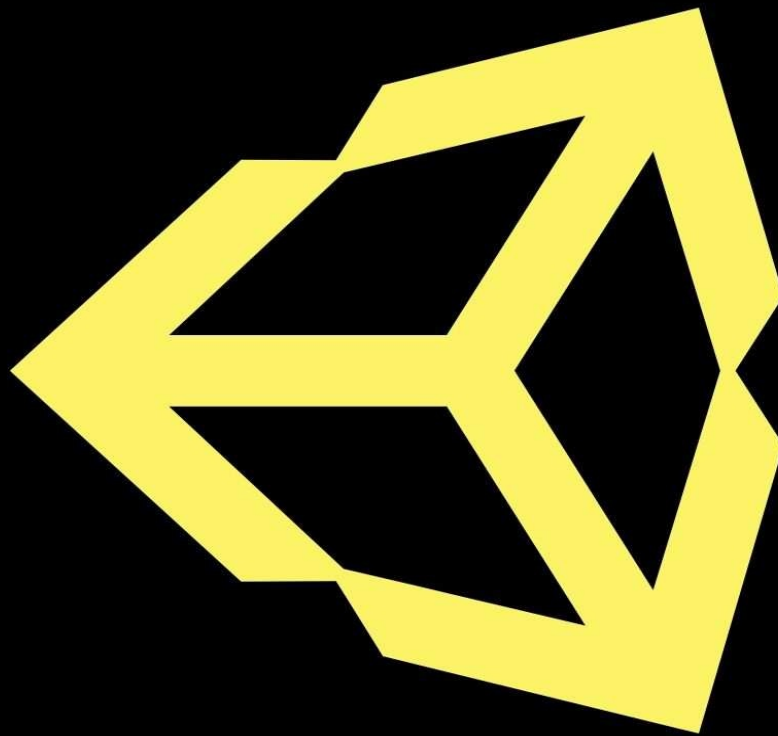


UNITY 2D GAME DEVELOPMENT

Beginner's Guide to 2D game development with Unity



MEM LNC

2020-6

John Bach

1nd edition

Unity 2 d game development

Beginner's Guide to 2D game development with Unity

2nd edition

By
Moaml mohammed

2020

.

"Programming isn't about what you know; it's about what you can figure out . ” - *Chris Pine*

Who this book is for?

If you don't know anything about programming in general, writing code, writing scripts, or have no idea where to even begin, then this book is perfect for you. If you want to make games and need to learn how to write C# scripts or code, then this book is ideal for you.

WHO THIS BOOK IS FOR? 5

INTRODUCTION.....

8

2D OR 3D PROJECTS 8

FULL 3D 9

ORTHOGRAPHIC 3D 10

FULL 2D 11

2D GAMEPLAY WITH 3D GRAPHICS 11

WHY DO WE USE GAME ENGINES? 14

QUICK STEPS TO GET STARTED WITH UNITY ENGINE15

CHAPTER

ONE.....
20

SETTING THE SCENE AND CAMERA IN UNITY3D 20

SCENE BUILDING AND BACKGROUND PROCESSING 24

BUILDING A CAMERA CONTROL SYSTEM 49

BUILDING A SCENE LANDSCAPE 72

CHAPTER

2.....
81

MOTION IN PROGRAMMING 81

CHAPTER III

.....97

CREATE AND LAUNCH PROJECTILES 98

SPECIAL ATTACKS FOR PROJECTILES 114

EJECTOR INDUSTRY 122

CHAPTER

IV.....
..... 144

TOUCH SCREEN INPUT IN UNITY3D AND EXPORT FOR PHONES 145

TIPS FOR DESIGNER GAMES.....
.....163

THE GAME DESIGNER PUTS HIMSELF IN THE PLAYER'S PLACE 164

GAME TESTERS	165
SKILLS	166
THE ABILITY TO COMMUNICATE WITH DIFFERENT PEOPLE	167
THE SKILL OF WORKING WITHIN A TEAM	168
ABLE TO HANDLE MATTERS WHEN NECESSARY	169
ABLE TO DRAW AROUND IT	170
CAPABLE OF CREATIVITY	171

Introduction

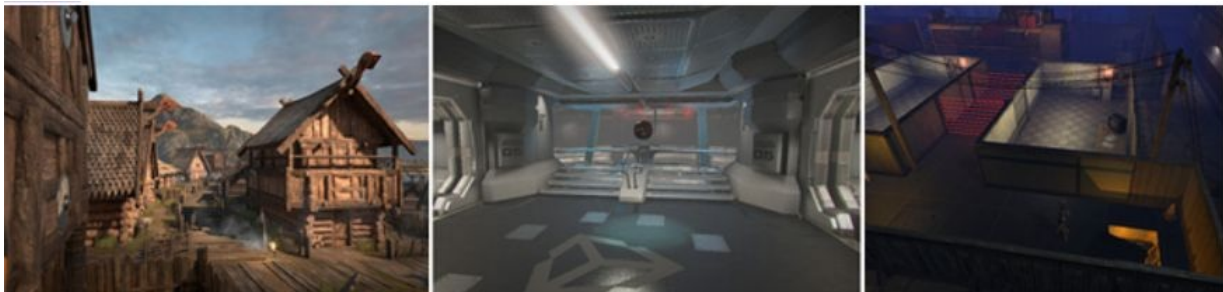
Unity is a cross-platform development platform initially created for developing games but is now used for a wide range of things such as: architecture, art, children's apps, information management, education, entertainment, marketing, medical, military, physical installations, simulations, training, and many more. Unity takes a lot of the complexities of developing games and similar interactive experiences and looks after them behind the scenes so people can get on with designing and developing their games. These complexities include graphics rendering, world physics and compiling. More advanced users can interact and adapt them as needed but for beginners they need not worry about it. Games in Unity are developed in two halves; the first half -within the Unity editor, and the second half -using code,

specifically C#. Unity is bundled with MonoDeveloper or Visual Studio 2015 Community for writing C#.

2D OR 3D PROJECTS

Unity is equally suited to creating both 2D and 3D games. But what's the difference? When you create a new project in Unity, you have the choice to start in 2D or 3D mode. You may already know what you want to build, but there are a few subtle points that may affect which mode you choose. The choice between starting in 2D or 3D mode determines some settings for the Unity Editor -such as whether images are imported as textures or sprites. Don't worry about making the wrong choice though, you can swap between 2D or 3D mode at any time regardless of the mode you set when you created your project. Here are some guidelines which should help you choose.

