Unity3D
C# quick start guide

Εισαγωγή

Το παρόν αρχείο κατασκευάστηκε με σκοπό να αποτελεί έναν εύκολο και γρήγορο οδηγό στο κομμάτι του προγραμματισμού scripts για το Unity3D προσανατολισμένο σε αναγνώστες χωρίς ιδιαίτερη εμπειρία πάνω στο αντικείμενο. Παρακάτω θα υπάρχουν αναφορές σε τεχνικά ζητήματα τόσο της γλώσσας C# όσο και χαρακτηριστικών του Unity3D τα οποία, ωστόσο, δεν θα έπρεπε να αποτελούν το κύριο σημείο εστίασης. Αντιθέτως, ο λόγος που εμπεριέχονται στον οδηγό αυτό είναι ώστε να δικαιολογηθούν ορισμένες συμπεριφορές στα παραδείγματα και στον τρόπο συγγραφής του κώδικα. Τα παραδείγματα είναι αυτά τα οποία θα συμβάλλουν πολύ περισσότερο τόσο στη κατανόηση της λειτουργίας των scripts εντός του περιβάλλοντος του Unity3D όσο και ως αναφορές για τους πιο συχνούς μηχανισμούς που εμφανίζονται στα παιχνίδια.

Επίσης, είναι σημαντικό να αναφερθεί πως σε καμία περίπτωση ο παρόν οδηγός δεν θα πρέπει να θεωρηθεί ως η μοναδική ή η καλύτερη λύση σε ό,τι αφορά το scripting στο Unity3D. Σκοπός είναι αυτό το αρχείο να βοηθήσει στο να ξεκινήσει κανείς να μαθαίνει το σκεπτικό γύρω από τον κώδικα στο συγκεκριμένο περιβάλλον. Από εκεί και πέρα τα υπόλοιπα μπορεί να τα μάθει από άλλες πηγές (Unity tutorials, videos, forums) και κυρίως με την εξάσκηση και την εμπειρία.

Τέλος, να σημειωθεί πως προαπαιτούμενες γνώσεις ΔΕΝ χρειάζεται να υπάρχουν για να αξιοποιηθεί ο οδηγός. Θα ήταν πολύ χρήσιμο να υπήρχαν προηγούμενες γνώσεις προγραμματισμού (C/C++), κυρίως σε ό,τι αφορά τη σύνταξη, αλλά επειδή, όπως αναφέρθηκε, τα τεχνικά ζητήματα δεν είναι τα πιο σημαντικά, τα παραδείγματα θα είναι αρκετά για μια δυναμική αρχή στο χώρο του scripting για το Unity3D.

Περιεχόμενα

Κεφάλαιο 1: C#5

1. Κλάσεις5
2. Προσβασιμότητα πεδίων6
3. Getters – Setters8
4. Constructors9
5. Static πεδία10
6. Κληρονομικότητα12
7. Namespaces13

Επίλογος κεφαλαίου 114

Κεφάλαιο 2: API15

1. Αναγνώριση όρων15
2. Editor και autocomplete17

Κεφάλαιο 3: Βασικές έννοιες, κλάσεις, συναρτήσεις και συνθήκες19

1. Public & Private πεδία19
2. Prefabs21
3. Awake/Start/Update23
4. Vector324
5. Quaternion25
6. Transform26
7. GameObject29
8. Find & GetComponent29
9. Rigidbody30
10. Inputs32
11. Colliders & Triggers35
12. Coroutines38
13. Raycasting40

Κεφάλαιο 4: Παραδείγματα42

1. Απλό 2D character controller42

2. Συμπεριφορά για όπλο με χρήση Raycasting43

Κεφάλαιο 1: C#

Η C# αποτελεί μια γλώσσα με αντικειμενοστραφή χαρακτηριστικά που δημιουργήθηκε και αναπτύχθηκε από τη Microsoft. Η σύνταξή της παραπέμπει σε μεγάλο βαθμό σε αυτή των γλωσσών C και C++ με ορισμένα επιπλέον στοιχεία τα οποία καθιστούν την ανάπτυξη εφαρμογών σημαντικά πιο εύκολη.

Στα πλαίσια που μας αφορά, η C# έχει τα εξής χαρακτηριστικά τα οποία χρησιμοποιούνται πιο συχνά στη συγγραφή scripts για το Unity3D:

1. Κλάσεις
   Η C# είναι, μεταξύ άλλων, μια αντικειμενοστραφής γλώσσα, γεγονός που της επιτρέπει να διαχειρίζεται τύπους δεδομένων σε μορφή κλάσεων όπως προσδιορίζει το αντικειμενοστραφές μοντέλο. Εφόσον δεν είναι ήδη γνωστό, μια κλάση αποτελεί έναν τύπο δεδομένων ο οποίος περιλαμβάνει πεδία τα οποία μπορούν να έχουν τη μορφή είτε άλλων τύπων δεδομένων (int, float, αντικείμενα άλλων κλάσεων κλπ) είτε συναρτήσεων-μελών που αναφέρονται στα αντικείμενα που παράγει η κλάση.

   Είναι σημαντικό να γίνει ξεκάθαρο όσο το δυνατόν νωρίτερα πως η έννοια κλάση και αντικείμενο διαφέρουν με τον εξής τρόπο: η κλάση ορίζει τα πεδία που θα έχει το αντικείμενο, σχεδιάζοντας την «αρχιτεκτονική» του. Το αντικείμενο αποτελεί μια μεταβλητή ποσότητα που περιλαμβάνει ό,τι ορίζει η κλάση και καταλαμβάνει στη μνήμη τον χώρο που ορίζουν τα πεδία της. Για να εκφραστεί με μια αναλογία, η κλάση θα αποτελούσε τα σχέδια κατασκευής ενός σπιτιού, ενώ κάθε σπίτι που θα χτιζόταν βάσει των σχεδίων αυτών, θα αποτελούσε αντικείμενο της κλάσης.

   Ο ορισμός μιας κλάσης έχει την εξής μορφή:

```
class MyClass
{
	/*Pedia*/
}
```

Οπότε, αν η κλάση είχε τύπους δεδομένων και συναρτήσεις ως πεδία θα είχε την εξής μορφή:

```
1	class MyClass
2	{
3		int myInt;
4		float myFloat = 0.2f;
5		string myString;
6
7		void myFunction()
8		{
9
10		}
11	}
```

Αναλύοντας το παραπάνω έχουμε:

- Στις γραμμές 3 – 5 δηλώνονται 3 διαφορετικά πεδία από 3 διαφορετικούς τύπους δεδομένων: ακέραιο αριθμό, πραγματικό αριθμό και συμβολοσειρά αντίστοιχα.
- Στο 1ο και 3ο πεδίο δεν έχει οριστεί αρχική τιμή, οπότε το κάθε ένα θα έχει την αντίστοιχη «μηδενική» τιμή του. Οπότε το myInt θα είναι ίσο με 0 και το myString θα είναι ίσο με την κενή συμβολοσειρά “”.
- Στο 2ο πεδίο αναθέτουμε τιμή 0.2 στο myFloat. Για να προσδιοριστεί πως ο αριθμός ανήκει σε κάποιο πεδίο float, προσθέτουμε το γράμμα f μετά το νούμερο (0.2f).
- Στις γραμμές 7 – 9 δηλώνεται η συνάρτηση myFunction() η οποία δεν παίρνει κανένα όρισμα και δεν επιστρέφει κανένα αποτέλεσμα. Στην προκειμένη περίπτωση δεν εκτελεί καμία συγκεκριμένη λειτουργία.

2. Προσβασιμότητα πεδίων

Οι κλάσεις μπορεί να έχουν πεδία τα οποία να είναι καλύτερο να διατηρούνται χωρίς να μπορούν να προσπελαστούν από πηγές εκτός της κλάσης. Εναλλακτικά, μπορεί κάποια πεδία να είναι καλύτερο να μεταβάλλονται εκτός της κλάσης γιατί μπορεί να εξαρτώνται από κάποια είσοδο του χρήστη, ή από κάποιον άλλο εξωτερικό παράγοντα. Για το λόγο αυτό χρησιμοποιούνται τα εξής keywords για να προσδιοριστούν κατά πόσο τα πεδία μπορούν να προσπελαστούν από πηγές εκτός της κλάσης:

- public: Όπως ορίζει και το όνομα, το συγκεκριμένο keyword δηλώνει πως το πεδίο (τύπος δεδομένων ή συνάρτηση-μέλος) στο οποίο αναφέρεται είναι δημόσιο και έξω από τη κλάση. Επομένως, το συγκεκριμένο πεδίο ενός αντικειμένου της κλάσης μπορεί να προσπελαστεί και ενδεχομένως να μεταβληθεί από κάποια πηγή εκτός της κλάσης. Αξίζει να σημειωθεί πως στα πλαίσια του Unity3D ένα πεδίο που ορίζεται ως public μπορεί να μεταβληθεί όχι μόνο μέσω κώδικα αλλά και από τον ίδιο τον editor του Unity3D (βλ. παραδείγματα).
- private: Κατά αντιστοιχία, το keyword αυτό δηλώνει πως το πεδίο στο οποίο αναφέρεται είναι ιδιωτικό της κλάσης και δεν μπορεί να προσπελαστεί από κάποια εξωτερική πηγή.

Για παράδειγμα έστω πως είχαμε την παραπάνω κλάση και είχαμε μεταβάλλει τα πεδία ως εξής:

```
class MyClass
{
	public int myInt;
	private float myFloat = 0.2f;
	public string myString;

	public void myFunction()
	{
		myFloat++;
	}
}
```

- Έχουμε ορίσει τα myInt, myString και myFunction ως public πεδία ενώ το myFloat ως private.
- Η συνάρτηση myFunction δεν έχει πρόβλημα να διαχειριστεί το myFloat διότι, αν και είναι private, και τα δύο πεδία ανήκουν στην ίδια κλάση.

Έστω τώρα πως είχαμε το εξής στο ίδιο αρχείο .cs:

```
1	class MyOtherClass
2	{
3		void myOtherFunction()
4		{
5			MyClass myClassObject = new MyClass();
6			myClassObject.myInt++;
7			myClassObject.myFunction();
8		}
9	}
```

- Αυτή η καινούρια κλάση περιέχει μόνο μια συνάρτηση ονόματι myOtherFunction().
- Μέσα στη συνάρτηση αυτή δημιουργείται ένα αντικείμενο της κλάσης MyClass με όνομα myClassObject.
- Στη γραμμή 5 δημιουργείται το αντικείμενο και αρχικοποιείται με τον τρόπο που φαίνεται. Το new δηλώνει πως κατασκευάζουμε ένα νέο αντικείμενο. Το MyClass() αναφέρεται στη κλήση της συνάρτησης constructor της κλάσης MyClass (βλ. 4. Constructors). Ουσιαστικά η γραμμή αυτή δηλώνει τη χρήση ενός τέτοιου αντικειμένου και κάνει τις απαραίτητες δεσμεύσεις στη μνήμη για να κατασκευάσει το αντικείμενο.
- Το αντικείμενο myClassObject έχει όλα τα πεδία που περιγράφει η κλάση MyClass με τις προσβασιμότητες που προσδιορίστικαν. Επομένως, αφού το πεδίο myInt είχε δηλωθεί ως public, η συνάρτηση της κλάσης MyOtherClass μπορεί να το προσπελάσει και να το μεταβάλλει[1].
- Η συνάρτηση myFunction() είναι επίσης public οπότε μπορεί να κληθεί μέσα από τη συνάρτηση της MyOtherClass οπότε μεταβάλλεται και η τιμή της myFloat έμμεσα από τη συνάρτηση. Ωστόσο, το πεδίο myFloat είναι private, οπότε στα πλαίσια της κλάσης MyOtherClass, το myClassObject.myFloat είναι αόρατο και μη προσβάσιμο.

