjects σε ιεραρχική δομή. Με αυτό εννοούμε ότι μπορεί να υπάρχουν ιεραρχίες από GameObjects.

Δηλαδή η έννοια του Child και Parent Gameobject, όπως βλέπουμε στην Εικόνα 6.1

Εικόνα 4.1.6.2. Ιεραρχία αντικειμένων στο Hierarchy Tab



Δημιουργήσαμε ένα empty GO και το μετονομάσαμε σε CapsuleP. Παρατηρούμε ότι το CapsuleP είναι ο **PARENT** του Capsule gameObject και το Capsule είναι το CHILD του CapsuleP. Η σχέση που έχει το parent με το child gameobject είναι ιδιαίτερη, καθώς αν κάνουμε κάποιο transformation στον parent, το ίδιο θα συμβεί και στο child, δηλαδή an ο parent θα μετακινηθεί κατά 2 τιμές στον άξονα X, τότε το ίδιο θα συμβεί και με το child.

Ομοίως στα Scaling και Rotation transforms.

Επίσης, αν ένα **parent gameobject** είναι απενεργοποιημένο στο **Scene** (δηλαδή ανενεργό), τότε θα είναι ανενεργά και όλα τα **child game objects**.

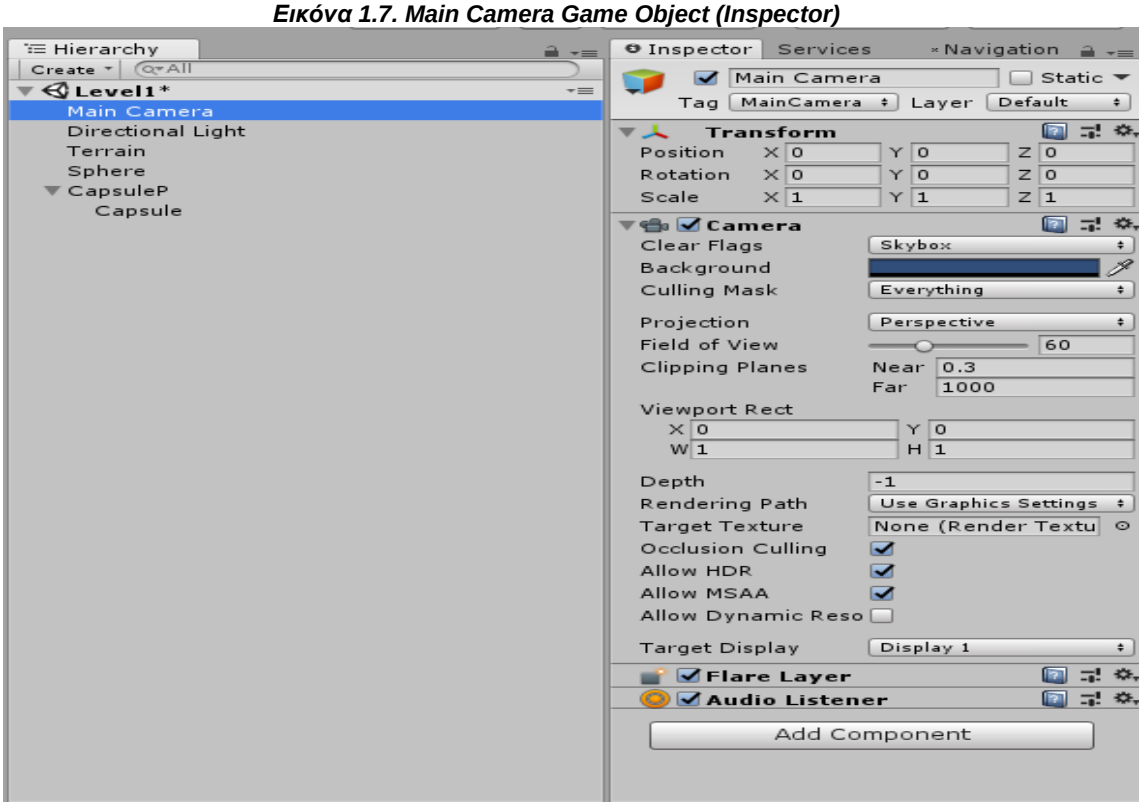
4.1.7 Παράθυρο 7: Inspector

Ίσως από τα πιο σημαντικά παράθυρα στην Unity, ο Inspector δείχνει κάθε φορά, αναλόγως το αντικείμενο που έχω επιλέξει μέσα στο Scene, όλα τα Components του συγκεκριμένου Game Object.

Κάθε Game Object έχει συγκεκριμένα Components πάνω του ανάλογα με την φύση του gameobject.

Έτσι, μπορούμε να “ψαχουλέψουμε”και να δούμε από τι Components-«Συστατικά» αποτελείται κάθε gameobject. Για παράδειγμα, στην Εικόνα 1.7 έχουμε επιλέξει το **Game object “Main Camera”**

Εικόνα 1.7. Main Camera Game Object (Inspector)



4.1.8 Παράθυρο 8: Transformation Widgets

Τα Transformation Widgets, μας επιτρέπουν να κάνουμε 3D επεξεργασία (transform manipulations) των Game Object μας. Το κλασικό εικονίδιο με το «**χεράκι**» μας βοηθάει να κάνουμε Pan μέσα στο 3D Scene μας.

Το 2ο εικονίδιο αφορά το "**Translate**" Tool (Μετακίνηση), και με αυτό κάνουμε Translate τα αντικείμενα μας, μέσα στον 3D χώρο, δηλαδή τα μετακινούμε ως προς τους άξονες

X, Y και Z.

Το 3ο εικονίδιο αφορά το "**Rotate**" Tool (Περιστροφή), με το οποίο περιστρέφουμε τα 3D αντικείμενά μας στον 3D χώρο, δηλαδή περιστροφές γύρω από συγκεκριμένους άξονες X,Y,Z. Οι περιστροφές αυτές υπολογίζονται με Euler συντεταγμένες.

Το 4ο εικονίδιο αφορά το "**Scale**" Tool (Μεγέθυνση), με το οποίο μεγεθύνουμε τα 3D αντικείμενα μας στον 3D χώρο, ως προς συγκεκριμένους άξονες X, Y και Z ή και όλους μαζί αν θέλουμε να κάνουμε uniform scaling.

Το 5ο εικονίδιο έχει έρθει με την έκδοση 4.6 της Unity, η οποία προσθέτει το uGUI στα tools που έχουμε. Το συγκεκριμένο tool μας επιτρέπει να επιλέξουμε οποιοδήποτε αντικείμενο στο scene και να κάνουμε **"2D Transformations**" σε αυτό, δηλαδή να το μεγεθύνουμε, μετακινήσουμε, περιστρέψουμε, όπως θα κάναμε σε κάποιο πρόγραμμα 2D για User Interfaces, για παράδειγμα όπως το Expression Blend.

Εικόνα 1.8. Transformation Widgets

4.1.9 Παράθυρο 9: Pivot/Center & Local/Global Transform Modes

Με αυτές τις επιλογές μπορούμε να αλλάξουμε τον τρόπο που θα συμπεριφερθεί σε συγκεκριμένα Transformation, δηλαδή αν θα μετακινήσουμε ένα αντικείμενο βάσει τις

περιστρέψουμε 2 αντικείμενα από ένα συγκεκριμένο κέντρο μεταξύ τους ή γύρω από τους δικούς τους άξονες.

Εικόνα 1.9 Pivot/Center & Local/Global Transform Modes

4.1.10 Παράθυρο 10: Play Modes

Σε αυτό το panel μπορούμε να δούμε 3 βασικά κουμπιά τα οποία είναι χρήσιμα κατά την διάρκεια της δοκιμής κώδικα (Testing).

PLAY:

Πατώντας το, ξεκινάει το "**Play Mode**" της Unity και τρέχει το παιχνίδι μας! Το Game View ζωντανεύει και μπορούμε να παίξουμε κανονικά..! Η εκτέλεση σταματάει όταν ξαναπατήσουμε το play button. Μια παρατήρηση: Να πούμε σε αυτό το σημείο ότι όταν είμαστε σε Play mode, καλό θα ήταν να γνωρίζουμε πως ό,τι αλλαγές κάνουμε στο Scene μας, **στα gameobjects μας ή και στα components τους δεν αποθηκεύονται όταν σταματήσουμε την εκτέλεση και χάνουμε ό,τι κάναμε.**

.

PAUSE: Αν θελήσουμε ποτέ να σταματήσουμε προσωρινά και όχι εντελώς την εκτέλεση για να δούμε μερικές τιμές στον inspector ή για να δούμε στο scene view σε τι θέση βρίσκονται μερικά gameobjects, μπορούμε να πατήσουμε το pause button. Για να συνεχιστεί η εκτέλεση ξαναπατάμε το pause.

STEP: Στην σπάνια περίπτωση που θέλουμε να είμαστε πολύ σχολαστικοί όσον αφορά την εκτέλεση των scripts μας, τις αλλαγές που γίνονται στο παιχνίδι και τις τιμές να αλλάζουν στον inspector καρέ-καρέ (frame by frame), τότε θα χρησιμοποιήσουμε το step κουμπάκι, καθώς θα προχωρήσει την εκτέλεση του παιχνιδιού κατά ένα καρέ (frame) και θα ξαναγίνει pause αυτόματα. Έτσι, έχουμε πλήρη έλεγχο στην εκτέλεση του παιχνιδιού μας και μας λύνει τα χέρια στην κυριολεξία όταν θα πάμε να κάνουμε debugging.

4.2. Τα Κύρια Αντικείμενα στο Χώρο: GameObjects

4.2.1 Τι είναι ένα GameObject;

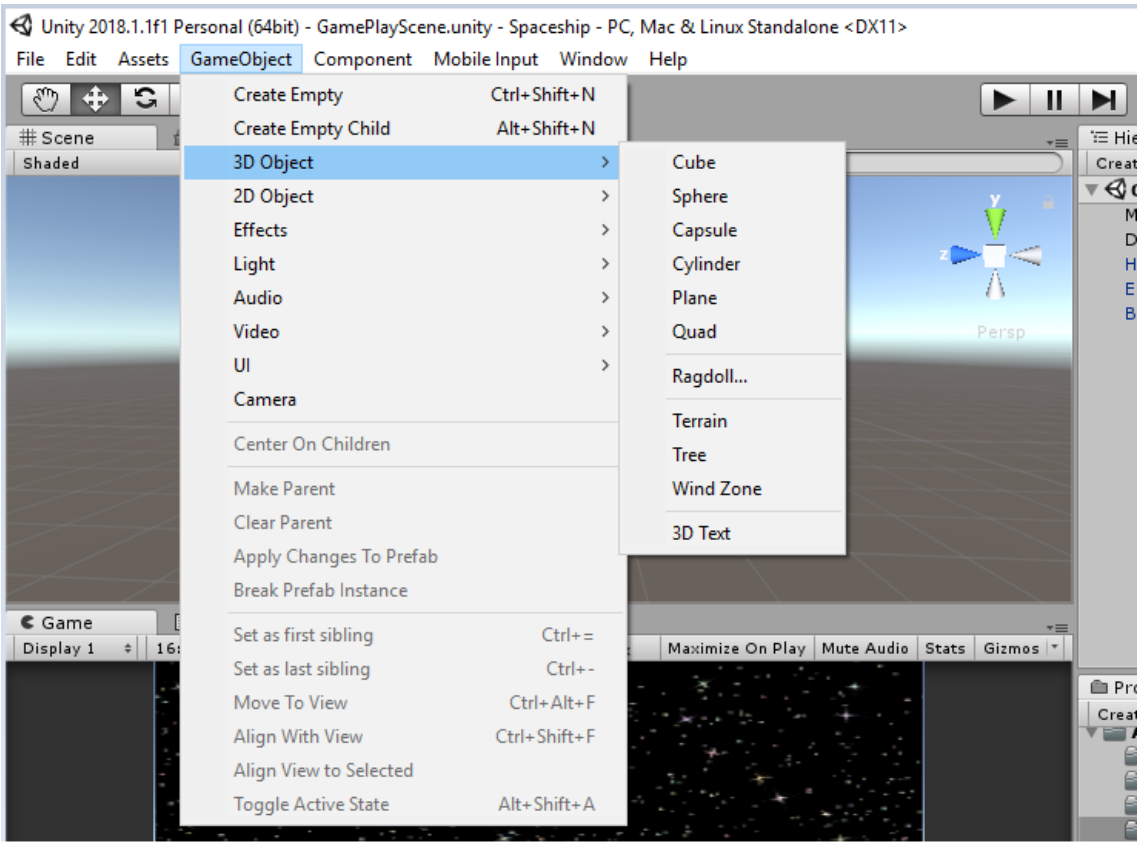
Ουσιαστικά οτιδήποτε υπάρχει μέσα στο Hierarchy είναι ένα GameObject...!

4.2.2 Πώς δημιουργώ ένα GameObject;

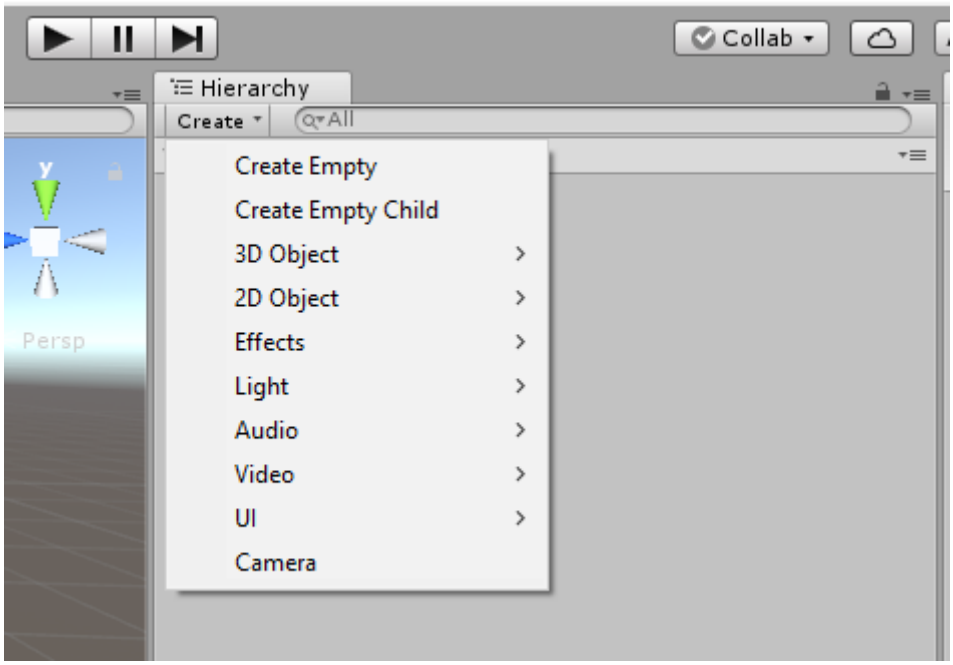
Μπορούμε πολύ εύκολα να δημιουργήσουμε ένα GO. Η παρακάτω εικόνα μας δείχνει τον τρόπο δημιουργίας.

Εικόνα 4.2.2.1.a Δημιουργία GameObject

Μπορώ να έρθω εδώ στην καρτέλα GameObjects ή ακόμα πιο γρήγορα πηγαίνοντας και πατώντας το κουμπί "Create New" στο Hierarchy (εικόνα β).



Εικόνα 4. 2.2.1.b Δημιουργία GameObject



4.2.3 Βασικοί κανόνες για τα GameObjects

Ας αρχίσουμε να αναφέρουμε κάποιους βασικούς «νόμους» ή κανόνες που ισχύουν για τα GameObjects, οι οποίοι θα μας βοηθήσουν να τα κατανοήσουμε ακόμα καλύτερα σαν έννοιες.

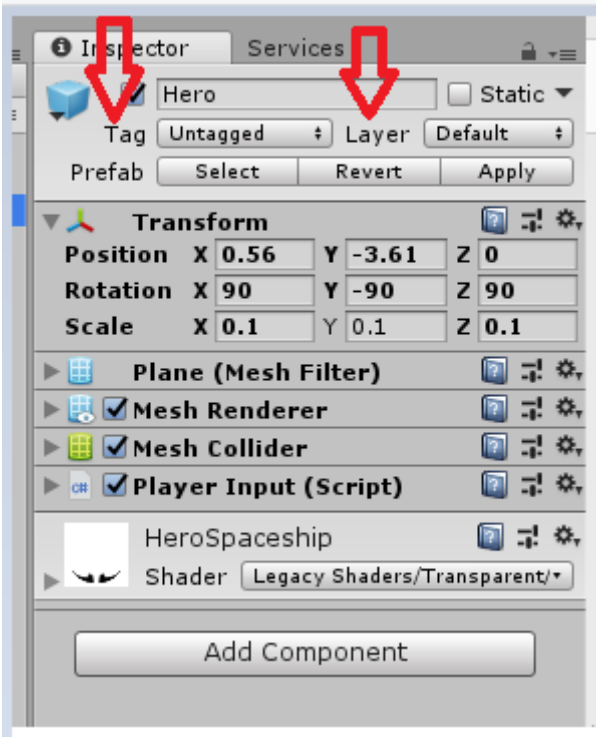
- **Ένα GameObject μπορεί να είναι ενεργοποιημένο ή απενεργοποιημένο**

Αυτό σημαίνει ότι ένα GameObject μπορεί να υπάρχει στο Scene κανονικότατα και να είναι "κρυμμένο" μέχρι να χρειαστεί να "εμφανιστεί" ή μπορεί να απενεργοποιείται προσωρινά για λόγους Gameplay. Προσοχή όμως, επειδή είναι απενεργοποιημένο δεν σημαίνει ότι δεν πιάνει χώρο στην μνήμη. Όταν είναι απενεργοποιημένο ένα GameObject είναι σαν να μην λειτουργεί κανένα από τα Components του, αλλά υπάρχει στη μνήμη. Δηλαδή, αν απενεργοποιήσω την σφαίρα, θα εξαφανιστεί από το Scene και Game View αλλά θα υπάρχει στο Hierarchy.

- **Ένα GameObject έχει ή δεν έχει Tag και ανήκει σε κάποιο Layer**

Κάθε GameObject μπορεί να έχει ένα Tag ή να είναι από μόνο του "Untagged" και να ανήκει σε κάποιο layer, όπως π.χ. το αρχικό που είναι το "Default". Τα Tags / Layers που έχει ένα GameObject υπάρχουν και αυτά στον inspector, ακριβώς κάτω από εκεί που τα ενεργοποιούμε / απενεργοποιούμε

Εικόνα 4.2.3 Το Tag και το Layer του GameObject

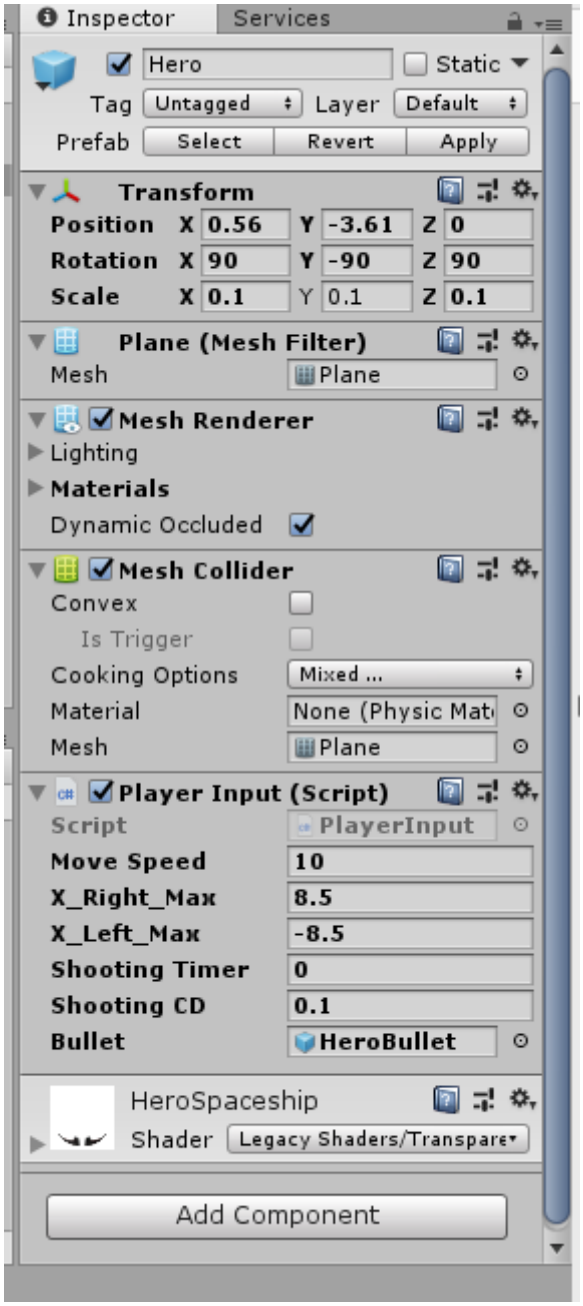


- **Ένα GameObject έχει τουλάχιστον 1 Component και μπορεί να έχει ως N components πάνω του.**

Κάθε GameObject θα μπορούσε να έχει 1 μόνο Component, δηλαδή το Transform Component, το οποίο είναι το default (βασικό, προ-επιλεγμένο) component κάθε gameobject και θα μπορούσε να έχει πολύ περισσότερα.

Για παράδειγμα, στην εικόνα 2.3.3 βλέπουμε τα 5 Components που έχει το GameObject μας. Το έξτρα κομμάτι κάτω από το “Player Input” Component είναι Material, κάτι που χρειάζεται κάθε 3D model ή γραφικό γενικά για να μπορέσει να το “ζωγραφίσει” η Unity στην οθόνη μας.

Εικόνα 2.3.3



4.3. ΤΑ ΒΑΣΙΚΑ ΣΥΣΤΑΤΙΚΑ ΚΑΘΕ GAMEOBJECT: COMPONENTS

4.3.1 Τι είναι ένα Component;

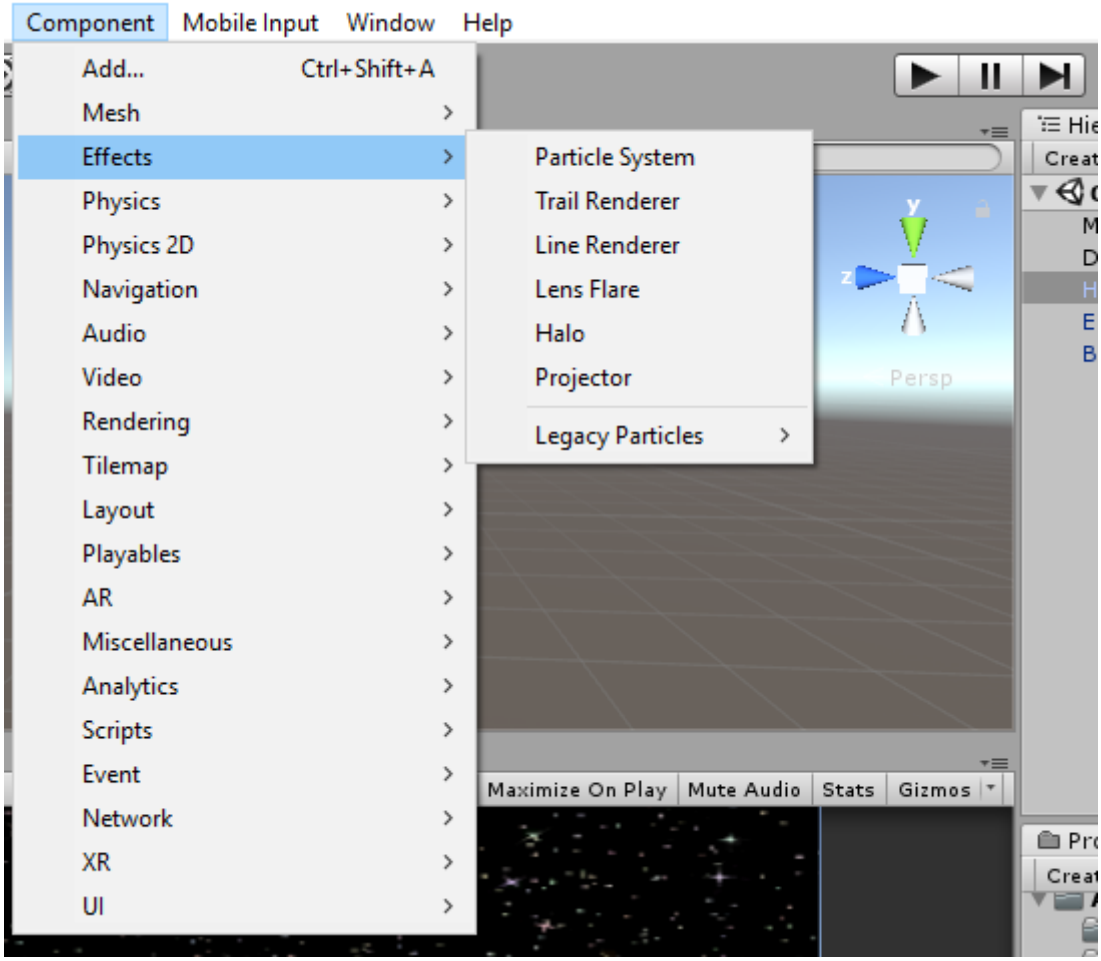
Κάθε Component συμβολίζει κάποια διαφορετική λειτουργία του GameObject. Ένα Component μπορεί να προσθέτει Physics και συγκρούσεις σε ένα GameObject, ένα άλλο να δημιουργεί ήχους, ένα άλλο να παράγει particles και ένα άλλο να του δίνει την ιδιότητα να παράγει φως! Κάθε Component μπορώ να το κατηγοριοποιήσω σε 2 κατηγορίες: Built-in της Unity, και αυτά που φτιάχνουμε εμείς δηλαδή τα Scripts. Και κάπου εδώ αξίζει να σημειωθεί πως και τα Scripts θεωρούνται Components.

4.3.2 Προσθήκη Component σε GameObject

Πάμε να δούμε, όμως, πώς προσθέτω Components σε ένα Game Object; Πολύ απλά, πρέπει πρώτα να επιλέξω το GameObject που θέλω να του προσθέσω νέο Component, π.χ. ας επιλέξουμε το Particle System. Από εκεί και ύστερα, υπάρχουν 2 τρόποι

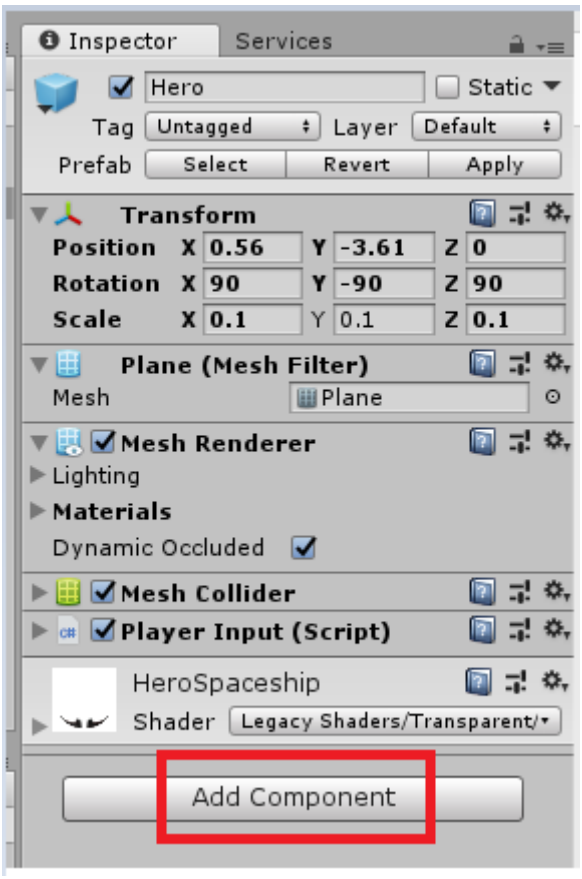
Εικόνα 4.3.2.1α Προσθήκη Component: 1ος τρόπος

Πρώτον, μπορούμε να πάμε πάνω στην Unity και να επιλέξουμε από την καρτέλα components το component που θέλουμε να του προσθέσουμε.



Υπάρχει και ένας δεύτερος τρόπος, ο οποίος ενδεχομένως να βολεύει περισσότερο, καθώς είναι ίδιος με τον πρώτο, απλώς μας παρέχεται και η δυνατότητα να κάνουμε Search! Πάμε κατευθείαν στον Inspector και πηγαίνουμε κάτω κάτω. Εκεί θα βρούμε το κουμπί “Add Component”, το πατάμε και βλέπουμε και βλέπουμε την παρακάτω εικόνα.

Εικόνα 3.2.1$^{\beta}$ Προσθήκη Component: 2^{ος} τρόπος



4.3.3 Η δομή ενός Component

Κάθε Component εμφανίζεται στον inspector για το συγκεκριμένο gameobject που έχουμε επιλέξει και περιέχει έναν συγκεκριμένο αριθμό από properties, τα οποία μπορούμε να πειράξουμε. Για παράδειγμα, ας δούμε το “Light” Component του GameObject “Directional Light” στην Εικόνα 3.3.1

Εικόνα 4.3.3.1 Παράδειγμα Light Component

Βλέπουμε εδώ ότι το Light έχει Properties τα οποία μπορούμε να πειράξουμε, όπως το Type που επηρεάζει το είδος του φωτός που θα έχουμε στο Scene, το Color, που είναι το χρώμα του φωτός, Intensity που συμβολίζει το πόσο έντονο είναι ένα φώς κ.α. Όλα μπορούμε να τα πειράξουμε αν θέλουμε. Κάθε GameObject έχει πάνω του συγκεκριμένα Components τα οποία απλά εξυπηρετούν αυτόν τον σκοπό του GameObject.

4.4. ΟΡΓΑΝΩΣΗ ΤΩΝ SCENE ΜΑΣ ΜΕ TAGS & LAYERS

4.4.1 Τι είναι τα Tags;

Όταν γράφω κώδικα, δε θέλω να κάνω έλεγχο αν με χτύπησε ένα μεγάλο ή μικρό Zombie. Με ενδιαφέρει αν ήταν Zombie.Αφλου θα επιτεθεί στον χαρακτήρα μου ούτως η άλλως. Οπότε εδώ έρχονται τα Tags! Θα πρέπει να «Tag-άρουμε» κάθε Zombie με το Tag “Zombie”. Έτσι, λοιπόν, με τα Tags μπορούμε να «μαρκάρουμε» κάποια αντικείμενα, τα οποία μπορεί να μην είναι παρόμοια μεταξύ τους, αλλά να ανήκουν όλα σε μια ευρύτερη ομάδα. Π.χ μπορεί να έχω για εχθρούς Goblins, Trolls, Ogres,

Ghouls, Dragons, κ.ά. Είναι μεν διαφορετικοί, αλλά είναι όλοι «εχθροί». Θα τους μαρκάρω λοιπόν όλους με το Tag “Enemy”, εφόσον θέλω στον κώδικα να μπορώ να τους χτυπήσω όλους με το σπαθί μου, και ας έχουν διαφορετικές άμυνες ή επιθέσεις. Δεν παύουν να είναι όλοι εχθροί και να έχουν όλοι ενέργεια (Health) κτλ.

4.4.2 Τι είναι τα Layers

Τα Layers πρόκειται για μια λειτουργία της Unity η οποία, όπως και τα Tags, χρησιμοποιείται για να ομαδοποιήσει κάποια Game Objects μεταξύ τους. Χρησιμοποιούνται κυρίως από τις κάμερες έτσι ώστε μια κάμερα να ξέρει πόσα από τα Layers να «ζωγραφίσει» στην οθόνη. Μπορεί εμείς π.χ. να έχουμε 5 Layers, αλλά να θέλουμε ο παίχτης να βλέπει μόνο 4 από αυτά στο παιχνίδι. Το 5ο Layer, για παράδειγμα, να είναι ένα βοηθητικό Layer, αποτελούμενο από GameObject - βοηθούς

(Σφαίρες, Κύβοι και οτιδήποτε άλλο δεν θέλουμε να βλέπει ο παίχτης, που θα προσφέρουν κάποια ουσιαστική βοήθεια στον σχεδιαστή) που θα βλέπει ο Game Designer όταν σχεδιάζει επίπεδα στην Unity. Επίσης, χρησιμοποιούνται από τα φώτα και ακόμα από το σύστημα φυσικής της Unity στο οποίο μπορούμε να ορίσουμε ότι π.χ. τα GameObjects που ανήκουν στο Layer 1 να μην μπορούν να αλληλοεπιδράσουν φυσικά (κρούσεις κτλ.) με το Layer 2 αλλά να μπορούν με το 3.