Full 3D



3D games usually make use of three-dimensional geometry, with materials and textures rendered on the surface of these objects to

make them appear as solid environments, characters and objects that make up your game world. The camera can move in and around the scene freely, with light and shadows cast around the world in a realistic way. 3D games usually render the scene using perspective, so objects appear larger on screen as they get closer to the camera. For all games that fit this description, start in 3D mode.

Orthographic 3D



Sometimes games use 3D geometry, but use an orthographic camera instead of perspective. This is a common technique used in games which give you a bird's-eye view of the action, and is sometimes called "2.5D". If you're making a game like this, you should also use the editor in 3D mode, because even though there is no perspective, you will still be working with 3D models and assets. You'll need to switch your camera and scene view to Orthographic though. (scenes above from Synty Studios and BITGEM)

Full 2D



Many 2D games use flat graphics, sometimes called sprites, which have no three-dimensional geometry at all. They are drawn to the screen as flat images, and the game's camera has no perspective. For this type of game, you should start the editor in 2D mode.

2D gameplay with 3D graphics



Some 2D games use 3D geometry for the environment and characters, but restrict the gameplay to two dimensions. For example, the camera may show a “side scrolling view” and the player can only move in two dimensions, but the game still uses 3D models for the obstacles and a 3D perspective for the camera. For these games, the 3D effect may serve a stylistic rather than functional purpose. This type of game is also sometimes referred to as “2.5D”. Although the gameplay is 2D, you will mostly be manipulating 3D models to build the game so you should start the editor in 3D mode.

Perhaps the gaming industry is one of the most difficult industries in this era, and that is in many ways that start with technical challenges,

passing through an audience that is difficult to satisfy and ruthless even for the major companies if their products are not at the required level, and not an end to fierce competition and high failure rates and the difficulty of achieving profits that cover high production costs.

On the other hand, there are features of this industry that make survival in it possible. On the technical side, for example, the vast majority of games are not free of similar functions and repetitive patterns of data processing, which makes the reuse of the software modules of previous games in order to create new games possible. This, in turn, contributes to overcoming technical obstacles and shortening time and effort.

When you talk about making a game, you are here to mention the big process that involves dozens and possibly hundreds of tasks to accomplish in many areas. Making a game means producing, marketing, and publishing it, and all the administrative, technical, technical, financial, and legal procedures and steps involved in these operations. However, what is important for us in this series of lessons is the technical aspect which is game development, which is the process of building the final software product with all its components. This process does not necessarily include game design, as the design process has a broader perspective and focuses on such things as the story, the general characteristic of the game, the shapes of the stages and the nature of the opponents, as well as the rules of the game, its goals and terms of winning and losing.

Returning to the game development process, we find that many specializations and skills contribute to this process. There are painters, model designers, animation technicians, sound engineers, and director, in addition to - of course - programmers. This comprehensive view is

important to know that the programmer's role in producing the game is only an integral role for the roles of other team members, though this image is beginning to change with the emergence of independent developers Indie Developers who perform many tasks besides programming.

Why do we use game engines?

If we wanted to talk in more detail about the role of programmers in the games industry, we will find that even at the level of programming itself there are several roles that must be taken: there are graphics programming and there are input systems, resource import systems, artificial intelligence, physics simulation and others such as sound libraries and aids. All of these tasks can be accomplished in the form of reusable software modules as I mentioned earlier, and therefore these units together constitute what is known as the Game Engine. By using the engine and software libraries that compose it, you are reducing yourself to the effort needed to build an I / O system, simulate physics, and even a portion of artificial intelligence. What remains is to write the logic of your own game and create what distinguishes it from other games. This last point is what the next series of lessons will revolve around, and although the task seems very small compared to developing the entire game, it is on its smallness that requires considerable effort in design and implementation as we will see.

Quick steps to get started with Unity Engine

If you did not have previous experience with this engine, you can read this quick introduction, and you can skip it if you have dealt with this

engine previously. I will not elaborate on these steps since there are many lessons, whether in Arabic or English, that you take, but here we are to make sure that each series reader has the same degree of initial knowledge before starting.

The first step: **download and install the engine**

To download the latest version of the engine, which is 19, go directly to the website <http://unity3d.com> and then download the appropriate version for the operating system that you are using, knowing that the free version of the engine has great potential and it meets the purpose for our project in this series of lessons.

Step two: **create the project**

Once the engine is running after installing it, the start screen will appear, click New Project to display a screen like the one you see in the image below. All you have to do is choose the type 2D and then choose the name and location of the new project that you will create, and then click on Create Project.

1.The name defaults to New Unity Project but you can change it to whatever you want. Type the name you want to call your project into theProject namefield.

2.The location defaults to your home folder on your computer but you can change it.EITHER(a) Type where you want to store your project on your computer into

theLocationfield. OR(b) Click on the three blue dots '...'. This brings up your computer's Finder (Mac OS X) or File Explorer (Windows OS).

3. Then, in Finder or File Explorer, select the project folder that you want to store your new project in, and select "Choose".

4. Select 3D or 2D for your project type. The default is 3D, coloured red to show it is selected. (The 2D option sets the Unity editor to display its 2D features, and the 3D option displays 3D features. If you aren't sure which to choose, leave it as 3D; you can change this setting later.)

5. There is an option to select Asset packages...to include in your project. Asset packages are pre-made content such as images, styles, lighting effects, and in-game character controls, among many other useful game creating tools and content. The asset packages offered here are free, bundled with Unity, which you can use to get started on your project. EITHER: If you don't want to import these bundled assets now, or aren't sure, just ignore this option; you can add these assets and many others later via the Unity editor. OR: If you do want to import these bundled assets now, select Asset packages...to display the list of assets available, check the ones you want, and then click on Done.

6. Now select Create project and you're all set!

Step Three: **Get to know the main program windows**

At first we got 4 major windows in Unity. Here is a summary of its functions:

Scene Window: It you use to build the game scene, add different objects to it and distribute it in 2D space. Initially this window contains only one object which is the camera.

Hierarchy: contains a tree arrangement that contains all the objects that have been added to the scene and helps you in organizing the relationships between them, as it is possible to add objects as children to other beings so that the son being is affected by the parent being as we will see. Initially this window contains only one object which is the camera.

Project Browser: Displays all files inside the project folder, whether they were added to the scene or not added. The project initially contains one folder called Assets, and inside it we will add all other files and folders.

Inspector Properties Window: When selecting any object from the scene hierarchy, scene window, or project browser, its properties will appear in this window and you can change it from there.

In this introduction, we have reviewed what appears from the Unity3D interface at first glance, with a simple introduction to the game industry, we will embark on the next lessons in a practical project through which we learn how to create a real complete game!