Μπορούν επιπλέον να υπάρχουν public και private κλάσεις το οποίο αφορά το κατά πόσο είναι ορατές ή όχι σε άλλες κλάσεις.

3. Getters – Setters

Οι getters/setters (ή accessors/mutators) αποτελούν συναρτήσεις της κλάσης που αφορούν ξεχωριστά τα πεδία της και ορίζουν το κατά πόσο μπορούν να μεταβληθούν ή όχι. Η C# έχει έναν ξεχωριστό τρόπο με τον οποίο το ορίζει αυτό και παρουσιάζεται στο παρακάτω παράδειγμα:

```
public class MyClass
{
        public float myFloat {get; set;}
        public float myOtherFloat {get;}

        public void myFunction()
        {
                myFloat = myOtherFloat / 2;
        }
```

[1] Να σημειωθεί πως η μεταβολή του myInt εφαρμόζεται μόνο στο αντικείμενο myClassObject. Αν κατασκευαστεί ένα αντικείμενο myClassObject2 από την κλάση MyClass το πεδίο myInt του θα είναι όπως ορίζει η κλάση, δηλαδή 0.

```
}
```

- Το πεδίο myFloat έχει και get και set ιδιότητες που του επιτρέπουν να διαβαστεί και να μεταβληθεί.
- Το πεδίο myOtherFloat έχει μόνο την ιδιότητα get, γεγονός που σημαίνει πως το συγκεκριμένο πεδίο, αν και public, μπορεί να διαβαστεί αλλά όχι να μεταβληθεί, ούτε καν από κάποια συνάρτηση εντός της κλάσης.
- Η συνάρτηση myFunction() δείχνει πως ενώ το myFloat μπορεί να μεταβληθεί, το πεδίο myOtherFloat χρησιμοποιείται μόνο για να διαβάσουμε τη τιμή του. Αυτές οι μεταβλητές λέγονται και read-only.

4. Constructors

Ο constructor αποτελεί συνάρτηση μιας κλάσης η οποία καλείται καθώς δημιουργείται ένα αντικείμενο αυτής της κλάσης. Χρησιμοποιείται για την αρχικοποίηση των τιμών των πεδίων του αντικειμένου καθώς και για τη δέσμευση του απαραίτητου χώρου στη μνήμη. Ο ορισμός μιας συνάρτησης constructor γίνεται με τον εξής τρόπο:

```
public class MyClass
{
	private int myInt;
	public float myFloat;

	public MyClass(int myInt, float myFloat)
	{
		this.myInt = myInt;
		this.myFloat = myFloat;
	}

	public MyClass(int myInt)
	{
		this.myInt = myInt;
		myFloat = 5.2f;
	}

	public MyClass(float myFloat)
	{
		this.myFloat = myFloat;
		myInt = 3;
	}
}
```

- Ο constructor είναι public συνάρτηση που δεν έχει τύπο επιστροφής και το όνομά της είναι ακριβώς αυτό της κλάσης.

- Όπως και με τις συναρτήσεις, και στους constructor μπορούν να κατασκευάζονται πολλές συναρτήσεις με το ίδιο όνομα εφόσον τα ορίσματά τους είναι διαφορετικά σε αριθμό ή/και τύπους.
- Ο 1ος constructor παίρνει ως ορίσματα ένα int και ένα float τα οποία αναθέτει στις τιμές των πεδίων του αντικειμένου.
- Ο 2ος constructor παίρνει ως όρισμα μόνο ένα int το οποίο αναθέτει στο πεδίο myInt και αναθέτει σταθερή τιμή στο πεδίο myFloat ενώ ο 3ος constructor κάνει το αντίστροφο.
- Το `this` πρόκειται για ένα keyword το οποίο αναφέρεται στο αντικείμενο το οποίο καλεί τη συγκεκριμένη συνάρτηση. Ο λόγος που τοποθετείται στους constructor είναι για λόγους σαφήνειας, καθώς τα ονόματα των ορισμάτων είναι ίδια με αυτά των πεδίων της κλάσης.
- Η χρήση των constructor αυτών γίνεται με τον εξής τρόπο:

```
class MyOtherClass
{
	void myOtherFunction()
	{
		MyClass myClassObject1 = new MyClass(1,1.2f);
		MyClass myClassObject2 = new MyClass(2);
		MyClass myClassObject3 = new MyClass(0.4f);
	}
}
```

- Το myClassObject1 χρησιμοποιεί τον 1ο constructor, ενώ τα myClassObject2 και myClassObject3 χρησιμοποιούν τον 2ο και 3ο constructor αντίστοιχα για να αρχικοποιηθούν.
- Το αν τα πεδία είναι public ή private δεν έχει σημασία καθώς ο constructor που τους αναθέτει τιμές ανήκει στην κλάση.
- Επομένως, τα πεδία των αντικειμένων θα έχουν την εξής μορφή:

	myClassObject1	myClassObject2	myClassObject3
myInt	1	2	3
myFloat	1.2	5.2	0.4

5. <u>Static πεδία</u>

Εντός των κλάσεων μπορεί να υπάρχουν πεδία τα οποία θα ήταν χρήσιμο να σχετίζονται με την κλάση αλλά να μην εξαρτώνται από ένα και μόνο αντικείμενο. Για παράδειγμα, έστω πως υπάρχει η κλάση Car με ένα πεδίο int counter το οποίο

αυξάνεται με κάθε constructor για να μετράει πόσα αμάξια υπάρχουν. Αυτό θα είχε μια τέτοια μορφή:

```
class Car
{
	public int counter = 0;

	public Car()
	{
		counter++;
	}
}
```

Ωστόσο κάτι τέτοιο δεν θα δουλέψει με τον τρόπο αυτό, διότι το πεδίο counter θα απευθύνεται σε ένα μόνο αντικείμενο και για κάθε αμάξι που φτιάχνεται το counter θα ξεκινάει από 0 και θα πηγαίνει στο 1 λόγω του constructor.

Εδώ εισέρχεται η έννοια του static πεδίου. Αν αντί για " `public int counter = 0;`" η κλάση είχε "`public static int counter = 0;`", το πεδίο counter δεν θα απευθυνόταν σε κάθε αντικείμενο ξεχωριστά αλλά σε μια σταθερή μεταβλητή counter που συνδέεται μονάχα με την κλάση. Οπότε θα μπορούσε να υπάρξει το εξής:

```
class Car
{
	public static int counter = 0;

	public Car()
	{
		counter++;
	}
}


class MyOtherClass
{
	void myOtherFunction()
	{
		Car car1 = new Car();
		Car.counter++;
	}
}
```

- Στην κλάση MyOtherClass κατασκευάζεται ένα αντικείμενο της κλάσης Car με όνομα car1 το οποίο καλεί τον constructor του ο οποίος αυξάνει κατά 1 τον μετρητή που αντιστοιχεί στην κλάση Car.
- Με την εντολή car.counter++; αυξάνεται ξανά ο μετρητής counter της κλάσης Car, χωρίς κάποια χρηστικότητα παρά μόνο για τους σκοπούς του παραδείγματος. Για να γίνει αυτό έπρεπε πρώτα να προσπελαστεί μέσω της κλάσης Car και ΟΧΙ μέσω του αντικειμένου car1. Μάλιστα το car1.counter δεν υφίσταται καν σαν πεδίο.

6. <u>Κληρονομικότητα</u>

Το κεφάλαιο της κληρονομικότητας στον αντικειμενοστραφή προγραμματισμό έχει γενικά πολύ υλικό, οπότε εδώ θα αναγραφούν τα πιο βασικά.

Η κληρονομικότητα αφορά μια τεχνική οργάνωσης κλάσεων με βάση τα κοινά τους χαρακτηριστικά σε μια διάταξη από πιο γενικές σε πιο ειδικές κλάσεις. Για παράδειγμα, έστω η κλάση Computer με χαρακτηριστικά RAM και CPU. Ωστόσο, πλέον και ένα smartphone μπορεί να έχει RAM και ένα μοντέλο CPU, ενώ έχει π.χ. και αριθμό τηλεφώνου και αυτονομία μπαταρίας σε ώρες. Αντί να κατασκευάσουμε 2 εντελώς διαφορετικές κλάσεις Computer και Smartphone που έχουν και οι 2 πεδία RAM και CPU μπορούμε να θεωρήσουμε πως το Smartphone κληρονομεί τα χαρακτηριστικά αυτά από τη κλάση Computer. Σε C# αυτό περιγράφεται έτσι:

```
class Computer
{
	public int RAM;
	public string CPU;

	public Computer(int ram, string cpu)
	{
		RAM = ram;
		CPU = cpu;
	}
}

class Smartphone : Computer
{
	public string number;
	public float battery_life;

	public Smartphone(int ram, string cpu, string number, float battery_life) :
	base(ram,cpu)
	{
		this.number = number;
		this.battery_life = battery_life;
	}
}
```

- Το ότι μια κλάση κληρονομεί από μια άλλη εκφράζεται στο όνομα της κλάσης χρησιμοποιώντας : και το όνομα της γενικότερης κλάσης από την οποία κληρονομεί.
- Η κλάση Smartphone έχει επίσης τα πεδία RAM και CPU ως public μεταβλητές από τη κλάση Computer.
- Ο constructor της κλάσης Smartphone περιλαμβάνει ως ορίσματα πεδία τόσο για την ίδια τη κλάση όσο και για το constructor της κλάσης Computer. Τα πεδία που αφορούν τον constructor της κλάσης Computer περνάνε ως ορίσματα στην κλήση του base constructor, δηλαδή του constructor της κλάσης από την οποία κληρονομεί η κλάση Smartphone.
- Ο base constructor καλείται όπως φαίνεται στον κώδικα με : base() έπειτα από τα ορίσματα του constructor της κλάσης-παιδί, όπου ως ορίσματα στην base περνάνε τα απαραίτητα όπως ορίστηκαν από τον constructor της κλάσης-γονιού.