4.5. ΟΡΙΣΜΟΣ ΠΡΟΤΥΠΩΝ ΚΑΙ ΕΠΑΝΑΧΡΗΣΙΜΟΠΟΙΗΣΗ ΜΕΣΩ PREFABS

4.5.1 Τι είναι τα Prefabs

Το Prefab είναι μια δομή της Unity, η οποία μας επιτρέπει να φτιάχνουμε ένα GameObject, να το σχεδιάζουμε όπως θέλουμε και να του περνάμε ό,τι Components θέλουμε και με ό,τι τιμές θέλουμε στα Properties τους.

Έπειτα, η Unity αποθηκεύει αυτή τη δομή (ή template αν θέλετε) ως έχει στα Assets του παιχνιδιού μας και μπορούμε να χρησιμοποιήσουμε αυτή τη δομή για να φτιάξουμε όσα GameObjects (που μοιάζουν στο αρχέτυπο) θέλουμε! Επίσης, μας επιτρέπει να αλλάζουμε το αρχέτυπο μία φορά και να αλλάζουν όλα τα δημιουργημένα GameObjects μέσα στο scene!

Για να το πούμε όσο πιο απλά γίνεται: Θέλουμε να φτιάξουμε πάρα πολλά αντικείμενα με συγκεκριμένη δομή από GameObjects (Ιεραρχία/Components/Values); Με τα prefabs μπορούμε.

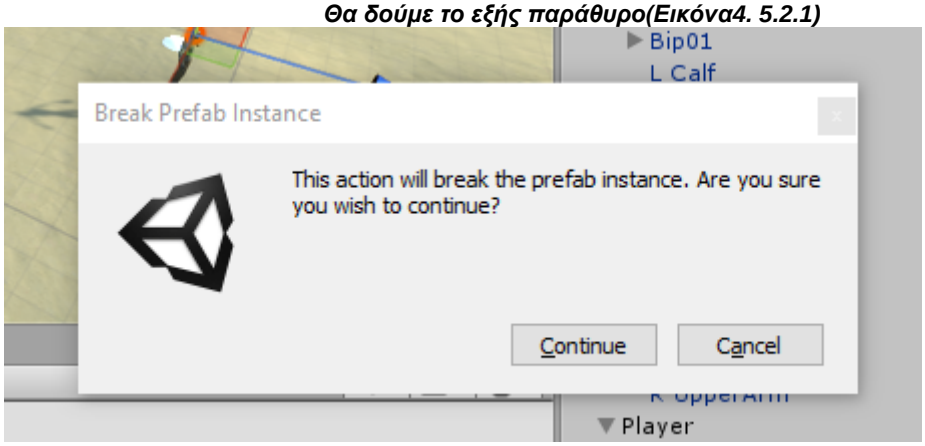
Φτιάχνουμε το αντικείμενο μας μία φορά στην Unity, το κάνουμε Prefab, και από εκεί χρησιμοποιούμε το Prefab για να φτιάξουμε όλα τα υπόλοιπα!

4.5.2 Οι επιλογές Apply, Select και Revert

Έχοντας το Prefab στα Assets μας, μας είναι πολύ εύκολο να φτιάξουμε αρκετά GameObjects τύπου “PointLights”, αρκεί να τα κάνουμε Drag and Drop από τα Assets στο Scene ή στο Hierarchy.

Τι συμβαίνει όμως εάν κατά λάθος σβήσουμε μέσα στο Hierarchy ένα κομμάτι που ανήκει σε Prefab;

Θα δούμε το εξής παράθυρο(Εικόνα4. 5.2.1)



Πατώντας continue λέμε πως “σπάει” το Prefab, και θα δούμε τα GameObjects που έχουν απομείνει χωρίς αυτό που μόλις σβήσαμε αλλάζουν χρώμα και γίνονται άσπρα ξανά, καθώς δεν υπακούνε πια σε κανένα αρχέτυπο Prefab.

Στην περίπτωση που θέλουμε να κάνουμε μία αλλαγή σε ένα GameObject και θέλουμε

αυτή να γίνει σε Prefab Level (όλα τα Prefabs δηλαδή), ένας τρόπος για να το επιτύχουμε αυτό είναι να πάμε κατευθείαν στο GameObject, να κάνουμε την αλλαγή που θέλουμε και να πατήσουμε το κουμπάκι “Apply”, που πλέον βρίσκεται στον Inspector μόνο αν το GameObject που έχουμε επιλέξει υπακούει κάποιο αρχέτυπο Prefab.

4.6. Η ΜΗΧΑΝΗ ΓΡΑΦΙΚΩΝ ΤΗΣ UNITY3D

4.6.1 Rendering & Drawing

Με τον όρο Render εννοούμε την πράξη κατά την οποία «εμφανίζονται» γραφικά στην οθόνη. Αν κάτι έχει Renderer, τότε αυτό σημαίνει ότι του δίνεται η δυνατότητα να εμφανιστεί το γραφικό του στην οθόνη, είτε είναι 2D είτε είναι 3D.

Το Rendering έχει κάποια διαφορά από το Drawing! Με τον όρο Draw ουσιαστικά αναφερόμαστε στο πρακτικό «ζωγράφισμα» γραφικών στην οθόνη. Η Unity παίρνει τα assets που θα πρέπει να ζωγραφίσει από το Assets folder, είτε είναι 3D είτε 2D και τα ζωγραφίζει στην οθόνη, δηλαδή καταγράφει την θέση, το σχήμα, την δομή, τον τρόπο αλληλεπίδρασης του περιβάλλοντος και τον τρόπο με τον οποίο θα εμφανιστεί ένα γραφικό στην οθόνη, αφού τα τοποθετήσει!

4.6.2 Draw Calls

Τα draw calls είναι ουσιαστικά «μέθοδοι που αναλαμβάνουν να ζωγραφίσουν γραφικά στην οθόνη».

Η Unity όταν είναι να ζωγραφίσει γραφικά κάνει ένα draw call για κάθε object που έχει γραφικό στην οθόνη αυτή τη στιγμή μέσω του graphics API, το οποίο μπορεί να είναι είτε OpenGL είτε Direct3D.

Όλα τα drawcalls γίνονται ΑΝΑ ΚΑΡΕ. Οπότε όπως θα καταλαβαίνετε, αυτό δημιουργεί αρκετή επιβάρυνση, όσον αφορά την CPU, τον επεξεργαστή μας δηλαδή, άρα και το πόσο πιο αργά ή γρήγορα θα τρέχει το παιχνίδι μας. Θυμηθείτε σε αυτό το σημείο και κρατήστε το εξής: κάθε γραφικό στην οθόνη μπορεί να είναι πολύπλοκο ή απλό. Αυτό επηρεάζεται από αρκετούς παράγοντες, π.χ. αν είναι 2D ή 3D γραφικό.

Το 3D αμέσως σημαίνει ότι θα πρέπει να υπάρχειπληροφορία βάθους, 3ου άξονα δηλαδή, οπότε αμέσως έχουμε πληροφορίες για μια επιπλέον μεταβλητή: -50% ταχύτητα.

Άλλος παράγοντας είναι το φως. Μπορεί ένα αντικείμενο να φωτίζεται ή όχι. Το φως επηρεάζει κατά πάρα πολύ το πώς γίνεται render ένα γραφικό, καθώς ανάλογα με τη δομή του δέχεται και διαφορετική πληροφορία φωτός. Επίσης, ένα γραφικό 2D μπορεί να είναι μικρό σε μέγεθος (16 x 16) ή τεράστιο (4096 x 4096), οπότε και το μέγεθος επηρεάζει την ταχύτητα.

Τέλος, ακόμα και τα χρώματα, τα DPI καθώς και η χρωματική παλέτα η ίδια που χρησιμοποιήθηκε για να φτιαχτεί το γραφικό παίζει ρόλο στην πληροφορία που θα χρειαστεί να υπολογίσει η Unity πριν προχωρήσει στην διαδικασία να ζωγραφίσει κάτι για να το πάρει ο renderer! Ανάλογα λοιπόν με το πόσο εύκολα ζωγραφίζονται τα γραφικά στην οθόνη επηρεάζεται και η ταχύτητα με την οποία θα ζωγραφίζονται, κάτι που μας κάνει το παιχνίδι πιο γρήγορο.

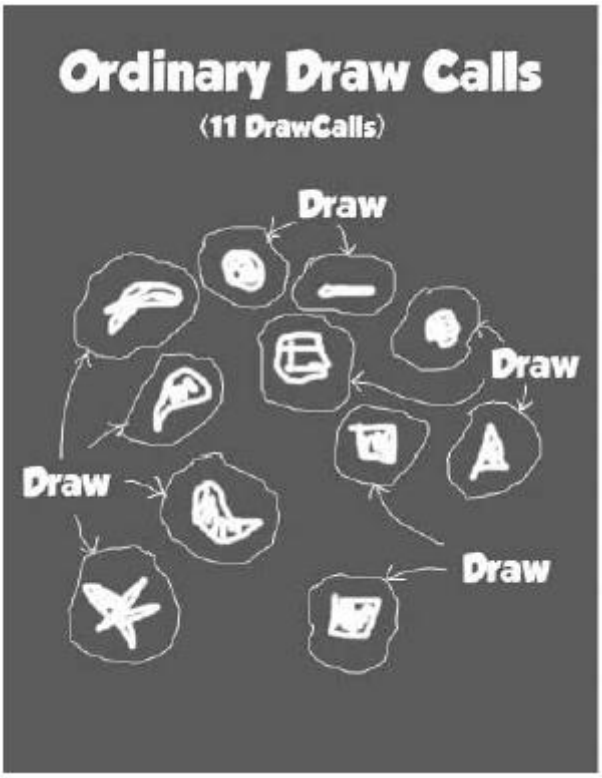
Ευτυχώς όμως η Unity έχει built-in σύστημα, το οποίο βοηθάει αυτό και μόνο ξεκάθαρα και ονομάζεται **Batching**!

4.6.3 Batching

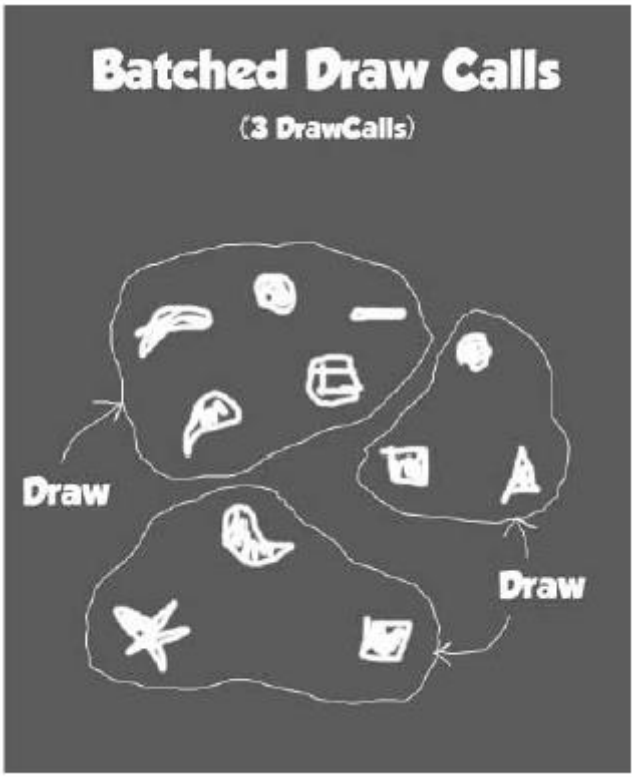
Πρόκειται για το σύστημα της Unity το οποίο αναλαμβάνει με έξυπνο τρόπο να ομαδοποιήσει γραφικά τα οποία έχουν συγκεκριμένη “ομαδοποιήσιμη” συμπεριφορά και να τα ζωγραφίσει όλα μαζί με ένα και μόνο ένα draw call!

Όσα περισσότερα gameobjects καταφέρει η Unity να ομαδοποιήσει τόσο πιο ισχυρό είναι το batching καθώς και η απόδοση του συστήματος στο τελικό παιχνίδι.

Εικόνα 4.6.3.1 Batch Draw Calls

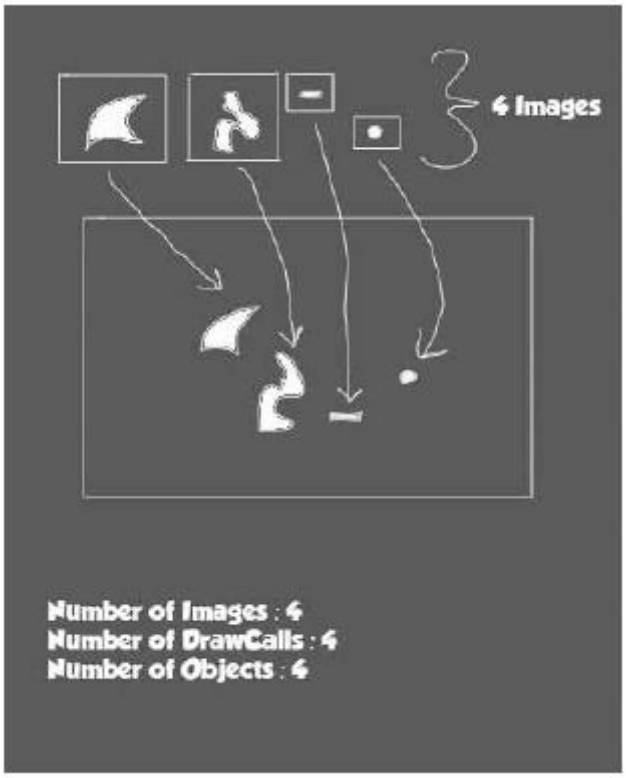


Εικόνα 4.6.3.2 Batched Draw Calls



4.6.4 Image Atlasing & Sprite Sheets

Μια άλλη καλή μέθοδος για να γλιτώνουμε χώρο στον GPU, καθώς και ταχύτητα είναι να μην χρησιμοποιούμε 4 εικόνες για 4 διαφορετικά αντικείμενα, αλλά να χρησιμοποιήσουμε ένα Image Atlas, δηλαδή μια μεγάλη εικόνα (σε δύναμη του 2 πάντα), που περιέχει μέσα της όλες τις εικόνες που θα χρειαστούμε για να κάνουμε render. Αυτή είναι μια μέθοδος που κάνουν κατά κόρον οι animators όταν θέλουνε να δημιουργήσουνε animations σε χαρακτήρες. Φυσικά, θα μπορούσαμε να έχουμε 10 εικόνες αλλά αυτό σημαίνει 10 διαφορετικά αντικείμενα στην μνήμη, ενώ θα μπορούσαμε να έχουμε ένα!


4 Images
Number of Images : 4
Number of DrawCalls : 4
Number of Objects : 4

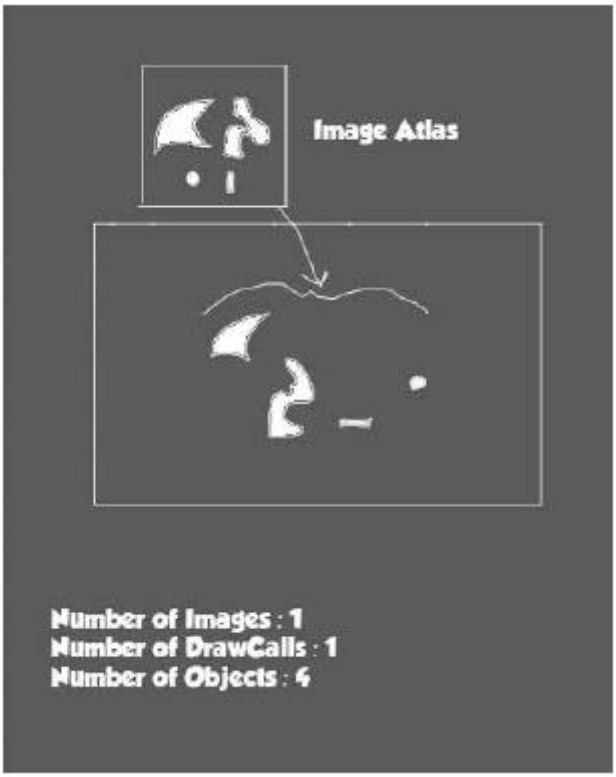


Image Atlas
Number of Images : 1
Number of DrawCalls : 1
Number of Objects : 4

4.6.5 Sprites

Με τον όρο Sprites αναφερόμαστε σε 2D εικόνες, οι οποίες συνήθως προέρχονται από κάποιο Spritesheet και χρησιμοποιούνται κατά κόρον σε 2D παιχνίδια, καθώς είναι πολύ απλά στην δομή και την χρηστικότητά τους, και δεν περιέχουν τις επιπλέον πληροφορίες που περιέχουν οι υφές (Textures).

4.6.6 Textures (Υφές)

Με τον όρο Texture εννοούμε την υφή που θα έχει ένα 3d μοντέλο ή την εικόνα που θα αντιπροσωπεύει τον ήρωά μας στο παιχνίδι. Από την στιγμή που η Unity απέκτησε 2D toolset, ο κόσμος σταμάτησε να χρησιμοποιεί textures για τα 2d αντικείμενά του και το γύρισε σε Sprites.

Με ένα texture μπορούμε να αποθηκεύσουμε πολλή περισσότερη

πληροφορία απ' ό,τι ένα sprite και να τα χρησιμοποιήσουμε πάνω σε 3D επιφάνειες, με σκοπό να δώσουμε την ψευδαίσθηση της υφής. Επιπλέον, τα textures μπορούμε να τα χρησιμοποιήσουμε σαν tiles πάνω σε Unity Terrain, ή ακόμα και σε 3d models και να δώσουμε εφέ, τα οποία είναι πολύ ρεαλιστικά για παιχνίδια

4.6.7 Shaders

Όλο το rendering της Unity γίνεται με την βοήθεια των shaders. Οι shaders δεν είναι παρά μικρά script, τα οποία έχουν γραφτεί σε HLSL (γλώσσα προγραμματισμού shaders), τα οποία πρακτικά παραμετροποιούν το hardware μας (δηλαδή την GPU μας) στο πώς θα γίνονται τελικά render ορισμένα αντικείμενα

Πρακτικά λοιπόν με τους shaders αυτό που κάνουμε είναι να εμφανίζουμε το ίδιο texture αλλά με διαφορετικές πληροφορίες, δηλαδή διαφορετική εμφάνιση. Υπάρχουν shaders που προσδίδουν παραπάνω στρώμα υφής στα ήδη υπάρχοντα textures, κάνοντας τα να φαίνονται ανάγλυφα, υπάρχουν shaders που προσδίδουν γυαλιστερή υφή σε ένα texture και άλλα πάλι κόβουν εντελώς την πληροφορία του φωτός, δίνοντας τους την δυνατότητα να είναι «αυτόφωτα» και να φαίνονται σαν να είναι σε διαφορετικό Unity Layer.

4.6.8 Material (Υλικό)

Το Material, το οποίο ουσιαστικά πρόκειται για τον συνδυασμό Texture με Shader που θέλουμε να εφαρμοστεί πάνω σε ένα GameObject. Καλώς ή κακώς, στην Unity, κάθε gameobject που εμφανίζεται στην οθόνη έχει είτε ένα Sprite (και εκεί τελειώνει η υπόθεση αν μιλάμε για 2D) είτε ένα Material.

Το Material λοιπόν καθορίζει ποιο texture θα εφαρμοστεί πάνω στο 3d μοντέλο του οποίου πειράζουμε τον τρόπο που θα κάνει render τώρα, και επίσης, με ποιον Shader.

Επίσης, με ένα material μπορούμε να προσαρμόσουμε το offset που θα έχει ένα texture πάνω του, αν θα είναι tiled, αν θα το μεγεθύνουμε σαν texture ή μικρύνουμε, έτσι ώστε πχ ένα texture να χωράει 4 φορές πάνω σε μια επιφάνεια κτλ.

Μπορείτε να φανταστείτε ένα material απλά να είναι Texture + Shader μαζί με λίγη παραπάνω πληροφορία όσον αφορά το Tiling. Συγκεκριμένα, το Tiling αφορά σε ποιο σημείο πάνω σε μια εικόνα θα μπορέσει να παρουσιαστεί αναδίπλωση ως προς τον άξονα X ή Y.

Με τον όρο αναδίπλωση εννοούμε κατά πόσο θα θέλαμε να έχει προχωρήσει το 1ο pixel της εικόνας (το πάνω-αριστερά) κατά τον X και τον Y άξονα εντός της επιφάνειας στην οποία υλοποιείται κάποιο Texture; Τα pixels που «χάθηκαν» κατά την αναδίπλωση έχουν αναδιπλωθεί στις αντίστοιχες πλευρές που σημειώθηκε η αναδίπλωση εξού και ο όρος «αναδίπλωση» ή αλλιώς Offset. Το Offset χωρίζεται σε offset στον X και offset στον Y άξονα. Αθροιστικά τα 2 offsets αυτά αποτελούν το Tiling

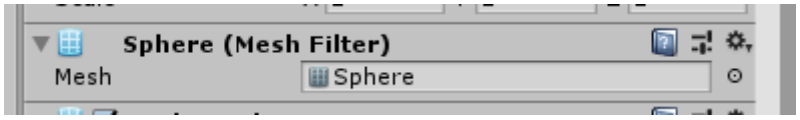
4.7.ΜΕΛΕΤΗ ΤΩΝ MESH FILTER ΚΑΙ MESH RENDERER COMPONENTS

4.7.1 Mesh Filter Component

Με το MeshFilter Component ουσιαστικά λέμε στην Unity τι ακριβώς θα γίνει rendered. Δηλαδή ποιό “Mesh” θα φορτώσει και θα εμφανίσει στον χρήστη. Γι’ αυτό και το λέμε “MESH Filter”. Το Mesh είναι πρακτικά το 3d μοντέλο μας. Κάθε αντικείμενο στον 3D χώρο καλό είναι να το γνωρίζουμε σαν “Mesh” και όχι σαν «Μοντέλο», καθώς η Unity έχει διαφορετική ορολογία στα Models, τα οποία αφορούν humanoids (ανθρωποειδή), monsters (τέρατα), πρακτικά οτιδήποτε έχει γίνει σε 3d package (όπως 3Ds Max / Maya) και έχει animations (Κινήσεις) τα οποία πατάνε σε κάποιο Rig (Σκελετό).

Η μόνη ιδιότητα που θα δούμε σε ένα Mesh Filter είναι το ίδιο το Mesh το οποίο περιμένουμε να γίνει rendered. Αν πατήσουμε πάνω στο κυκλάκι που έχει στα δεξιά μας εμφανίζει όλα τα Meshes που μπορούμε να επιλέξουμε για να γίνουν render. «Το Mesh Filter καθορίζει το 3D αντικείμενο που θα γίνει rendered».

Εικόνα 4.7.1 Meshes για το Mesh του Filter



4.7.2 Mesh Renderer

Το Mesh Renderer Component είναι το Component υπεύθυνο για το rendering του Mesh στο παιχνίδι μας. Πάμε να δούμε από τι ιδιότητες αποτελείται, όπως φαίνεται στην εικόνα 7.2.

Εικόνα 4.7.2 Mesh Renderer Properties

Όπως βλέπουμε, είναι ένα Component που μπορούμε να το απενεργοποιήσουμε και ενεργοποιήσουμε. Όποτε δεν είναι ενεργός ο Mesh Renderer, τότε δεν γίνεται Render το

GameObject. ΠΡΟΣΟΧΗ, όμως, γιατί το Game Object και οι Colliders του είναι ενεργοί,

απλά δεν φαίνονται στην οθόνη μας!

Η ιδιότητα Materials, μας ενημερώνει για το πόσα Materials είναι φορτωμένα πάνω στο GameObject μας την ίδια στιγμή! Μπορούμε να του αλλάξουμε το Size και να του δώσουμε τιμή 2. Τότε, θα μπορούμε να του περάσουμε ένα δεύτερο material και να γίνονται render και τα 2 materials την ίδια στιγμή με διαφορετικούς Shaders και διαφορετικές ρυθμίσεις στο tiling των texture τους!

4.8 Θεωρία Μηχανής Φυσικής Της UNITY3D

Η Unity έχει δικιά της Physics Engine built-in, οπότε δεν θα χρειαστεί να γράψουμε κάποιου είδους

κώδικα για να μπορέσουμε να φτιάξουμε τα Physics σε ένα game. Μάλιστα,

χρησιμοποιεί την **NVIDIA PhysX Physics** engine για **3D** και την **Box2D** για 2D physics. Με τα physics μπορούμε σε ένα game να προσθέσουμε την έννοια της δύναμης, της βαρύτητας, της ορμής, της δύναμης του αέρα καθώς και άλλα διάφορα εφέ που προσδίδουν ρεαλισμό στο παιχνίδι μας.

4.8.1 Collision (ή αλλιώς σύγκρουση)

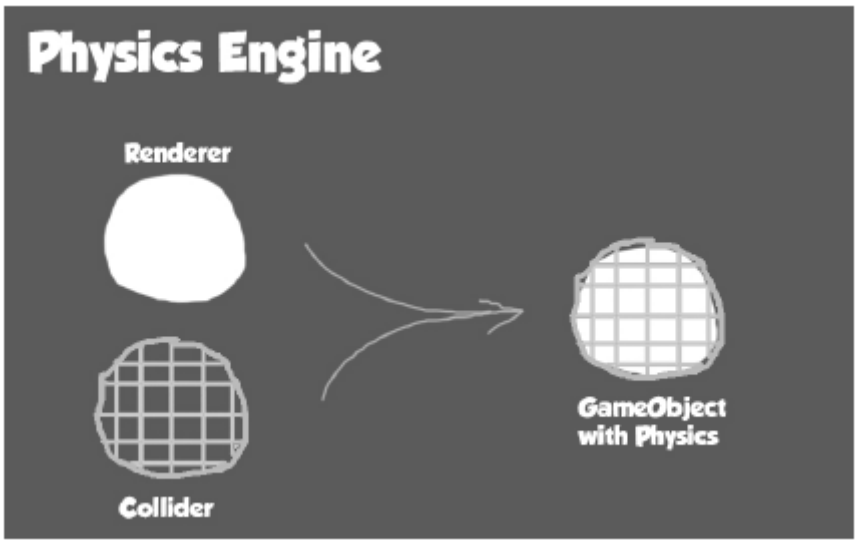
Θα λέμε ότι έχουμε Collision στην Unity, δηλαδή σύγκρουση μεταξύ 2 αντικειμένων ή απλά «επαφή» 2 αντικειμένων, αν και μόνο αν οι Colliders τους έχουν επαφή μεταξύ τους.

4.8.2 Colliders (ή αλλιώς αισθητήρες σύγκρουσης)

Χωρίς Colliders δεν υπάρχουν physics. Ως Collider **ορίζεται ένα εξωτερικό Mesh το οποίο συνήθως περιβάλλει το δικό μας Mesh, και χρησιμοποιείται καθαρά ως Mesh που ελέγχει για Collisions**. Ας σκερφτολυμε ένα στρώμα εξωτερικό που καλύπτει ολοκληρωτικά το 3D μοντέλο μας ή το GameObject μας σε σημεία που θα θέλαμε εμείς να έχουμε collision.

Αυτό το στρώμα συγκρίνει η Unity αν έχει αγγίξει ή είναι μέσα σε άλλα στρώματα και έτσι υπολογίζει τα Collisions, δηλαδή τις συγκρούσεις. Ας δούμε την παρακάτω εικόνα.

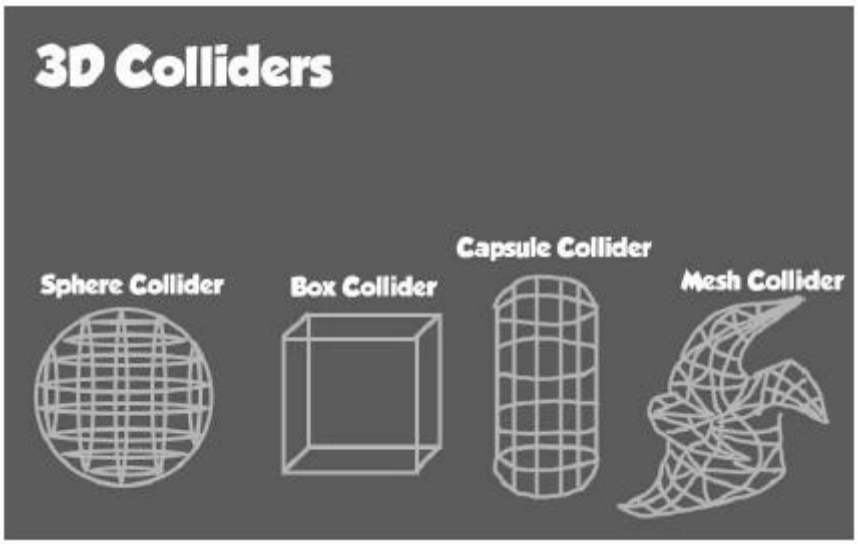
Εικόνα 4.8.2 Ο collider σε σχέση με το GameObject



Ας σκεφτούμε κάτι άλλο. Θα μπορούσε π.χ. ένα GameObject να μην έχει Collider. Αυτό το object δεν θα το συμπεριλάμβανε καν η Physics Engine της Unity στους υπολογισμούς της και θα μπορούσαν όλα τα αντικείμενα να περάσουν από μέσα του, λες και αυτό είναι φάντασμα.

Βασικά, είναι μια καλή ιδέα να μην βάζουμε colliders στα φαντάσματα μας στα παιχνίδια μας, εάν φτιάχνουμε ένα Spooky Ghost Story παιχνίδι. Αυτό με την προϋπόθεση ότι το φάντασμα δεν αλληλεπιδρά με το περιβάλλον μέσω physics. Επίσης, υπάρχουν πολλών είδη Colliders, και για 3D και για 2D, ανάλογα την χρήση που θέλουμε να κάνουμε.

Εικόνα 4.8.2β Τα διάφορα είδη colliders



Sphere Collider: Πρόκειται για μια σφαίρα που συνήθως καλύπτει σφαιρικά μοντέλα και GameObjects ή σφαιρικά μέρη των GameObjects.

Box Collider: Όπως και με το Sphere, έτσι και ο Box Collider πρόκειται για έναν κύβο που καλύπτει άλλα αντικείμενα.

Capsule Collider: Ο καλύτερος collider όσον αφορά humanoid characters. Είναι όπως το sphere collider, αλλά μπορεί να γίνει stretched ως προς τον Υ και να καλύψει ένα συγκεκριμένο ύψος, ανάλογα το αντικείμενο που θέλουμε να δράσει ως collider του.

Mesh Collider: Ο πιο "ακριβός" Collider από πλευράς υπολογιστικών πόρων, καθώς αναπαράγει το ίδιο μοντέλο ως Collider με το μοντέλο του οποίου δρα ως Collider έτσι ώστε να είναι ένα τέλειο Collider που καλύπτει κάθε mesh και surface του μοντέλου. Καλές πρακτικές αποφεύγουν την χρήση αυτού του Collider, καθώς είναι πολύ ακριβός (αφού πρακτικά είναι σαν να έχουμε το μοντέλο 2 φορές στην οθόνη και κάθε polygon του να υπολογίζεται για collisions), και γονατίζει" κυριολεκτικά το μηχάνημά μας λόγω των πολλών υπολογισμών.

Οι καλές πρακτικές γενικά λένε να χρησιμοποιούμε όσο το δυνατό πιο απλούς Colliders για να χαρακτηρίσουμε τα Collisions των αντικειμένων μας. Κάθε GameObject μπορεί να έχει παραπάνω από έναν Collider γι' αυτόν ακριβώς τον λόγο! Απλώς μην ξεχνάμε: πιο πολλοί Colliders και πιο πολύπλοκοι Colliders = Μεγαλύτερη επεξεργαστική ισχύ για τα Physics.