Chapter one

Setting the scene and camera in Unity3D

In the previous lesson, we learned about Unity Engine and how to download it and create a new project on it, we will start in this lesson in a practical project to learn how to make a full real game. In the first lesson, we will talk about setting up the game scene and controlling the camera.

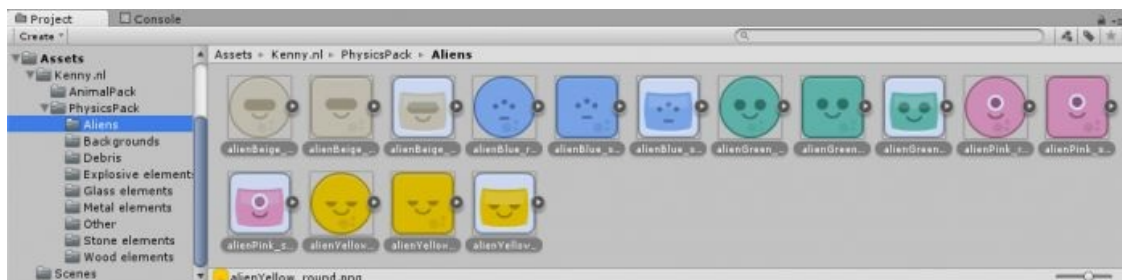
In this example, I will be using a set of free graphics offered under the Creative Commons CC0 license, available at <http://kenney.nl>. The first set is graphics for physical games like Angry Birds, which is the game we are trying to emulate its mechanics across this series. As for the second group, they are simple animal drawings that we will use in place of the birds in the original game, as they represent projectiles. However, we will add some other graphics and sound files as needed. In this series, I will adopt placing each source (image file or sound file) in a folder with the name of the site from which the source was retrieved, so that you can come back to these sites and download files.

After downloading and decompressing these graphics, copy them into the Assets folder of your project. You can copy them via the operating system or drag them directly into the folder in Unity Browser. Once you add these files to the project, Unity will recognize them as sprites, which are the most common types of graphics in 2D games. In order to get the highest possible quality we will modify some graphics

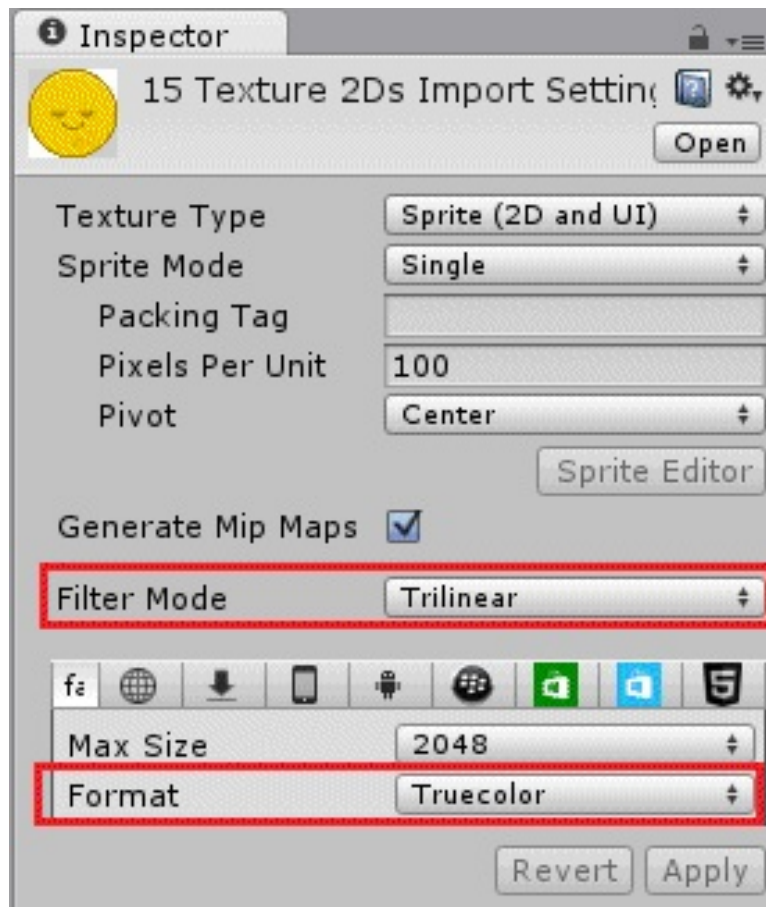
properties, I'll set an example here on one folder and then we have to repeat the process for all graphics folders.

Note: Graphics groups contain two different storage methods: the first is to store each image separately in a separate file, and the second is to collect the images in one file called a sprite sheet. In these lessons, we will discuss the first method, which is separate shapes, for easy work on them. Note that the second method is more efficient in terms of performance and easier in the case of motion graphics.

As you can see in the following figure, I have chosen all the elements of the Monster Graphics folder, which will represent the opponents in this game.



After that, I modified the properties of importing these graphics as follows:



By changing the filter type to Trilinear and coordinating colors to Truecolor, we ask Unity not to compress these images so that they remain at their original size and maintain their quality. Although compression is better for performance, you will notice that Unity tells you that most of these graphics are not compressible because they do not achieve the rule that length and width are equal to logarithmic numbers of base 2 (for example 64, 128, 512, etc.). Don't forget to click Apply to save the changes after they are done.

Scene building and background processing

After changing the properties of all the graphics we imported, we can start preparing the scene. The first step will be to add the background and floor on which the game will revolve. An important specification of the import process is the Pixels Per Unit. The number 100 here means that a 100-pixel streak in the original image covers a distance of one unit in the Unity space. If you assume that the unit is one meter, a box size of 10 x 10 pixels equals a box of 10 x 10 cm in the game space. Fortunately for us, the set of images that we imported are suitable for each other, for example, monster images are 70 x 70 pixels while the backgrounds are much larger and reach 1024 x 1024 pixels.