7. Namespaces

Για καλύτερη οργάνωση του κώδικα αλλά και για καλύτερη απόκρυψη δεδομένων όπου χρειάζεται, ορίζονται namespaces, δηλαδή, κομμάτια κώδικα που μπορεί να περιέχουν μία ή και πολλές κλάσεις μαζί. Ο τρόπος που ορίζονται τα namespaces είναι ο εξής:

```
namespace myNamespace
{
	class MyClass
	{
		public void myFunction() { }
	}
}
```

Τα κομμάτια κώδικα στα namespaces μπορούν να προσπελαστούν με δύο τρόπους: είτε εισάγοντας στο script ολόκληρο το namespace στην αρχή του script με χρήση της εντολής using, είτε αξιοποιώντας το συγκεκριμένο κομμάτι εντός του κώδικα σε ένα σημείο. Χρησιμοποιώντας το από πάνω namespaces οι 2 μέθοδοι φαίνονται παρακάτω:

1.
```
using myNamespace;

class MyOtherClass
{
	void myOtherFunction()
```

```
        {
                MyClass myClassObject = new MyClass();
                myClassObject.myFunction();
        }
    }
```

2. ```
   class MyOtherClass
   {
           void myOtherFunction()
           {
                   myNamespace.MyClass myClassObject = new myNamespace.MyClass();
                   myClassObject.myFunction();
           }
   }
   ```

Επίλογος κεφαλαίου 1

Το πρώτο κεφάλαιο είχε ως σκοπό να παρέχει σε εισαγωγικό επίπεδο ορισμένες τεχνικές λεπτομέρειες και όρους πάνω στη γλώσσα που θα χρησιμοποιηθεί για τη συγγραφή των scripts στο Unity3D. Δεν είναι απαραίτητη η γνώση αυτών των τεχνικών λεπτομερειών και, μάλιστα, πολλά από τα παραπάνω δεν θα χρειαστούν για τη συγγραφή των scripts μέχρι κάποιο πιο ψηλό επίπεδο. Ωστόσο, κρίθηκε χρήσιμο να αναφερθούν κυρίως για τη διασταύρωση των εννοιών με τα παραδείγματα με σκοπό τη καλύτερη κατανόησή τους. Άλλωστε, μπορεί κάποιος απλά να δει και από τα περιεχόμενα πως εδώ υπάρχουν λίγα λόγια για τους όρους αυτούς ώστε να ανατρέξει στον παρόν οδηγό ή στο διαδίκτυο για πιο πολλές πληροφορίες.
   ```

Κεφάλαιο 2: API

Το API (Application Program Interface) αποτελεί το σύνολο από συναρτήσεις, κλάσεις, interfaces κλπ τα οποία χρησιμοποιούνται για την ανάπτυξη εφαρμογών σε συγκεκριμένα περιβάλλοντα. Εφόσον δεν αναφερόμαστε σε εφαρμογές κονσόλας, για να γραφτούν προγράμματα σε περιβάλλοντα όπως συγκεκριμένα λειτουργικά συστήματα ή, όπως στην περίπτωσή μας, εντός άλλων προγραμμάτων χρησιμοποιούνται οι αντίστοιχες βιβλιοθήκες με τα απαραίτητα εργαλεία. Για το λόγο αυτό κιόλας δεν είναι απαραίτητες οι εις βάθος γνώσεις της C# για να γραφτούν scripts εντός του Unity3D, καθώς θα γίνεται κυρίως χρήση του API που παρέχει το ίδιο το πρόγραμμα.

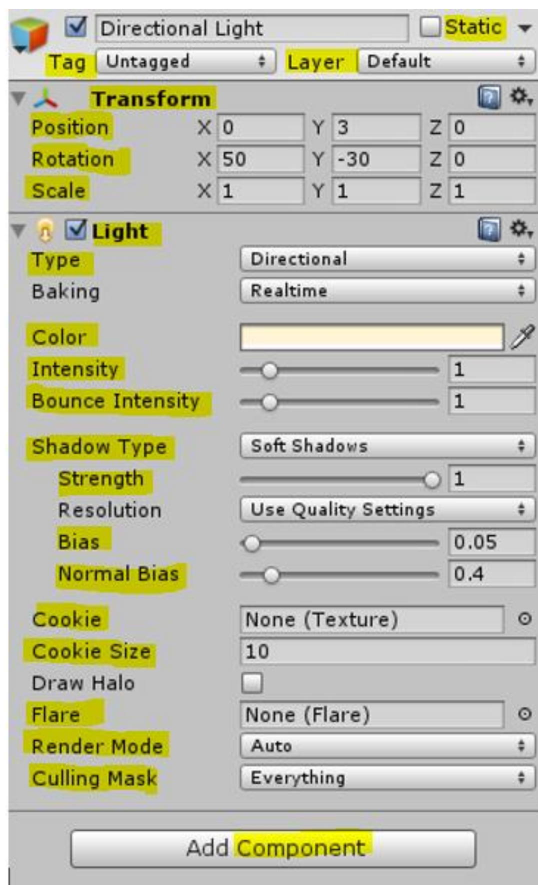
Σημειώνεται, επιπλέον, πως σε καμία περίπτωση δεν απαιτείται εκτενής γνώση ολόκληρου του API του Unity3D (όπως και για οποιοδήποτε περιβάλλον αξιοποιείται το API). Τα πιο βασικά στοιχεία θα εφαρμοστούν στα παραδείγματα και για οτιδήποτε άλλο υπάρχει online documentation πάνω στις έτοιμες βιβλιοθήκες του Unity3D (http://docs.unity3d.com/ScriptReference/).

Στο σύντομο αυτό κεφάλαιο οι βασικοί στόχοι είναι να γίνει κατανοητή η χρήση του API μέσω του editor (MonoDevelop/ Visual Studio) για γρήγορη και εύκολη συγγραφή των scripts καθώς και να εξοικειωθεί ο αναγνώστης με την αναγνώριση των όρων εντός του Unity3D ώστε να διευκολυνθεί στη χρήση τόσο του API όσο και του online documentation.

1. Αναγνώριση όρων

   Το API του Unity3D χρησιμοποιεί συγκεκριμένους όρους για να αναπαραστήσει διάφορες έννοιες τους οποίους και είναι βασικό να γνωρίζει κανείς ώστε να χρησιμοποιήσει αποτελεσματικά το API. Για παράδειγμα, μπορεί να θέλει κάποιος να ψάξει μια συνάρτηση που θα μετακινεί ένα αντικείμενο προς μία κατεύθυνση, ωστόσο ο όρος move, όπως ίσως σκεφτόταν κάποιος, δεν υφίσταται στο API του Unity3D. Για τον σκοπό αυτό χρησιμοποιείται η συνάρτηση Translate της κλάσης Transform.

   Για να υπάρχει μια αρκετά καθαρή εικόνα ως προς το τι όροι υπάρχουν στο API, αρκεί να παρατηρηθεί το ίδιο το editor του Unity3D. Ακολουθεί ένα screenshot από το inspector tab του Unity3D όταν είναι επιλεγμένο ένα directional light:



Στο παρόν παράδειγμα σημειώνονται 3 βασικά μέρη (components) του αντικειμένου:

Το 1ο δεν φαίνεται άμεσα αλλά είναι το ανοιχτό γκρι στη κορυφή που έχει το όνομα του αντικειμένου, το tag, το layer και το αν είναι static ή όχι. Αυτό είναι μια από τις γενικότερες κατηγορίες component και ονομάζεται Game Object.

Το 2ο είναι το Transform και αποτελεί το component που χαρακτηρίζει πως το αντικείμενο βρίσκεται στον χώρο. Τα πεδία του προσδιορίζουν τη θέση, τη κλίση και τη κλίμακα του αντικειμένου.

Το 3ο είναι το Light και, όπως προδίδει το όνομα, αφορά την ιδιότητα του αντικειμένου να εκπέμπει φως. Τα πεδία στο συγκεκριμένο component αφορούν τις ρυθμίσεις του.

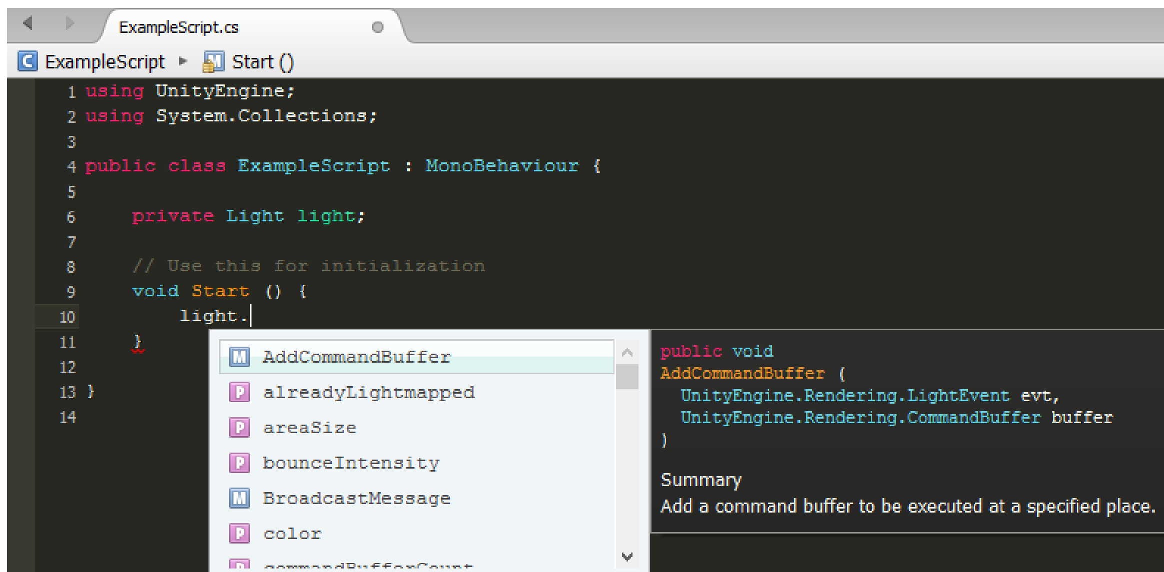
Οι λέξεις που είναι σημειωμένες με κίτρινο είναι εκείνες που αν προσπελάσει κανείς την αντίστοιχη κλάση του component θα τις βρει και στο API. Επομένως θα μπορούσε κανείς να γράψει το ακόλουθο:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class ExampleScript : MonoBehaviour {
 5
 6     private Light light;
 7
 8     // Use this for initialization
 9     void Start () {
10         light.intensity = 5;
11         light.color = Color.red;
12         light.type = LightType.Point;
13     }
14
15 }
16
```

Γενικός κανόνας: Αν υπάρχει κάπου σαν λέξη μέσα στο editor του Unity3D, υπάρχει μεγάλη πιθανότητα να υπάρχει και στην αντίστοιχη κλάση του API.

2. <u>Editor και autocomplete</u>

Όπως και στις δικές μας κλάσεις, για να προσπελαστούν τα πεδία του αντικειμένου χρησιμοποιείται η τελεία («.»). Είτε στο MonoDevelop, είτε στο Visual Studio, όταν χρησιμοποιείται η τελεία εμφανίζεται ένα μενού με όλες τις δυνατές επιλογές πάνω στο αντικείμενο που μόλις γράφτηκε. Όσο γράφονται και άλλα γράμματα ή λέξεις που αντιστοιχούν σε κάποιες από τις επιλογές, όλες οι άλλες επιλογές αφαιρούνται από το μενού και είναι εφικτό να επιλεχθεί κάποια από αυτές τις επιλογές με βελάκια ή με το ποντίκι ή με άλλα μέσα, ανάλογα με τον κειμενογράφο. Ακολουθούν παραδείγματα της συγκεκριμένης λειτουργίας:



Στο συγκεκριμένο παράδειγμα αξίζουν να σημειωθούν τα εξής:

- Οι επιλογές που εμφανίζονται ξεπερνούν κατά πολύ όσα δείχνει το editor. Αυτό συμβαίνει λόγω όλων των κλάσεων από τις οποίες κληρονομεί η κλάση Light, από άλλες επιλογές της ίδιας της C# ή και επειδή υπάρχουν πεδία τα οποία είναι προσπελάσιμα μόνο μέσω κώδικα.
- Για την κάθε επιλογή εμφανίζεται η σύνταξή της, τι επιστρέφει και τι ορίσματα δέχεται καθώς και, σε πολλές περιπτώσεις, μια μικρή περιγραφή για το τη λειτουργία της.