4.8.3 Rigidbody

Πάμε να δούμε μια έννοια πολύ βασική για τα Physics των game μας. Το Rigidbody Component. Πρόκειται για ένα Component που ουσιαστικά λέει στην Unity «υπολόγισε με ως Game Object στα Physics». Πρακτικά με αυτό εννοούμε ότι ο Collider δεν είναι αρκετός για να υπολογίζονται Physics. Ο Collider το μόνο που κάνει είναι να παρέχει το μέσο για να γίνουν physics, αλλά δεν ενεργοποιεί το ίδιο το Physics engine.

Οπότε όσα Game Objects θέλουμε να υπολογίζονται στα Physics και να μπορούμε να γράψουμε κώδικα για τα Collisions που κάνουν στον 3d χώρο θα πρέπει να έχουν αυτό το Component (Rigidbody), το οποίο τους επιτρέπει να υπολογίζονται στα collisions του Physics engine.

4.8.4 Τα είδη των συγκρούσεων (Collision Types)

Όταν έχουμε Collision μπορούμε να έχουμε 2 είδη collision έτσι όπως έχουν φτιαχτεί στην Unity. Υπάρχει το καθαρόαιμο Collision το οποίο ουσιαστικά έχει να κάνει με την φυσική κρούση/επαφή 2 αντικειμένων με Colliders και RigidBodies.

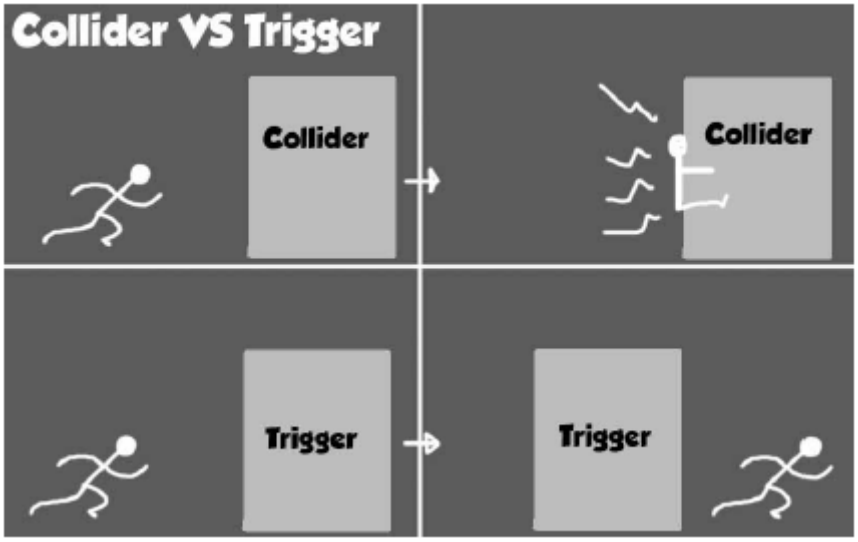
Επίσης, υπάρχει ένας ακόμα τύπος Collider που ονομάζεται Trigger. Ουσιαστικά πρόκειται πάλι για Collision, με την διαφορά ότι δεν υπάρχει κρούση. Αυτό που εννοούμε εδώ είναι ότι μπορεί ένα Game Object να μπει μέσα στον Collider και να βγει από την άλλη πλευρά του.

Αυτού του είδους ο Collider μας βοηθάει να σχεδιάσουμε και υλοποιήσουμε διάφορα εφέ όπως πχ δηλητηριασμένο δωμάτιο, όπου όσο είσαι μέσα στο δωμάτιο δέχεσαι Damage αλλά δεν έχεις κάποιου είδους σύγκρουση.

Επίσης, μπορεί να χρησιμοποιηθεί σαν «αισθητήρας» για κάποιο Scripted event ή Cinematic. Οι Triggers λοιπόν αν και δεν παρέχουν κρούση, παρέχουν τέτοιου είδους Collisions, τα οποία βοηθάνε και λύνουν άλλης φύσης προβλήματα.

Για δικιά μας ευκολία μπορούμε να θεωρήσουμε ότι υπάρχουν τα απλά Collisions μεταξύ 2 σωμάτων (κρούση) και τα Trigger Collisions μεταξύ 2 σωμάτων (κάποιο να έχει μπει μέσα στο άλλο χωρίς να υπάρχει φυσική επαφή). Λεπτομέρειες για αυτό στην παρακάτω εικόνα.

Εικόνα 4.8.4 Collider και Trigger



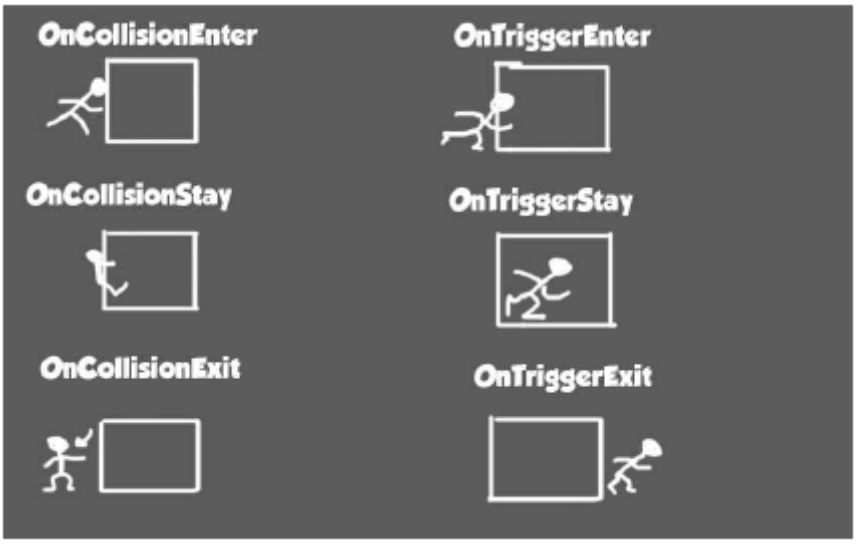
4.8.5 Ο κώδικας στην Unity περί Physics

Στην Unity τα Physics υπολογίζονται κανονικά μέσα στο Cycle των καρέ που τρέχει με συγκεκριμένα FPS. Υπάρχει μια μέθοδος, **η FixedUpdate( ),** η οποία όπως και η Update τρέχει σε κάθε καρέ, καμία φορά όμως και 2 και 3 φορές σε κάθε καρέ, ανάλογα με τον όγκο υπολογισμών που έχει να κάνει.

Ονομάζεται FixedUpdate, τρέχει ανά πολύ συγκεκριμένα προκαθορισμένα χρονικά διαστήματα σε όλες τις συσκευές και τρέχει πάντα πριν την Update στην ροή εκτέλεσης μεθόδων. Είναι υπεύθυνη για όλα τα Collisions που γίνονται στο game και είναι η συγκεκριμένη μέθοδος που κάνει τους ελέγχους για τα Collisions που πιθανώς να έχουμε γράψει στα Scripts μας.

Ας δούμε ποιες είναι οι κύριες εντολές-μέθοδοι που μπορούμε να γράψουμε για να υλοποιήσουμε Physics.

Εικόνα 8.5 Οι 3 εκδοχές μεθόδων για απλό και Trigger Collision



Ουσιαστικά στην ίδια την Unity, εάν μέσα σε οποιοδήποτε script γράψουμε π.χ.

```
Void OnCollisionEnter (…)
{
        Some Code here…
}
```

αναφερόμαστε αμέσως στα Collisions που θα συμβαίνουν όταν ένα αντικείμενο έρχεται σε επαφή με ένα άλλο, και θα τρέξει μια φορά.

Πρακτικά τρέχει πάντα όταν γίνεται η επαφή. Το **OnCollisionStay( ),** από την άλλη, θα τρέχει σε κάθε καρέ, εφόσον υπάρχει Collision (επαφή) σε κάθε καρέ.

Το **Enter** και **Exit** αφορούν το καρέ στο οποίο γίνεται έξοδος και είσοδος στο Collision, δηλαδή μια φορά. Αυτό σημαίνει πως εάν σταματήσει να υπάρχει επαφή, θα τρέξει η **OnCollisionExit().**

4.8.6 ΜΕΛΕΤΗ ΤΩΝ ΔΙΑΦΟΡΩΝ COLLIDER ΚΑΙ RIGIDBODY COMPONENTS

Εδώ θα αναφερθούμε επιγραμματικά στους διάφορους colliders που μπορούν να έχουν πάνω τους τα Game Objects

4.8.6.1 Sphere Collider

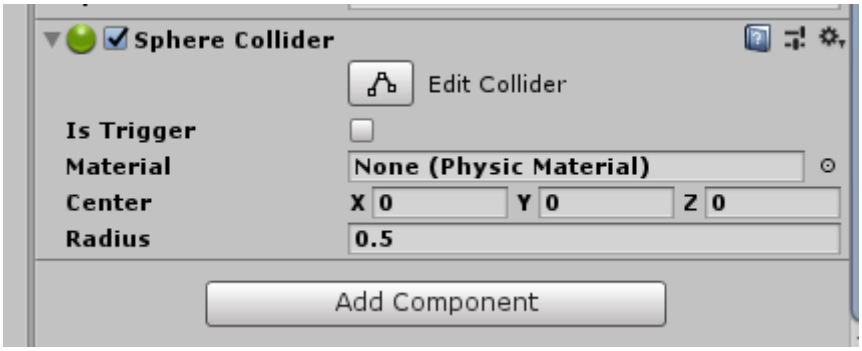
Έχουμε τη δυνατότητα να πειράξουμε τον Collider, αν πατήσουμε πάνω στο "**Edit Collider**" button μας δίνεται η δυνατότητα να "παίξουμε" με το σχήμα των Colliders. Αν πλησιάσουμε αρκετά κοντά στον Collider, μπορούμε να δούμε μερικές τελείες, οι οποίες αν πατήσουμε πάνω τους και σύρουμε το ποντίκι, τότε θα αλλάξει σχήμα ο Collider.

Η ιδιότητα **IsTrigger** αφορά το αν ο Collider αυτός θα είναι Trigger ή κανονικός Collider).

Η ιδιότητα **Material** δέχεται ένα Physic Material (όχι δηλαδή τα Materials που γνωρίζουμε!): Τα Physic Materials ουσιαστικά είναι materials που αφορούν την συμπεριφορά ενός αντικειμένου με το περιβάλλον, δηλαδή να έχει συμπεριφορά ξύλου, μετάλλου, πάγου, λάστιχου κ.λ.π. Αφορά δηλαδή ιδιότητες όπως αναπήδηση στις επιφάνειες (bounciness), τριβή (Linear Drag) και την γωνιακή τριβή (Angular Drag) που θα έχουν οι Colliders.

Οι ιδιότητες **Center** και **Radius** αφορούν τον Sphere Collider συγκεκριμένα και είναι πρακτικά η θέση του κέντρου της σφαίρας στον 3D χώρο σύμφωνα με Local Coordinates (θυμηθείτε την 1η ενότητα του μαθήματος και το Radius είναι η ακτίνα της σφαίρας, πόσο μεγάλη θα είναι.

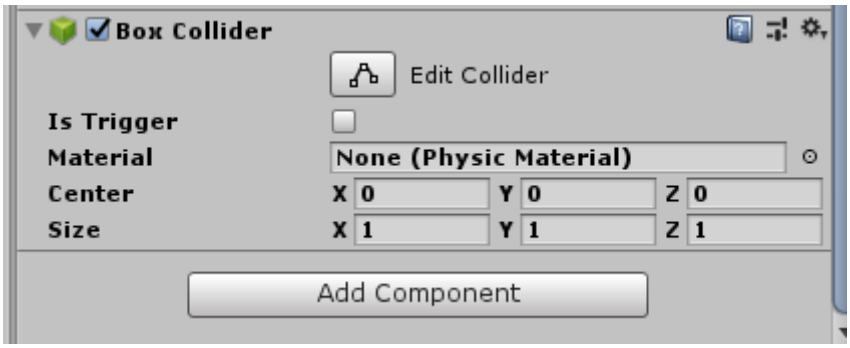
Εικόνα 8.6.1 Sphere Collider Component



4.8.6.2 Box Collider

Center: ομοίως με το Sphere
Size: πρόκειται για το μέγεθος του κύβου (scale) ως προς τους 3 άξονες.

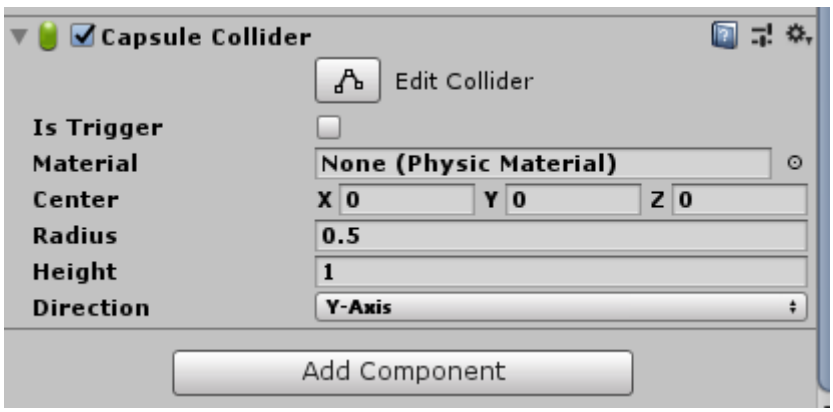
Εικόνα 4.8.6.2 Sphere Collider Component



4.8.6.3 Capsule Collider

Capsule Collider ενδείκνυται για ανθρωποειδή μοντέλα.
Το **Height** αναφέρεται στο πόσο ψηλό είναι το Capsule.
Το **Direction** αναφέρεται στον άξονα στον οποίο θα δέχεται ύψος το Capsule.

Εικόνα 4.8.6.3 Capsule Collider Component



4.8.6.4 Mesh Collider

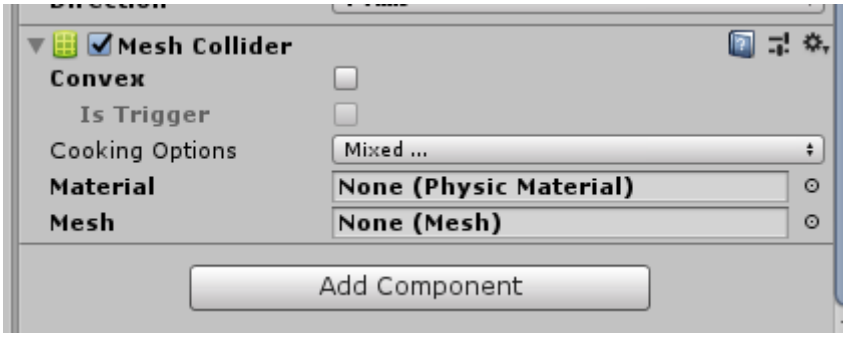
Ο Mesh Collider, σε αντίθεση με τα άλλα Colliders, μόνο το Material και το is Trigger έχει κρατήσει.

Convex: Αν αυτή η ιδιότητα είναι ενεργοποιημένη, τότε αυτός ο Mesh Collider θα κάνει Collisions με άλλα Mesh Colliders.

Smooth Sphere Collisions: Όταν είναι ενεργοποιημένη και αυτή η ιδιότητα, τότε έχουμε smoothing εφέ όταν έχουμε επαφή με λείο έδαφος (smooth terrain), π.χ. για να κάνουμε εξομοίωση μιας μπάλας που κυλάει σε έναν λόφο.

Mesh: Αναφέρεται στο Mesh το οποίο θα χρησιμοποιήσει για να φτιάξει το Collider Mesh.

Εικόνα 4.8.6.4 Mesh Collider Component

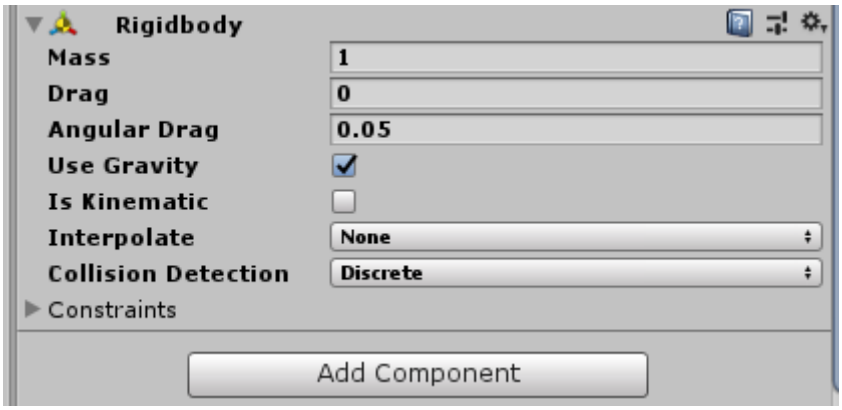


4.8.6.5 Rigidbody Component

Mass: Πρόκειται για την μάζα του σώματος την οποία θα χρειαστεί για να υπολογίσει δυνάμεις φυσικής (π.χ. βαρύτητα) καθώς και κρούσεις.
Drag: Πόση αντίσταση θα έχει το αντικείμενο στον αέρα, κάτι που επηρεάζει το πόσο θα κινείται ένα αντικείμενο στον 3D χώρο μέσω δυνάμεων φυσικής (Force).
Angular Drag: Ίδιο με το Drag, απλώς αφορά την γωνιακή αντίσταση, δηλαδή την αντίσταση στον αέρα βάσει περιστροφών (Torque).
Use Gravity: Πρακτικά αν είναι ενεργοποιημένο, τότε έχουμε την έλξη της βαρύτητας στο αντικείμενό μας και θα το τραβήξει κάτω, δηλαδή προς τον global αρνητικό άξονα των Y της Unity.
Is Kinematic: Αν είναι ενεργοποιημένο, τότε δεν επηρεάζεται από τα physics, δηλαδή δεν θα μπορέσει κάποιο άλλο αντικείμενο να το σπρώξει ποτέ ή να το μετακινήσει, ή να το αναγκάσει να κάνει κάποια περιστροφή, παρ' όλα αυτά, υπάρχει ακόμα Collision μαζί του. Χρήσιμο όταν θέλουμε να φτιάξουμε κινούμενες πλατφόρμες ή να πραγματοποιήσουμε νοητές κρούσεις (Triggers) με αντικείμενα τα οποία δεν κινούνται με δυνάμεις φυσικής αλλά με animations.
Interpolate: Αν παρατηρήσουμε ότι τα Rigidbody μας κινούνται «άγαρμπα», τότε θα πρέπει να ενεργοποιήσουμε μια από τις πιθανές του τιμές στον Inspector. Δεν χρειάζεται να του δώσουμε ιδιαίτερη σημασία αυτή τη στιγμή πάντως.
Collision Detection: Πρόκειται για τον τρόπο που θα ελέγχει τα collisions η Unity για το συγκεκριμένο Rigidbody. Ασχολούμαστε με αυτό συνήθως όταν έχουμε να κάνουμε με πολύ γρήγορα αντικείμενα, τα οποία περνάνε μέσα από άλλα και δεν προλαβαίνει η μηχανή να δει αν έγινε collision. Έχουμε λοιπόν 3 τύπους collision detection:
Discrete (Διακριτός): Πρόκειται για την προκαθορισμένη τιμή και τον πιο κλασικό τρόπο που ελέγχουμε collisions σε ένα scene ανά τακτά χρονικά διαστήματα (βλέπε FixedUpdate).
Continuous (Συνεχές): Χρησιμοποιείται συνήθως για τα σταθερά αντικείμενα στον 3D χώρο που θέλουμε να ελέγξουν για collision με πολύ γρήγορα αντικείμενα.
Continuous Dynamic (Δυναμικός Συνεχές): Χρησιμοποιείται στα πάρα πολύ γρήγορα αντικείμενα που κάνουν κρούσεις με άλλα δυναμικά αντικείμενα στον

3D χώρο και λόγω ταχύτητάς τους, δεν προλαβαίνει ποτέ το Engine το ίδιο να υπολογίσει Collision.
111
Constraints: Αφορά τους περιορισμούς που θα έχει το αντικείμενο στον 3D χώρο. Συγκεκριμένα, μπορούμε να «κλειδώσουμε» την δυνατότητά του να περιστρέφεται ή και να κινείται ακόμα σε έναν συγκεκριμένο άξονα. Γι' αυτό, υπάρχουν αυτά τα Checkboxes κάτω.
Οι επιλογές Continuous και Continuous Dynamic, ακριβώς επειδή ελέγχονται μονίμως, καλό θα είναι να αποφεύγονται αν μπορούμε να χρησιμοποιήσουμε το Discrete, μιας και χρειάζονται πολλούς υπολογιστικούς πόρους για να τρέξουν σωστά.

Εικόνα 4.8.6.5 RigidBody Component



4.9. Animations Στην Unity3d Και Τεχνητή Νοημοσύνη

4.9.1 Το σύστημα των Animations της Unity3D

Η Unity υποστηρίζει animations. Όταν θέλουμε να κάνουμε κάποια συγκεκριμένη κίνηση και δεν μπορούμε να την κάνουμε βάσει κώδικα, γιατί δεν μπορούμε να γράψουμε κάτ ειροκίνητα, παρά μόνο δυναμικά, τότε ερχόμαστε στην ανάγκη του animation system της Unity.!

Το **Animation System** είναι ό,τι πρέπει για συγκεκριμένες κινήσεις στον χώρο. Και όταν λέμε συγκεκριμένες, εννοούμε πως ό,τι τιμές πειραχτούν κατά την «εγγραφή» της κίνησης, αυτές οι τιμές είναι που θα «παιχτούν» στο στάδιο της αναπαραγωγής του animation.

4.9.2 Keyframe-based Animation Systems & Tweening

Με τον όρο **KeyFrame** εννοούμε ότι μπορούμε μέσα σε ένα πεπερασμένο και καθορισμένο χρονικό διάστημα, όπως για π.χ. 8 δευτερόλεπτα ή 4 λεπτά ή 10 ώρες, να σχεδιάσουμε λεπτομερώς μια κίνηση χωρίς να ασχοληθούμε με όλα τα καρέ.

Ο τρόπος που δουλεύει το σύστημα είναι ο εξής:

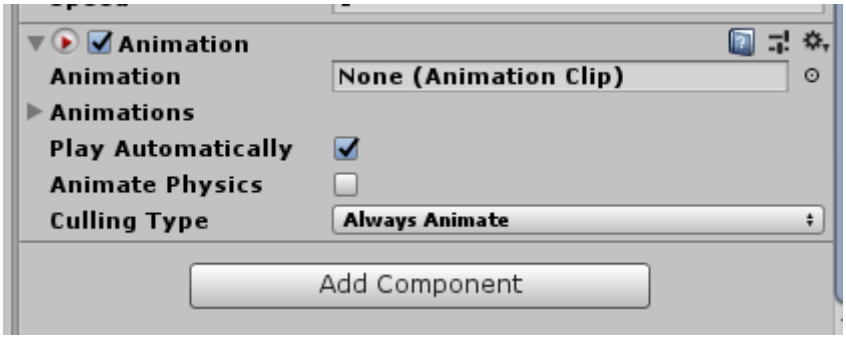
Ορίζουμε σε συγκεκριμένες χρονικές στιγμές του διαστήματος την ακριβή θέση, περιστροφή ή Scaling που θέλουμε να έχει ένα GameObject στο Transform Component του και από εκεί και ύστερα το σύστημα αναλαμβάνει να φτιάξει όλη την μετάβαση του αντικειμένου μεταξύ των συγκεκριμένων χρονικών στιγμών που έχουμε ορίσει. Αυτό μπορεί επίσης να γίνει για οποιοδήποτε συνδυασμό από properties που έχουν συγκεκριμένα components. Ας πούμε ένα παράδειγμα.

Το μόνο που μας ενδιαφέρει είναι οι θέσεις που θα έχει τις συγκεκριμένες χρονικές στιγμές και όχι το πώς θα πάει εκεί. Αυτό είναι δουλειά του συστήματος και ονομάζεται **Motion Tween**. Και θα το δούμε πρακτικά στην ανάπτυξη του **ZombieAnimatorController** στη συνέχεια των ενοτήτων μας.

4.9.3 Animation Component

Ας δούμε προσεχτικά πό τι αποτελειται το Animation Component (εικόνα 9.3)

Εικόνα 4.9.3 Το Animation Component



1. Το “**Animation**” Property ουσιαστικά έχει το τρέχον φορτωμένο Animation το οποίο θα μπορεί να παίξει ανά πάσα στιγμή.
2. Το **Animations** είναι μια λίστα με animations, η οποία μπορεί να γίνει accessed από τα Scripts που έχουμε γράψει και θέλουμε να αλληλοεπιδράσουμε μαζί του.
3. Το bool “**Play Automatically**” όταν είναι true, το animation παίζει με το που ξεκινάει το παιχνίδι.
4. Το bool “**Animate Physics**”, ουσιαστικά, μας ρωτάει αν θα θέλαμε να αλληλοεπιδρά το Animation μας με Physics.
5. Τέλος, το **Culling Type** καθορίζει πότε δεν θα παίζει το Animation. Είναι αρκετά προχωρημένο για το scale του project μας, οπότε δεν θα του δώσουμε ιδιαίτερη σημασία.

Με το **Animation Component** χρησιμοποιούμε τα Animations που έχουμε φτιάξει στο Animation Window

4.10 ΜΕΛΕΤΗ ΤΟΥ PARTICLE SYSTEMS COMPONENT

Στα παιχνίδια μας έχουμε και άλλου είδους οντότητες, οι οποίες είναι άλλοτε σε υγρή ή αέρια μορφή, κοινώς πολύ δύσκολο να τις αναπαραστήσουμε μόνο με Meshes ή Sprites. Για εφέ όπως μετακινούμενα υγρά, καπνό, σύννεφα, φωτιές και μαγικά ξόρκια, υπάρχει μια διαφορετική προσέγγιση στα γραφικά, συγκεκριμένων συστημάτων, γνωστά ως σωματίδια ή Particle Systems.

4.10.1 Εισαγωγή στα Particle Systems

Τα Particles (σωματίδια) είναι μικρές, απλές εικόνες ή meshes, τα οποία εμφανίζονται στην οθόνη και κινούνται σε μεγάλους αριθμούς μέσω ενός Particle System. Κάθε μικρό particle συμβολίζει ένα μικρό μέρος από μια υγρή ή άμορφη οντότητα, και το οπτικό εφέ που δημιουργεί η κίνηση όλων των particles μαζί δημιουργεί την ψευδαίσθηση μιας ολοκληρωμένης οντότητας.

Χρησιμοποιώντας ένα νέφος καπνού σαν παράδειγμα, κάθε Particle θα συμβόλιζε ένα πολύ μικρό μέρος αυτού του νέφους.

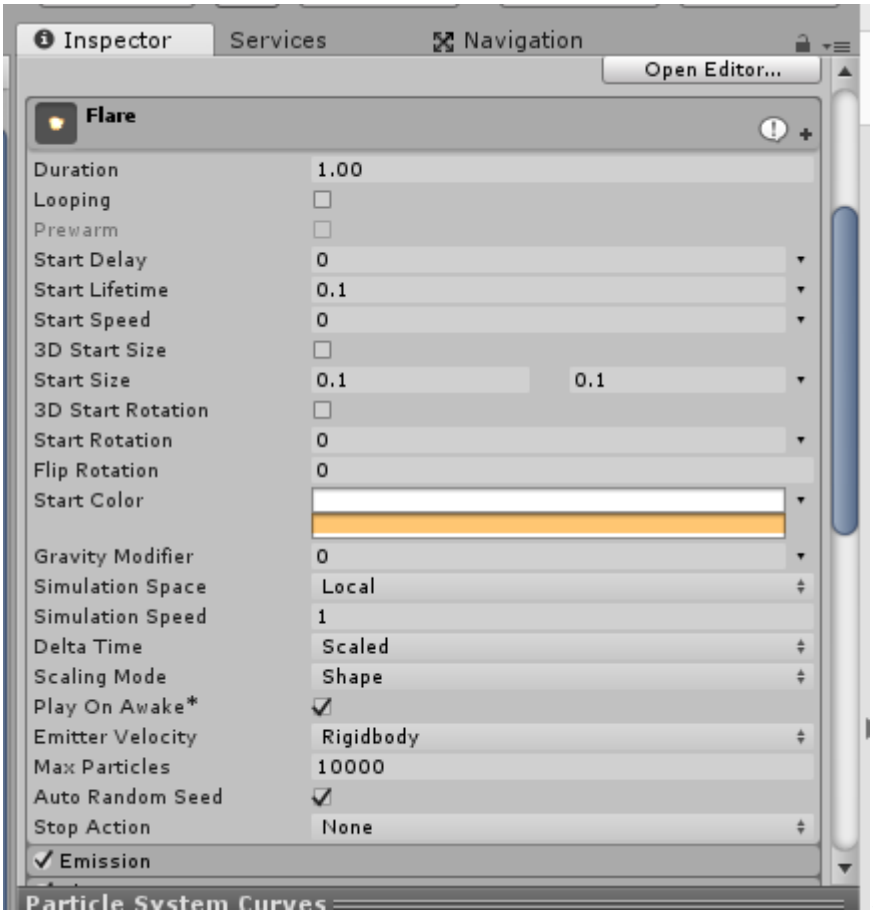
Αν όμως όλα τα Particles μαζευόντουσαν σε μια περιοχή όλα μαζί με μεγάλη πυκνότητα, τότε θα φαινόταν ότι σε αυτήν την περιοχή υπάρχει ένα μεγαλύτερο νέφος.
Μερικές βασικές έννοιες που θα πρέπει να κρατήσουμε για τα Particles είναι οι εξής:

6. Κάθε **Particle** (σωματίδιο) έχει προκαθορισμένο χρόνο ζωής (lifetime), τυπικά μερικά δευτερόλεπτα, κατά την διάρκεια των οποίων μπορεί να παρουσιάσει μεταβολές. Ξεκινάει η «ζωή» του όταν εκπέμπεται ή αλλιώς Emmited από το Particle System.
7. Το Particle System κάνει **emit** (εκπέμπει) particles σε τυχαίες θέσεις μέσα σε μια περιοχή του χώρου, δομημένη σαν μια σφαίρα, ημισφαίριο, κώνος, κουτί ή κάποιο άλλο Mesh.
8. Τα Particles φαίνονται στην οθόνη μέχρι κάποια στιγμή να τελειώσει ο χρόνος ζωής τους, όπου και αφαιρούνται από το Particle System.
9. Το **Emmision Rate** του συστήματος επηρεάζει τον ρυθμό παραγωγής particles για κάθε δευτερόλεπτο, αν και μερικές φορές το Rate μπορεί να είναι και αυτό τυχαίο ελάχιστα.
10. Η επιλογή μας όσον αφορά το Lifetime και το Emmision Rate του συστήματος επηρεάζει τον **μέσο αριθμό** Particles που θα φαίνονται ανά πάσα στιγμή στην οθόνη.