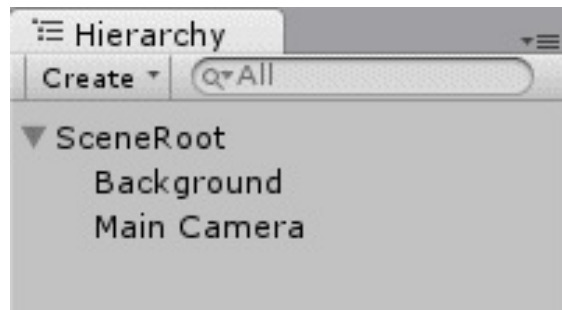
Let's consider from the beginning that this series is not only for learning programming games, it is not only for learning the Unity engine, but you can also say that it will also teach you the best programming and organizational practices for the scene to get the best result with this engine. For this we will prepare the scene of the game in a special way so that the scene contains one object we call the root object of the scene SceneRoot and then we will add all elements of the scene in the form of children and grandchildren of this object. To add a new object, simply select the menu:

Game Object> Create Empty

The new object will then appear in the scene hierarchy as GameObject, choose it and change its name to SceneRoot and its location to the point of origin (0, 0). As mentioned in the introduction to this chapter, you can change these values from the Inspector Properties window as shown in the image below. Note that this object is "empty" as its name indicates, meaning that it is only in memory and cannot be seen in the scene even though it is in the middle of it,

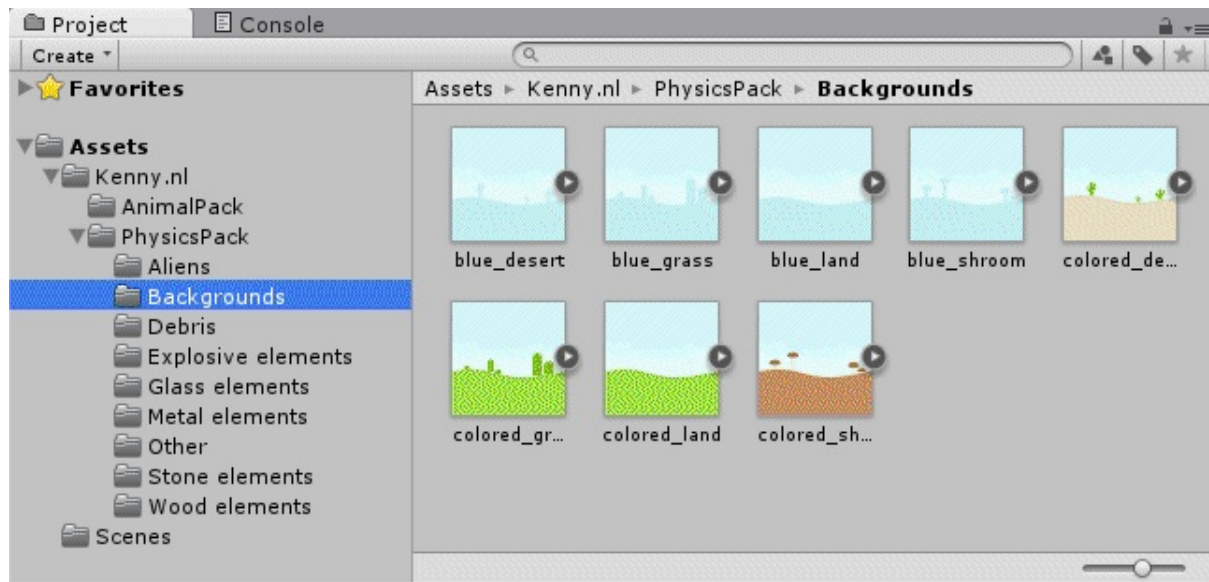
we will come to these details according to our need for it.

Then we will add another blank object to represent the background of the scene, all you have to do is click the right mouse button on the SceneRoot object in the hierarchy and choose Create Empty in order to add a sub-object (son) to the root object and name it Background. Finally you have to drag the Main Camera object into the root object so that the scene hierarchy is like this.

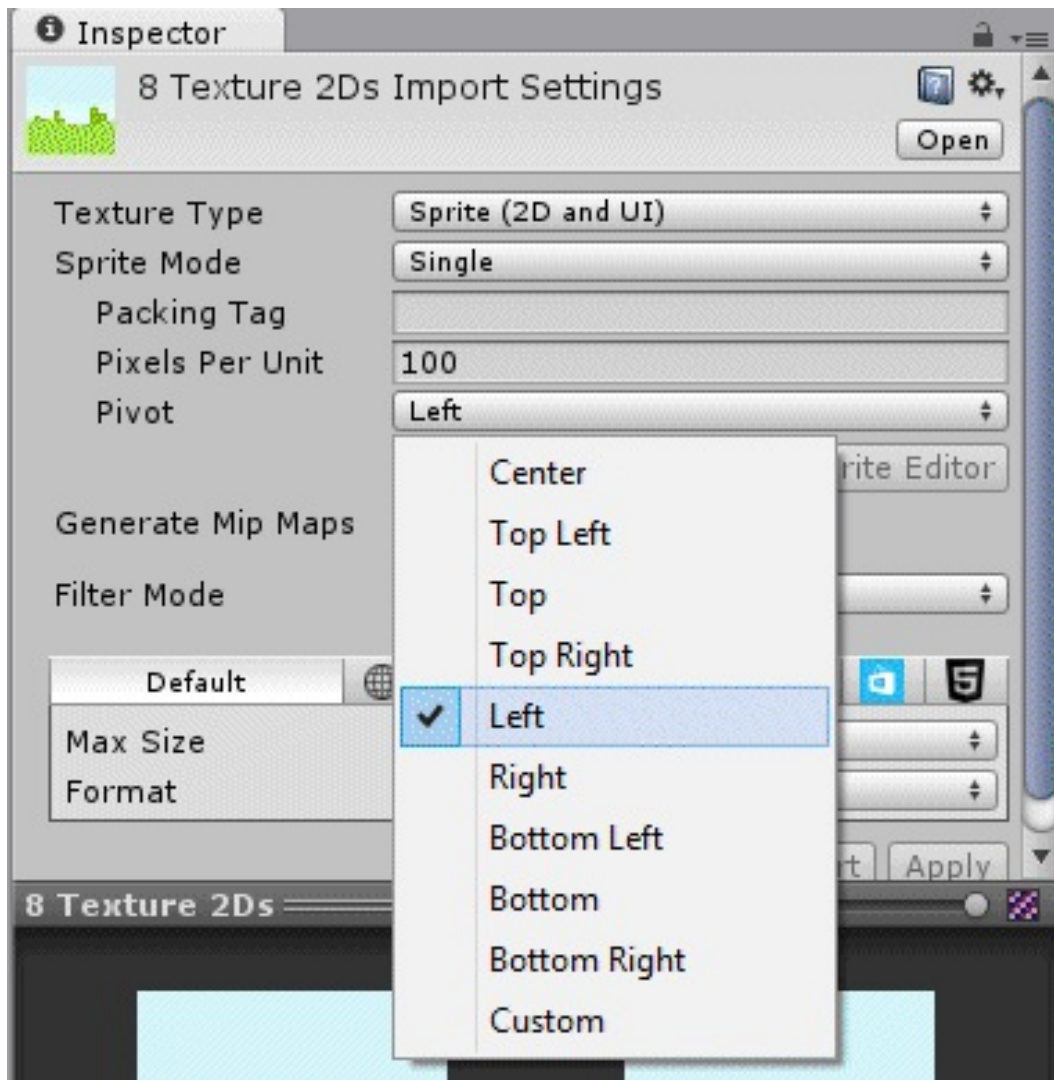


Now is the time to save the scene that we are working on, as is the case in 99% of the programs you can save via Control + S, which is a matter of saving the scene and not saving the entire project, as the project is only a set of files that can be modified and saved separately. Create a new folder inside Assets and name it Scenes and then save the scene inside with the name GameScene.