```
ExampleScript.cs
ExampleScript ▸ Start ()
1  using UnityEngine;
2  using System.Collections;
3
4  public class ExampleScript : MonoBehaviour {
5
6      private Light light;
7
8      // Use this for initialization
9      void Start () {
10         light.in
11     }
12
13 }
14
```

```
GetComponentInChildren
GetComponentInParent
GetComponentsInChildren
GetComponentsInParent
GetInstanceID
intensity
```

```
public float
intensity { get; set; }

Summary
The Intensity of a light is multiplied with the Light color.
```

Στο παράδειγμα αυτό οι επιλογές του μενού είναι πολύ λιγότερες γιατί έχουν μείνει μόνο αυτές που εμπεριέχουν σε κάποιο σημείο τα γράμματα “in”.

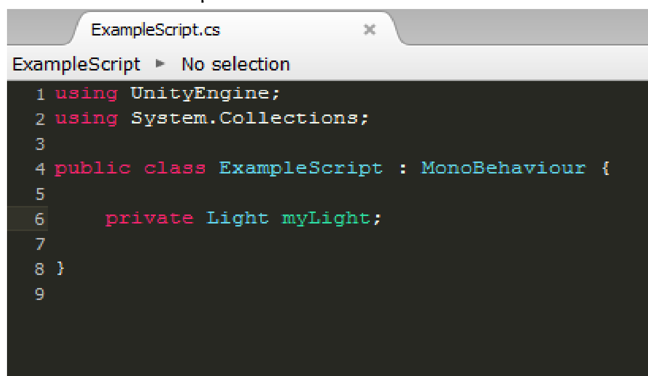
Αξίζει επίσης να παρατηρηθεί πως δίπλα από κάθε πεδίο υπάρχει ένα γράμμα. Το γράμμα αυτό περιγράφει περί τίνος πρόκειται το πεδίο. Με P συμβολίζεται ένας τύπος δεδομένων όπως float ενώ με M μια συνάρτηση (από το method, που επίσης χαρακτηρίζει μια συνάρτηση).

Κεφάλαιο 3: Βασικές έννοιες, κλάσεις, συναρτήσεις και συνθήκες

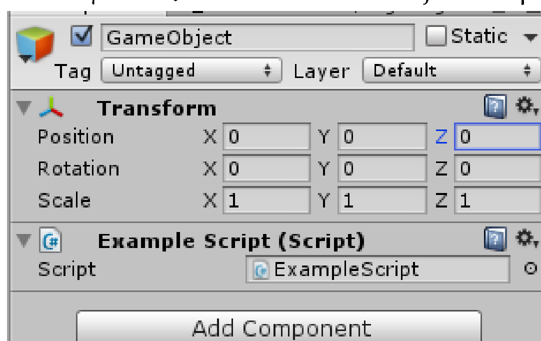
Στο κεφάλαιο αυτό θα γίνει μια σύντομη παρουσίαση των βασικών εργαλείων που θα χρησιμοποιηθούν στα παραδείγματα του κεφαλαίου 4. Ο λόγος που οι έννοιες αυτές παρουσιάζονται σε ξεχωριστό κεφάλαιο και όχι στην επεξήγηση του κάθε παραδείγματος είναι καθαρά για αποφυγή επανάληψης πληροφοριών και για να υπάρχει ένας πιο γρήγορος τρόπος να ανατρέξει κανείς σε αυτές. Άλλωστε, είναι καλύτερο να παρουσιαστούν τα εργαλεία που αποτελούν τα παραδείγματα από πιο πριν ώστε να είναι πιο εύκολη και γρήγορη η κατανόησή τους.

1. <u>Public & Private πεδία</u>

   Στο 1ο κεφάλαιο αναφέρθηκαν οι ιδιότητες public και private των πεδίων ως προς τη προσβασιμότητα. Εντός του API του Unity3D, οι ιδιότητες αυτές αποκτούν διαφορετική σημασία, καθώς το αν ένα πεδίο είναι public ή όχι ορίζει αν θα είναι προσβάσιμο και μέσω του editor. Έστω, λοιπόν, πως υπάρχει το εξής script στο project το οποίο ενσωματώνεται σε ένα άδειο Game Object:



   Επιλέγοντας αυτό το Game Object θα βλέπαμε αυτό στον inspector:



Παρατηρείται πως μαζί με τα άλλα components του αντικειμένου εμφανίστηκε ένα component με το όνομα του script το οποίο ενσωματώθηκε στο αντικείμενο.

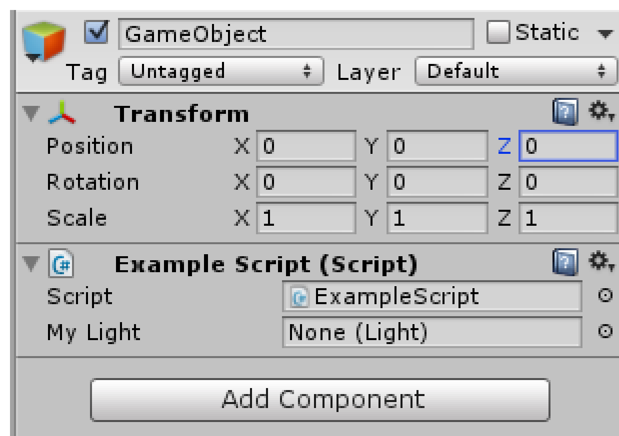
Κάθε script που ενσωματώνεται σε αντικείμενο είναι ένα component του οποίου ο τύπος έχει το όνομα του ίδιου του script.

Εφόσον το πεδίο myLight δηλωθεί σαν public, όμως, θα γίνουν οι εξής αλλαγές:

ExampleScript.cs

election

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class ExampleScript : MonoBehaviour {
5
6     public Light myLight;
7
8 }
```



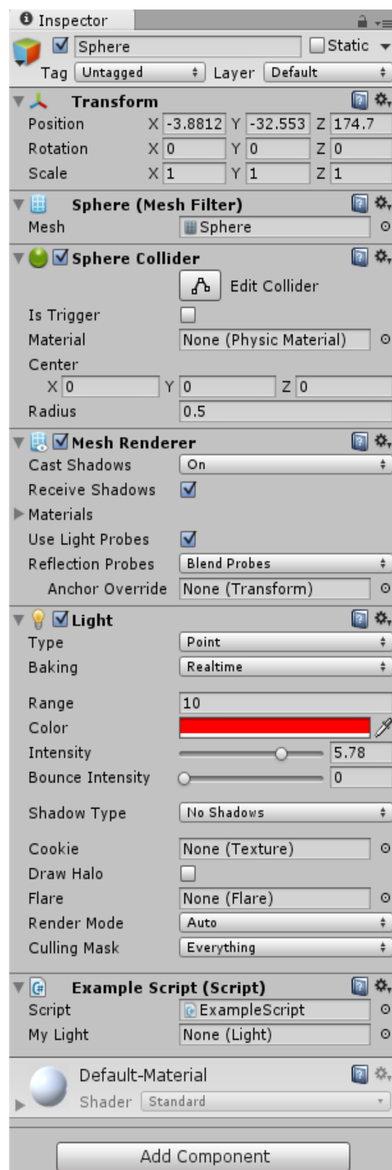
Παρατηρείται πως στο component του script εμφανίστηκε ένα πεδίο με το όνομα My Light το οποίο δέχεται κάποιο αντικείμενο το οποίο έχει Light σαν component. Δηλαδή θα μπορούσε στο πεδίο αυτό να γίνει drag and drop το directional light της σκηνής, για παράδειγμα, αλλά δεν θα μπορούσε να γίνει drag and drop το Main Camera ή κάποιο αντικείμενο που δεν έχει Light ως component.

Είναι προφανές πως η παρούσα λειτουργία των private/public πεδίων λειτουργούν για οποιοδήποτε τύπο δεδομένων.

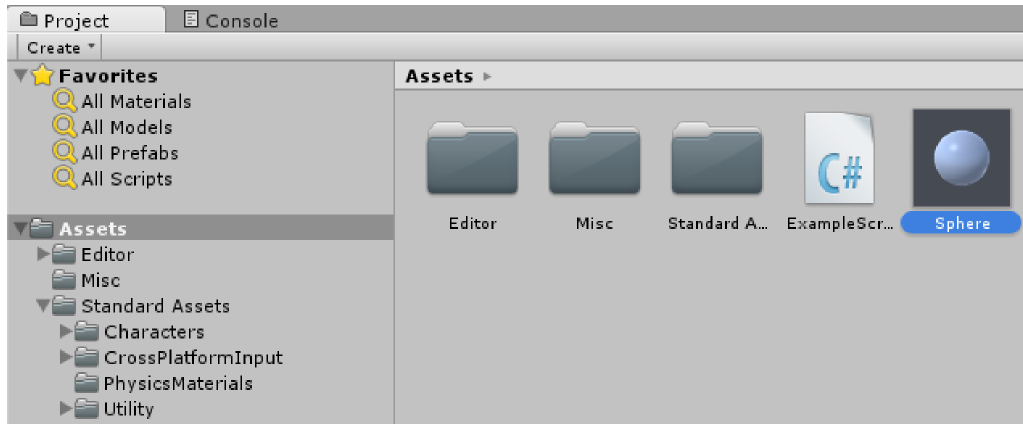
2. <u>Prefabs</u>

Ένα prefab είναι ένα σύνολο από αντικείμενα, components κλπ τα οποία αποθηκεύονται σε ένα Game Object το οποίο αποθηκεύεται στον φάκελο των assets. Χρησιμοποιείται πολύ συχνά σε projects του Unity3D διότι βοηθάει στην οργάνωση αντικειμένων που περιέχουν scripts και components με συγκεκριμένες ρυθμίσεις ώστε να μη χρειάζεται οι ρυθμίσεις αυτές να επαναπροσδιορίζονται συνεχώς.

- Κατασκευή ενός prefab: Έστω πως υπάρχει στη σκηνή ένα αντικείμενο με ένα Light Component του οποίου τις ρυθμίσεις έχουμε αλλάξει, όπως αυτό:



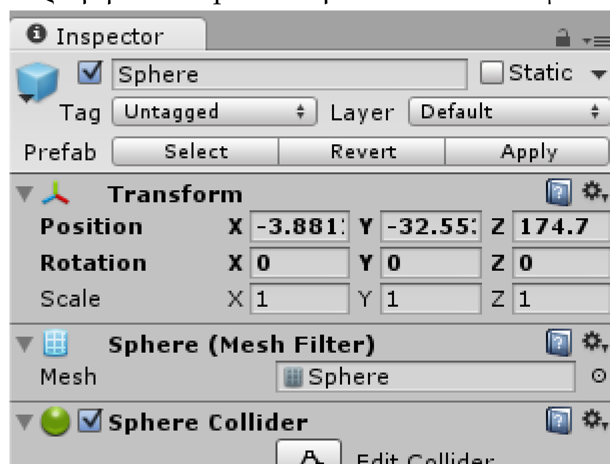
Για να αποθηκευτεί αυτό το αντικείμενο με αυτό το όνομα, τα components, τις ρυθμίσεις κλπ ως ένα prefab απλά γίνεται drag and drop από το hierarchy window στο project window και δημιουργείται το εξής asset:



Πλέον, για να τοποθετηθεί ένα ακόμα αντικείμενο με τις ίδιες ακριβώς ρυθμίσεις αρκεί να γίνει drag and drop του prefab στη σκηνή.

- Αλλαγές του prefab: Αλλαγές στις ρυθμίσεις των components ενός prefab μπορούν να γίνουν με 2 τρόπους. Μέσα από κάποιο αντικείμενο της σκηνής που προήλθε από αυτό το prefab ή μέσα από το ίδιο το prefab. Να σημειωθεί πως μια από τις πιο χρήσιμες ιδιότητες των prefabs είναι πως όταν οι εφαρμογές εφαρμοστούν στο prefab, οι ρυθμίσεις θα αλλάξουν σε ΚΑΘΕ αντικείμενο που προήλθε από αυτό το prefab.