4.10.2 Το Particle System Component – Core Module

Το Particle System Component έχει ένα Core Module (κύρια λειτουργία) που αφορά όλο το σύστημα, και από εκεί και ύστερα έχει μικρά υπό modules που προσθέτουν ποικιλία και fx στα particles μας. Ας δούμε το βασικό Module για αρχή (εικόνα 10.2)

Εικόνα 4.10.2 Core Module



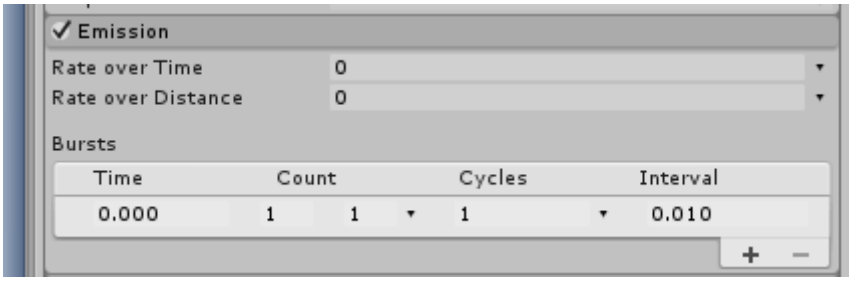
- **Duration**: Πόση ώρα θα είναι ενεργός ο Particle System από την στιγμή που θα ξεκινήσει.
- **Looping**: Καθορίζει αν το System θα ξεκινάει από το μηδέν ξανά, όταν φτάσει το μέγιστο του Duration του.
- **Prewarm**: Αν είναι ενεργοποιημένο, τότε το σύστημα θα ξεκινήσει από το μηδέν έχουμε ήδη ολοκληρώσει ένα cycle του emission του (πρέπει να είναι και το Looping ενεργοποιημένο).
- **Start Delay**: Καθορίζει την καθυστέρηση σε δευτερόλεπτα, πριν αρχίσει να γίνεται
- emission των particles.
- **Start Lifetime**: Καθορίζει τον χρόνο ζωής για κάθε particle.
- **Start Speed**: Καθορίζει την ταχύτητα που θα έχει κάθε particle.
- **Start Size**: Καθορίζει το μέγεθος που θα έχει κάθε particle.
- **Start Rotation**: Καθορίζει την γωνία περιστροφής που θα έχει κάθε particle όταν δημιουργείται.
- **Start Color**: Καθορίζει το αρχικό χρώμα κάθε particle.
- **Gravity Multiplier**: Πολλαπλασιάζει την βαρύτητα του Physics συστήματος της Unity και την εφαρμόζει στα particles. Αν είναι μηδέν, τότε δεν έχουμε βαρύτητα.
- **Inherit Velocity**: Αν το particle system κινείται, τα particles θέλουμε να έχουν την ίδια ταχύτητα με αυτό;
- Simulation Space: Τα particles θα πρέπει να δρουν σε τοπικά XYZ (και να μετακινείται με τον parent) ή σε global (και να μένουν στον χώρο τους);
- **Play on Awake**: Θα πρέπει να ξεκινήσει το Particle System να κάνει emit particles μ το που ξεκινήσει το GameObject να είναι ενεργό;
- **Max Particles**: Μέχρι πόσα particles θέλουμε να υπάρχουν ταυτόχρονα στηνοθόνη.
- Αν το όριο το «πιάσουμε», τότε το σύστημα δεν θα δημιουργεί νέα particles μέχρι κάποιο από τα πρώτα διαγραφούν από το σύστημα λόγω lifetime.

4.10.3 Το Particle System Component – Emission Module

Χρήσιμο όταν θέλουμε να επηρεάσουμε τον ρυθμό παραγωγής particles,

Εικόνα 4.10.3 Emmision Module

.



- **Rate**: Πρόκειται για το Emission Rate που προαναφέραμε: Ο αριθμός των particles που θα γίνονται emit κάθε μονάδα χρόνου ή απόστασης (αναλόγως τι θα επιλέξουμε στο dropdown).
- **Bursts**: Αν το Rate είναι σε time mode, μας επιτρέπει να διαλέξουμε χρονικές στιγμέςόπου τα particles θα κάνουν burst (παραγωγή μεγάλου αριθμού particles σε πολύ μικρό χρονικό διάστημα)

4.11. ΗΧΟΣ ΚΑΙ ΜΟΥΣΙΚΗ

Η Unity, για να εξομοιώσει τα εφέ της θέσης, χρειάζεται τους ήχους να προέρχονται από πηγές (**Audio Sources**) οι οποίες είναι attached πάνω σε αντικείμενα (Ένα Audio Source είναι κατ' επέκταση Component).

Οι ήχοι που εκπέμπονται τότε μπορούν να ληφθούν από έναν ακροατή, ο οποίος θεωρείται ο δέκτης του ήχου (Audio Listener), και τις περισσότερες φορές ο δέκτης είναι η main camera.

Ευτυχώς, σε αυτό το μάθημα θα ασχοληθούμε αποκλειστικά με αυτό: Τον Audio Source και τον Audio Listener. Πρόκειται για τα 2 βασικότερα Components που αφορούν την εκπομπή ήχου και την λήψη ήχου σε ένα οποιοδήποτε Scene.

4.11.1 Audio Listener Component

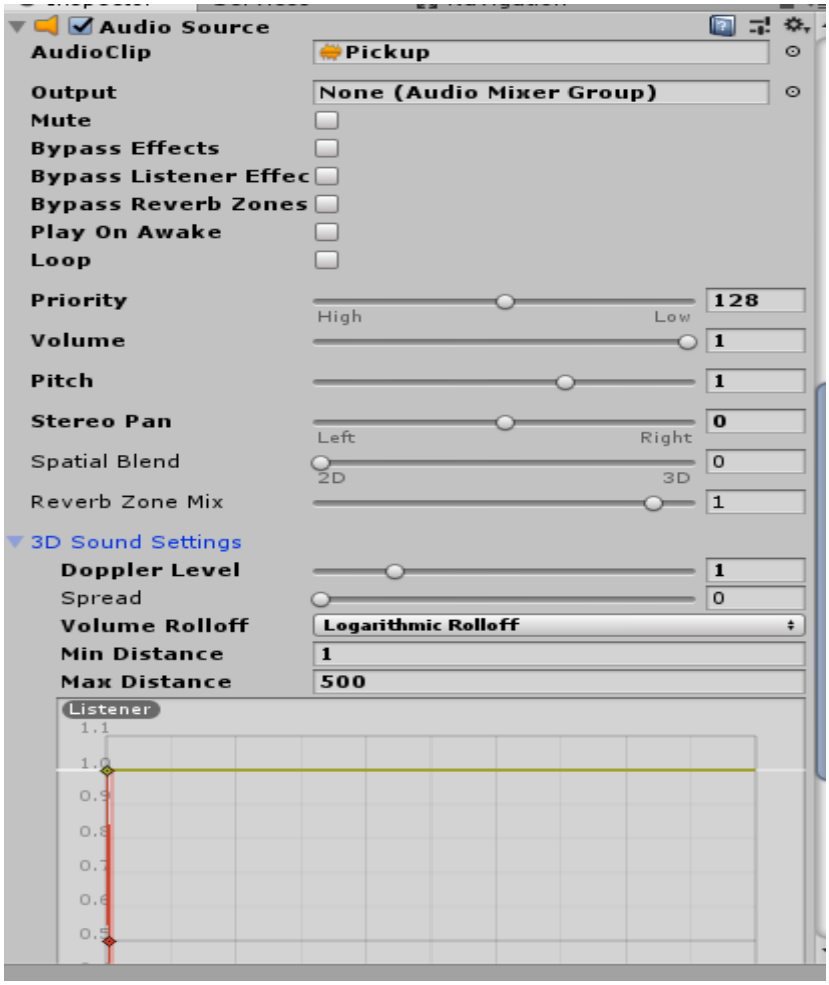
Κάθε Scene πρέπει να **έχει αυστηρά έναν και μόνο έναν Audio Listener** attached σε κάποιο αντικείμενο για να μην μπερδευτούν τα αυτιά μας με τους ήχους που θα ακούμε αλλά και η Unity με τους ήχους που θα λαμβάνει.

Ο δέκτης λοιπόν είναι attached στην main camera, αυτό σημαίνει ότι σε ένα 3D σύστημα ήχων και ένα scene διαμορφωμένο όπως στην εικόνα 3 θα παίξει ένας ήχος στο Audio Source που βλέπουμε εκεί, και όσο πιο κοντά πλησιάζουμε σε αυτόν τόσο πιο δυνατά θα ακούγεται ο ήχος που εκπέμπει, εφόσον ο ήχος είναι 3D ήχος, γιατί υπάρχουν και Audio Sources που εκπέμπουν τους λεγόμενους 2D ήχους, ήχους δηλαδή που ανεξάρτητα από την απόσταση Audio Listener με Audio Source θα ακούγονται πάντα στην ίδια ένταση

4.11.2 Audio Source Component

Πάμε να δούμε τι ακριβώς έχει πάνω του λοιπόν ένα Audio Source Component.

Εικόνα 4.10.3 Τα Properties σε ένα Audio Source Component



Αρκετά πιο πολύπλοκος από τον Audio Listener, κάτι που είναι πολύ λογικό, καθώς ο Audio Listener δεν μπορεί να αλλάξει τον τρόπο που ακούει. Απλά ακούει. Αυτό είναι όλο. Ο Source όμως μπορεί να αλλάξει τον τρόπο που θα στείλει έναν ήχο ή τον τρόπο που θα τον «διαμορφώσει» καθώς τον στέλνει. Ένα Audio Source έχει τα εξής properties:

- **Audio Clip**: πρόκειται για το ηχητικό κομμάτι που είναι φορτωμένο πάνω στον Source, και αυτό που θα παίξει αν καλέσουμε τη μέθοδό του, Play.
- **Mute**: Αν το ενεργοποιήσουμε, όπως είναι λογικό, δεν θα ακούσουμε τίποτα ποτέ από αυτό το Source.
- **Bypass Effects / Listener Effects / Reverb Zones:** Πρόκειται για λειτουργίες που μας επιτρέπουν να φιλτράρουμε τα εφέ που θα αφορούν ήχους (παραμορφώσεις κτλ.)
- **Play on Awake**: Όπως στα Animation και Particle System Components, αν αυτό είναι ενεργοποιημένο, τότε ξεκινάει να παίζει ήχος από την στιγμή που ενεργοποιείται το GameObject.
- **Loop**: Αν είναι ενεργοποιημένο, τότε ο ήχος παίζει ξανά και ξανά ασταμάτητα.
- **Priority:** Πρόκειται για ένα property που αφορά την προτεραιότητα του ήχου σε ένα Scene, δηλαδή να ακούγεται ψηλότερα ή χαμηλότερα σε σχέση με άλλους ήχους που συνυπάρχουν στο Scene (το πιο σημαντικό: 0, το λιγότερο σημαντικό: 256 και προκαθορισμένα έχει τιμή 128).
- **Volume:** Πόσο δυνατά θέλουμε να ακούγεται ο ήχος, η ένταση δηλαδή.

- **Pitch:** Η ταχύτητα playback του ήχου. Το φυσιολογικό είναι 1. Προσοχή, μεγαλύτερες και μικρότερες τιμές αυξάνουν ή μειώνουν όχι μόνο την ταχύτητα αλλά και το ύψος των ήχων που ακούγονται.
- **3D/2D Sound Settings**: Πρόκειται για ρυθμίσεις συγκεκριμένα για 2D / 3D ήχους, όπως π.χ. πόσο δυνατό θα είναι το Doppler Effect, πόση θα είναι η μέγιστη και ελάχιστη απόσταση που θα ακουστεί ένας ήχος (όσον αφορά το 3D) ή το Panning του ήχου σε 2D και 3D κατάσταση.

ΚΕΦΑΛΑΙΟ 5. Υλοποίηση Παιχνιδιού – Ανάπτυξη Μηχανισμών

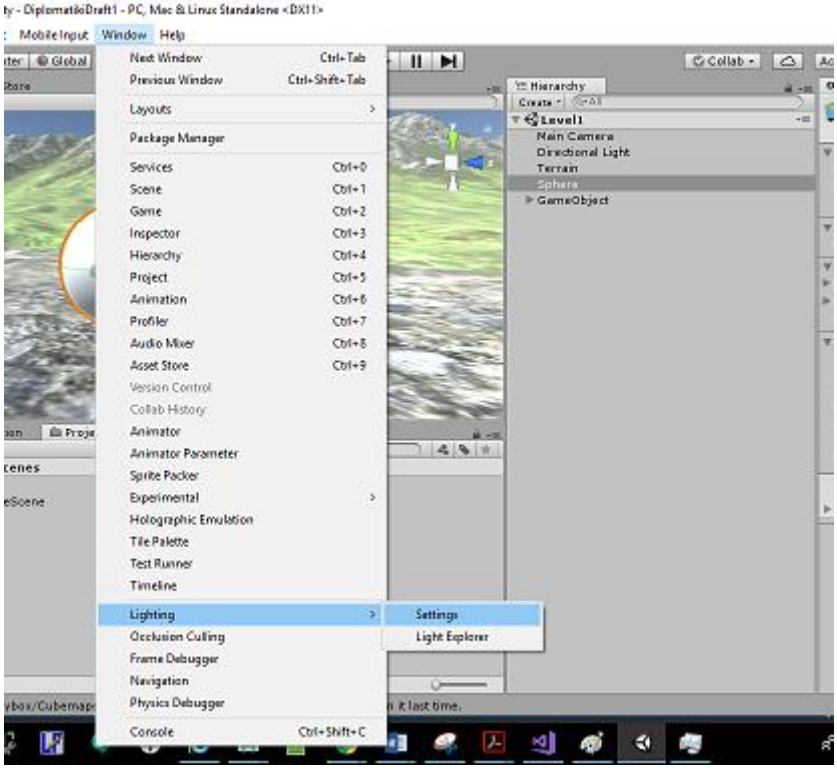
5.1. ΦΩΤΙΣΜΟΣ (LIGHTING)

5.1.1 Γενικός φωτισμός περιβάλλοντος – Sky Box

Η Unity θεωρεί πως ο γενικός φωτισμός που θα έχει κάθε Scene θα είναι σκοτεινός, με την προοπτική ότι θα πρέπει να τα φωτίζουμε εμείς αργότερα! Οπότε θα πάμε να αυξήσουμε τον γενικό φωτισμό περιβάλλοντος (Ambience Lighting).

Θα πάμε Windows → Lighting → Settings

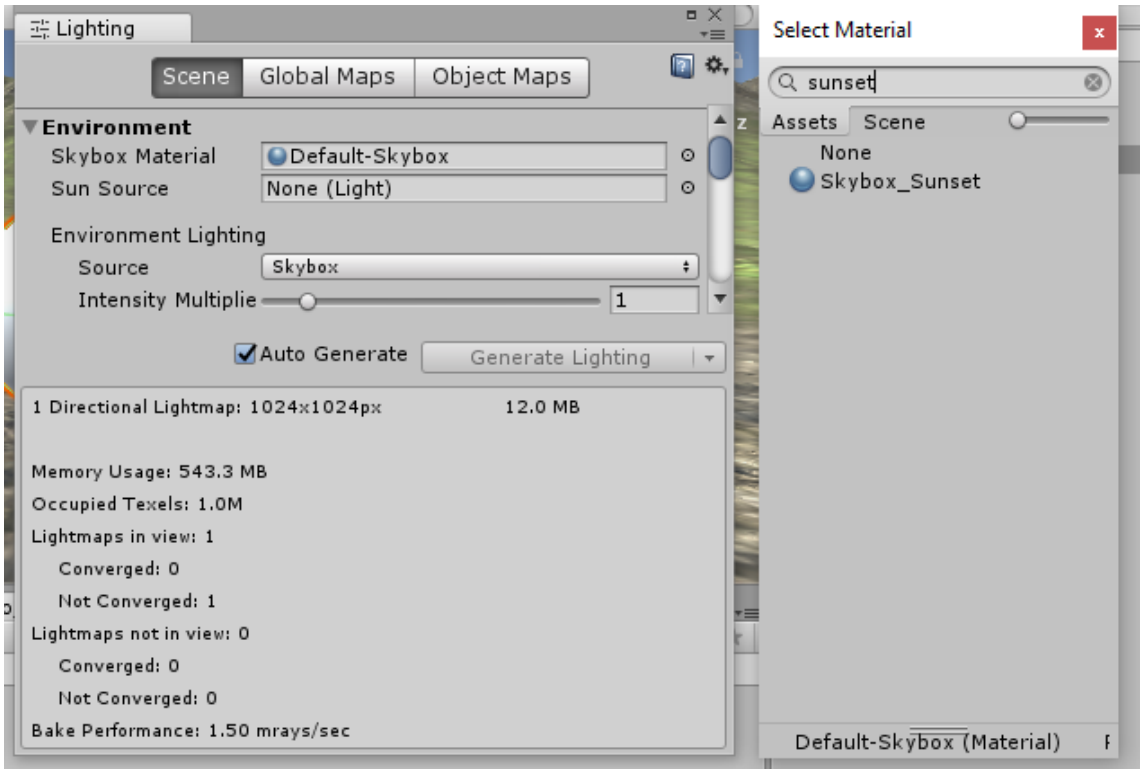
Εικόνα 5.1.1 Windows → Lighting → Settings



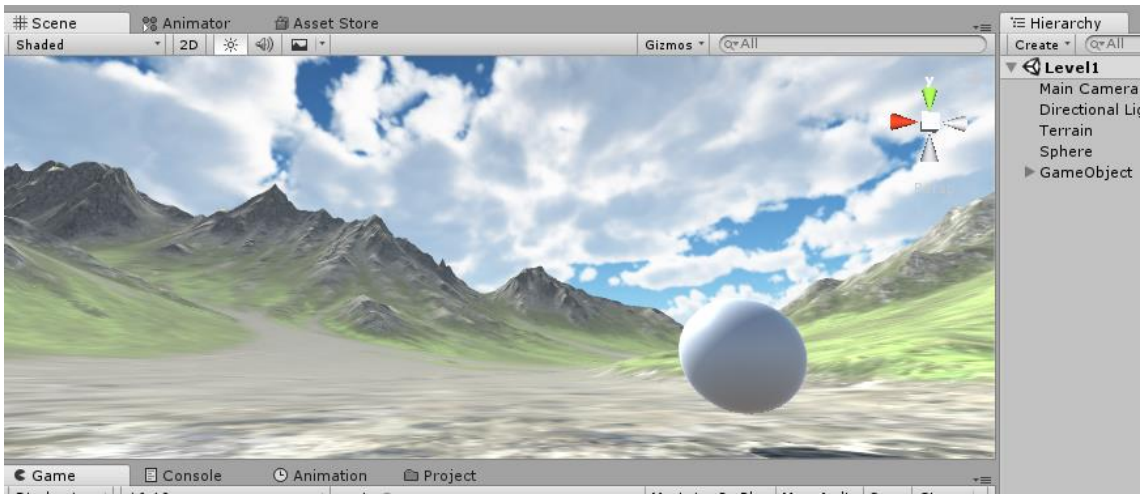
Στο tab scene θα κλικάρουμε από το TextBoxArea Environment το κυκλάκι δίπλα από την επιλογή SkyBox Material. Αφού το κλικάρουμε θα ανοίξει το διπλάνο παράθυρο ζητώντας να επιλέξουμε το Material. Γράφουμε Sunset DayTime.

Εν συνεχεία στην επιλογή SunSource θα βάλουμετο GO Directional Light που εχουμε στο Scene μας.

Εικόνα 5.1.1β SkyBox Material



Εικόνα 5.1.1γ Scene view μετά την εισαγωγή του Sunset Daytime SkyBox



5. 2. Δημιουργία Οντότητας – Εχθρός

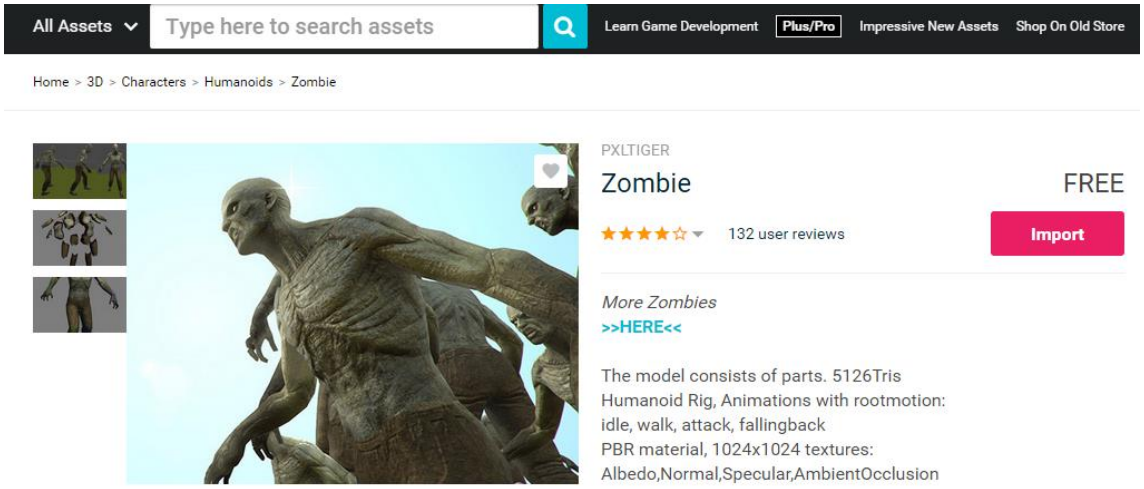
5.2.1 Ενσωμάτωση του Ζόμπι στο Scene

Θα εισέλθουμε στο Asset Store όπου μας παρέχει μερικά γραφικά μοντέλα – scripting εντελώς δωρεάν. Σαφώς υπάρχουν γραφικά μοντέλα (Pro version) για τα οποία θα πρέπει κάποιος να πληρώσει ώστε να μπορέσει να τα αποκτήσει.

Στη περίπτωση μας θα εισάγουμε ένα Zombie που θα αποτεέσει και τον κύριο εχθρό μας.

Πάμε λοιπόν στην καρτέλα Window → Asset Store

Στο SearchBox θα πληκτρολογήσουμε Zombie και θα βάλουμε στο φίλτρο price Range μηδέν.



Στην κατηγορία Asseets δημιουργούμε ένα φάκελο με την ονομασία Imported. Εκεί μέσα θα αποθηκεύσουμε το φάκελο του Zombie που κατεβάσαμε πριν από λιγο από το Asset Store.

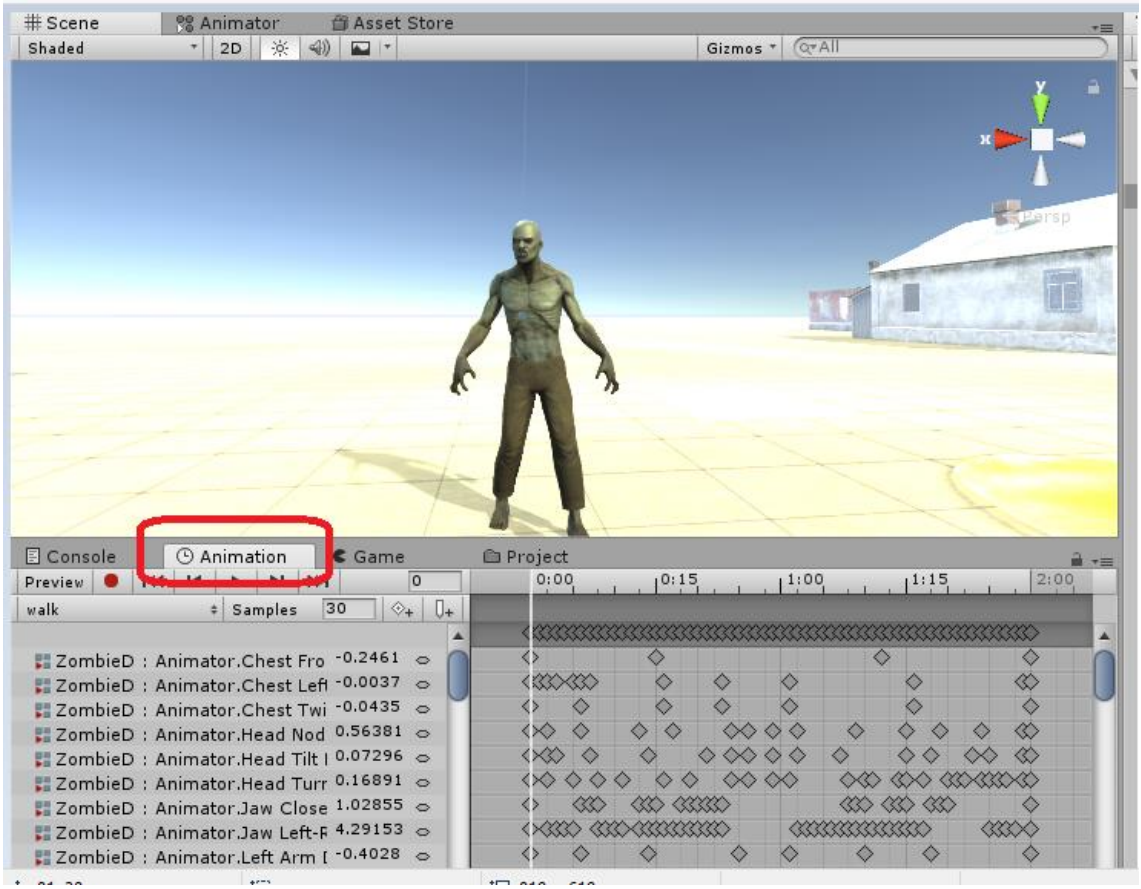
Εν συνεχεία κάνουμε drag and drop το prefab Zombie μέσα στο Scene μας.

5.2.2 Το Animation System του εχθρού

Ανοίγουμε το Animation Window της Unity (εικόνα 1). Στην συνέχεια, θα κάνουμε drag and drop την καρτέλα που γράφει "**Animation**" δίπλα από την Console tab, ώστε όλο το παράθυρο να μπει στο κάτω μέρος του project.

Επιλέγουμε τον εχθρό και θα παρατηρήσουμε ότι το κουμπάκι "**REC**" στο animation window είναι πλέον ενεργό! Ας το πατήσουμε για να αποθηκεύσουμε το νέο μας animation.

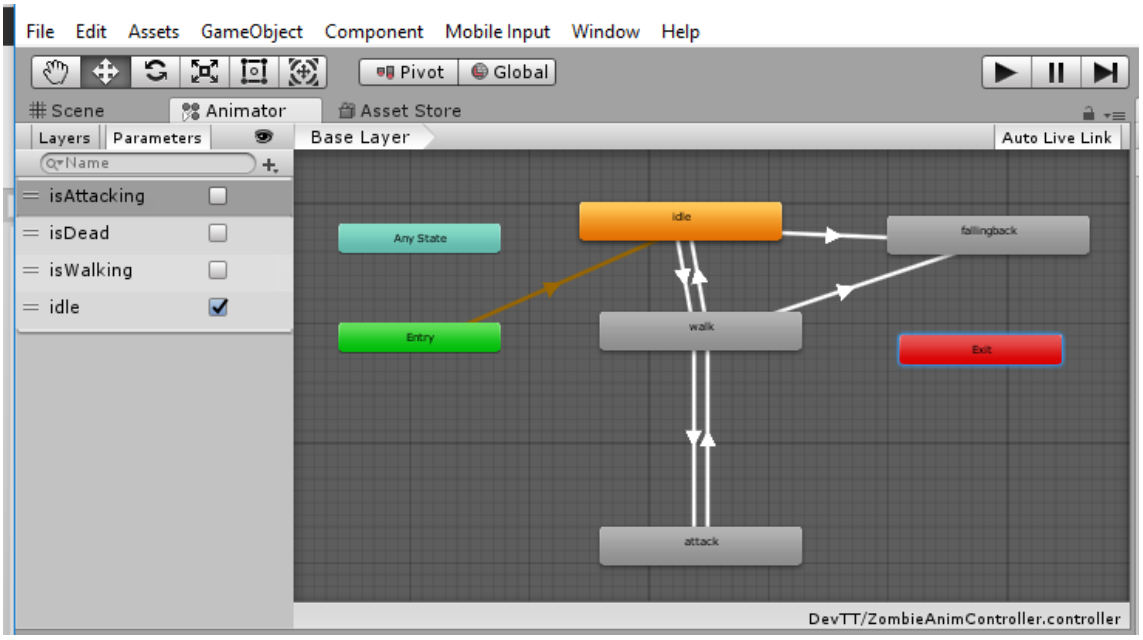
Για λόγους πρακτικούς θα φτιάξουμε έναν φάκελο "Animations" και το animation αυτό θα το ονομάσουμε "**Walk**" και θα το αποθηκεύσουμε μέσα σε αυτόν τον φάκελο.



Ορίζουμε σε συγκεκριμένες χρονικές στιγμές του διαστήματος την ακριβή θέση, περιστροφή ή Scaling που θέλουμε να έχει το GameObject στο **Transform** Component του και από εκεί και ύστερα το σύστημα αναλαμβάνει να φτιάξει όλη την μετάβαση του αντικειμένου μεταξύ των συγκεκριμένων χρονικών στιγμών που έχουμε ορίσει. Αυτό μπορεί επίσης να γίνει για οποιοδήποτε συνδυασμό από properties που έχουν συγκεκριμένα components.

Ας πούμε ένα παράδειγμα. Έχω ένα χρονικό διάστημα 20 δευτερολέπτων. Στο δευτερόλεπτο μηδέν (0), στην αρχή δηλαδή ορίζουμε ότι το αντικείμενο θα έχει X = 10. Μετά πάμε τον δείκτη μας στην χρονική στιγμή t = 5, δηλαδή στην μέση, και ορίζουμε X = 50 και μετά στην χρονικήστιγμή t = 10, δηλαδή στο τέλος, και ορίζουμε ότι το αντικείμενο θα έχει X = -10.

Παρατηρούμε ότι προστέθηκε ένας **animator Component** στο GameObject μας. Θα κάνουμε double click και θα ανοίξει το παρακάτω παράθυρο.



Στο παράθυρο θα δούμε όλα τα animations όπως το Idle, Walk,Attack , FallingBack καθώς και τα λεγόμενα Transitions, τη μετάβαση δηλαδή που θα κάνει ο animator από το ένα animation στο επόμενο ή και πίσω σε περίπτωση που το έχουμε ορίσει.

Στην κατάστηαση Idle ο εχθρός βρίσκεται σε «ηρεμία». Δεν κινέιται καθόλου. Για να κινηθεί πρέπει να δημιουργήσουμε Transition. Κάνουμε δεξί κλικ → Make Transition και το σύρουμε στο animation Walk.

Αντίστοφα κάνουμε και το transition από κατάσταση Walk → se Idle.

Προκειμένου να υλοποιηθούν οι εκάστοτε μεταβάσεις θα πρέπει να γράψουμε κώδικα.

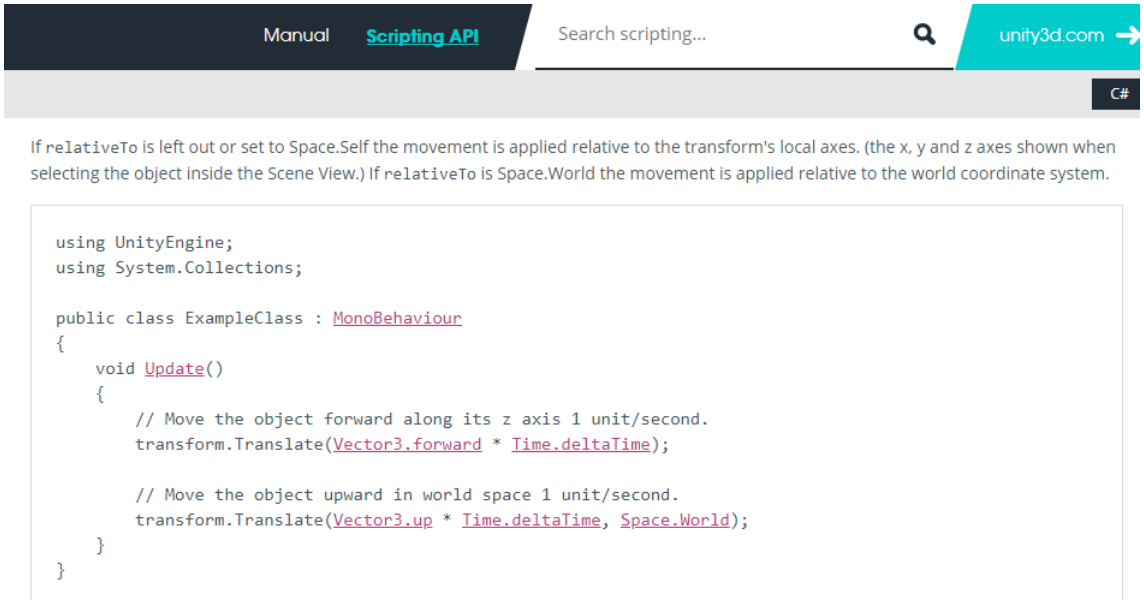
Πως θα μπορεί όμως ο κώδικας να επικοινωνεί με τον Animator Component και πως θα καλούνται οι μεταβάσεις?