Let's agree from now that we'll be building one scene for the whole game, and we're going to change both the background, the floor and the object locations to make different stages. The first object will be the scene background, for which there are several images that can be used. These 8 images are in the Backgrounds folder of the first set of photos as you see in the image below. What distinguishes these background images is that each of them applies its right side to the left side completely, which makes us able to repeat it horizontally in succession without the player noticing that there are limits between them.



Since we are dealing with a game with a casual scene, we have to repeat the background more than once horizontally, and be 4 times. Before we start adding backgrounds to the scene, let's make it easier for ourselves to make the pivot for all background images the left side of the image instead of the middle, as you see in the following image:



But what does it mean if the focal point of the image is its left side, not its center? The two pictures below answer this question.

In both cases, the location of the image has not changed, which is (0, 0), i.e. in the middle of the scene. In the default position (left image), the anchor of the image is in the middle of it, that is, the position of the image in space is the middle of it, and the left and right sides of the image extend between the points $x = -5.12$ and $x = + 5.12$. In the second case, we changed the anchor to the far left side of the image horizontally while remaining in the middle vertically. This way, the image runs from the point $x = 0$ left to the point $x = + 10.24$ right.

Note that the values of these points are consistent with the fact that one unit = 100 pixels where the width of the image in units is $10.2 = 1024/100$

With this, we can compute the four image locations that we will stack next to each other from left to right around the point of origin, as follows:

The image on the far left will be at 20.48- and it extends to 10.24-.

The image to the left of the origin point will be at 10.24- and extends to 0.0.

The image to the right of the point of origin will be in position 0.0 and extend to 10.24+.

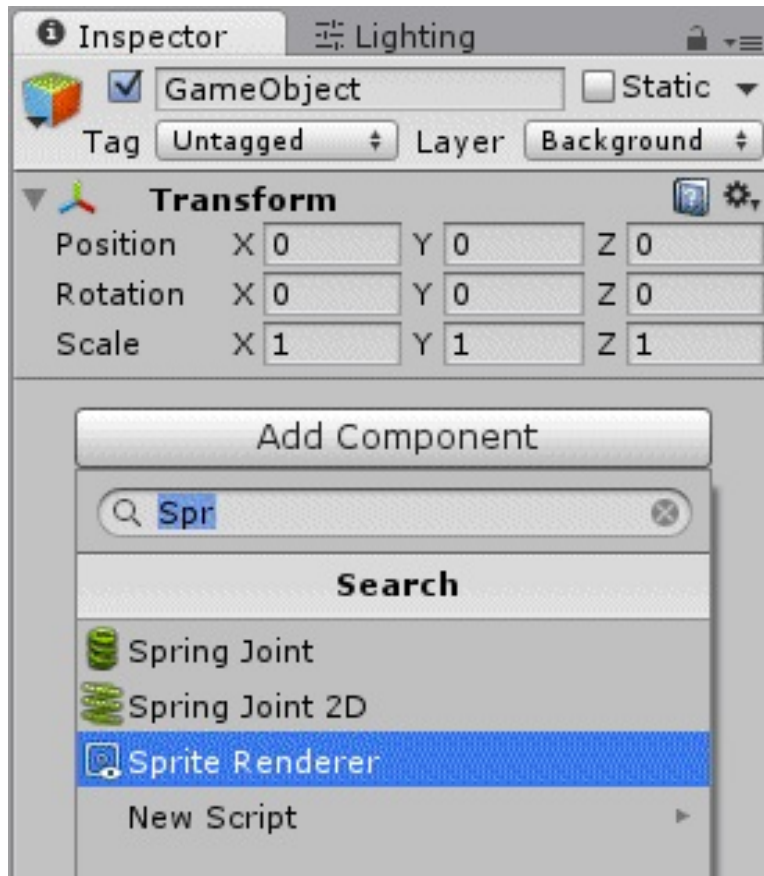
The image on the far right will be at 10.24+, and extends to 20.48+.

Instead of manually placing this image, we will write a script so that we give it an image and ask it to repeat it a number of times to create the background for us. This applet will work as follows: We will give it all the background images we have in the form of a matrix so that each image has its own number, in addition to that we will be able to provide it with the number of repetitions required for the image in order to draw the background. When running the game, we must choose the background number that we want to appear on the screen, and this number will be from 0 to 7. We will return to this applet theme shortly.

Before going into details I would like to point out that the objects in the Unity engine are all similar, and they are in their original empty position invisible, and what distinguishes each object from others and

gives it its own characteristics is the set of components that are added to it. The only component in the empty object is Transform, which is found in all engine objects without exception, as it gives the object its location, rotation and size in the two-dimensional (or three-dimensional) space in addition to that it determines its relationships with other objects such as the father and sons relationship that we see in the hierarchy of the scene. Based on this fact, what you see in the Inspector properties screen when you choose any object is the set of components added to it.

What we're going to do now is build a prefab template, which is an object to which we add some components, and then we can create several copies of it while running the game. The good thing about these templates is that they can be modified from one place, no matter how many objects they are created from, as any modification to the original template (such as adding or deleting a component) will be reflected in all objects that follow this template. The template we're going to have is for the object that will display the background image, which will initially need one component - the Sprite Renderer for rendering 2D images on the screen. To create the template, we first add an empty object to the scene and then we add the mentioned component to it through the Add Component button, which gives you a list of all available components. For ease of access, you can search by writing the name of the component Sprite Renderer and then choosing it as follows:



After adding the component we are ready to convert this object to a template, by dragging it from the hierarchy into any folder in the project, in this case I'm going to create a new folder inside Assets and call it Prefabs because in this work we will need a large number of templates as we will see. Then change the template name to BGElement. After creating the template we no longer need the object in the scene so we can delete it using the Delete key. Be careful here that you delete the object from the hierarchy and do not delete the template from the project, they have the same name, but the difference is great between the two!

To add a new applet to the project, first let's create a folder for applets within Assets and call it Scripts. Then you can add a new applet in the #C language by clicking on the folder with the right mouse button and then choosing:

Create> C # Script.