Για τον πρώτο τρόπο, αρκεί να γίνουν οι αλλαγές στις ρυθμίσεις του αντικειμένου και έπειτα να γίνει κλικ στο Apply που θα έχει εμφανιστεί στην κορυφή του inspector εφόσον το αντικείμενο προήλθε από κάποιο prefab:



Αν ο χρήστης δεν πατήσει Apply, οι αλλαγές αυτές που έγιναν θα ισχύουν μόνο στο παρόν αντικείμενο και δεν θα περαστούν σε άλλα αντικείμενα που προήλθαν από το ίδιο prefab.

Για τον δεύτερο τρόπο αρκεί ο να επιλεχθεί το prefab μέσα από το project window και να γίνουν οι αλλαγές που ο χρήστης επιθυμεί.

3. Awake/Start/Update

Οι συναρτήσεις Awake/Start/Update αποτελούν συναρτήσεις του API οι οποίες καθορίζουν τι θα εκτελείται κατά τη διάρκεια του παιχνιδιού και αποτελούν τα βασικά δομικά χαρακτηριστικά κάθε συμπεριφοράς. Πιο συγκεκριμένα:

- void Awake() {} : Οτιδήποτε βρίσκεται εντός των αγκυλών θα εκτελείται κατά την εκκίνηση του παιχνιδιού. Χρησιμοποιείται κυρίως για αρχικοποίηση πεδίων, όπως components, τα οποία θέλουμε να έχουν αρχικοποιηθεί πριν από οποιοδήποτε άλλο συμβάν ώστε να μην υπάρξουν προβλήματα με πεδία που δεν έχουν προλάβει να αρχικοποιηθούν.
- void Start() {} : Οτιδήποτε βρίσκεται εντός των αγκυλών θα εκτελείται μετά την εκκίνηση του παιχνιδιού και τη στιγμή που το script είναι ενεργό όσο τρέχει το παιχνίδι. Σημειώνεται πως αν το script δεν είναι ενεργό κατά την εκκίνηση του παιχνιδιού, τα στοιχεία της Awake θα εκτελεστούν κανονικά, ενώ τα στοιχεία της Start θα εκτελεστούν όταν το script γίνει ενεργό.
- void Update() {} : Οτιδήποτε εντός των αγκυλών της συνάρτησης αυτής θα εκτελείται σε ΚΑΘΕ frame, και για τον λόγο αυτό αποφεύγεται να χρησιμοποιούνται χρονοβόρες διαδικασίες εντός της συνάρτησης. Έχει πάρα πολλές χρήσεις με τις πιο κοινές να είναι ο έλεγχος συνθηκών και εισόδου από τον χρήστη. Η Update συναντάται επίσης σε άλλες δύο μορφές με την ίδια σύνταξη, ως FixedUpdate και ως LateUpdate. Η FixedUpdate καλείται σε σταθερά χρονικά διαστήματα ανάμεσα σε frames και χρησιμοποιείται κυρίως για διαδικασίες που αφορούν physics, όπως συναρτήσεις της κλάσης Rigidbody. Η LateUpdate καλείται επίσης σε κάθε frame αλλά αφού κληθούν οι συναρτήσεις της Update. Κυρίως χρησιμοποιείται για οργάνωση της σειράς εκτέλεσης των συναρτήσεων. Για παράδειγμα αν μια κάμερα ακολουθεί έναν παίκτη, η κίνησή της θα πρέπει να εκτελείται στη LateUpdate καθώς ο παίκτης ενδεχομένως να έχει κινηθεί κατά τη διάρκεια της Update.

Σημειώνεται πως κατά τη δημιουργία ενός καινούριου script μέσα από τον editor του Unity3D, το script θα έχει ήδη έτοιμες για συμπλήρωση τις συναρτήσεις Start και Update με την εξής μορφή:

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class Example : MonoBehaviour {
5
6     // Use this for initialization
7     void Start () {
8
9     }
10
11     // Update is called once per frame
12     void Update () {
13
14     }
15 }
```

4. <u>Vector3</u>

Το Vector3 αποτελεί κλάση του API η οποία αναπαριστά τρισδιάστατα διανύσματα. Χρησιμοποιούνται για να προσδιοριστεί η θέση ενός αντικειμένου ή η κατεύθυνσή του στο χώρο.

- Αρχικοποίηση ενός Vector3: Η αρχικοποίηση ενός αντικειμένου της κλάσης Vector3 γίνεται με τον εξής τρόπο:

```
4 public class Example : MonoBehaviour {
5
6     private Vector3 vector;
7     private float float1,float2,float3;
8
9     // Use this for initialization
10     void Start () {
11         vector = new Vector3(float1,float2,float3);
12     }
```

Όπου τα float1, float2 και float3 είναι πραγματικοί αριθμοί οι οποίοι χρησιμοποιούνται για να δώσουν τιμές στα πεδία x, y και z, αντίστοιχα, του αντικειμένου vector. Τα πεδία x, y και z αναπαριστούν την απόσταση στον κάθε έναν από τους τρεις άξονες. Σημειώνεται πως στον constructor του Vector3 υπάρχει overload που επιτρέπει να μην εισαχθεί τιμή στο πεδίο του z, στην οποία περίπτωση το πεδίο αυτό θα ήταν ίσο με το 0.

- Κατεύθυνση: Η κλάση Vector3 περιέχει ορισμένα static πεδία τα οποία παρέχουν κατευθύνσεις στον χώρο. Τα πεδία αυτά εκφράζονται ως εξής:
  - Vector3.up
  - Vector3.down
  - Vector3.right
  - Vector3.left
  - Vector3.forward
  - Vector3.back

  Κάθε ένα από τα πεδία αυτά αντιστοιχεί σε ένα τρισδιάστατο διάνυσμα το οποίο αποτελεί βάση του τρισδιάστατου χώρου και έχει μορφή τριών αριθμών εκ των οποίων οι δύο είναι 0 και ένας είναι 1 ή -1 ανάλογα με την κατεύθυνση. Για παράδειγμα το Vector3.right αντιστοιχεί στο διάνυσμα (1,0,0), το Vector3.left στο (-1,0,0) κ.ο.κ..

  Η ανάθεση μιας τέτοιας τιμής σε ένα αντικείμενο της κλάσης Vector3 γίνεται ως εξής:

```
4 public class Example : MonoBehaviour {
5
6     private Vector3 vector;
7
8     // Use this for initialization
9     void Start () {
10        vector = Vector3.right;
11    }
```

5. <u>Quaternion</u>

   Το Quaternion είναι η κλάση η οποία χρησιμοποιείται για να αναπαραστήσει τη κλίση ενός αντικειμένου και αντιστοιχεί στο τετραδόνιο, μια έννοια στο χώρο των μαθηματικών στην οποία δεν υπάρχει νόημα να εμβαθύνουμε. Έχει αντίστοιχα πεδία x, y, z και w (όπως το Vector3 έχει τα x, y, z) τα οποία όμως έχουν νόημα να μεταβληθούν απευθείας μόνο από άτομα που γνωρίζουν τη μαθηματική έννοια στην οποία αντιστοιχούν. Εναλλακτικά, εντός του κώδικα θα χρησιμοποιείται η αναπαράσταση του Quaternion σε γωνίες Euler, όπως άλλωστε εμφανίζεται και στον editor του Unity3D στα πεδία που αφορούν rotation.

- Αρχικοποίηση ενός Quaternion:

```
4 public class Example : MonoBehaviour {
5
6     private Quaternion quaternion;
7     private float float1, float2, float3;
8
9     // Use this for initialization
10    void Start () {
11        quaternion = Quaternion.Euler(float1, float2, float3);
12    }
13
```

Όπου τα float1, float2, float3 αντιστοιχούν στις γωνίες περιστροφής στον x, y και z άξονα αντίστοιχα.

6. <u>Transform</u>

Το Transform αποτελεί μια από τις πιο κοινές κλάσεις καθώς σχεδόν κάθε αντικείμενο περιέχει το component Transform. Το component αυτό καθορίζει τις ιδιότητες ενός αντικειμένου το οποίο βρίσκεται στον χώρο. Δηλαδή, μέσω αυτού του component προσδιορίζονται ιδιότητες όπως η τοποθεσία, η κλίση ή η κλίμακα ενός αντικειμένου, σε παγκόσμιες ή τοπικές συντεταγμένες, και παρέχονται λειτουργίες για τη μετακίνησή του.

Σχεδόν κάθε αντικείμενο για το οποίο γράφουμε κάποιο script έχει ένα αντικείμενο transform της κλάσης Transform το οποίο απευθύνεται στον εαυτό του.

Τα πεδία που αφορούν στην τοποθεσία, κλίση και κλίμακα του αντικειμένου στο οποίο είναι ενσωματωμένο το script είναι τα εξής:

- transform.position
- transform.rotation
- transform.lossyScale

Τα πεδία αυτά αναφέρονται στη τοποθεσία, τη κλίση και τη κλίμακα ως προς τον υπόλοιπο κόσμο. Σε περίπτωση όμως που ένα αντικείμενο είναι παιδί ενός άλλου και θέλουμε να αναφερθούμε στα χαρακτηριστικά αυτά σε σχέση με το αντικείμενο-γονιό χρησιμοποιούνται τα εξής αντίστοιχα πεδία:

- transform.localPosition
- transform.localRotation
- transform.localScale

Σημειώνεται πως τα πεδία transform.position, transform.lossyScale, transform.localPosition και transform.localScale είναι τύπου Vector3 ενώ τα πεδία transform.rotation και transform.localRotation είναι τύπου Quaternion και για να τα προσπελάσουμε ως Vector3 χρησιμοποιούμε τα πεδία transform.rotation.eulerAngles και transform.localRotation.eulerAngles. Ακολουθεί παράδειγμα αξιοποίησης των πεδίων αυτών:

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class Example : MonoBehaviour {
5
6     private Vector3 vector;
7
8     // Use this for initialization
9     void Start () {
10        transform.position = vector;
11        transform.rotation = Quaternion.Euler(vector);
12        transform.lossyScale = new Vector3(transform.lossyScale.x + 1, 1, 1);
13    }
```

Υπάρχουν αρκετοί τρόποι να μεταβληθούν τα πεδία αυτά όπως φαίνεται στο παράδειγμα, και οι διαφορετικοί αυτοί τρόποι μπορούν να εφαρμοστούν και στα άλλα πεδία. Για παράδειγμα, θα μπορούσαμε κάλλιστα να χρησιμοποιήσουμε το εξής:

```
7     void Start () {
8         transform.rotation = Quaternion.Euler(new Vector3(transform.lossyScale.x + 45,
9                                                           transform.position.y,
10                                                          transform.rotation.eulerAngles.z));
11    }
```

(Εννοείται πως τα παραδείγματα δεν έχουν κάποια συγκεκριμένη λειτουργικότητα.)

Οι ίδιες ακριβώς λειτουργίες μπορούν να πραγματοποιηθούν με τους ίδιους τρόπους και στα αντίστοιχα local πεδία.