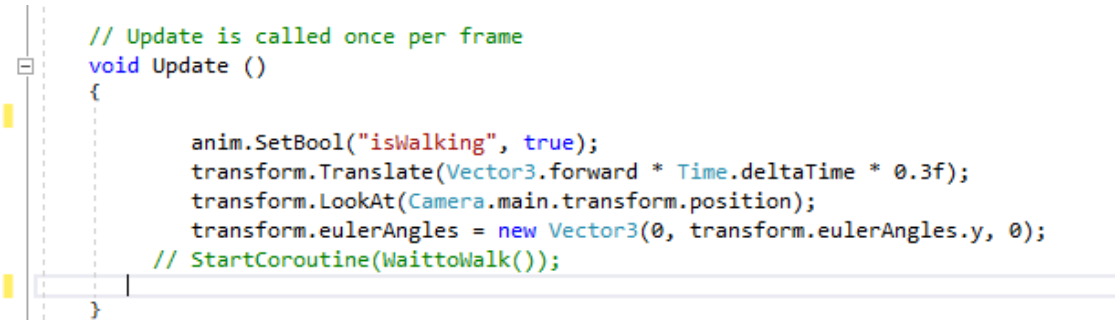
Η απάντηση στο δεύτερο σκέλος της ερωώτησης υλοποιείται μέσω των παραμέτρων "**Parameters**".

Δημιουργουμε τις εξης παραμέτρους τύπου Boolean:

- Idle
- IsWalking
- isAttacking
- isDead

5.2.3 Το Script Κίνσηης Και Animation SystemΤου Εχθρού





Ας μελετήσουμε για αρχή τι κάνει ο κώδικας μέσα στην Update.

- Με την λέξη **transform** αναφερόμαστε στο Transform Component του gameobject που έχει αυτό το Script πάνω του. Εναλλακτικά μια παρόμοια εντολή

θα ήταν η GetComponent Transform.

- Η **Translate** είναι μια μέθοδος του transform, η οποία μας επιτρέπει να κάνουμε

κίνηση (**Translate**) στον 3D χώρο και δέχεται ένα Vector3 και άλλες παραμέτρους.

Η μεταβλητή αυτηχρειάζεται 3 παραμέτρους (για τα X, Y και Z), σε αντίθεση με την Vector2 που χρειάζεται 2 (για το X και Y).

- Παρατηρούμε πως ορίζουμε το Z axis (Vector3.forward) και την πολλαπλασιάζουμε επί Time.deltaTime. Η **Time.deltaTime** είναι μια μεταβλητή της κλάσης **Time**.
  Η κλάση Time εκφρλαζει τον γραμμικό χρόνο που έχει περάσει από την στιγμή που τρέχει η εφαρμογή!
- Στην δικιά μας περίπτωση όμως έχουμε **Time.deltaTime**, η οποία συγκεκριμένη ορίζει τον χρόνο που περνάει για να πάμε από το πέρας του καρέ Ν στην αρχή του καρέ Ν+1. Αυτό το χρειαζόμαστε οπωσδήποτε για να μπορέσει να τρέχει σε όλων των ειδών τις πλατφόρμες με τον ίδιο ρυθμό ανεξάρτητα από την ταχύτητα του PC/Tablet/Laptop ή Mobile, στο οποίο τρέχουμε το παιχνίδι. Πιο αργά συστήματα κάνουν μεγαλύτερες κινήσεις, και πιο γρήγορα συστήματα κάνουν μικρότερες κινήσεις,

- **Transform.eulerAngles** Σε αυτή τη γραμμή κώδικα ορίζουμε μόνο την τιμή σε μοίρες για το y.ενώ για το x και z θέτουμε τιμές μηδέν.Αυτό το κάνουμε γιατί παρατηρήσαμε ότι ο εχθρός ερχόταν μεν καταπάνω μας αλλά δεν περπατούσε στο έδαφος αλλλα με κλίση προς τα πάνω. Παρακάτω η Unity μας παρέχει βοήθεια για να λύσουμε αυτό το issue.

Description

The rotation as Euler angles in degrees.

The x, y, and z angles represent a rotation z degrees around the z axis, x degrees around the x axis, and y degrees around the y axis.

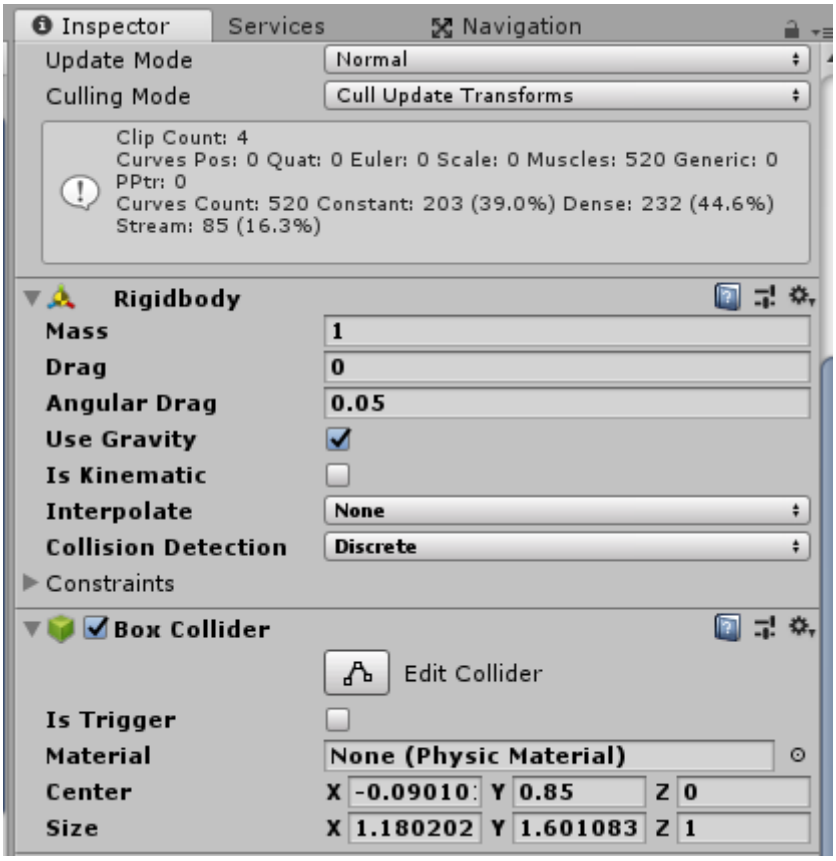
Only use this variable to read and set the angles to absolute values. Don't increment them, as it will fail when the angle exceeds 360 degrees. Use Transform.Rotate instead.

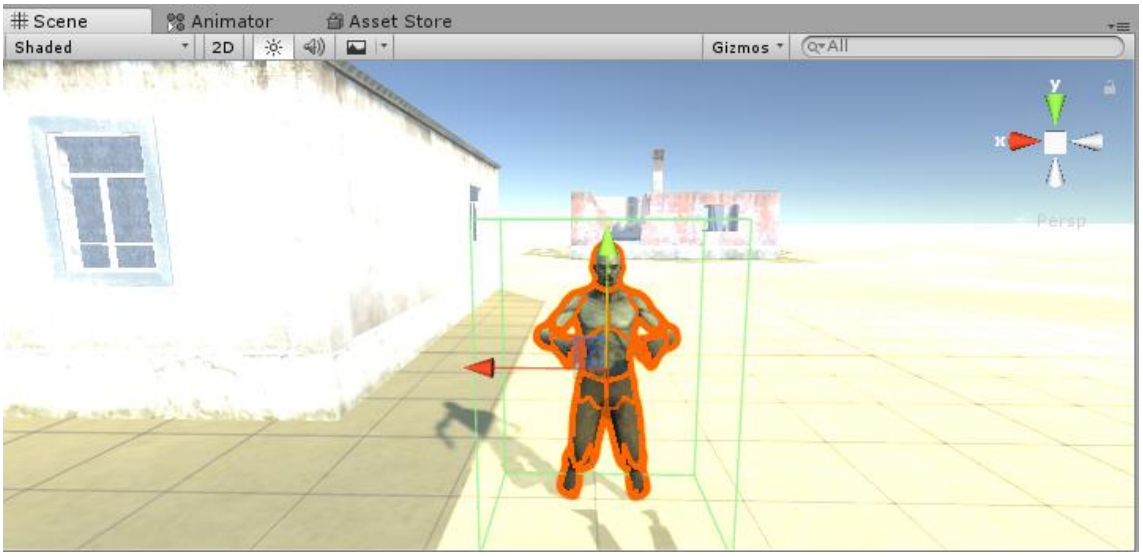
```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    public float yRotation = 5.0F;
    void Update() {
        yRotation += Input.GetAxis("Horizontal");
        transform.eulerAngles = new Vector3(10, yRotation, 0);
    }
```

5.2.4 ΝΟΗΜΟΣΥΝΗ ΕΧΘΡΩΝ Ι: ΕΠΙΘΕΣΗ

Στην παρούσα ενότητα θα αναπτύξουμε το script της επίθεσης του εχθρου. Αρχικά πρέπει να πούμε στην Unity να υπολογίσει το GameObject στα Physics. Επομένως στο GO Zombie προσθέτουμε RigidBody

Επίσης προσθέτουμε και Collider. Ο Collider το μόνο που κάνει είναι να παρέχει το μέσο για να γίνουν physics, αλλά δεν ενεργοποιεί το ίδιο το Physics engine. Οπότε όσα Game Objects θέλουμε να υπολογίζονται στα Physics και να μπορούμε να γράψουμε κώδικα για τα Collisions που κάνουν στον 3d χώρο θα πρέπει να έχουν αυτό το Component (Rigidbody), το οποίο τους επιτρέπει να υπολογίζονται στα collisions του Physics engine.


Inspector
Services
Navigation
Update Mode
Normal
Culling Mode
Cull Update Transforms
Clip Count: 4
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 520 Generic: 0
PPtr: 0
Curves Count: 520 Constant: 203 (39.0%) Dense: 232 (44.6%)
Stream: 85 (16.3%)
Rigidbody
Mass
1
Drag
0
Angular Drag
0.05
Use Gravity
Is Kinematic
Interpolate
None
Collision Detection
Discrete
Constraints
Box Collider
Edit Collider
Is Trigger
Material
None (Physic Material)
Center
X -0.09010
Y 0.85
Z 0
Size
X 1.180202
Y 1.601083
Z 1


Scene
Animator
Asset Store
Shaded
2D
Gizmos
All
Persp

```
emply-CSharp                                    CollisionWithPlayer
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.UI;
5
6   public class CollisionWithPlayer : MonoBehaviour {
7       public bool zombieIsThere;
8       float timer;
9       int timeBetweenAttack;
10      Animator anim;
11      public GameObject bloodyScreen;
12      AudioSource attackSound;
13
14      private PlayerHealth playerHealthscriptvar;
15
16      // Use this for initialization
17      void Start()
18      {
19          anim = GetComponent<Animator>();
20          timeBetweenAttack = 1;
21
22          // with the 3 lines below we have a connection between the two scripts this and PlayerHealth
23          GameObject PlayerHealthObject = GameObject.FindWithTag("Player");
24
25          if(PlayerHealthObject != null)
26          {
27              playerHealthscriptvar = PlayerHealthObject.GetComponent<PlayerHealth>();
28          }
29          AudioSource [] audios = GetComponents<AudioSource>();
30          attackSound = audios[0];
31      }
```

```
void OnCollisionEnter(Collision col)
{
    if (col.gameObject.tag == "Player")
    {
        zombieIsThere = true;
    }
}

void OnCollisionExit(Collision col)
{
    if (col.gameObject.tag == "Player")
    {
        zombieIsThere = false;


        anim.SetBool("isAttacking", false);
        bloodyScreen.gameObject.SetActive(false);
    }
}
```

Δημιουργούμε το script με όνομα **CollisionWithPlayer.cs.** Το σκεπτικό που κρύβεται πίσω από τον κώδικα είναι ότι πως στην περίπτωση που υπάρχει collision μεταξύ του Player και του zombie θα τρέχει η μέθοδος OnCollisionEnter ενω αν σταματήσου να έχουν επαφή θα τρέξει η μέθοδος OnCollisionExit().

Μέσα στη OnCollisionEnter τρέχει μία if Statement η οποία προκειμένου να υολοπιήσει το block του κώδικα πρέπει να τρέξει τη συνθήκη if (col.gameObject.tag). Εδώ πρακτικά μας λέει ότι αν το collision είναι του player τότε τρέξε τον κώδικα. Εδώ λοιπόν βάλαμε μία ετικέτα στον χαρακτήρα μας Player.

Στην αρχή του script ορίζουμε τις μεταβλητές.Εδώ θα μας απασχολήσουν οι εξής:

• **Bool zombieIsThere**: που θα παίνει τη τιμή True όταν εισέρχεται το Zombie στο collision του Player και αντίστοροφα

• **Int timeBetweenAttack:** την οποία μεταβλητη την αρχικοποιούμε στη Start() μέθοδο ίσον με 1. Στη συνέχεια στην Update() μέθοδο φτίαχνουμε ένα **Timer** τον οποίο θα τον μηδενίζουμε όταν καλούμε τη μέθοδ Attack(); Έτσι μπορούμε να προσθέσουμε μία δεύτερη συνθήκη στην if statement μέσα στην Update() μέθοδο. Τότε το Zombieμπορεί να επιτεθεί στον Player κάθε 1 δευτερόλεπτο.

```
    // Update is called once per frame
    void Update()
    {

        timer += Time.deltaTime;

        if (zombieIsThere && timer >= timeBetweenAttack)
        {
            Attack();
        }


    }
```

Μέσα στη μέθοδο Attack() ερχόμαστε να υλοποιήσουμε το **animation** του **isAttacking.**

Στη Start() μέθοδο αποθηκεύουμε το αποτέλεσμα που επιστρέφει η GetComponent<Animator> στη μεταβλητή anim τύπου Animator όπως την ορίσαμε αρχικά.
Όταν θα τρέχει η Attack() θα τρέχει και η εντολη anim.SetBool("isAttacking",true).

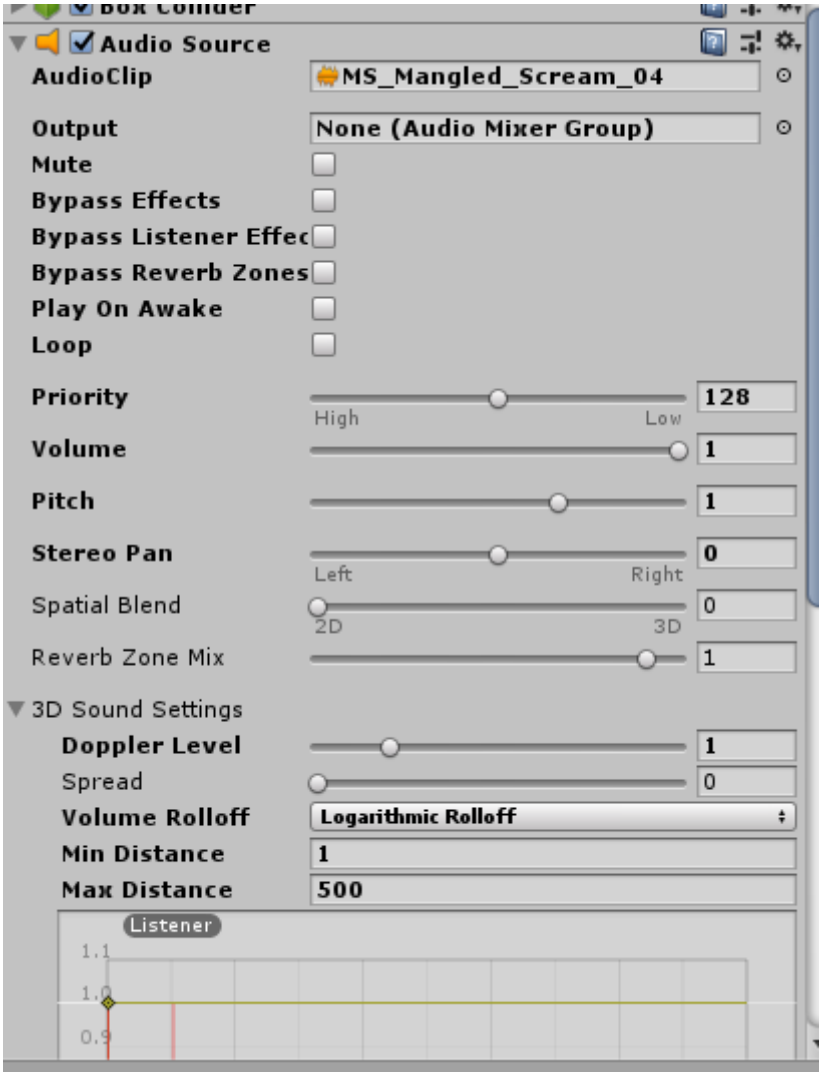
```
void Attack()
{
    timer = 0f;
    // GetComponent<Animator>().Play("attack");
    //anim.SetBool("isWalking", false);
    anim.SetBool("isAttacking",true);
    bloodyScreen.gameObject.SetActive(true);
    StartCoroutine(waitTwoSeconds());
    int deductedhealth = playerHealthscriptvar.health -= 5;
    Text healthComingFromScript= playerHealthscriptvar.healthText;

   string stringHealth = (deductedhealth).ToString();
    healthComingFromScript.text = "" + stringHealth;
    attackSound.Play();
}
```

5.2.5 Νοημοσύνη Εχθρών: Περιπολία

5.3 Εισαγωγή των Ηχητικών Assets του Εχθρού Και Χρήση Στο Παιχνίδι Με C#

Εισάγουμε από το Asset Store ηχητικά εφε που θα δώσουν στο Zombie μας κάποιο ακουστικό ερέθισμα.

Σε αυτό το σημείο είμαστε έτοιμοι να καλέσουμε τα ηχητικά εφέ από τον Gameplay κώδικα προσθέτοντας του ένα Component τύπου **AudioSource** , και αυτό μέσω της εντολής **AddComponent**. Και ενσωματώνουμε στο Audio Clip το sound από το asset που μόλις κατεβάσαμε.



Στον ορισμό των μεταβλητών πρόσθέτουμε άλλη μια μεταβλητή τύπου AudioSource

AudioSource attackSound;

Στη μέθοδο Start() θα προσπελάσουμε το component AudioSource με τη με τη μέθοδο GetComponent.

```
    }
    AudioSource [] audios = GetComponents<AudioSource>();
    attackSound = audios[0];
}
```

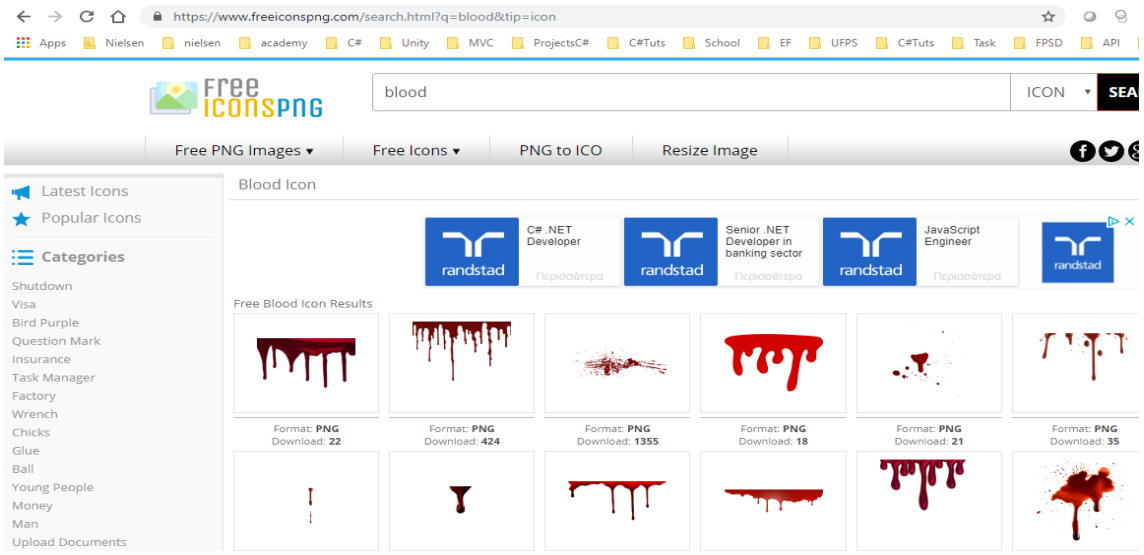
Επειδη θέλουμε ο εχθρός να βγάζει παραπάνω από έναν ήχους δημιουργούμε ένα πίνακα τύπου AudioSource με reference μεταβλητή audios και φυσικά αρχικοποιούμε το πρώτο index του πίνακα με τη μεταβλητη attackSound.

Μετά το μόνο που απομένει είναι να καλέσουμε την μέθοδο Play( ) touAudioSource, η οποία έχει την ίδια λογική αυτής για το Animation Component, δηλαδή θα ξεκινήσει να παίζει τον ήχο μας.

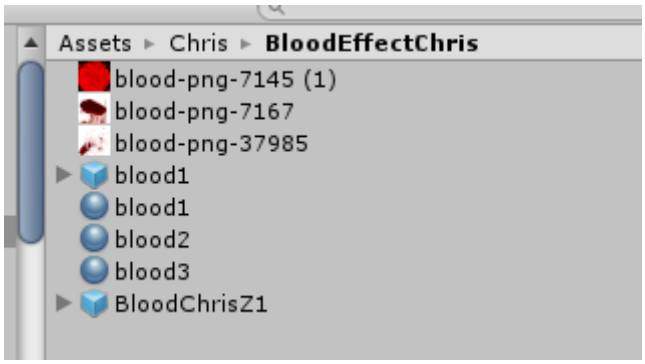
```
void Attack()
{
    timer = 0f;
    // GetComponent<Animator>().Play("attack");
    //anim.SetBool("isWalking", false);
    anim.SetBool("isAttacking",true);
    bloodyScreen.gameObject.SetActive(true);
    StartCoroutine(waitTwoSeconds());
    int deductedhealth = playerHealthscriptvar.health -= 5;
    Text healthComingFromScript= playerHealthscriptvar.healthText;

   string stringHealth = (deductedhealth).ToString();
    healthComingFromScript.text = "" + stringHealth;
    attackSound.Play();
}
```

5.4 Δημιουργία Αίματος Εφέ Μέσω Particle System Και Ενσωμάτωση Του Στον Εχθρό

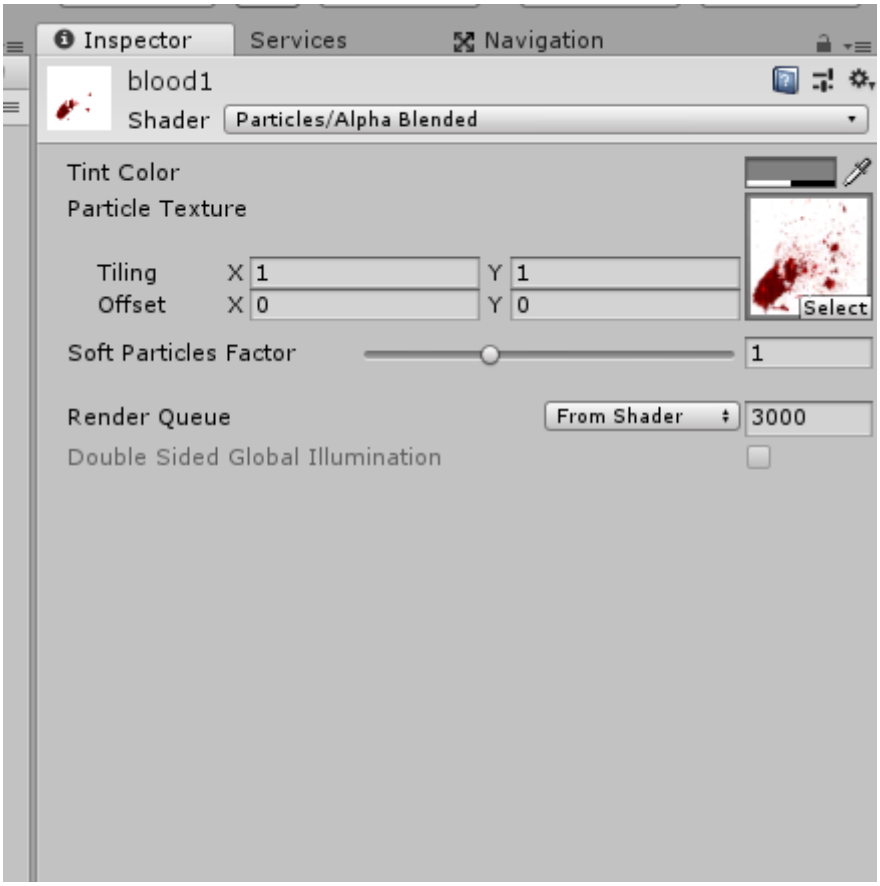
Θα περιηγηθούμε στην ιστολσελίδα www.freeiconspng.com και θα κατεβάσουμε τρεις φωτογραφίες τύπου png με εφφε το αίμα του εχθρού μας.



Θα δημιουργήσουμε ένα φάκελο BloodEffects στα Assets μας και θα με drag n drop θα τα εισάγουμε στο φάκελο αυτό.

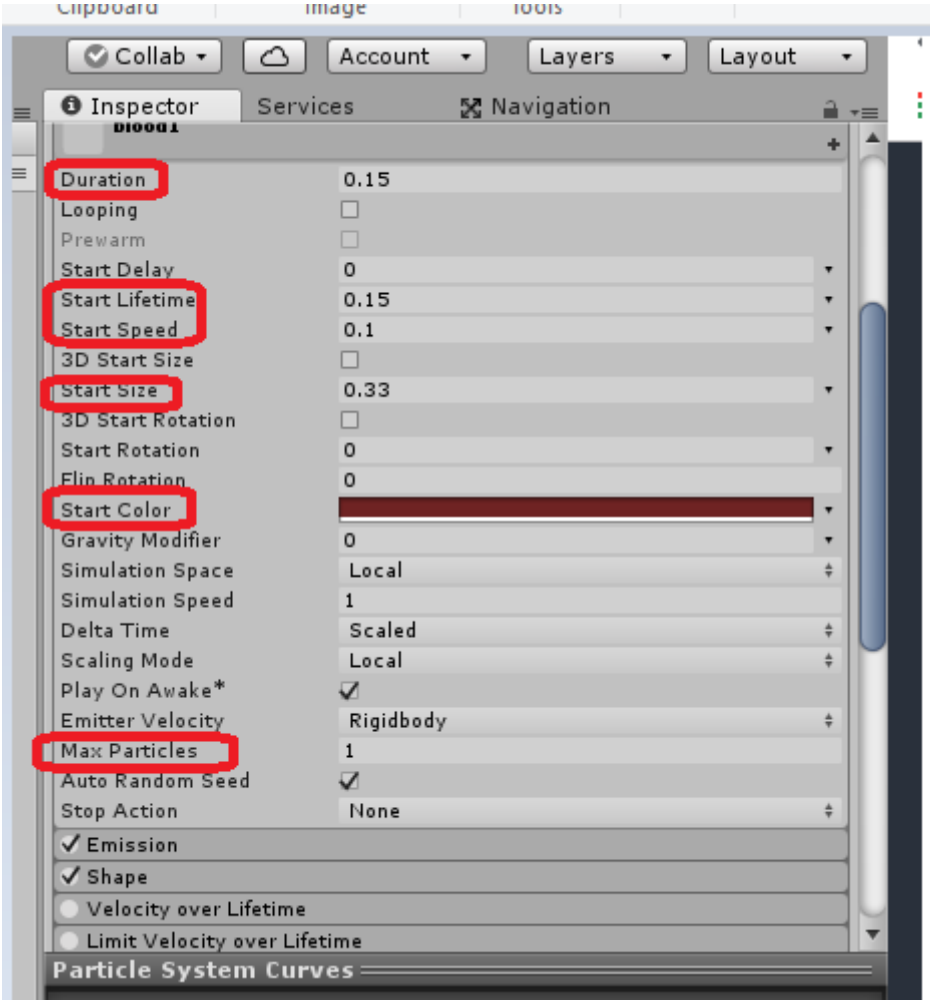


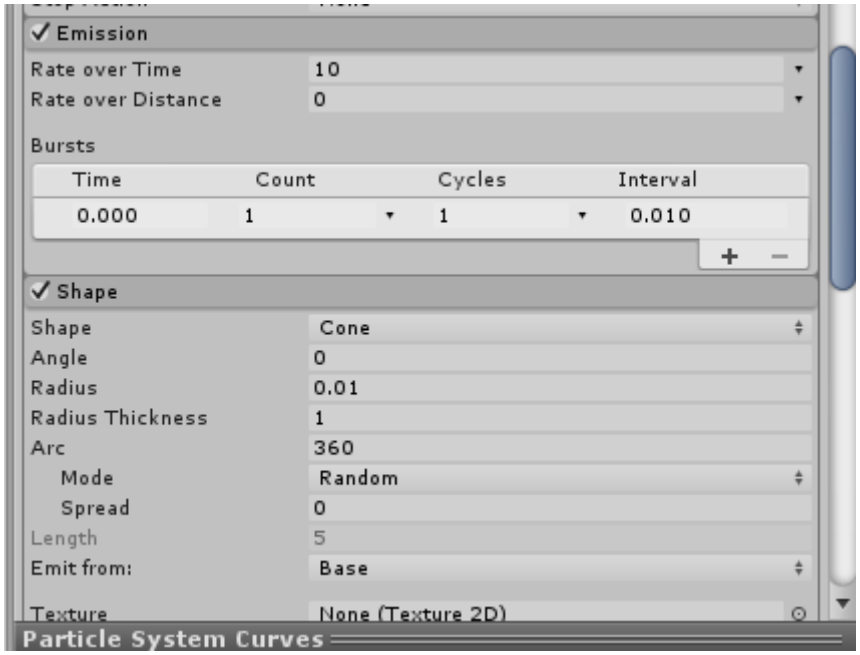
Η επόμενη κίνηση μας είναι να δημιουργήσουμε το material. Δεξί κλικ Create → Material Shader = Particles/Alpha Belended και στο κουμπί select θα εισάγουμε τη φώτο μας. Duplicate Ctrl + D και στο κάθε material θα προσθέσουμε την εκάστε εικόνα.



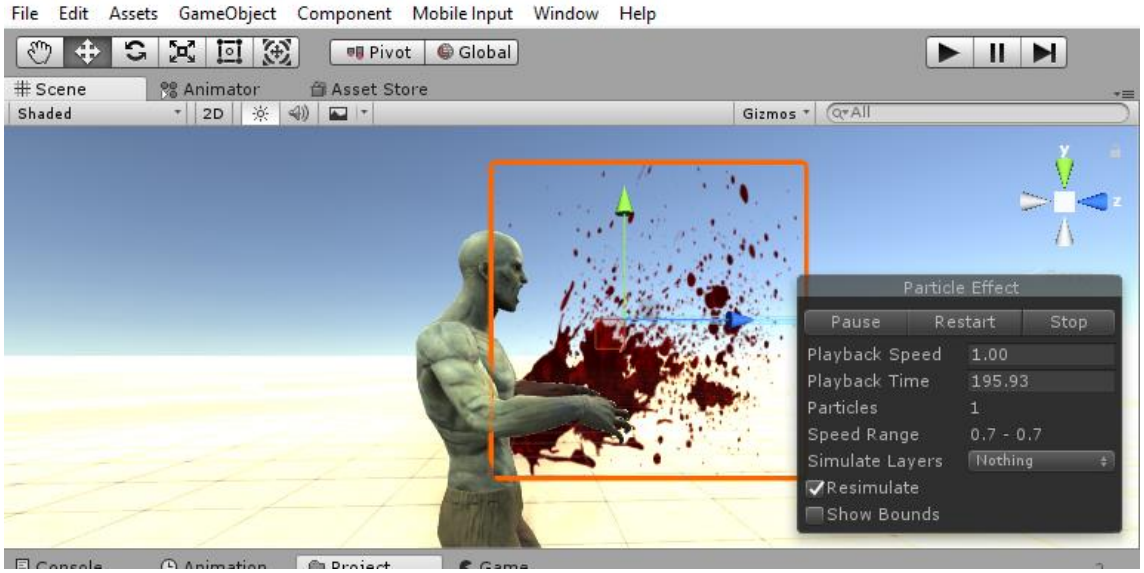
Πάμε στον εχθρό μας και δημιουργούμε ένα κενό GO το ονομάζουμε Blood1 και το κάνουμε παιδί του Zombie. Δεξί κλικ και κλικάρουμε Particle System. Παίρνουμε το material blood1 από τα assets μας και το ρίχνουμε drag n drop στο Particle System. Πειράζουμε τις αντίστοιχες τιμές όπως δείχνει η εικόνα παράτω.