Enter the name BackgroundManager and then press Enter and then open the file in the MonoDevelop or Visual Studio editor by double-clicking. Let's see the applet we will write in the following narration and then discuss it together in detail.

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class BackgroundManager: MonoBehaviour {
```

```
// Template used to build background elements
```

```
public GameObject bgElementPrefab;
```

```
// Matrix contains all available background images
```

```
public Sprite [] selectionList;
```

```
// The number of times the background image is repeated
```

```
public int repeatCount = 4;
```

```
// Background image currently displayed
```

```
private int selectedIndex = -1;
```

```
// "Background" is a reference to the named son object
private Transform bgGameObject;

// called once upon start up
void Start () {
    // "Background" search the children for the object with the
name
    bgGameObject = transform.FindChild ("Background");
    // Initially choose the first image as the background
    ChangeBackground (0);
}

// It is called once when each frame is rendered
void Update () {

}

// It changes the background image currently displayed
public void ChangeBackground (int newIndex) {
    if (newIndex == selectedIndex) {
        // No need to change the image
        return;
    }
}
```

// Delete the current background images

```
for (int i = 0; i < bgGameObject.childCount; i++) {  
    Transform bgSprite = bgGameObject.GetChild (i);  
    Destroy (bgSprite.gameObject);  
}
```

// Store the new background image from the array according to the given location

```
Sprite newSprite = selectionList [newIndex];
```

// What is the width of the background image in pixels?

```
float width = newSprite.rect.width;
```

// How high is the background image in pixels?

```
float height = newSprite.rect.height;
```

// How many pixels per unit?

```
float ppu = newSprite.pixelsPerUnit;
```

// Calculate length and width using units

```
width = width / ppu;
```

```
height = height / ppu;
```

// Calculate the horizontal position of the first image on the far left

```

float posX = -width * repeatCount * 0.5f;

// Calculate the new scene boundaries
Vector2 boundsSize = new Vector2 (width * repeatCount,
height);
Bounds newBounds = new Bounds (Vector2.zero,
boundsSize);

// Send a message telling the boundaries of the scene to
change
BroadcastMessage ("SceneBoundsChanged", newBounds);

// We can now start building the new background
for (int i = 0; i <repeatCount; i ++) {
    // Create a new object using the template
    GameObject bg = (GameObject) Instantiate
(bgElementPrefab);
    // Specify the name of the object
    bg.name = "BG_" + i;
    // Fetch the image rendering component in the object
    SpriteRenderer sr = bg.GetComponent <SpriteRenderer>
();
    // Specify the image to render as the image selected from
the array
    sr.sprite = newSprite;
    // Adjust the horizontal position using the previously

```


calculated value

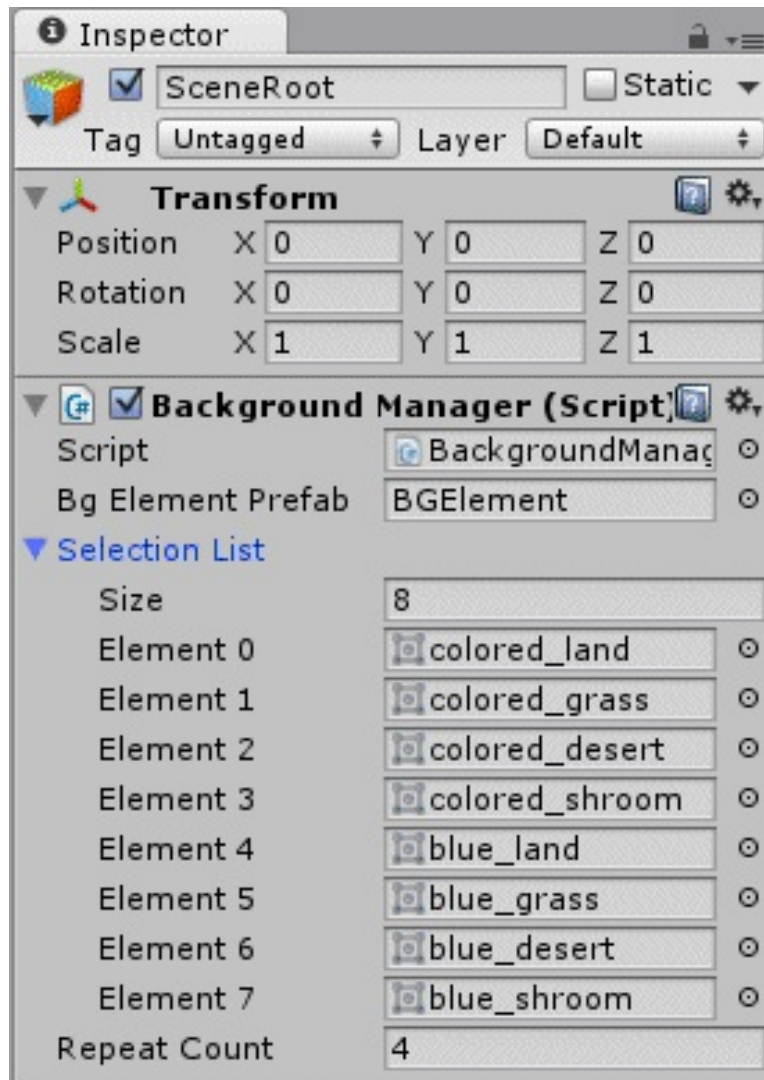
```
    bg.transform.position = new Vector2 (posX, 0);  
        // Return the background image a step back in the  
drawing arrangement  
        sr.sortingOrder = -1;  
        // Cap for the created background element "Background"  
Select the empty object  
        bg.transform.parent = bgGameObject;  
        // Add a square shaped collision component  
        bg.AddComponent <BoxCollider2D> ();  
        // Add the image width to the horizontal location value for  
calculating the new location for the next item  
        posX += width;  
    }  
  