Επίσης, η κλάση Transform περιέχει πεδία που μας επιτρέπουν να προσδιορίσουμε Vector3 αντικείμενα ως προς το συγκεκριμένο transform. Τα πεδία αυτά είναι τα:

- transform.forward
- transform.right
- transform.up

Δεν υπάρχει λόγος να υπάρχουν τα αντίστοιχα back, left και down καθώς αντιστοιχούν απλά στις αρνητικές τιμές των παραπάνω. Η χρήση τους γίνεται με τον ίδιο ακριβώς τρόπο όπως τα πεδία της κλάσης Vector3 που ορίζουν κατευθύνσεις, και για το λόγο αυτό δεν κρίνεται σκόπιμο να παρατεθούν παραδείγματα.

Επιπλέον, εντός της κλάσης Transform περιλαμβάνονται και οι εξής χρήσιμες συναρτήσεις:

- transform.Translate(Vector3 Translation);
- transform.Rotate(Vector3 EulerAngles);

Η συνάρτηση transform.Translate ουσιαστικά μεταφέρει ένα Transform προς την κατεύθυνση του Vector3. Η μεταφορά αυτή γίνεται στιγμιαία και λειτουργεί με ένα σκεπτικό τηλεμεταφοράς. Ωστόσο, σε πολλές περιπτώσεις η συνάρτηση αυτή χρησιμοποιείται για να προσομοιώσει κίνηση με το να μετακινείται το αντικείμενο κατά μικρά διαστήματα συνεχόμενα (π.χ. στη συνάρτηση Update). Ένα παράδειγμα της συγκεκριμένης συνάρτησης εν δράσει είναι το εξής:

```
14      void Update () {
15          transform.Translate(transform.forward * 5 * Time.deltaTime);
16      }
```

Το συγκεκριμένο κομμάτι κώδικα μεταφέρει το αντικείμενο στο οποίο είναι προσαρτημένο το script προς την ευθεία για αυτό κατεύθυνση συνεχόμενα. Ο αριθμός 5 προσδιορίζει τη ταχύτητα με την οποία το αντικείμενο μετακινείται. Όσο πιο μεγάλος ο αριθμός στη θέση του 5 τόσο μεγαλύτερη η ταχύτητα.

Τέλος, αξίζει να παρατηρηθεί το πεδίο Time.deltaTime. Το συγκεκριμένο κομμάτι παρέχει το χρόνο ανάμεσα στα frames του παιχνιδιού και είναι ΑΠΑΡΑΙΤΗΤΟ για να λειτουργούν τέτοιες συναρτήσεις ομαλά σε διαφορετικά συστήματα όπου τα fps είναι διαφορετικά.

Η συνάρτηση transform.Rotate λειτουργεί και συντάσσεται με παρόμοιο τρόπο όπως η transform.Translate, μόνο που η λειτουργία της είναι να περιστρέφει το αντικείμενο γύρω από τον άξονα EulerAngles που προσδιορίζεται αντίστοιχα με κάποια κατεύθυνση πολλαπλασιασμένη με έναν αριθμό που δηλώνει τη ταχύτητα περιστροφής και με το Time.deltaTime για τους παραπάνω λόγους.

7. GameObject

Όπως αναφέρθηκε, το GameObject είναι η γενικότερη κατηγορία που χαρακτηρίζει σχεδόν κάθε αντικείμενο που μπορεί να υπάρξει σε μια σκηνή, από τον παίκτη μέχρι ένα κουμπί του User Interface. Επειδή ακριβώς η κλάση αυτή είναι τόσο γενική δεν έχει πολλά πεδία ή συναρτήσεις που μας αφορούν. Ωστόσο, κρίνεται σκόπιμο να αναφερθεί η εξής συνάρτηση:

gameObject.SetActive(bool value);

Σημειώνεται πως η παρούσα συνάρτηση εφαρμόζεται σε αντικείμενο της κλάσης GameObject και ΔΕΝ είναι static πεδίο της κλάσης [όπως το GameObject.Find() που θα αναφερθεί παρακάτω]. Συνεπώς, η συνάρτηση αυτή αφορά στο αντικείμενο στο οποίο είναι προσαρτημένο το script. Η λειτουργία της συνάρτησης αυτής είναι, προφανώς, να ενεργοποιεί και να απενεργοποιεί το αντικείμενο και, κατά συνέπεια να το εμφανίζει και να το κρύβει αντίστοιχα.

8. Find & GetComponent

Για λόγους καλής πρακτικής, σε πολλές περιπτώσεις δεν θα θέλαμε να υπάρχουν πολλά public πεδία για αποφυγή του υπερβολικού drag and drop, οπότε θα ήταν πιο χρήσιμο να είχαμε private πεδία. Για να δώσουμε τα απαραίτητα components στα πεδία αυτά μέσω κώδικα χρησιμοποιούνται συχνά συναρτήσεις όπως η Find και η GetComponent. Η σύνταξη των δύο αυτών συναρτήσεων είναι η εξής:

- GameObject.Find(string name);
- GetComponent<[Τύπος Component]>(); ή
  transform.GetComponent<[Τύπος Component]>(); ή
  gameObject. GetComponent<[Τύπος Component]>();

Στην περίπτωση της συνάρτησης Find δεν αναφερόμαστε σε συγκεκριμένο αντικείμενο αλλά στην κλάση GameObject διότι η συνάρτηση Find είναι static. Αυτό διακρίνεται από το γεγονός πως το GameObject.Find(string name); έχει το πρώτο γράμμα κεφαλαίο, γεγονός που υποδεικνύει αναφορά στην κλάση και όχι σε συγκεκριμένο αντικείμενο.

Αντίθετα, η συνάρτηση GetComponent αναφέρεται σε κάποιο συγκεκριμένο transform ή gameObject προκειμένου να ανακτήσει το component του αντίστοιχου τύπου.

Ακολουθούν παραδείγματα χρήσης των παραπάνω συναρτήσεων:

```
7     void Start () {
8         AudioListener listener = GetComponent<AudioListener>();
9         Light light = GameObject.Find("Lamp").GetComponent<Light>();
10        Transform item = GameObject.Find("Item").transform;
11    }
```

Παρατηρούνται τα εξής:

- Στη γραμμή 8 δημιουργείται μια αναφορά τύπου AudioListenerη οποία αντιστοιχίζεται στο component AudioListener που έχει το αντικείμενο στο οποίο προσαρτάται το script. Σε περίπτωση που δεν βρεθεί το component αυτό, θα υπάρξει error εντός του παιχνιδιού.
- Στη γραμμή 9 δημιουργείται μια αναφορά τύπου Light η οποία αντιστοιχίζεται στο component Light ενός αντικειμένου στη σκηνή με το όνομα "Lamp". Είναι σημαντικό να υπάρχει ακρίβεια στο όνομα.
- Στη γραμμή 10 δημιουργείται μια αναφορά τύπου Transform η οποία αντιστοιχίζεται στο component Transform του αντικειμένου με το όνομα "Item" όπως ακριβώς και στη γραμμή 9. Το συγκεκριμένο παρατίθεται για να τονιστεί το γεγονός πως το πεδίο transform θα μπορούσε κάλλιστα να αντικατασταθεί από μια συνάρτηση GetComponent<Transform>.

Σημειώνεται πως η συνάρτηση Find ψάχνει σε όλα τα αντικείμενα της σκηνής για να βρει το αντικείμενο που ψάχνει, ενώ η συνάρτηση GetComponent ψάχνει σε όλα τα components του αντίστοιχου αντικειμένου. Για το λόγο αυτό αυτές οι συναρτήσεις μπορούν να είναι αρκετά κοστοβόρες και δεν θα ήταν σοφό να χρησιμοποιούνται σε συναρτήσεις όπως η Update.

9. <u>Rigidbody</u>

Η κλάση Rigidbody είναι η κλάση που περιλαμβάνει όλες τις λειτουργίες που αφορούν την αναπαράσταση ιδιοτήτων ενός αντικειμένου ως αντικείμενο στον φυσικό κόσμο. Ένα αντικείμενο με Rigidbody component θα μπορεί να το επηρεάζει η βαρύτητα, να αντιδράει με το περιβάλλον του καθώς και να δέχεται και να δυνάμεις. Με βάση τις συναρτήσεις τις κλάσης αυτής προσομοιώνουμε φυσικές ιδιότητες σε ένα αντικείμενο, και για τον λόγο αυτό, ένα αντικείμενο με το component αυτό θα είναι αρκετά κοστοβόρο σε σχέση με αντικείμενα χωρίς το Rigidbody component. Πλέον, καθώς και η Unity3D σαν engine αλλά και οι υπολογιστές εξελίσσονται, υπάρχει δυνατότητα υποστήριξης πολλών αντικειμένων με Rigidbody ταυτόχρονα, χωρίς,

ωστόσο, αυτό να σημαίνει πως δεν θα έπρεπε να είμαστε σχετικά φειδωλοί με αυτά τα αντικείμενα.

Η συνάρτηση της κλάσης που θα συναντάται πιο συχνά θα είναι η AddForce, η οποία εμφανίζεται ως εξής:

```
4 public class Example : MonoBehaviour {
5
6     private Rigidbody rb;
7
8     // Use this for initialization
9     void Start () {
10        rb = GetComponent<Rigidbody>();
11        rb.AddForce(Vector3.up * 5);
12    }
```

Για να έχουμε πρόσβαση στο Rigidbody component του αντικειμένου (εφόσον αυτό υπάρχει) το αποθηκεύουμε σε μια private μεταβλητή χρησιμοποιώντας την GetComponent. Ειδάλλως,κάθε φορά που θέλαμε να αναφερθούμε σε κάποιο πεδίο ή συνάρτηση του component αυτού θα έπρεπε να χρησιμοποιούσαμε την συνάρτηση GetComponent, γεγονός κακό για την επίδοση του script όπως αναφέρθηκε παραπάνω.

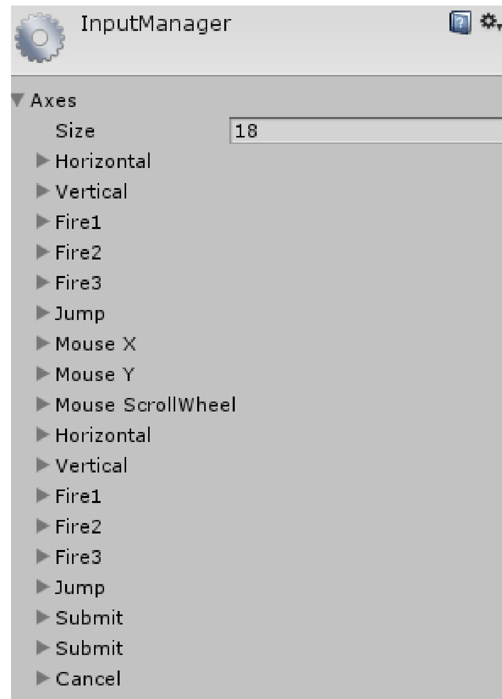
Η AddForce συντάσσεται όπως και η συνάρτηση Translate της κλάσης Transform. Δέχεται σαν κυρίως όρισμα ένα Vector3 το οποίο δηλώνει την κατεύθυνση προς την οποία ασκούμε δύναμη στο αντικείμενο. Στο συγκεκριμένο παράδειγμα, κατά την αρχή του προγράμματος ασκείται μια δύναμη προς τα πάνω με παράγοντα 5 στο αντικείμενο.

Να σημειωθεί πως, αφού ισχύουν διαφορετικές υλοποιήσεις της φυσικής σε 3D και 2D χώρο, υπάρχει σαν κλάση το Rigidbody2D που περιλαμβάνει αντίστοιχες λειτουργίες με αυτές του Rigidbody, απλά εφαρμοσμένες στο 2D χώρο.