Την ίδια διαδικασία ακολουθούμε και για τα 2 επόμενα material.





Βγάζουμε το GO με τα επισυναπτόμενα materials από το Zombie και το ρίχνουμε στα Assets μας και το κάνουμε prefab. Αφού το κάμνουμε prefab το διαγράφουμε από το scene μας.



Στη συνέχεια θα γράψουμε το script μας για να έχει λειτουργικότητα το Particle system του εχθρού που μόλις δημιουργήσαμε.

Ο κώδικας θα ενσωματωθεί στο script GunShoot.cs. Το script αναλύεται περεταίρω στην επόμενη ενότητα που δημιουργούμε τον Player. Ας δούμε το reference για το prefab bloodZombie.

```
public GameObject _hitMarkerPrefabZombie;
// private GameObject _impactEffectCrate;
```

Δηλώνουμε τοπική μεταβλητή ένα GameObject με όνομα bloodGO. Στην συνέχεια, εξισώνουμε αυτό το GO με το αποτέλεσμα της μεθόδου Instantiate, το οποίο αποτέλεσμα το δίνουμε σαν "GameObject" στο bloodGO. Αυτό ακριβώς κάνει η εντολή **as GameObject** στο τέλος του κώδικα. «**Πάρε αυτό το αποτέλεσμα, σαν να είναι GameObject**.

Η Instantiate είναι μια μέθοδος, η οποία δημιουργεί (Instantiates) GameObjects στο runtime. Δέχεται 3 ορίσματα:

1. ένα **GameObject** (αυτό που θα δημιουργήσει),
2. ένα **Vector3** το οποίο θα είναι η αρχική θέση του gameobject, και ένα
3. **Quaternion** πρόκειται για μια μεταβλητή που εκφράζει περιστροφή στον 3D χώρο.

Το Quaternion μπορεί να εκφραστεί με την βοήθεια πινάκων γραμμικής άλγεβρας και εκφράζει την περιστροφή ενός αντικειμένου ως προς το X (Yaw), το Y (Pitch), το Z (Roll) καθώς και το W, ένα νοητό διάνυσμα από την κάμερα προς το GameObject που περιστρέφεται.

Οπότε ας δούμε τι ακριβώς κάνουμε: Instatiate (Bullet, transform.position, Quaternion.identity) Δηλαδή λέμε στην θέση που βρίσκεται ο ήρωας μας (transform.position), με περιστροφή ακριβώς όπως είναι στο prefab (Quaternion.LookRotation), δημιούργησε μου ένα GameObject που είναι τύπου hitMarkerPrefabZombie.

```
}
GameObject bloodGO = Instantiate(_hitMarkerPrefabZombie, hitInfo.point, Quaternion.LookRotation(hitInfo.normal)) as GameObject;
```

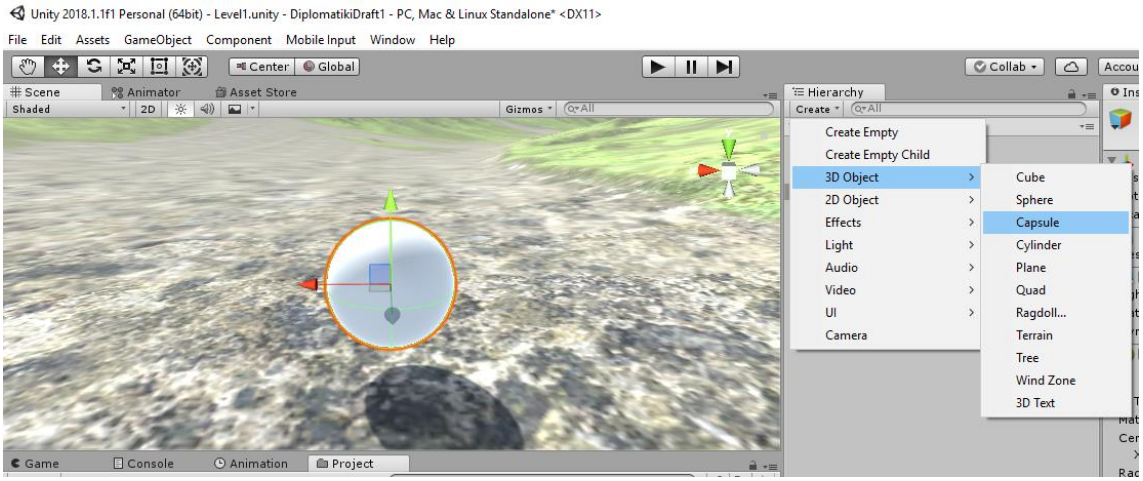
5.5. ΔΗΜΙΟΥΡΓΙΑ ΟΝΤΟΤΗΤΑΣ – ΚΕΝΤΡΙΚΟΣ ΗΡΩΑΣ

5.5.1 Μηχανισμός Κίνησης Του Παίχτη

Σε αυτήν την υποενότητα θα ασχοληθούμε με την κλάση Input της Unity, η οποία μας επιτρέπει να γράψουμε κώδικα που αφορά τον χειρισμό του παιχνιδιού.

Πάμε να φτιάξουμε ένα 3D αντικείμενο Capsule όπως φαίνεται στην εικόνα 3.1. Αυτό το αντικείμενο θα είναι και ο παίχτης μας.θα κάνουμε Rename και θα τον ονομάσουμε Player.

Εικόνα 3.1 Δημιουργία 3d Capsule(Παίχτης)



5.1.2 Κίνηση μέσω της Input.GetAxis() και του Time.deltaTime

Ας ξεκινήσουμε με την κίνηση του παίχτη μας. Πάμε στον φάκελο “Scripts” να φτιάξουμε ένα C# script “Player.cs”. Στην συνέχεια, θα ανοίξουμε το Script και θα γράψουμε τον κώδικα που φαίνεται στην εικόνα 3.1.1

Η ανάλυση του κώδικα γίνεται μέσω comments οπως θα παρατηρήσετε. Τα comments εχουν πράσινο χρώμα και περιγράφουν λεπτομερώς το νόημα της κάθε γραμμής. Είναι γραμμένος στα Αγγλικά αφού η όλη εργασία βασίστηκε επι των πλείστων σε documentation Scripting της Unity.

Παρ' όλα αυτά θα περιγράψουμε κάποια σημεία του κώδικα στα ελληνικά για την καλύτερη κατανόηση από την πλευρα του αναγνώστη μας.

_controller = GetComponent<CharacterController>();

Αποκτάμε ένα reference στο “Character Controller” Component μέσω της εντολής GetComponent<>. Αυτή η εντολή μας δίνει πρόσβαση στο CharacterController component του τρέχοντος game object που τρέχει το συγκεκριμένο script.

Όταν θέλω να μετακινήσω αντικείμενα στο 3D χώρο χρησιμοποιώ τη μέθοδο GetAxis της κλάσης Input.

Για παράδειγμα, μπορώ να πω ότι το Horizontal σημαίνει «Δεξί Βελάκι» και “D” ως προς τα θετικά και «Αριστερό Βελάκι» και“A” ως προς τα αρνητικά. Εάν πατήσω το D, ας πούμε, η

εντολή Input.GetAxis("Horizontal") θα μου επιστρέψει την θετική τιμή και την οποία μπορώ να πολλαπλασιάσω με το Speed που θα έχω ορίσει και το Time.deltatime.

Αντιθέτως αν πατήσω A, θα επιστρέψει αρνητικό sensitivity και θα περπατήσει οχαρακτήρας μου προς την αντίθετη πλευρά. Εάν δεν πατήσω τίποτα, θα επιστρέψειμηδέν. Έτσι, δεν θα έχω κίνηση.

.

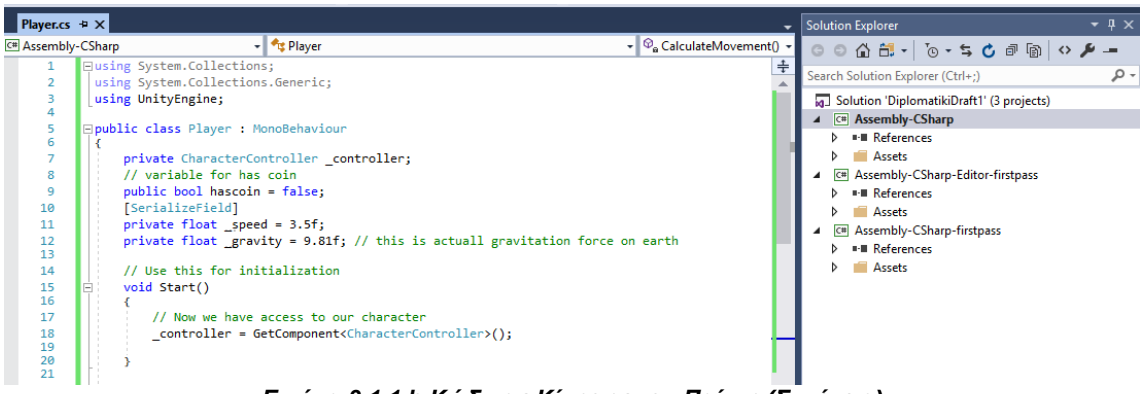
Vector3 direction = new Vector3(horizontalInput, 0, verticalInput); // we have the direction where to move

Έχουμε δημιουργήσει (μέσω της λέξης κλειδί “new”) μια μεταβλητή τύπου Vector3, πρόκειται για ένα διάνυσμα στον 3D χώρο, το οποίο θα το χρησιμοποιήσουμε για να εκφράσουμε την κατεύθυνση). Η Vector3 μεταβλητή δέχεται 3 παραμέτρους όταν δημιουργείται μια για την X μια για την Y και μία για την Z.

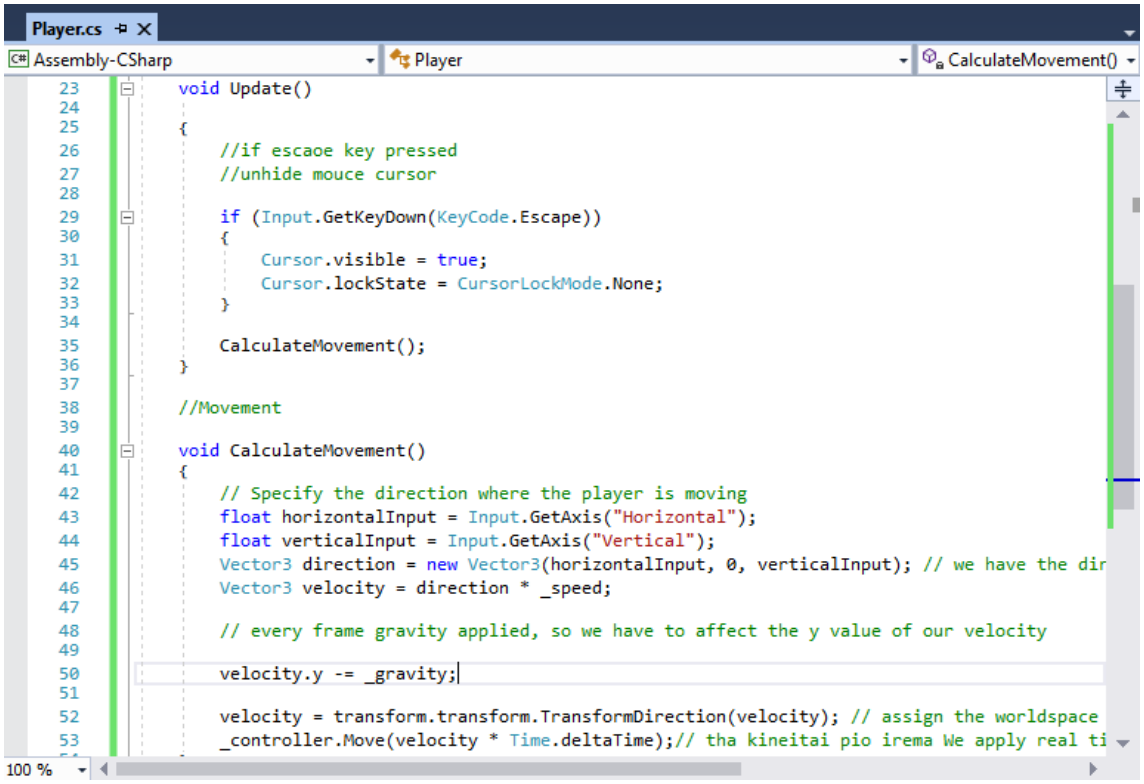
Η Time.deltaTime είναι μια μεταβλητή της κλάσης Time, που εκφράζει τον γραμμικό χρόνο που έχει

περάσει από την στιγμή που τρέχει η εφαρμογή!

Εικόνα 3.1.1.a Κώδικας Κίνησης του Παίχτη



Εικόνα 3.1.1.b Κώδικας Κίνησης του Παίχτη (Συνέχεια)




```
23      void Update()
24
25      {
26          //if escaoe key pressed
27          //unhide mouce cursor
28
29          if (Input.GetKeyDown(KeyCode.Escape))
30          {
31              Cursor.visible = true;
32              Cursor.lockState = CursorLockMode.None;
33          }
34
35          CalculateMovement();
36      }
37
38      //Movement
39
40      void CalculateMovement()
41      {
42          // Specify the direction where the player is moving
43          float horizontalInput = Input.GetAxis("Horizontal");
44          float verticalInput = Input.GetAxis("Vertical");
45          Vector3 direction = new Vector3(horizontalInput, 0, verticalInput); // we have the dir
46          Vector3 velocity = direction * _speed;
47
48          // every frame gravity applied, so we have to affect the y value of our velocity
49
50          velocity.y -= _gravity;
51
52          velocity = transform.transform.TransformDirection(velocity); // assign the worldspace
53          _controller.Move(velocity * Time.deltaTime);// tha kineitai pio irema We apply real ti
```

Εικόνα 3.1.1.c Κώδικας Κίνησης του Παίχτη(Συνέχεια)

```
//Movement

void CalculateMovement()
{
    // Specify the direction where the player is moving
    float horizontalInput = Input.GetAxis("Horizontal");
    float verticalInput = Input.GetAxis("Vertical");
    Vector3 direction = new Vector3(horizontalInput, 0, verticalInput); // we have the direction where to move
    Vector3 velocity = direction * _speed;

    // every frame gravity applied, so we have to affect the y value of our velocity

    velocity.y -= _gravity;

    velocity = transform.transform.TransformDirection(velocity); // assign the worldspace values to my velocity
    _controller.Move(velocity * Time.deltaTime);// tha kineitai pio irema We apply real time
}
```

Ας προσθέσουμε και μία επιπλέον κίνηση στον χαρακτήρα μας αυτόν του jump, Παρακάτω στην εικόνα 3.1.2 βλέπουμε τον κώδικα που γράψαμε.

Εικόνα 3.1.2.a Κώδικας Jump Character

```
//Variables for Jump
private float verticalVelocity;
//private float gravity = 14.0f;
private float jumpForce = 6.0f;
```

Ολοκληρώνουμε τον κώδικα προσθέτοντας τις παρακάτω γραμμές στη μέθοδο CalculateMovement() του Player.cs script

Εικόνα 3.1.2 b Κώδικας Jump Character

```
//Space key pressed then Jump
if (_controller.isGrounded)
{
    verticalVelocity = -_gravity * Time.deltaTime;
    if (Input.GetKeyDown(KeyCode.Space))
    {
        verticalVelocity = jumpForce;
    }
}
else
{
    verticalVelocity -= _gravity * Time.deltaTime;
}
```

5.2 Pivot Points

Στην Unity, αυτά θεωρούνται **VERTICES** ενός 3D model. Το σημείο το οποίο βρίσκεται στο κέντρο του GO, ονομάζεται Pivot. Το Pivot πρόκειται για το σημείο αναφοράς, βάσει του οποίου, δηλαδή, υπολογίζεται η θέση, γίνεται η περιστροφή και η μεγέθυνση. Καλές γραφιστικές πρακτικές είναι κάποιο αντικείμενο/3D model να έχει το pivot του σε κεντρικό σημείο για να γίνονται σωστά τα translations στον 3D χώρο. Εκτός αν το 3D μοντέλο είναι ένα ανθρωποειδές (Humanoid) με κάποιο σκελετικό σύστημα (Skeletal System), όπου τότε μια καλή πρακτική είναι το pivot point να βρίσκεται στο χαμηλότερο σημείο κάτω από τα πόδια του!

Προσοχή! Αφού γνωρίζουμε συνοπτικά τα Pivot Points ας λύσουμε το bug που συνατήσαμε.

Παρατήρησα λοιπόν ότι στην περίπτωση που επισυνάψουμε το script LookY στον Player θα διαπιστώσουμε σε Real Time Play οτι ο Player κινείται αντίθετα δηλαδή όταν πατάμε το D κατευθύνεται δεξια και αντίστοιχα συμβαίνει και με ταυπόλοιπα κουμπιά.

Για να το **διορθώσουμε** αυτό αλλάζουμε την τιμή στον άξονα y του transform position από 180 στο 0 στη main Camera.Τώρα ο παίχτης μας κινείται σύμφωνα με τη κατεύθυνσh που του δίνουμε όταν πατάμε τα αντίστοιχα κουμπάκια(A,D,W,S)

Υπάρχει ένα δεύτερο bug πρόβλημα**, όταν κοιτάει πάνω κινείται μπροστά, Οταν κοιτάει στο πάτωμα κινείται μπροστά.**Για αυτό το λόγο πρέπει να δημιουργήσουμε ένα είδος Pivot για τον παίχτη μας.

Δημιουρούμε ένα κενό GO ()CTRL +Shift+N) και το ονομάζουμε LookY. Το επισυνάπτουμε στον Player σαν Child Object και μετά επισυνάπτουμε τη Main Camera στο LookY. Τοποθετούμε το LookY Script στο GO.Με αυτή τη λύση μπορουμε να κοιτάμε πάνω και κάτω χωρίς να κινούμαστε μπροστά.

5.3. Πλοήγηση και Εισαγωγή Ορίων Μονοπατιού

5.3.1 Το Σύστημα Πλοήγησης (Explore the Navigation System in Unity And Use Navigation Mesh to constrain your Player)

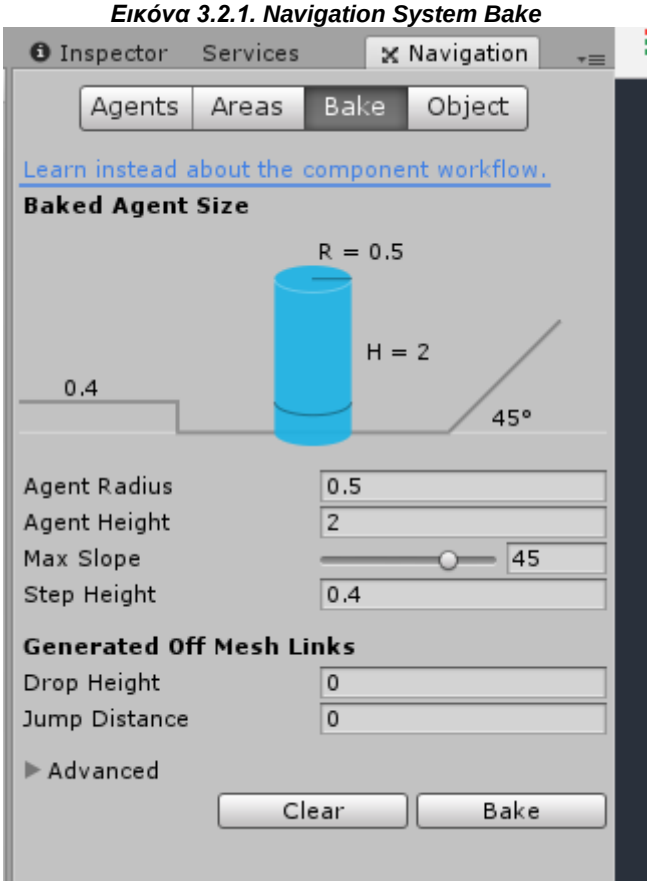
Το Navigation System μας επιτρέπει να δημιουργήσει χαρακτήρες και αντικείμενα που μπορούν να κινηθούν έξυπνα στον κόσμο του παιχνιδιού, γνωστοί ως πράκτορες (agents).

Το σύστημα πλοήγησης χρησιμοποιεί πλέγματα πλοήγησης τα οποία δημιουργούνται αυτόματα από τη γεωμετρία της σκηνής μέσω της διαδικασίας baking (ψησίματος). Δυναμικά εμπόδια τροποποιούν επίσης την πλοήγηση των πρακτόρων κατά το χρόνο εκτέλεσης της εφαρμογής.

Βάσει αυτής της δυνατότητας που μας προσφέρει η Unity3D, θα ρυθμίσουμε την επιφάνεια `πάνω στην οποία μπορεί να περπατήσει ο παίχτης καθώς και να περιορίσουμε το χώρο στον

οποίο μπορει να απομακρυνθεί. Έτσι αν πλοηγηθούμε στο τέος του Κόσμου ο παίχτης μας δε θα πέσει.
Πάμε Window → Navigation . Θα μας ανοίξει το παρακάτω παράθυρο.Επιλέγουμε το environment στην περίπτωση μας το World. Και πατάμε Bake.

Εικόνα 3.2.1. Navigation System Bake



Τέλος μπορούμε να δημιουργήσουμε υποτυπώδη τεχνητή νοημοσύνη την οποία κατέχουν είτε αντικείμενα είτε χαρακτήρες. Δηλαδή, στην παρούσα διπλωματική, οι εχθροί έχουν ρυθμιστεί έτσι ώστε να κινούνται στο χώρο αυτόνομα, ικανοί να φτάνουν στον προορισμό που τους έχει δοθεί, διανύοντας το πιο γρήγορο μονοπάτι.

5.4 Μηχανισμός Πυροβολισμού

Σε αυτήν την
υποενότητα θα προσθέσουμε την ικανότητα στον Player να μπορεί να πυροβολεί! Αυτό θα το πετύχουμε βήμα βήμα.

Θα κάνουμε το όπλο παιδι του Main Camera ωστε όπου κοτάζουμε την κάμερα να βλέπουμε το όπλο μας. Τοποθετούμε το όπλο στον Player.Θα φτιάξουμε τα Clipping Panes από σε 0.1

Δημιουργούμε ένα script και το ονομάζουμε GunShoot.cs. Θα το επισυνάψουμε στο όπλο που του ήρωα.

Πάμε να ορίσουμε τις μεταβλητές που θα χρειαστούμε:

```
[SerializeField]
private float _damageEnemy = 10f; // variable damageEnemy
public float _range = 100f; // how far the bullets go
public float _shootingTimer;
public float _shootingCD = 0.4f; // time between shootings, shoots every 4 sec, this goes for pistol
```

Δημιουργούμε μια μέθοδο Shoot(); Μέσα στη μέθοδο θα καλέσουμε τη συνάρτηση Raycast();

```
(Physics.Raycast(rayOrigin, out hitInfo,_range))

 Debug.Log("Hit something" + hitInfo.transform.name);
```

Το **Raycast** είναι μια μέθοδος που μας βοηθάει να εντοπίσουμε τη ‘σύγκρουση’. Συγκεκριμένα ,στέλνει μια ακτίνα (αόρατη) με σημείο εκκίνησης από τον ήρωα μας δηλαδή την Camera που ειναι attached στον χαρακτήρα μας προς κατεύθυνση εκέι που κοιτάζει η Camera και εάν υπάρχει κάποιο αντικείμενο κατά τη διάρκεια αυτής της διαδρομής ‘συγκρούεται’ με αυτό και

μπορούμε κατ'αυτόν τον τρόπο να αναφερθούμε και να τροποποιήσουμε το αντικείμενο σύγκρουσης .
Για να εντοπιστεί αυτή η σύγκρουση πρέπει το αντικείμενο να έχει ως component έναν Collider , οπότε στη δυναμική δημιουργία των εχθρών θα προσθέσουμε και τους colliders.


```
// RayCasting
Ray rayOrigin = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
RaycastHit hitInfo;
```

Η μεταβλητή hitInfo αποθηκεύει την πληροφορία «Τι χτύπησαμε όταν εκτοξεύσαμε την αόρατη ακτίνα».
Πότε θα ρίχνουμε αυτή την ακτίνα? Όταν πατάμε το αριστερό κλικ του mouse.
Θυμίζουμε πως το "GetMouseButton" τρέχει συνέχεια, γιατ ελέγχει αν είναι πατημένο κάτι συγκεκριμένο ανά καρέ, ενώ το "GetMouseButtonDown" τρέχει μία φορά και επιστρέφει true την στιγμή που πατήσουμε ή αφήσουμε κάποιο κουμπί από το πληκτρολόγιο.

```
// Update is called once per frame
void Update()
{
    //    Shoot();
    //}
    if (Input.GetMouseButton(0) && currentAmmo > 0)
    {

        Shoot();
    }
```

Επίσης, στον έλεγχο λέμε ότι ο τρέχων χρόνος πρέπει να είναι μεγαλύτερος από το Cooldown πυροβολισμού που θα ορίσουμε εμείς στο 0.4 μέσω του inspector. Τέλος, έξω από την if παρατηρούμε ότι αυξάνουμε την μεταβλητή ShootingTimer κατά Time.deltaTime σε κάθε καρέ. Έτσι, ο ShootingTimer θα αυξάνεται σταθερά ανάλογα με την ώρα.

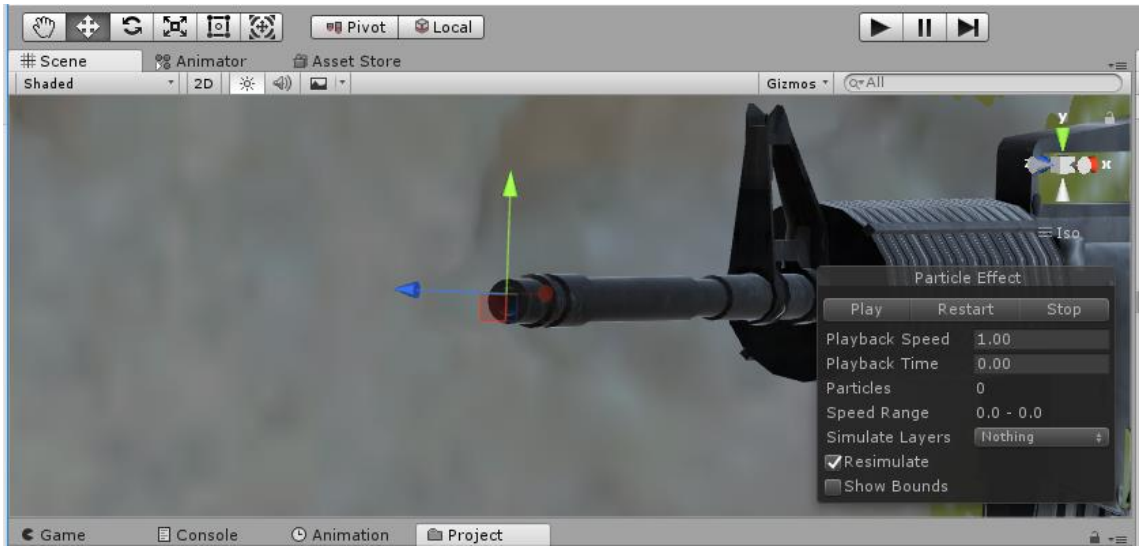
Σε 1 δευτερόλεπτο π.χ., η ShootingTimer θα είναι 1, σε 2 θα έχει τιμή 2 κτλ. Με την προϋπόθεση ότι μπήκαμε στην if, πρέπει οπωσδήποτε να μηδενίσουμε τον timer, έτσι ώστε να ξαναπάει μηδέν και να ξαναμετράει μέχρι να ξεπεράσει το 0.4.

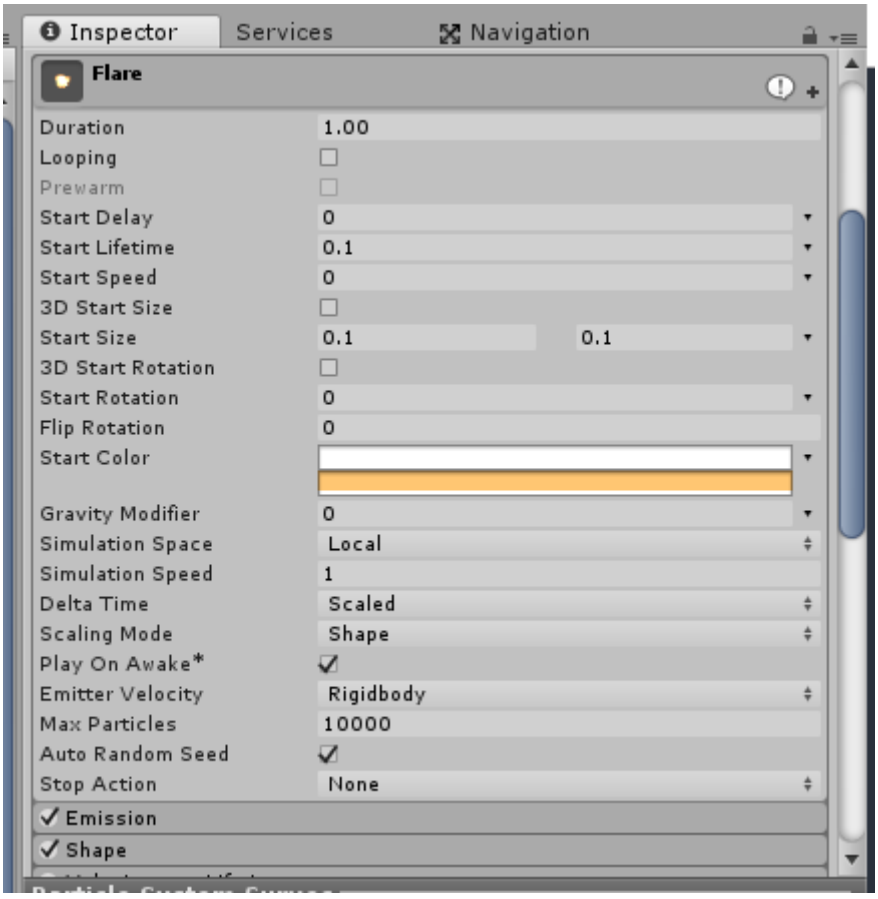
Έτσι, έχουμε καταφέρει να υλοποιήσουμε το Cooldown σύστημα.

5.5 Δημιουργία Εκπυρσοκρότησης (Muzzleflash) μέσω Particle System

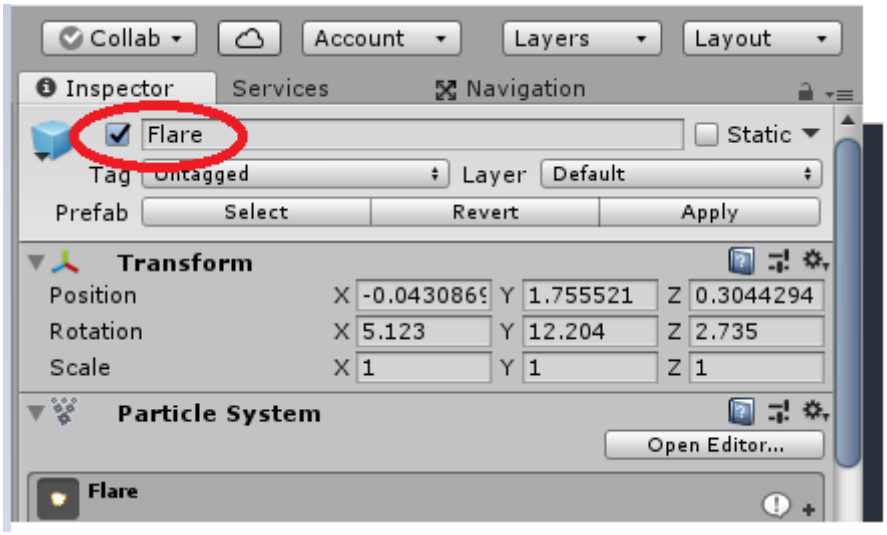
Εισάγουμε από τα standard Assets της Uinty και συγκεκριμένα από το Particles το εφφέ flare μέσα στο scene μας καο το τοποθετούμε στην άκρη της κάννης του όπλου μας.