    // Send a message informing you that the background image  
has changed and attach the location of the new photo  
    BroadcastMessage ("BackgroundChanged", newIndex);  
}  
}
```

Unity treats applets as components that can be added to any object in a scene or template in a project (templates at the end are objects). For us, this is an essential applet related to scene management as a whole, not to the player or opponent. Therefore, we will add it to the root object by simply dragging the BackgroundManager.cs file from the project browser to the SceneRoot object, either in the hierarchy or in

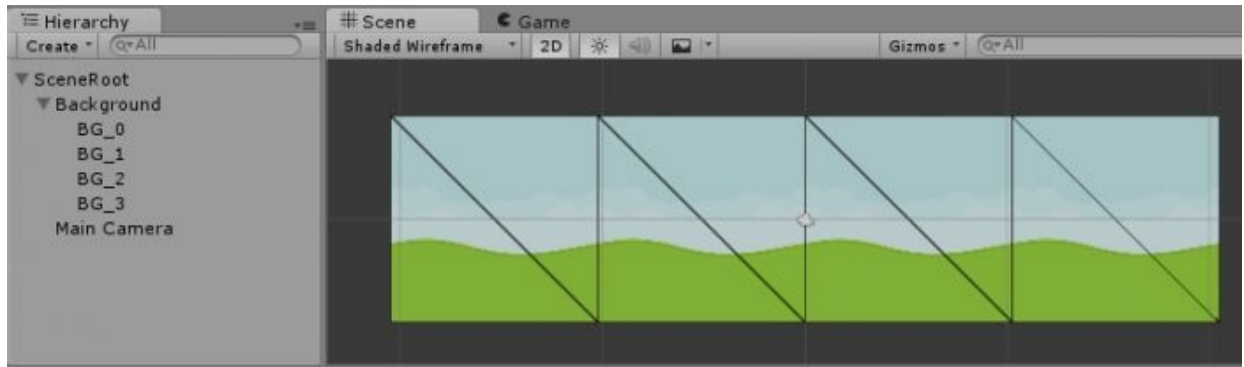
its properties screen.

After adding it, you will notice that the applet appeared in the root object properties screen and the general variables `bgElementPrefab`, `selectionList`, and `repeatCount` appeared so that the first appears as an empty box and the second appears as a list. You can specify the number of its elements from the size field, and the third in the form of a text box that accepts a numerical value.

Now let's see how we can define the values of these variables and it will start with `bgElementPrefab`. As expected, this field is intended to define the template that will later be used to create the background elements, which is the `BGElement` we have already created. To link the template to this variable, all you have to do is drag it from the project browser into the field you see in the properties screen. Then change the number of items in the `selectionList` to 8 and then drag each cell from one of the background images in the `Backgrounds` folder. The third variable, `repeatCount`, has a default value of 4 which is the value we used when defining it. The shape of the root object will now become:



If you try to run the game now, you will notice that the background image in the first item appears in the list, and you will notice that it repeated horizontally 4 times correctly as you see below, and you also notice the four objects of these images that were added as children of the background object in the hierarchy.



We will now quickly discuss the most important background mechanisms for the BackgroundManager applet. All programs in Unity take almost the same format, starting with defining the necessary variables, and then the two most frequently used functions are `() Start` and `() Update`. As shown in the comments on the code, the `Start ()` function is called only once at start-up if the object was already in the scene, or once it was added to the scene if it was not there before. Then the game enters into what we call the update loop or the frame rendering loop where the `Update ()` function is called when each frame is rendered as long as the game is running. In this case, we do not need to use `(Update)`.

Note that we started the applet by defining a number of variables, some of which are public and others private. For Unity, the general variables are of particular importance because it displays them as cells in the properties screen, allowing you to change their values without modifying the code as we have seen before. The following is an explanation of these variables:

`bgElementPrefab` is used as a reference to access the template that we will use to create background elements when they are built.

`selectionList` An array containing Sprite objects that stores a list of

all backgrounds that can be used.

repeatCount An integer that represents the number of times the background is repeated horizontally. Naturally, increasing this number will increase the width of the final background.

selectedIndex represents the background location currently displayed in the list, where the value 0 represents the first item.

bgGameObject We will use this variable to access the background object which we have set as the parent of all background images in the scene. It is worth noting here that in most cases we use a Transform variable to access objects, so there are many benefits we will know.

At startup, the `()` function is called, which first searches the children for the object named Background and stores it in the `bgGameObject` variable. The second step that this function performs is to choose the first item in the list automatically to be the background of the scene by calling `ChangeBackground ()` with the value 0.

The `ChangeBackground ()` function is the most important in this program, as it changes the background according to the value given to it when calling `newIndex`. If this value was originally equal to the value of the currently displayed `selectedIndex` value, there is no need to change anything, thus executing the function. Other than that, the function takes precautions for the presence of a currently displayed background and deletes it above all. The deletion is by passing all the `bgGameObject` sons and calling the `Destroy ()` function to delete them from the scene. The object deletion state is one of the exceptions to the rule on dealing with objects via the Transform component, where deletion must take place at the entire object level and not on a specific component, thus we call `bgSprite.gameObject` when deletion.

After deleting the previous background (if any) we need to add the new one. So the next step will be to store the value of the image in the newIndex element in the newSprite variable and then calculate the width of the original image in Unity space by dividing its width in pixels by the number of pixels in each unit. Since the frequency of the background image will revolve around the origin point, the farthest point to the left will be half the width of all the images that we will repeat, thus we calculate the posX value by multiplying half the total number of images repeatCount by the width of the single image width, and we convert the result to the negative value until we start from the left (Remember, we selected the image to the left of its image as its fulcrum in space).

In addition to the starting point, we calculated the length and overall width of the scene after building the new background, and then we stored these values in a variable Bounds type, this variable simply stores a point in space and extensions for this point on the x and y axes. Note that we define the span variable as a two-dimensional vector and define the value of x as the number of times the background image is multiplied by the width of one image and the value of y as the height of one image. After that we sent a message to all other programs, whether on the root object or any object in the children, to inform it that the boundaries of the scene have changed, and we used that function (BroadcastMessage). When calling this function, we give it the name of the message, which is SceneBoundsChanged, and we also attach the newBounds variable that contains the new scene boundaries. We will see later what it means to send a message, how to receive it, and why it is important. Throughout this set of lessons we will deal extensively with messages, so it is okay to know early how to send them. Messages are sent in Unity using one of the following functions:

- `BroadcastMessage` which sends a message to all the programs on the current object as well as to all the children and descendants of its descendants.
- `SendMessage` which sends the message only to all programs on the current object.
- `SendMessageUpwards` which sends the message to all applets of the current object as well as to all of its parents and grandparents in the hierarchy.

The iterative cycle following these steps is the one in which the background images in space are correctly constructed and distributed. We start with creating a new object `bg` using the `bgElementPrefab` template and then give it a sequential name according to its location in the background set from left to right (for example `BG_0` will be the background name on the far left and `BG_3` on the far right). Then we call the component `Sprite Renderer` and specify the image to be rendered, and here is the `newSprite` that we previously extracted from the `selectionList`. Again we dealt with the object directly and not with the `Transform` component, because the goal here is to add a new component to it.

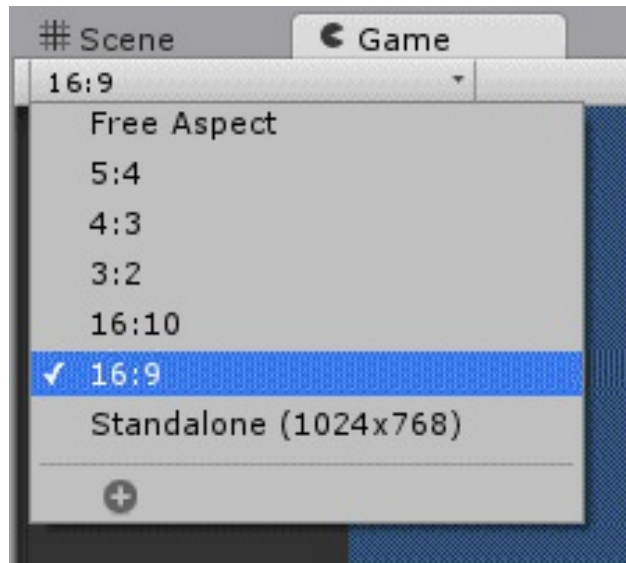
The next two steps deal with `bg.transform` as you see, the first defines its location in space `gb.transform.position` and the second determines its parent being and here is `bgGameObject` that we have already talked about. For the site, the background is centered vertically as you see where `y = 0`, and horizontally, the location is the `posX` value that we previously calculated. It is important here to realize the fact that the background images must appear behind the rest of the images in the game space. In order to do that, we have to tell the rendering component to draw the background before painting any other image so that the background does not cover the elements of the scene. This

we can do by reducing the value of the `sr.sortingOrder` variable where the default value is zero. Images with a lower value for `sortingOrder` are rendered first, followed by images with a higher or higher value.

Next, we add the Box Collider 2D collision detection component so that we can detect the player touching or clicking the background with the mouse. The last step is to increase the `posX` by the width of the image so that the next image is located to the right of the current, adjacent to its right side. After building the background we send another message which is `BackgroundChanged` and attach the new background element number, which we will see later its importance.

Building a camera control system

If you run the game, you will notice that the camera is looking directly into the center of the scene, and what we will do now is to add a mechanism to move it left and right, in addition to the possibility of rounding and distance. Before we start with the camera control details, we have to determine what the aspect ratio is. This value is important because it directly affects the size of the rectangle that defines the field of view of the camera. Since this game is designed for phones and plays incidentally, the ratio that we will adopt is 16: 9. To choose it, go from the Scene screen to the Game screen and choose this percentage from the top left of the screen as you see here:



Before starting writing the camera control code, we need to know the nature of this control and what the player can do with it. The control method will be two-dimensional, meaning that the player will be able to move the camera left and right, up and down. In addition, the player must be able to round and move the camera, all within certain limits. We will look at the camera control app shortly, and then we will explain some of its details. Finally, we will see how we can link the functions of this camera with the player's input.

But before that, let's learn one of the important principles in programming games, especially with the Unity engine, which is separation of concerns. This means that a single code is responsible for one task, and it is the only one authorized to control the details of this task. Thus, we will first write the code to control the camera, but it is completely separate from the player's input. These entries will be written in another code of their own, which will receive them and accordingly "asks" the camera control code to move them. This request may or may not be answered according to data that the camera can control. For example, if the player moves the camera to the far right of the scene and then tries to move it right beyond the scene, then the camera code will reject this request and the camera will

remain in place.

Let's start now with the CameraControl code that you see in the following narration, this code must be added to the camera object in order to control it:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class CameraControl: MonoBehaviour {
```

```
// Minimum size of the camera in the event of rounding
```

```
public float minSize = 2.5f;
```

```
// The height of the specified rectangle for the field of view
```

```
private float camHeight;
```

```
// Display the specified rectangle for the field of view
```

```
private float camWidth;
```

```
// A reference to the camera component
```

```
Camera cam;
```

```
// variable to store the boundaries of the current scene
```

```
private Bounds sceneBounds;
```