Σημειώνεται επίσης, πως η χρήση της AddForce γίνεται συχνά για να προσομοιωθεί το άλμα σε έναν χαρακτήρα.

10. <u>Inputs</u>

Εδώ θα αναφερθούν οι βασικές συνθήκες που χρησιμοποιούνται για την εισαγωγή εισόδου από τον χρήστη. Η είσοδος αυτή μπορεί να αφορά στην ανίχνευση εισόδου είτε από πληκτρολόγιο είτε από ποντίκι, και μπορεί να γίνει μέσω του Input Manager της Unity3D ή με πιο hard coded τρόπους.



- Μέσω Input Manager: Το Input Manager αποτελεί μέρος των ρυθμίσεων της Unity3D και είναι προσβάσιμο μέσω του μονοπατιού Edit > Project Settings > Input. Σε αυτό περιλαμβάνονται άξονες και κουμπιά που ορίζονται μέσα στη μηχανή και αντιστοιχίζονται σε διάφορους τρόπους εισόδου ανάλογα με τις επιλογές εξαγωγής του παιχνιδιού. Για παράδειγμα, ο άξονας Horizontal αντιστοιχεί στα κουμπιά A και D καθώς και στο αριστερό και δεξί βέλος του πληκτρολογίου, ενώ ταυτόχρονα αντιστοιχεί και στις οριζόντιες κινήσεις ενός joystick ή ενός χειριστηρίου. Όλη αυτή η πληροφορία μπορεί να προσπελαστεί μέσω των εντολών που θα δειχθούν παρακάτω. Σημειώνεται πως είναι καλύτερο να χρησιμοποιείται η είσοδος μέσω του Input Manager καθώς δίνει μεγαλύτερη ελευθερία στην ανάθεση πλήκτρων και μεθόδων εισόδου και επιτρέπει στον παίκτη την αλλαγή αυτών των ρυθμίσεων πιο εύκολα.
    - Input.GetAxis(string AxisName)/Input.GetAxisRaw(string AxisName)
      Οι συναρτήσεις αυτές επιστρέφουν τιμές από -1 έως 1 οι οποίες αντιστοιχούν στο ποια είσοδος δόθηκε βάσει του άξονα. Έστω το εξής παράδειγμα:

```
void Update()
{
    if (Input.GetAxis("Horizontal") > 0) {
        //Move right
    }
    else if (Input.GetAxis("Horizontal") < 0) {
        //Move left
    }
    if (Input.GetAxisRaw("Vertical") > 0) {
        //Move up
    }
    else if (Input.GetAxisRaw("Vertical") < 0) {
        //Move down
    }
}
```

Με τον τρόπο που παρουσιάζεται στο παράδειγμα θα ήταν δυνατό να κατασκευαστεί ένας controller για έναν χαρακτήρα. Εντός της Update θα γίνονται με τη σειρά έλεγχοι για το αν ο παίκτης πατάει κάτι που αντιστοιχεί στο «δεξιά», έπειτα στο «αριστερά», στο «πάνω» και στο «κάτω».

Το 1 στον οριζόντιο άξονα αντιστοιχεί στο «δεξιά» ενώ το -1 στο «αριστερά». Αντίστοιχα, το 1 στον κατακόρυφο άξονα αντιστοιχεί στο «πάνω» ενώ το -1 στο «κάτω».

Η διαφορά μεταξύ GetAxis και GetAxisRaw είναι πως με την πρώτη συνάρτηση υπάρχει μια μικρή χρονική μετάβαση από το 0 μέχρι το 1 ή το -1, παίρνοντας τις ενδιάμεσες πραγματικές τιμές, ενώ με την GetAxisRaw το αποτέλεσμα θα είναι μόνο -1, 0 ή 1 και όχι κάποια ενδιάμεση τιμή.

Στα πλαίσια του υπολογιστή, ο οριζόντιος άξονας αντιστοιχεί στα A, D/αριστερό βέλος, δεξί βέλος, ενώ ο κατακόρυφος στα W, S/πάνω βέλος, κάτω βέλος.

- Input.GetButton(string ButtonName)/
  Input.GetButtonDown(string ButtonName)/
  Input.GetButtonUp(string ButtonName)

  Οι συναρτήσεις αυτές επιστρέφουν bool και εκφράζουν το αν είναι πατημένο το κουμπί στο όρισμα, αν πατήθηκε μία φορά ή αν σταμάτησε να πατιέται αντίστοιχα. Ο τρόπος χρήσης του είναι ο παρακάτω:

```
void Update()
{
    if (Input.GetButton("Fire1")) {
        //Shoot
    }
    if (Input.GetButtonDown("Jump")) {
        //Jump
    }
    if (Input.GetButtonUp("Fire1")) {
        //Stopped shooting
    }
}
```

Σημειώνεται πως στα πλαίσια του υπολογιστή, το πλήκτρο που αντιστοιχεί στο "Fire1" είναι το αριστερό κλικ και το αριστερό Control, ενώ για το κουμπί "Jump" το κουμπί είναι το Space.
Η πληροφορία ως προς το σε ποια πλήκτρα αντιστοιχούν τα κουμπιά και οι άξονες βρίσκεται εντός του Input Manager.

- Hard coded λύσεις: Υπάρχουν και ορισμένες συναρτήσεις που ελέγχουν πότε κάποιο συγκεκριμένο πλήκτρο ή κουμπί πατήθηκε. Οι συναρτήσεις αυτές έχουν το μειονέκτημα πως είναι πολύ συγκεκριμένες και δεν απευθύνονται σε εύρος τρόπων εισόδου καθώς παίρνουν σαν όρισμα ένα κωδικό που απευθύνεται σε συγκεκριμένο πλήκτρο του πληκτρολογίου ή του ποντικιού. Επίσης, αν δεν γίνουν με τέτοιο τρόπο ώστε ο κωδικός αυτός να είναι δημόσιος, ο παίκτης ΔΕΝ θα μπορεί να αλλάξει τις λειτουργίες των πλήκτρων. Παρά τα μειονεκτήματα αυτά, κρίνεται σκόπιμο να γίνουν αναφορές στις συναρτήσεις αυτές:
    - Input.GetKey(KeyCode key)/
      Input.GetKeyDown(KeyCode key)/
      Input.GetKeyUp(KeyCode key)

      Αυτό το σύνολο συναρτήσεων λειτουργεί με ίδιο τρόπο όπως οι αντίστοιχες Input.GetButton με τη διαφορά ότι απευθύνονται αποκλειστικά σε είσοδο πληκτρολογίου και ότι σαν όρισμα παίρνουν ένα αντικείμενο τύπου KeyCode το οποίο έχει τη μορφή «KeyCode.πλήκτρο». Χρήση των συναρτήσεων φαίνεται στο παρακάτω παράδειγμα:

```
void Update()
{
    if (Input.GetKey(KeyCode.W)) {
        //Move up
    }
    if (Input.GetKey(KeyCode.S)) {
        //Move down
    }
    if (Input.GetKeyDown(KeyCode.Space)) {
        //Jump
    }
    if (Input.GetKeyUp(KeyCode.W)) {
        //Stop moving up
    }
}
```

- Input.GetMouseButton(int button)/
  Input.GetMouseButtonDown(int button)/
  Input.GetMouseButtonUp(int button)

  Οι συναρτήσεις αυτές δουλεύουν με όμοιο τρόπο με τις πιο πάνω, με τη διαφορά πως απευθύνονται σε πλήκτρα ποντικιού και παίρνουν σαν πιο συχνά ορίσματα τους αριθμούς 0,1 και 2 που αντιστοιχούν στο αριστερό, δεξί και μεσαίο κλικ αντίστοιχα. Χρήση τους φαίνεται στο εξής παράδειγμα:

```
void Update()
{
    if (Input.GetMouseButton(0)) {
        //Shoot
    }
    if (Input.GetMouseButtonUp(0)) {
        //Stop shooting
    }
    if (Input.GetMouseButtonDown(1)) {
        //Throw grenade
    }
}
```

- Σε όλα τα τελευταία παραδείγματα ισχύει πως όταν η συνάρτηση έχει "Down" στο τέλος ελέγχεται μόνο τη στιγμή που θα πατηθεί το πλήκτρο/κουμπί, όταν έχει "Up" ελέγχεται η στιγμή που θα σταματήσει να πατιέται και όταν δεν έχει κάποια από αυτές τις λέξεις στο τέλος, ελέγχεται συνέχεια αν πατιέται.
- Σημειώνεται επίσης πως θέλουμε τέτοιες συναρτήσεις ελέγχου εισόδου να είναι σε κάποια ταχεία επαναλαμβανόμενη διαδικασία όπως η Update, καθώς ο έλεγχος για είσοδο θέλουμε να γίνεται όσο γίνεται πιο τακτικά.

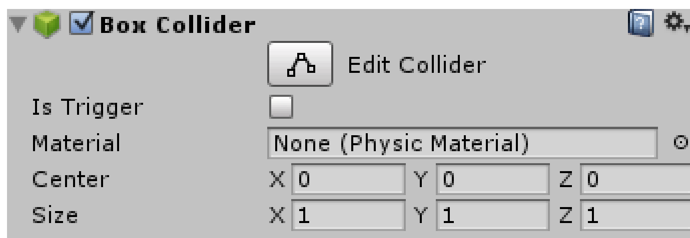
11. Colliders & Triggers

Ίσως από τα πιο σημαντικά στοιχεία ενός video game είναι τα events που συμβαίνουν με διάδραση στα αντικείμενα, όπως με τη σύγκρουση μαζί τους ή με την είσοδο σε μια συγκεκριμένη περιοχή. Τέτοια events μπορούν να χρησιμοποιηθούν σε πληθώρα εφαρμογών, όπως, για παράδειγμα, ένα σύστημα μάχης, ή ένα σύστημα που πυροδοτεί διαφορετικά event με την είσοδο του παίκτη σε έναν χώρο.

Τόσο οι colliders όσο και τα triggers αποτελούν περιοχές στο χώρο (2D ή 3D) οι οποίες μπορεί να εξυπηρετούν διάφορους σκοπούς. Η κύρια διαφορά τους είναι πως οι colliders εμποδίζουν στοιχεία με άλλους colliders να προσπελάσουν τη περιοχή και

πρακτικά αποτελούν «αόρατους τοίχους», ενώ τα triggers είναι αόρατες περιοχές στις οποίες οποιοδήποτε αντικείμενο έχει πρόσβαση. Συνεπώς, οι colliders υπάρχουν για να προσομοιώνουν τη σύγκρουση δυο φυσικών αντικειμένων (και να μεταφέρουν οποιαδήποτε πληροφορία αφορά τη σύγκρουση αυτή), ενώ τα triggers αφορούν στη μετάδοση πληροφοριών για τα αντικείμενα που εισέρχονται, μένουν ή εξέρχονται από αυτά. Σημειώνεται πως προκειμένου τα triggers να αναγνωρίσουν ένα αντικείμενο, αυτό το αντικείμενο θα πρέπει να έχει collider.