Ορίζουμε τις τιμές στις παραμέτρους όπως φαίνεται στην παρακάτω εικόνα.





Η λογική πίσω από την λειτουργικότητα του Muzzleflase είναι η δημιουργία κώδικα που θα ενεργοποιεί το GO όταν θα πυροβολούμε και θα το απενργοποιεί στην αντίθετη περίπτωση.



Ορίζουμε λοιπόν μία μεταβλητή η οπόια θα είναι το reference για αυτό το GO(muzzleflash).

```
public ParticleSystem _muzzleFlash; // variable muzzleflash
```

Μέσα στη μέθοδο Shoot() θα ενεργοποιήσουμε το Muzzleflash

```
// Muzzleflash
_muzzleFlash.Play();
```

Αντιθέτως όταν δεν θα πατάμε το αριστερό MouseClick δε θα παίζει το Muzzleflash

```
if (Input.GetMouseButton(0) && currentAmmo > 0)
{

    Shoot();
}
else
{
    _muzzleFlash.Stop();
    // _muzzleFlash.SetActive(false);
    //  _weaponAudio.Stop();
    shootSound.Stop();

    // anim.SetBool("PFire", false);
```

5.6 Υλοποιήση Εφέ Πρόσκρουσης Σφαίρας (Hitmarker) μέσω Particle System

Ανοίγουμε το script GunShhot.cs και ορίζουμε μία νέα μεταβλητή που θα είναι reference στο prefab που θα κάνουμε instantiate.

```
[SerializeField]
private GameObject _hitMarkerPrefab; // variable Hitmarker
```

Έχουμε εξηγήσει σε προηγούμενη ενότητα τη λογική της μεθόδου instantate(); Θα αναλύσουμε όμως τη σημασία της μεθόδου **Dstroy(**shootingGO,3);

```
//Instantiate the the hitMarker
GameObject shootingGO = Instantiate(_hitMarkerPrefab, hitInfo.point, Quaternion.LookRotation(hitInfo.normal));
Destroy(shootingGO, 3);
```

Καθαρισμός των χρησιμοποιημένων αντικειμένων

Από την στιγμή που μια σφαίρα δεν θα βρει κάποιο από τα Zombie, θα εκτοξευθεί και θα παραμείνει για πάντα στο scene μας ή μπορεί να ταξιδεύει και για πάντα. Αυτό εμείς δεν το θέλουμε, καθώς κάθε σφαίρα που αστοχεί είτε από μας είτε από τους εχθρούς θα υπάρχει στην μνήμη. Οπότε θα πρέπει με κάποιο τρόπο να τις καταστρέφουμε.

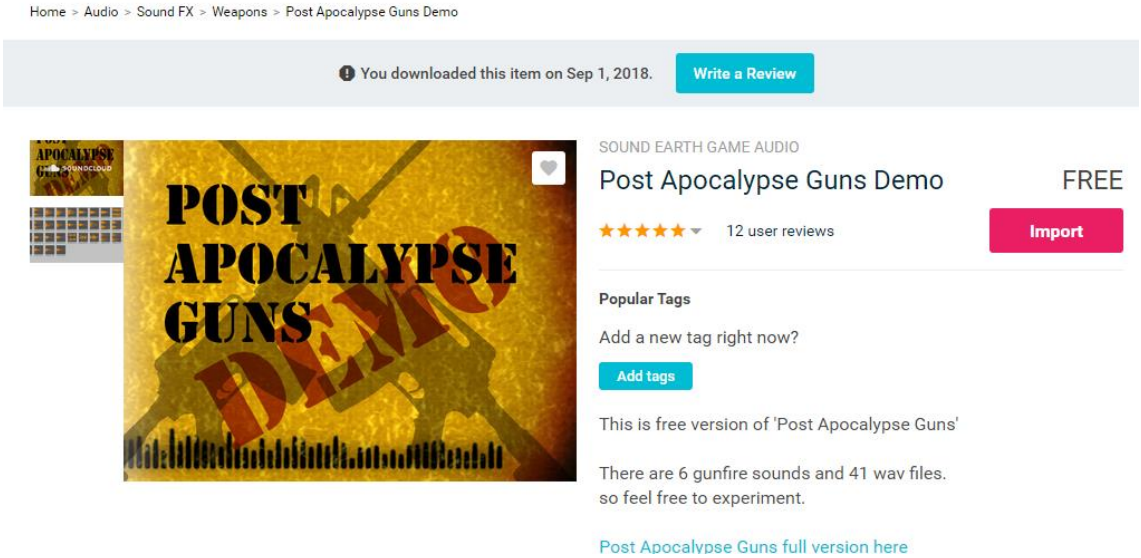
Η μέθοδος **Destroy** είναι η ακριβώς αντίθετη από την Instantiate. Πρακτικά της περνάμε σαν παράμετρο ένα GameObject που θέλουμε να καταστραφεί από το Scene. Στη συγκεκριμένη περνάμε δύο παραμέτρους

I. Την παράμετρο “shootingGO” η οποία κάνει την ίδια δουλειά με την μεταβλητή transform. Όπως η **transform** αναφέρεται στο Transform Component του GameObject που έχει το Script αυτό πάνω του, έτσι η **gameObject** αναφέρεται στο ίδιο το GameObject.

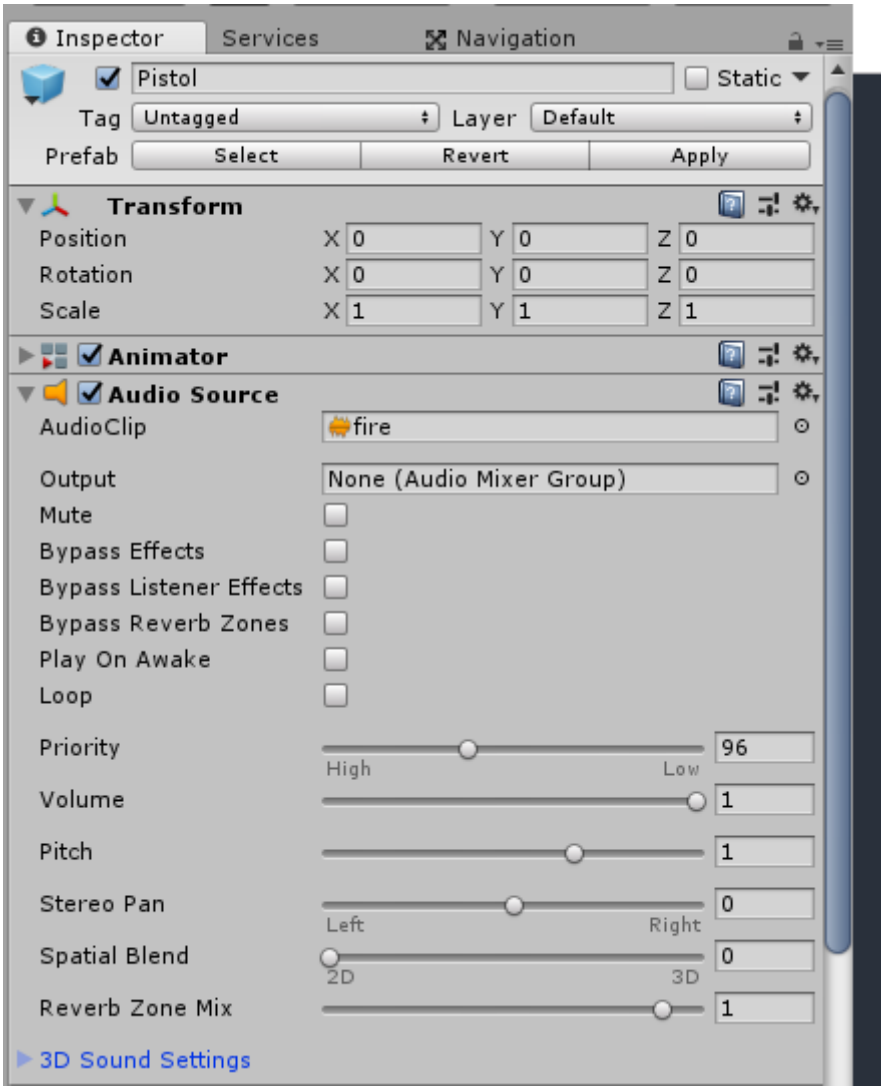
II. Την παράμετρο χρόνο, σε πόσο χρόνο θα καταστραφεί το GO.

5.7 Υλοποιήση Ηχητικών Εφέ του όπλου μας.

Το Asset Store είναι πολύτιμος βοηθός και σε αυτή την περίπτωση. Θα εισάγουμε ένα πακέτο ηχητικών εφέ εντελώς δωρεάν.



Θα επιλέξουμε απο το Hierarchy το όπλο μας και θα προσθέσουμε ένα Audio Source component. Με drag and drop θα ρίξουμε το Audio από το φάκελο που μόλις κατεβάσαμε μέσα το πεδίο AudioClip. Θα ξετικάρουμε την επιλογή Play on Awake και θα τικάρουμε το Loop.



Θέλουμε λοιπόν να ελέγχουμε πότε θα πάιζει το Audio Source όταν θα πατάμε αριστερό κλικ στο mouyse.Χρειαζόμαστε λγιαυτό ένα reference. Πάμε να γράψουμε το scriptaki μας.

Ορίζουμε τη μεταβλητή μας μέσα στο script GunShoot.cs

```
[SerializeField]
private AudioSource _weaponAudio; // variable refference to weapon Audio source

[SerializeField]
```

Μέσα στη μέθοδο Shoot() θα γράψουμε τον έλεγχο . Όταν δε θα πατάμε το αριστερό κλικ ο ήχος θα σταματλαει.

```
        //Audio
        if (shootSound.isPlaying == false)
        {
            shootSound.Play();
        }
        // Animation Playing
```

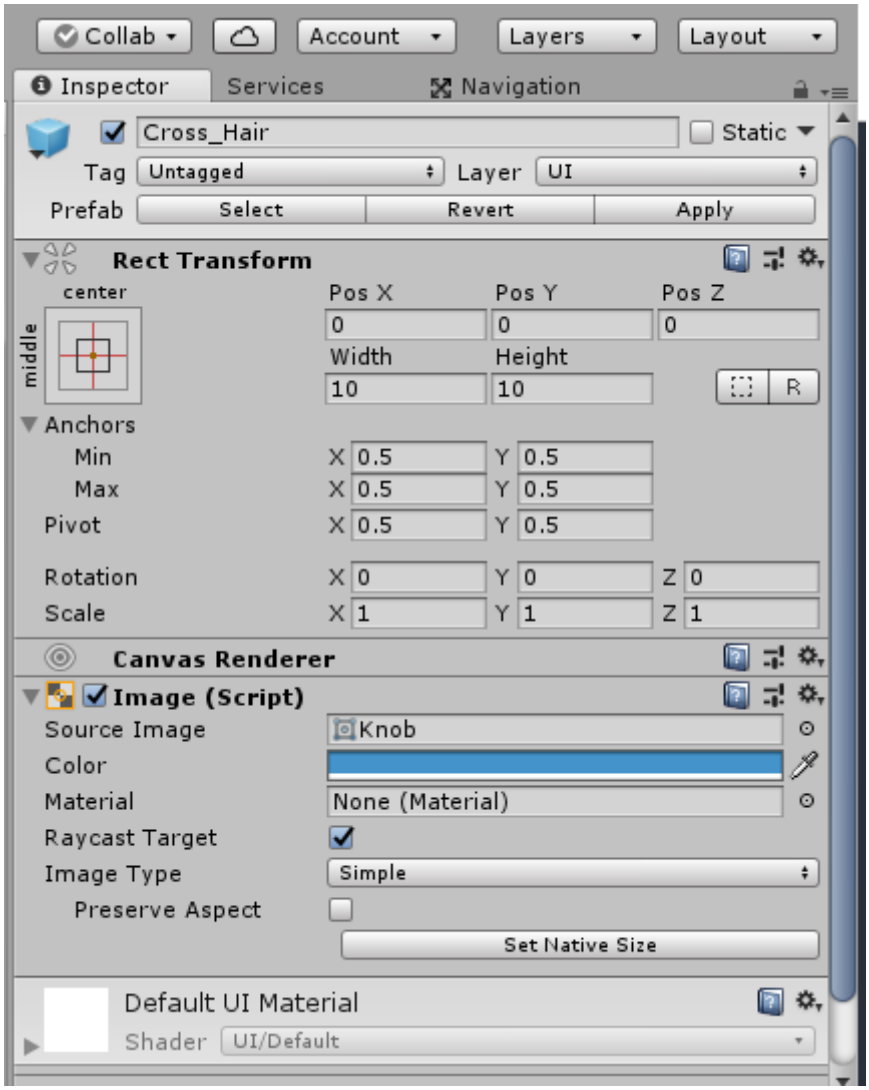
```
// Update is called once per frame
void Update()
{
    //    Shoot();
    //}
    if (Input.GetMouseButton(0) && currentAmmo > 0)
    {


        Shoot();
    }
    else
    {
        _muzzleFlash.Stop();
        // _muzzleFlash.SetActive(false);
        //  _weaponAudio.Stop();
        shootSound.Stop();

        // anim.SetBool("PFire", false);
```

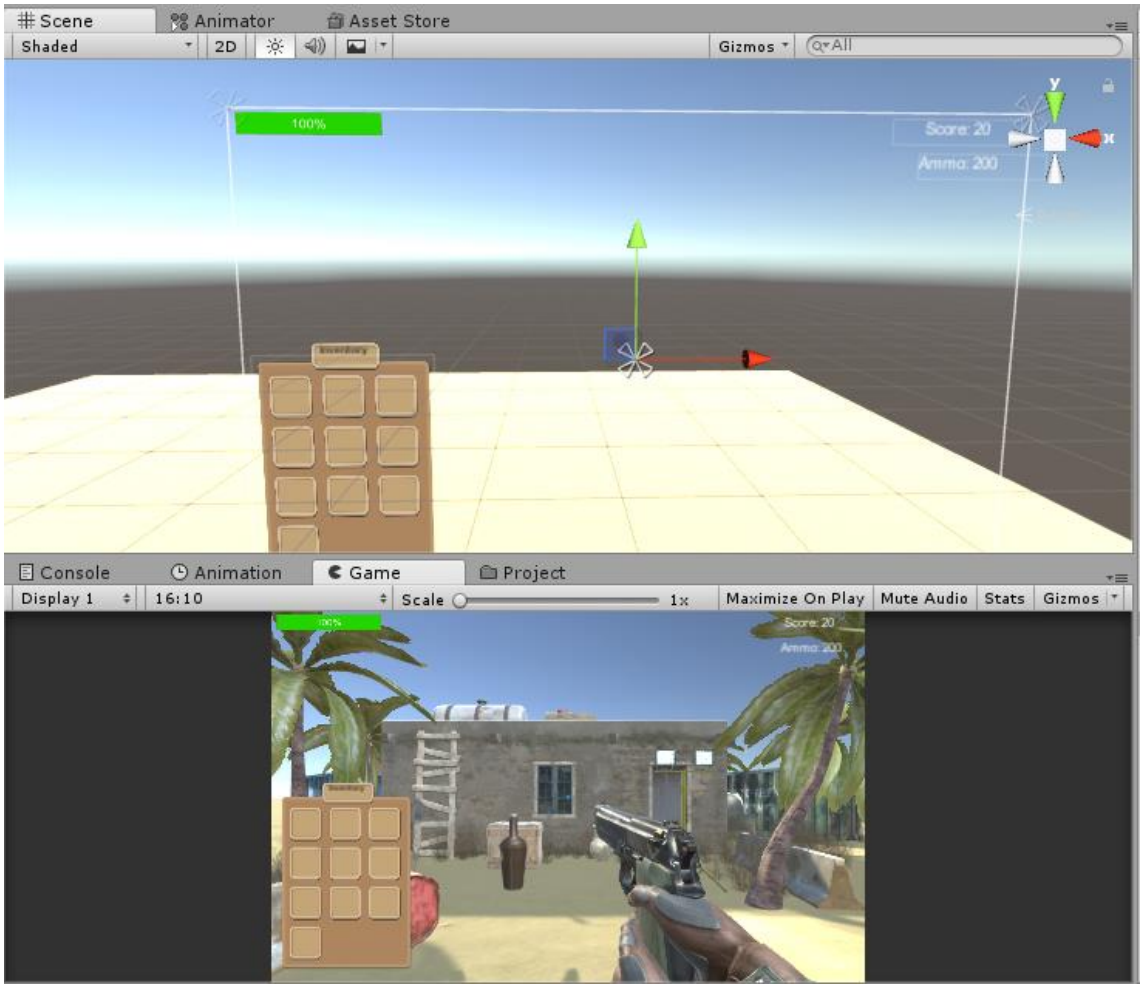
```
// Update is called once per frame
void Update()
{
    //    Shoot();
    //}
    if (Input.GetMouseButton(0) && currentAmmo > 0)
    {


        Shoot();
    }
    else
    {
        _muzzleFlash.Stop();
        //  muzzleFlash.SetActive(false);
        //  _weaponAudio.Stop();
        shootSound.Stop();

        // anim.SetBool("PFire", false);
```

5.8 Δημιουργία Στόχαστρο στο Όπλο - CrossHair.

Σε αυτή την ενότητα θα δημιουργήσουμε το Crosshair. Το Crosshair θα μας βοηθήσει να στοχεύουμε τον εχθρό.Για να το πετύχουμε αυτό θα στρέψουμε το όπλο μας στο κέντρο της οθόνης μας και εκεί θα δημιουργήσουμε το Crosshair.

Πάμε στο Hierarchy πατάμε δεξί κλικ → UI → Image. Στον inspector θα εισάγουμε το Sprite που θέλουμε, στη περίπτωση μας είναι το knob. Δίνουμε ένα χρώμα μπλε στό στόχο μας


Collab
Account
Layers
Layout
Inspector
Services
Navigation
Cross_Hair
Static
Tag
Untagged
Layer
UI
Prefab
Select
Revert
Apply
Rect Transform
center
middle
Pos X
Pos Y
Pos Z
0
0
0
Width
Height
10
10
R
Anchors
Min
X 0.5
Y 0.5
Max
X 0.5
Y 0.5
Pivot
X 0.5
Y 0.5
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Canvas Renderer
Image (Script)
Source Image
Knob
Color
Material
None (Material)
Raycast Target
Image Type
Simple
Preserve Aspect
Set Native Size
Default UI Material
Shader
UI/Default



Παρατηούμε όμως ένα πολύ σημαντικό πρόβλημα στο παιχνίδι μας. Ο κέρσορας είναι ορατός στο παιχνίδι μας. Όταν κινούμε τον παίχτη στα δεξια κινείται και ο κέρσορας.Αυτό δε το κάνει και πολύ λειτουργικό το παιχνίδ μας. Στόχος μας είναι όταν ξεκινάει το παιχνίδι ο κέρσορας να εξαφανίζεται.

Ανοίγουμε το script player.cs και μέσα θα γράψουμε τον κώδικα μας.

```
// Use this for initialization
void Start()
{

    //hide mouse cursor
    Cursor.visible = false;
    Cursor.lockState = CursorLockMode.Locked;

    // Now we have access to our character
    _controller = GetComponent<CharacterController>();


}
```

Στην περίπτωση που θα πατάμε το Escape key ο κέρσορας θα εμφανίζεται.

```
// Update is called once per frame
void Update()

{


    //if escape key pressed
    //unhide mouce cursor

    if (Input.GetKeyDown(KeyCode.Escape))
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }
```

5.9. Διεπαφές Χρήστη (User Interfaces)

Πρόκειται για όλων των ειδών τις πληροφορίες που μεταβιβάζονται στον παίχτη που παίζει το παιχνίδι και χρειάζεται να τις ξέρει για να έχει «τον έλεγχο». Για παράδειγμα, στο δικό μας παιχνίδι ο παίχτης θα ήταν καλό να γνωρίζει σε ποια περιοχή βρίσκεται (Area naming), πόση ενέργεια έχει, πόσους πόντους έχει, τι items έχει αποθηκεύσει στο inventory,πόσες σφαίρες του έχουν μείνει.

5.9.1. Τα βασικά του User Interface

Ας δούμε περιληπτικά όσον αφορά **μερικά UI Components**.

I. Canvas

Με το Canvas μπορούμε να ελέγξουμε πώς θα γίνουν render τα UI Elements μας. Κάθε UI Element πρέπει να είναι child του Canvas. Επίσης, είναι δυνατό να έχουμε παραπάνω από ένα Canvas στο Scene μας. Αν π.χ. πάμε να φτιάξουμε ένα Text ή ένα Button στην οθόνη χωρίς να υπάρχει Canvas, η Unity θα φροντίσει να δημιουργήσει ένα Canvas και θα μας φτιάξει μετά το Text ή το Button ή τέλος πάντων όποιο UI Element θέλουμε στην οθόνη μας, το οποίο και θα κάνει child στο Canvas αυτόματα.

II. Rect Transform

Το Transform Component στα 3D αντικείμενα μας, το οποίο περιέχει
Position, Rotation και Scale καθώς και Width, Height, X και Y.

III. Button

Πρόκειται ίσως για το πιο βασικό UI Element. Μπορεί να δεχτεί User Input και να τρέξει κάποιο event, το οποίο θα καλέσει συγκεκριμένο block κώδικα να εκτελεστεί. Ένα κουμπί μπορούμε να το χρησιμοποιήσουμε στο **main menu.**

IV. Text

Το Text είναι το δεύτερο πιο συνηθισμένο UI Element που θα συναντήσουμε και αφορά καθαρά κείμενο. Όταν εμείς θέλουμε να εμφανίσουμε κάποια γράμματα ή αριθμούς ή και τα 2 στην οθόνη, θα χρησιμοποιήσουμε το **Text UI Element**.

V. Event System

Κάθε UI Element μπορεί να έχει δικά του event και η δικιά μας δουλειά είναι να προσθέσουμε το σωστό Component που αναλαμβάνει αυτή τη δουλειά, να συνδέσουμε τα events και στη συνέχεια να καλέσουμε συγκεκριμένες μεθόδους από αυτά, οι οποίες πρέπει να είναι attached σε script στο ίδιο GameObject που θα έχει attached τα event Triggers.

Γνωρίζοντας αυτά, πάμε στα γρήγορα να φτιάξουμε το UI menu στην Unity!

5.9.2. Graphical User Interface

Ας σκεφτούμε ποιες πληροφορίες θέλουμε να εμφανίσουμε στην οθόνη και στη συνέχεια θα φτιάξουμε UI Elements για κάθε πληροφορία:

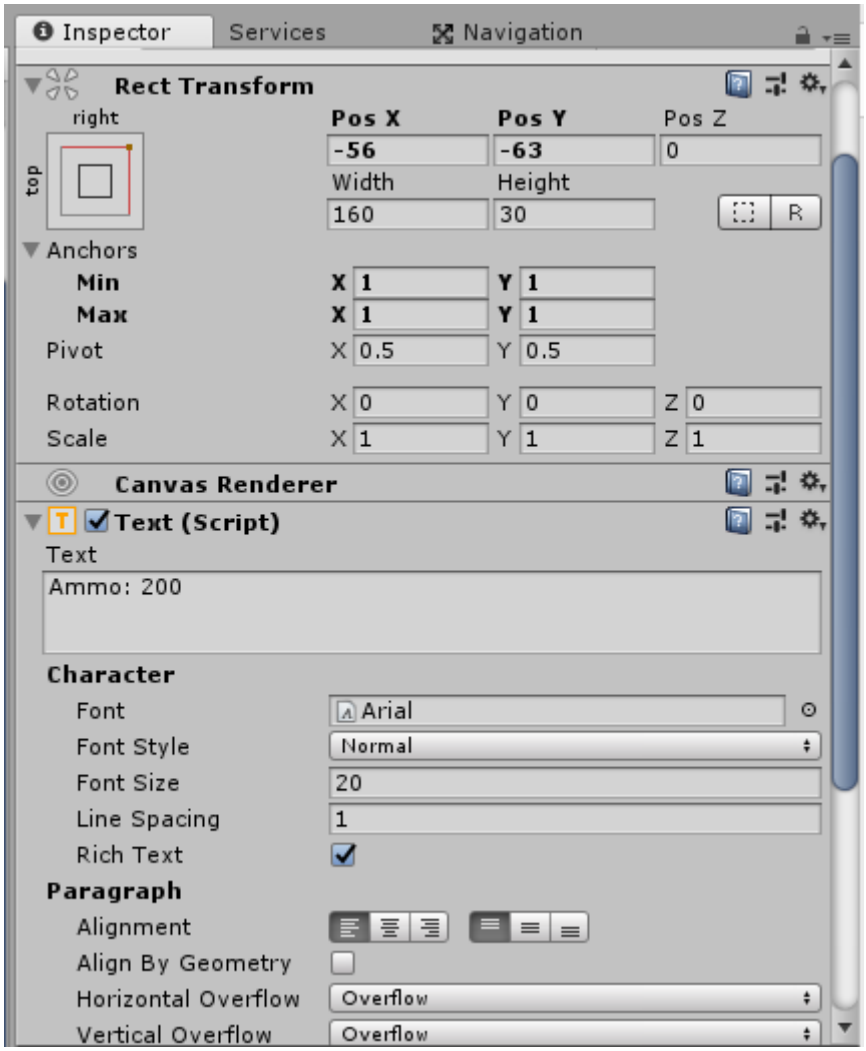
- Score
- Energy
- Ammunition

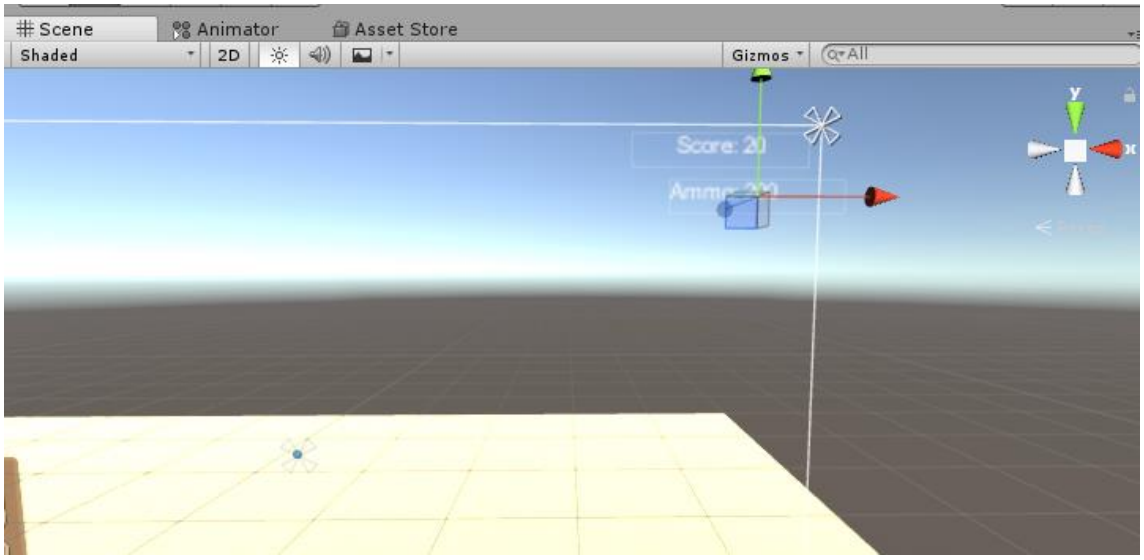
Συνεπώς όλο το GamePlay βασίζεται σε UI που θα έχει 4 κείμενα.

5.9.3 Graphical User Interface: Το πλάνο

Ξεκινάμε λοιπόν στην Unity δημιουργώντας το πρώτο μας Text (εικόνα 4.2). Θα δούμε ότι εμφανίστηκε στο κέντρο της οθόνης ένα κείμενο που γράφει “New Text” . Κατ’ αρχάς, θα του δώσουμε λευκό χρώμα στον Inspector, στο “Text” Component της Unity (ιδιότητα Color)και στη συνέχεια θα το μετακινήσουμε με το 2D Widget στο πάνω μέρος της οθόνης και δεξιά ώστε στο Game View να βλέπουμε αυτό που φαίνεται και στην εικόνα 4.2

Είναι σημαντικό να αγκιστρώσουμε ττην εικόνα επάνω και δεξιά με το Anchrs.Αυτό μας βοηθάει να κρατάει το ίδιο padding από την οθόμνη ανεξαρτήτως το resize που θα γίνεται όταν παίζουμε το παιχνίδι μας.





Στην συνέχεια, θα φτιάξουμε άλλα 2 κείμενα και θα τα τοποθετήσουμε στις γωνίες πάνω δεξιά και αριστερά. Επίσης, δεν θα ξεχάσουμε σε αυτά που είναι στις δεξιές πλευρές να έχουν Right Alignment και αυτά που είναι σε αριστερές να έχουν Left Alignment.

Ας δούμε πως φίνονται στο GameView στην παρακάτω εικόνα.

5.10. Υλοποιώντας την Ανανέωση του Score με C#.

Πάμε τώρα να φτιάξουμε το script που θα ανανεώνει το score μας κάθε φορά που θα σκοτώνουμε έναν εχθρο.

Θα ανοίξουμε το script **enemy.cs** και θα αναζητήσουμε τη μέθοδο DeductEnemyHealth().

Κάθε που ο παίχτης μας αφαιρεί ζωή από το Zombie θα ανανεώνεται το UI element Sore.Text.

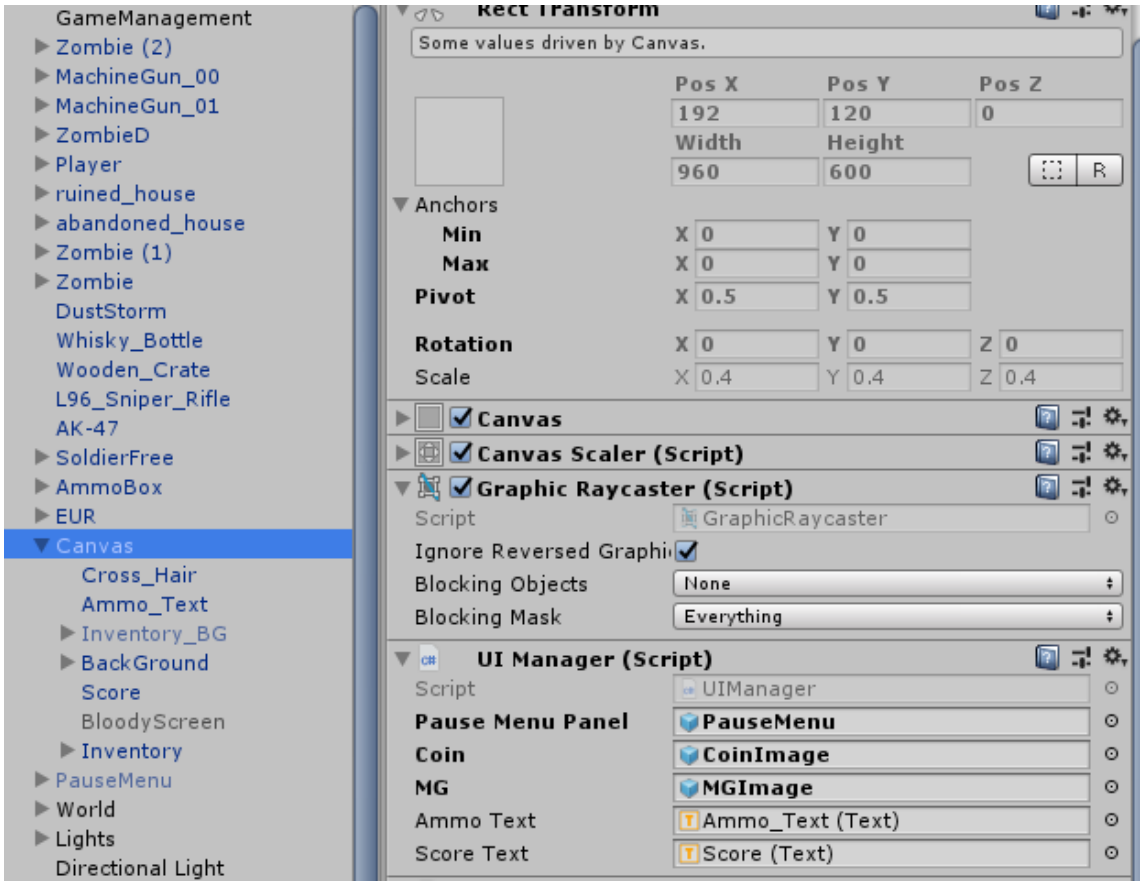
Αυτό το καταφέρνουμε με τη μέθοδο updadeScore. Τη μέθοδο αυτή την καλούμε αφού έχουμε ορίσει μία μεταβλητή _uiManager κλάσης UIManager.

```
UIManager _uiManager;
```

Στη συνέχεια μέσα στη μέθοδ Awake() αναζητούμε το component “Canvas” όπως δείχνει η παρακάτω εικόνα.

```
    // Use this for initialization
    void Awake () {
        _uiManager = GameObject.Find("Canvas").GetComponent<UIManager>();

public void DeductEnemyHealth(float deductHealth)
{
    enemyHealth -= deductHealth;
    bloodSound.Play();
    if (!isDead)
    {
        _uiManager.UpdateScore();
        if (enemyHealth <= 0)

        {
            EnemyDead();
        }
    }

}
```

Ποιά είναι η κλάση UIManager? Εχουμε δημιουργήσει ένα καινούριο script UIManager και το προσθετουμε σα component στον Canvas.



Στην παρακάτω εικόνα ορίζουμε τις απαραίτητες μεταβλητές για τα UI elements και στη συνέχεια γράφουμε 3 γραμμές κωδικα για th m;euodo UpdateScore().

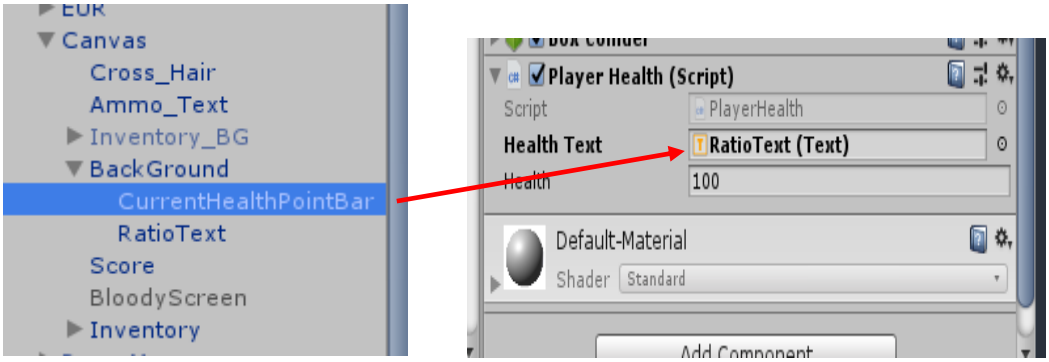
```
public class UIManager : MonoBehaviour
{
    [SerializeField]
    private GameObject _pauseMenuPanel;
    [SerializeField]
    private GameObject _coin;

    public GameObject _MG;
    // It is responsible for updating OnScreen elements
    [SerializeField]
    private Text _AmmoText;
    public Text scoreText;
    static int playerScore = 0;
    static int playerCash = 0;

    public void UpdateScore()
    {
        playerScore += 10;
        scoreText.text = "Score: " + playerScore;
    }
```

5.11. Υλοποιώντας την ανανέωση της Ενεργειακής μπάρας με C#;

Θα ξεκινήσουμε φτιάχνοντας ένα καινούριο script PlayerHealht.cs. Ορίζουμε δύο μεταβλητές, η πρωτη θα πάρει ένα refenrece που θα είναι το Text για Health και η δεύερη θα είναι η απόλυτη τιμή του παίχτη που θα μας δείχνει την ενέργεια του.

```
public class PlayerHealth : MonoBehaviour {

    public Text healthText;
    public int health;


    // Use this for initialization
    void Start ()
    {
        health = 100;
        //eference to the Image Healthbar

        // private ena script
    }

    // Update is called once per frame
    void Update ()

    {
        if (health<=0)
        {
            SceneManager.LoadScene("GameOver");
            Cursor.visible = true;
            Cursor.lockState = CursorLockMode.None;
        }
    }
}
```

Στη συνέχεια θα κάνουμε drag and drop στο reference το currentHealthPointBar από το Canvas.



Πηγαίνουνε στο script CollisionWithPlayer.cs και φτίαχνουμε μια καινούρια μεταβλητή τύπου PlayerHealth.

```
private PlayerHealth playerHealthscriptvar;
```

Στη συνέχεια με τις 3 παρακάτω γραμμές θα συνδέσουμε τα 2 script μεταξύ τους, πάρα πολυ χρήσιμος κώδικας!

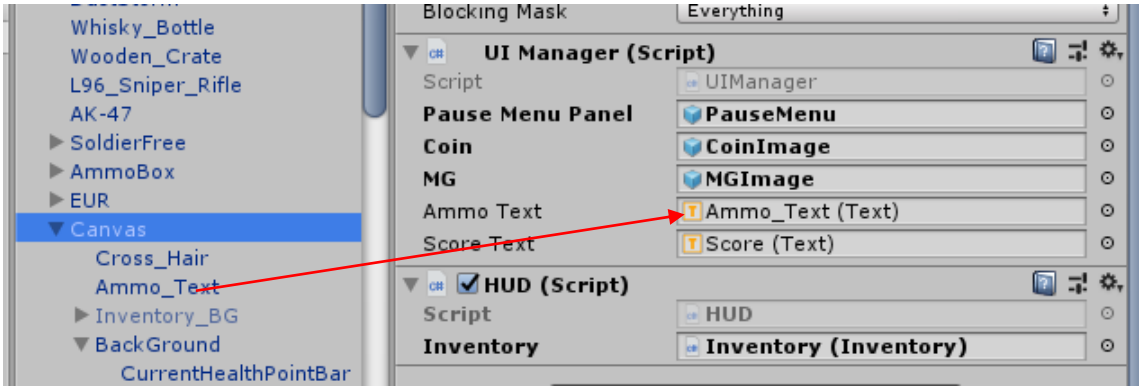
```
// with the 3 lines below we have a connection between the two scripts this and PlayerHealth
GameObject PlayerHealthObject = GameObject.FindWithTag("Player");

if (PlayerHealthObject != null)
{
    playerHealthscriptvar = PlayerHealthObject.GetComponent<PlayerHealth>();
}
AudioSource[] audios = GetComponents<AudioSource>();
```

```
int deductedhealth = playerHealthscriptvar.health -= 5;
Text healthComingFromScript = playerHealthscriptvar.healthText;

string stringHealth = (deductedhealth).ToString();
healthComingFromScript.text = "" + stringHealth;
```

5.12 Υλοποιώντας την ανανέωση του Ammunition Text με C#

Ξεκινάμε πάλι από την κλάση UIManage.cs και φτιάχνουμε μια μεταβλητή που θα χρησιμεύσει ως reference στο Ammunition UI element μας. Στη συνέχεια θα κάνουμε grag and drop όπως φαίνεται στην παρλατω εικόνα.

```
[SerializeField]
private Text _AmmoText;
```



Ορίζουμε μία νεά μέθοδο την UpdateAmmo()

```
public void UpdateAmmo(int count)
{
    _AmmoText.text = "Ammo: " + count;
}
```

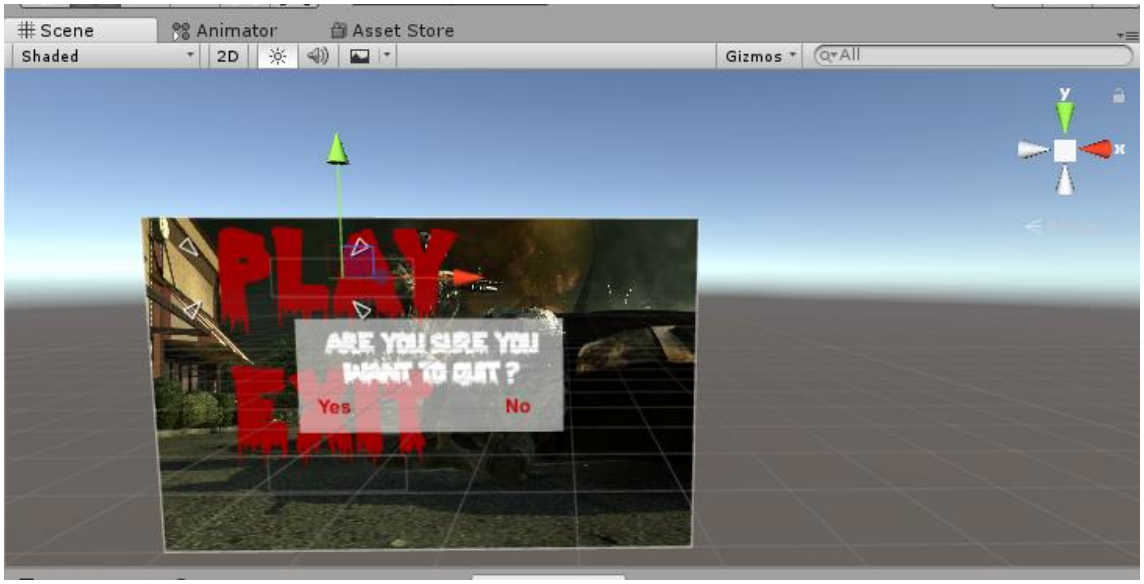
Και στο script GunShoot.cs θα αναζητήσουμε τη συνάρτηση Shoot() όπου θα γράψουμε την παρακάτω γραμμή.

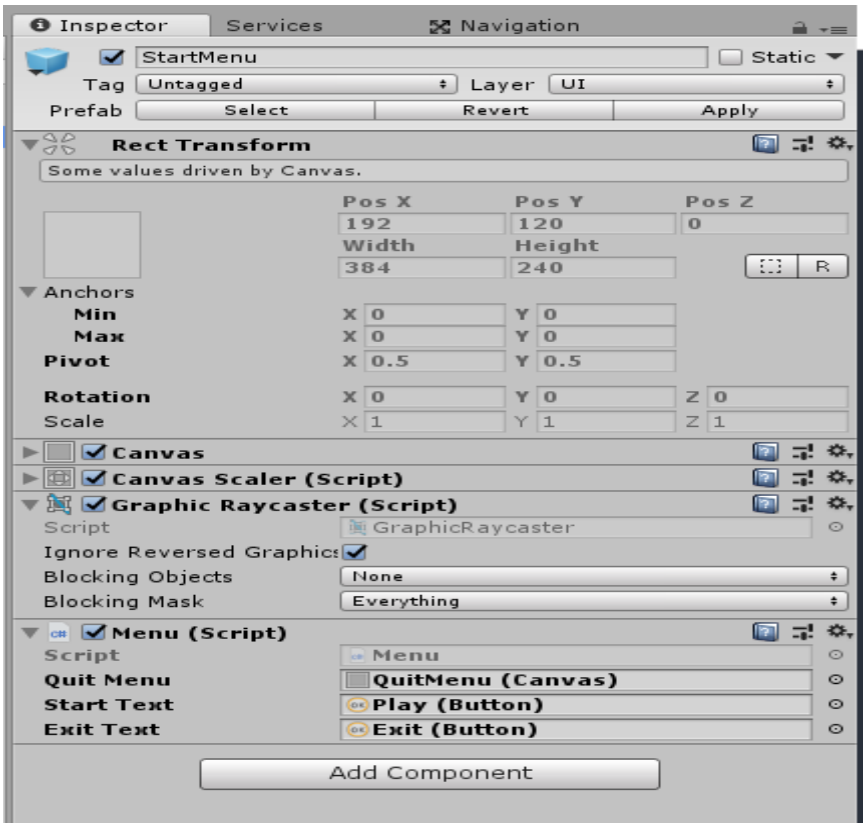
```
void Shoot()
{
            currentAmmo--;
            _uiManager.UpdateAmmo(currentAmmo);
```

5.13. Υλοποιώντας το Start, Pause και Game Over menu

Στο παιχνίδι αυτό έχουμε δημιουργήσει τρία διαφορετικά menu, το αρχικό μενού (Start), το μενού παύσης (Pause)και το μενού θανάτου (Game Over).

Κοινό χαρακτηριστικό και των τριών είναι η χρήση κουμπιού (Button) με τον ίδιο μηχανισμό και με τα ίδια properties . Δηλαδή, το αντικείμενο κουμπί χρησιμοποιείται με διαφορετικά ονόματα. Αυτό επιτυγχάνεται μέσω του κοινού script ButtonScript, που ανάλογα με την ονομασία του κουμπιού στο οποίο είναι επικολλημένο εκτελεί και την αντίστοιχη ενέργεια. Οι ενέργειες που μπορεί να εκτελέσει κάποιο κουμπί αναλόγως της ονομασίας του είναι είτε αλλαγή σκηνής είτε εμφάνιση υπομενού.


Scene
Animator
Asset Store
Shaded
2D
Gizmos
All
PLAY
ARE YOU SURE YOU WANT TO QUIT ?
Yes
No
EXIT

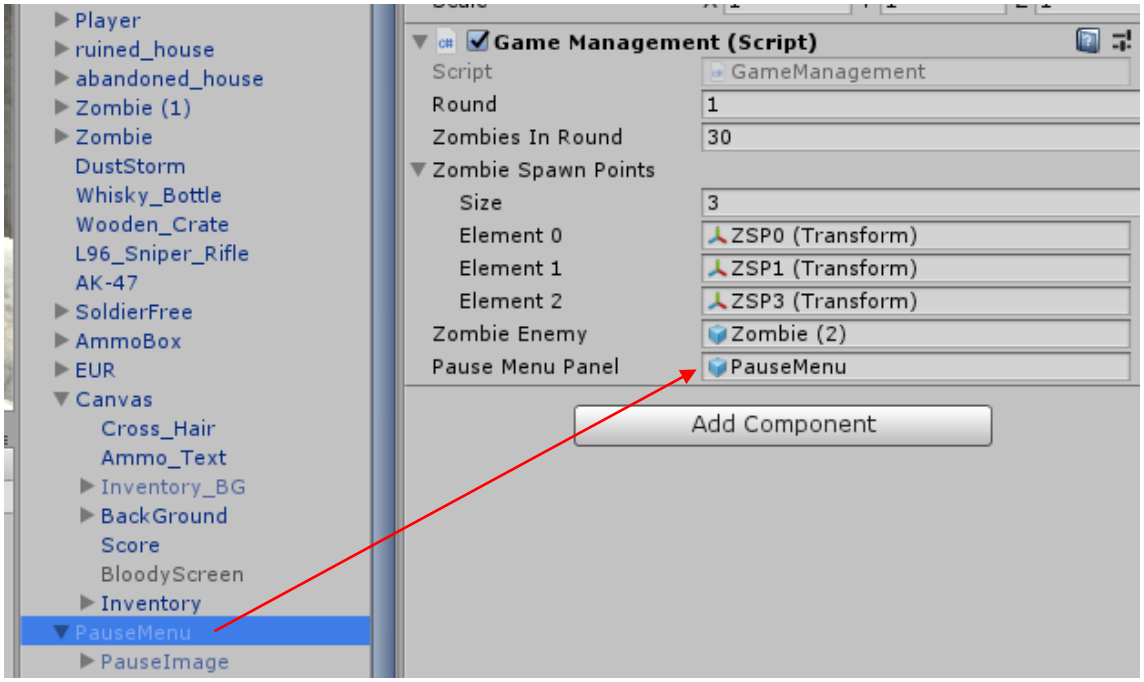

Inspector
Services
Navigation
StartMenu
Static
Tag
Untagged
Layer
UI
Prefab
Select
Revert
Apply
Rect Transform
Some values driven by Canvas.
Pos X
Pos Y
Pos Z
192
120
0
Width
Height
384
240
Anchors
Min
Max
Pivot
Rotation
Scale
Canvas
Canvas Scaler (Script)
Graphic Raycaster (Script)
Script
GraphicRaycaster
Ignore Reversed Graphics
Blocking Objects
None
Blocking Mask
Everything
Menu (Script)
Menu
Quit Menu
QuitMenu (Canvas)
Start Text
Play (Button)
Exit Text
Exit (Button)
Add Component

Στη συνέχεια παραθέτουμε τον κώδικα που αφορά το τι θα συμβαίνει όταν θα είμαστε στο Start menu.

```
public class Menu : MonoBehaviour
{
    public Canvas quitMenu;
    public Button startText;
    public Button exitText;


    void Start()
    {
        quitMenu = quitMenu.GetComponent<Canvas>();
        startText = startText.GetComponent<Button>();
        exitText = exitText.GetComponent<Button>();
        quitMenu.enabled = false;
    }

    public void ExitPress()

    {
        quitMenu.enabled = true;
        startText.enabled = false;
        exitText.enabled = false;

    }
    //public void OnClickPlay()
    //{
    //   // Application.LoadLevel("LevelOne");
    //    SceneManager.LoadScene("LevelOne");
    //}

                public void NoPress()
                {
                    quitMenu.enabled = false;
                    startText.enabled = true;
                    exitText.enabled = true;
                }

                public void StartLevel()
                {
                    SceneManager.LoadScene(1);
                }
                public void ExitGame()
                {

                    Application.Quit();
                }
            }
```

Ας δούμε κα το Pause menu:

Θα φτιάψουμε ένα script GameManagent.cs και θα κάνObject variable στον inspector και θα του δώσουμε το το Text που αφορά το Pause Menu.



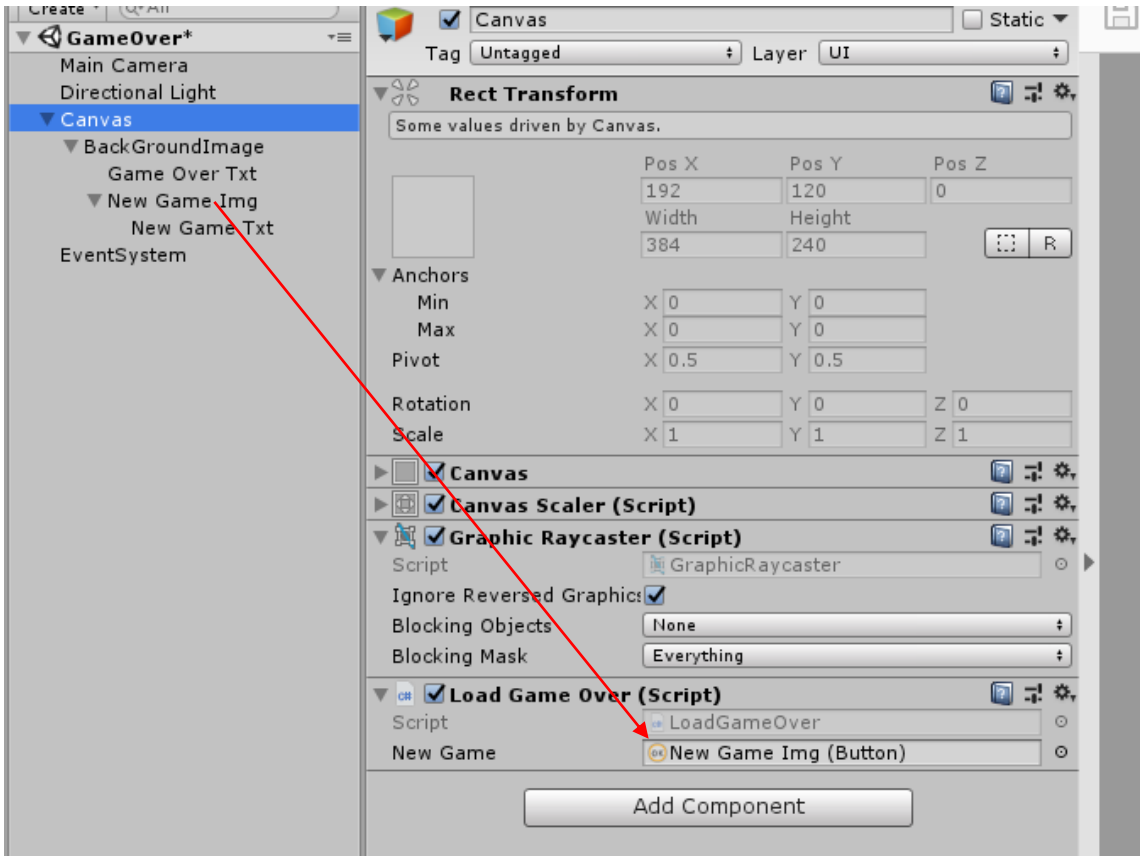
Όταν πατήσουμε το Key P το παιχνίδι ” παγώνει ”. Τότε μας εμφανίζεται το Pause menu όπως βλέπουμε παρακάτω.

Παρουσιάζουμε τον κώδικα που βοηθάει να κατανοήσουμε πως λειτουργεί το Pause menu.

```
// Update is called once per frame
void Update ()
{
    //if p key
    //pause game
    //enable pause menu

    if (Input.GetKeyDown(KeyCode.P))
    {
        _pauseMenuPanel.SetActive(true);

        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
        Time.timeScale = 0;
```

```
public void ResumeGame()
{
    _pauseMenuPanel.SetActive(false);
    Time.timeScale = 1;

}
```

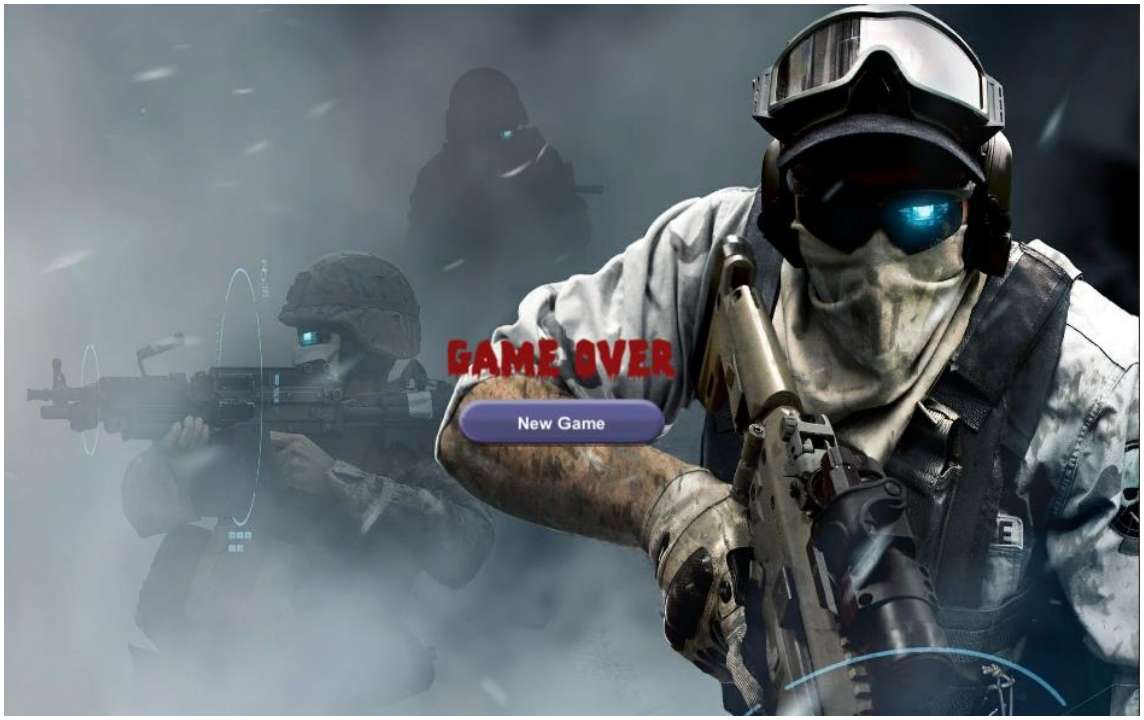
Ας κάνουμε κάτι τελευταίο τώρα. Έστω ότι θέλουμε να γράφει με μεγάλα γράμματα Game Over στο τέλος. Αυτό θα το πετύχουμε φτιάχνοντας ακόμα ένα UI Element τύπου Text, στο οποίο επίτηδες θα γράψουμε Game over.



Θα φτιάξουμε ένα καινούριο script και θα το ονομάσουμε LoadGameOver.cs

Η κλάση SceneManager περιέχει μία μέθοδο LoadScene η οποία παίρνει ως παραμετρο το ακριβές όνομα τοτ scene που θέλουμε να φορτώσουμε. Στην περίπτωση μας είναι το LevelOne.

```
You have mixed tabs and spaces. Fix this?
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5  using UnityEngine.UI;
6
7  public class LoadGameOver : MonoBehaviour {
8      public Button newGame;
9      // Use this for initialization
10     void Start () {
11         newGame.onClick.AddListener(loadGame);
12     }
13
14     void loadGame()
15     {
16         SceneManager.LoadScene("LevelOne");
17     }
18 }
19
```



ΚΕΦΑΛΑΙΟ 6. Συμπεράσματα και Πιθανές Βελτιώσεις

6.1 Πιθανές Βελτιώσεις

Το βασικό χαρακτηριστικό που το κάνει πιο προσιτό το Game Engine είναι οι βασικοί οδηγοί, και η πληθώρα των tutorials που υπάρχουν στο διαδίκτυο, αλλά και το Community Forum.

Οι πιθανές βελτιώσεις αφορούν μερικές σκέψεις και ιδέες ούτως ώστε να βελτιωθεί η λειτουργικότα του παιχνιδιού. Αυτές στόχο έχουν είτε να να προσθέσουν κάτι επιπλέον στο παιχνίδι, είτε να διορθώσουν ή να βετιώσουν κάτι που ήδη βρίσκεται στο περιβάλλον του παιχνιδιού μας.

Μία πιθανή βελτίωση είναι η μετατροπή τοου παιχνιδιου σε Network Gaming. Έτσι θα υπάρχει η δυνατότητα να συμμετέχουν και άλλοι ήρωες και να γίνουν συμπαίχτες μας.

Επίσης θα μπορούσαν να προστεθούν περισσότερα animations πχ στο όπλα καθώς εκπυρσοκροτεί ή καθώς γεμίζει πυρομαχικά το όπλο του.

Θα μπορόυσαμε να του προσθέσουμε πιο πολλά όπλα όπως μια χειρομβοβίδα η οποία θα προκαλεί μια πολύ δυνατή έκρηξη σκοτώνοντας μαζικούς εχθρούς.

Η ποικοιλία πολλαπλών διαφορετικών εχθρών θα μπορούσε να είναι μια καλή προσθήκη. Πχ θα μπορούσαμε να εντάξουμε στρατιώτες εχθρούς που θα μας πυροβολούν και αυτοί από απόσταση.

Στο Inventory μια καλη βελτίωση θα ήταν να προσθέταμε ένα scroll bar σε περίπτωση που στοιβάζει ο παίχτης μας πολλά items.

Θα μπορύσε να δοθεί η δυνατότητα με την απόκτης πολλών πόντων να αγοράσει περισσότερα όπλα ή να είναι το εισιτήριο να περάσει στην επόμενη πίστα.

Μιά καλή σκέψη είναι τα Cinematics καθώς θακάνουν το παιχίδι μας πιο αληθοφανές.

Η δυνατότητα του παίχτη μας να κάνει “Save Game” καθώς το παιχνίδι αποκτά μεγαλύτερη διάρκεια.

6.2 Βιβλιογραφία – Ιστότοποι

- Augmented Reality First Person Shooter /Survival Game
https://www.udemy.com/arkit-unity-3d-creating-augmented-reality-apps-with-c/learn/v4/overview

- The Ultimate Guide to Game Development with Unity
https://www.udemy.com/the-ultimate-guide-to-game-development-with-unity/learn/v4/overview

- Unity Adventure Game Tutorial #4 - Picking Objects Up & Money System
https://www.youtube.com/watch?v=an7DtvMWSaU&t=386s

- Unity Adventure Game Tutorial #5 - Inventory System
https://www.youtube.com/watch?v=UU6qUryy4c4

- Unity Adventure Game Tutorial #10 - Simple Quests
https://www.youtube.com/watch?v=hUxv60gjb6g&t=1138s
- Unity Adventure Game Tutorial #11 - Naming Areas of Your Scene
https://www.youtube.com/watch?v=44U72sIPvL4

- Unity Adventure Game Tutorial #9 - Use Inventory Objects

https://www.youtube.com/watch?v=aiBhZbTt5ss&index=9&list=PLZeS3mWWrHczWTbMxw6FRMwLmcA32jFnK

- Shooting with Raycasts - Unity Tutorial

https://www.youtube.com/watch?v=THnivyG0Mvo&t=231s

- Weapon Switching - Unity Tutorial

https://www.youtube.com/watch?v=Dn_BUIVdAPg

- Attacking Enemies, Health System, and Death Animation in Unity

https://codingchronicles.com/unity-vr-development/day-13-attacking-enemies-health-system-death-animation-unity

- Call of Duty Zombies Remake Unity 3D Tutorial: Part 7 Enemy Damage and Death Exo Zombies

https://www.youtube.com/watch?v=wLpgBM141k8&index=7&list=PLQ1JNmJxKtjWdGeftfELYYhZbg3d_mCaz

- World Assets
  http://devassets.com/
- Unity3D Character Controller - How To Jump
  https://www.youtube.com/watch?v=1Po3E0nZOpM

- Unity 3D - How to make a gun shoot (Raycast & Bulletholes)

https://www.youtube.com/watch?annotation_id=annotation_1930022423&feature=iv&src_vid=shDFvg0P71M&v=j5kNjJj9GXA

- SHATTER / DESTRUCTION in Unity (Tutorial)

https://www.youtube.com/watch?v=EgNV0PWVaS8

- FPS DESERT ENVIRONMENT!

https://www.youtube.com/watch?v=k9S9N3eYwjQ

- Triggered Animations in U nity3d

https://www.youtube.com/watch?v=AziTfHI5l3Y

- Unity3D Player Health - Zombie FPS

https://www.youtube.com/watch?v=VXNTiRGR19Y&index=2&list=PLqO_dK1hz3TMXDRmW-aNHGosp5LzNRhW5

- 2. Unity3D Enemy Movement - Zombie FPS
  https://www.youtube.com/watch?v=ZzPcbLhGLtU&index=3&list=PLqO_dK1hz3TMXDRmW-aNHGosp5LzNRhW5
- 3. Unity3D Enemy Attack - Zombie FPS
  https://www.youtube.com/watch?v=b74Wqufn8KU&list=PLqO_dK1hz3TMXDRmW-aNHGosp5LzNRhW5&index=4
- 15. Unity3D Player Spawn Tutorial - Zombie FPS
  https://www.youtube.com/watch?v=9rwI-ZJf7xE&index=30&list=PLqO_dK1hz3TMXDRmW-aNHGosp5LzNRhW5
- 11A. Unity3D Animation Tutorial Part 1 - Zombie FPS
  https://www.youtube.com/watch?v=C6N2FJXeVBA&list=PLqO_dK1hz3TMXDRmW-aNHGosp5LzNRhW5&index=24
- 11B. Unity3D Animation Tutorial Part 2 - Zombie FPS
  https://www.youtube.com/watch?v=qJcCfy1gtM8&list=PLqO_dK1hz3TMXDRmW-aNHGosp5LzNRhW5&index=25
- 11C. Unity3D Animation Tutorial Part 3 - Zombie FPS

https://www.youtube.com/watch?v=7uPzGWqDHdU&list=PLqO_dK1hz3TMXDRmW-aNHGosp5LzNRhW5&index=26

- SHOOTING/FOLLOW/RETREAT ENEMY AI WITH UNITY AND C# - EASY TUTORIAL https://www.youtube.com/watch?v=_Z1t7MNk0c4&index=2&list=PLBIb_auVtBwDgHLhYc-NG633rTbTPim9z
- PATROL AI WITH UNITY AND C# - EASY TUTORIAL https://www.youtube.com/watch?v=8eWbSN2T8TE&list=PLBIb_auVtBwDgHLhYc-NG633rTbTPim9z&index=3