Ο τρόπος εισαγωγής ενός collider γίνεται μέσω του αντίστοιχου component (Box collider, Sphere collider κλπ):



Προκειμένου το συγκεκριμένο component να λειτουργεί ως trigger αρκεί να γίνει checked το κουτί "Is Trigger". Παρατηρείται πως υπάρχουν πεδία που ορίζουν τι περιοχή θα καλύπτει αυτός ο collider καθώς και ένα πεδίο για Physic Material. Συνοπτικά, τα Physic Materials είναι στοιχεία που περιέχουν φυσικές ιδιότητες για έναν collider, όπως το πόση τριβή θα έχει ή το πόσο bouncy θα είναι. Εφόσον δεν δοθεί κάποιο, διατηρεί default τιμές.
Σημειώνεται, επίσης, πως για 2D αντικείμενα, υπάρχουν αντίστοιχοι 2D colliders/triggers (Box collider 2D κλπ).

Οι συναρτήσεις οι οποίες καλούνται ανάλογα με τη διάδραση με ένα collider/trigger είναι οι εξής:

- void OnCollisionEnter(Collision other)
- void OnCollisionStay(Collision other)
- void OnCollisionExit(Collision other)
- void OnTriggerEnter(Collider other)
- void OnTriggerStay(Collider other)
- void OnCollisionExit(Collider other)

Για 2D triggers/colliders οι συναρτήσεις έχουν την εξής μορφή:

- void OnCollisionEnter2D(Collision2D other)

- void OnCollisionStay2D(Collision2D other)
- void OnCollisionExit2D(Collision2D other)
- void OnTriggerEnter2D(Collider2D other)
- void OnTriggerStay2D(Collider2D other)
- void OnCollisionExit2D(Collider2D other)

Αξίζει να σημειωθεί πως οι συναρτήσεις που αφορούν collisions έχουν σαν όρισμα ένα Collision (δομή με πληροφορίες ως προς την επαφή-σύγκρουση των αντικειμέννω) ενώ οι συναρτήσεις που αφορούν triggers έχουν σαν όρισμα τον Collider του αντικειμένου με το οποίο ήρθε σε επαφή το παρόν αντικείμενο.

Όπως είναι ίσως εμφανές από το όνομα, ανάλογα με την κατάληξη (Enter, Stay ή Exit) η κάθε συνάρτηση θα εκτελεστεί κατά την πρώτη επαφή, όσο γίνεται επαφή ή όσο σταματάει να γίνεται επαφή με κάποιο αντικείμενο, αντίστοιχα με τις συνθήκες εισόδου (GetKeyDown/GetKey/GetKeyUp).

Τέλος, σημειώνεται πως προκειμένου να κληθούν οι συναρτήσεις αυτές, θα πρέπει τουλάχιστον ένα από τα 2 αντικείμενα που έρχονται σε επαφή να έχουν ένα Rigidbody component (και στην περίπτωση της οικογένειας συναρτήσεων OnCollision, θα πρέπει το Rigidbody component να είναι non-kinematic).

Ακολουθεί σύντομο παράδειγμα χρήσης συναρτήσεων των παραπάνω οικογενειών:

```
void OnCollisionEnter(Collision other)
{
    if (other.transform.tag == "Enemy")
    {
        //Get hit
    }
}

void OnTriggerStay(Collider other)
{
    if (other.tag == "Player")
    {
        //Lose life
    }
}
```

Ο έλεγχος για το tag γίνεται ώστε οι συναρτήσεις να καλούνται κατά την επαφή με συγκεκριμένα αντικείμενα, και όχι με κάθε collider στη σκηνή.

12. <u>Coroutines</u>

Τα coroutines αποτελούν ειδικό είδος συναρτήσεων οι οποίες έχουν την ιδιότητα να σταματάνε την εκτέλεσή τους και να τη συνεχίζουν στο επόμενο frame. Είναι πάρα πολύ χρήσιμα σε περιπτώσεις που αφορούν σταδιακή μεταβολή μιας τιμής καθώς μπορούν να συμπεριφέρονται σαν τη συνάρτηση Update, ενώ στη πραγματικότητα εκτελούνται παράλληλα με αυτή. Επιπλέον, τα coroutines είναι ο μοναδικός τρόπος στη C# να δηλωθεί πως κάποιες εντολές θα εκτελεστούν μετά το πέρασμα ορισμένων δευτερολέπτων. Ακολουθεί παράδειγμα που υποδεικνύει τον τρόπο σύνταξης της δήλωσης ενός coroutine:

```
private int value = 10;

IEnumerator CoRoutine()
{
    while (value > 0)
    {
        value--;
        yield return null;
    }
}
```

Το παρόν παράδειγμα μειώνει σε κάθε frame τη τιμή του value μέχρι αυτή να γίνει 0. Αν ο αντίστοιχος κώδικας ήταν σε μια απλή συνάρτηση, τότε το σώμα της συνάρτησης θα εκτελούταν σε ένα frame και απλά θα έκανε τη τιμή του value ίση με το 0 κατευθείαν.

```
IEnumerator CoRoutine()
{
    yield return new WaitForSeconds(10f);
    value = 0;
}
```

Το συγκεκριμένο coroutine όταν αρχίσει να εκτελείται θα περιμένει 10 δευτερόλεπτα και έπειτα θα εκχωρήσει το 0 στη τιμή του πεδίου value.

```
IEnumerator CoRoutine()
{
    while (value > 0)
    {
        value--;
        yield return new WaitForSeconds(1);
    }
}
```

Αυτό το παράδειγμα λειτουργεί ακριβώς όπως και το 1ο, με τη διαφορά πως μετά από κάθε μείωση του πεδίου value, η συνάρτηση θα περιμένει 1 δευτερόλεπτο.

Επίσης, η κλήση ενός coroutine δεν γίνεται όπως σε κάθε άλλη συνάρτηση, αλλά μέσω της συνάρτησης StartCoroutine. Για παράδειγμα:

```
void Start()
{
    StartCoroutine(CoRoutine());
}
```

Τα coroutines μπορούν να έχουν ορίσματα όπως μια οποιαδήποτε συνάρτηση, και δίνονται σαν ορίσματα εντός του coroutine που δίνεται ως όρισμα στη StartCoroutine().

Τέλος, αξίζει να επισημανθούν και άλλες 2 συναρτήσεις: η StopCoroutine([κάποιο coroutine]) και η StopAllCoroutines(). Όπως υποδεικνύει και το όνομα, οι συναρτήσεις αυτές χρησιμοποιούνται για να διακόψουν τη λειτουργία κάποιου συγκεκριμένου ή όλων των coroutines αντίστοιχα.

13. <u>Raycasting</u>

Οι ακτίνες (rays) έχουν μεγάλη χρησιμότητα στα παιχνίδια, καθώς αποτελούν νοητές ευθείες από ένα σημείο σε ένα άλλο αναγνωρίζοντας τυχόν αντικείμενα στην πορεία τους. Συνεπώς, μπορούν να βρουν εφαρμογή στη προσομοίωση του οπτικού πεδίου ενός χαρακτήρα, καθώς και σε μηχανισμούς που αφορούν όπλα. Το raycasting είναι η διαδικασία εκπομπής τέτοιων ακτίνων και η εφαρμογή της μπορεί να έχει την ακόλουθη μορφή:

```
void Update() {
    //Ray's constructor takes the starting position and a direction
    Ray ray = new Ray(transform.position, transform.forward);
    RaycastHit hit;
    if (Physics.Raycast(ray, out hit)) {
        if (hit.collider != null) {
            //We hit something!
            print(hit.collider.name); //Printing its name
        }
    }
}
```

Σημειώνεται πως υπάρχουν και άλλοι τρόποι για να επιτευχθεί το ίδιο αποτέλεσμα, καθώς και ειδικές παράμετροι που μπορούν να προστεθούν για να ορίσουν τη μέγιστη απόσταση της ακτίνας καθώς και ποια layers να αγνοήσει η ακτίνα.

Όπως και με κάθε στοιχείο που αφορά τη βιβλιοθήκη φυσικής της Unity, έτσι και το raycasting έχει ειδική σύνταξη σε περιβάλλον 2D η οποία μπορεί να είναι ως εξής:

```
void Update() {
    RaycastHit2D hit = Physics2D.Raycast(transform.position, transform.right);
    if (hit.collider != null) {
        //We hit something, in 2D!
        print(hit.collider.name);
    }
}
```

Το παράδειγμα αυτό εξυπηρετεί τόσο τον σκοπό της επίδειξης των 2D παραλλαγών για την κλάση RaycastHit και Physics όσο και τον σκοπό της επίδειξης ενός διαφορετικού τρόπου για εκπομπή ενός ray. Με παρόμοιο τρόπο θα μπορούσε να συνταχθεί και το 1ο παράδειγμα. Τέλος, αντίστοιχη ευελιξία διαχείρισης παρέχεται και στο 2D API, σε ό,τι αφορά απόσταση ακτίνας και layers που μπορεί να αγνοήσει η ακτίνα.

Κεφάλαιο 4: Παραδείγματα

1. Απλό 2D character controller

```
using UnityEngine;
using System.Collections;

public class MovementScript : MonoBehaviour {

    public float speed = 10f;
    public bool grounded = true;
    public float jumpforce = 100f;

    private float move;

    private Rigidbody2D rigidbody2D;

    void Awake() {
        rigidbody2D = GetComponent<Rigidbody2D>();
    }

    void Update () {
        move = Input.GetAxisRaw ("Horizontal");
        if (move > 0)
        {
            transform.localScale = new Vector3(1,1,1);
            transform.Translate(Vector3.right * speed * Time.deltaTime);
        }
        if (move < 0)
        {
            transform.localScale = new Vector3(-1,1,1);
            transform.Translate(Vector3.left * speed * Time.deltaTime);
        }
        if (Input.GetButton("Jump") && grounded)
        {
            rigidbody2D.AddForce(Vector2.up * jumpforce);
        }
    }

    public int isFacing()
    {
        if (transform.localScale.x == 1) return 1;
        if (transform.localScale.x == -1) return -1;
        else return 0;
    }

    void OnCollisionStay2D(Collision2D col)
    {
        if (col.collider.tag == "ground") grounded = true;
    }

    void OnCollisionExit2D(Collision2D col)
    {
        if (col.collider.tag == "ground") grounded = false;
    }

}
```

2. Συμπεριφορά για όπλο με χρήση Raycasting σε 2D περιβάλλον

```
public class Weapon : MonoBehaviour {

    public float fireRate = 0.5f;
    public float Damage = 10f;
    public LayerMask whatToHit;

    private float timeToFire = 0;
    private Transform firePoint;


    // Use this for initialization
    void Start () {
        firePoint = transform.FindChild ("FirePoint");
        if (firePoint == null) {
            Debug.LogError("No firepoint - WHAT?!");
        }
    }

    // Update is called once per frame
    void Update () {
        if (fireRate == 0f){
            if (Input.GetButtonDown("Fire1")) {
                Shoot();
            }
        }
        else {
            if (Input.GetButton("Fire1") && Time.time > timeToFire) {
                timeToFire = Time.time + 1/fireRate;
                Shoot();

            }
        }
    }

    void Shoot() {
        Vector2 mousePosition = new Vector2 (Camera.main.ScreenToWorldPoint (Input.mousePosition).x,
                                             Camera.main.ScreenToWorldPoint(Input.mousePosition).y);
        Vector2 firePointPosition = new Vector2 (firePoint.position.x, firePoint.position.y);
        RaycastHit2D hit = Physics2D.Raycast (firePointPosition, mousePosition - firePointPosition, 100, whatToHit);
        Debug.DrawLine (firePointPosition, (mousePosition - firePointPosition) * 100, Color.cyan);
        if (hit.collider != null) {
            Debug.DrawLine(firePointPosition, hit.point, Color.red);
            Debug.Log("We hit " + hit.collider.name + " and did " + Damage + " damage.");
        }
    }
}
```