// It is executed once at start up

void Start () {

// Fetch the camera component on the same object

cam = GetComponent <Camera> ();

// Update the camera dimensions

UpdateCamDimensions ();

}

// It is called once when each frame is rendered

void Update () {

}

// Receive a message informing you that the outline of the scene has changed

void SceneBoundsChanged (Bounds newBounds) {

sceneBounds = newBounds;

}

// This function moves the camera by a specified amount

public void Move (Vector2 distance) {

// Calculate the permissible traffic limits

float maxX = GetRightLimit ();

```
float minX = GetLeftLimit ();  
float maxY = GetUpperLimit ();  
float minY = GetLowerLimit ();
```

```
// In this particular case we are interested in the 3D site  
// z = -10 in order to ensure that the camera is on site  
//Always  
Vector3 dist3D = distance;
```

```
// Calculate the new site  
Vector3 newPos = transform.position dist3D;
```

```
// Stick to the movement's boundaries  
newPos.x = Mathf.Clamp (newPos.x, minX, maxX);  
newPos.y = Mathf.Clamp (newPos.y, minY, maxY);  
newPos.z = -10.0f;
```

```
// Change the location  
transform.position = newPos;
```

```
}
```

```
// This function changes the camera's proximity  
// Positive value means bringing the camera closer to the scene  
// Negative value means keeping the camera away from the  
scene
```

```

public void Zoom (float amount) {
    float newSize = cam.orthographicSize - amount;

    // Stick to the limits of rounding and banishment
    float maxSize = GetMaxSize ();
    newSize = Mathf.Clamp (newSize, minSize, maxSize);

    // Change the size of the field of view and thus the
approximation changes
    cam.orthographicSize = newSize;

    // Move the camera zero to ensure compliance with the scene
limits
    UpdateCamDimensions ();
    Move (Vector2.zero);
}

// Updates the width and height of the camera
void UpdateCamDimensions () {
    camHeight = cam.orthographicSize * 2;
    camWidth = camHeight * cam.aspect;
}

// The higher height of the camera is calculated
float GetUpperLimit () {

```

```
    return sceneBounds.max.y - camHeight * 0.5f;  
}
```

// The minimum height of the camera is calculated

```
float GetLowerLimit () {  
    return sceneBounds.min.y camHeight * 0.5f;  
}
```

// Calculates the maximum position of the camera to the right

```
float GetRightLimit () {  
    return sceneBounds.max.x - camWidth * 0.5f;  
}
```

// Calculates the maximum left of the camera

```
float GetLeftLimit () {  
    return sceneBounds.min.x camWidth * 0.5f;  
}
```

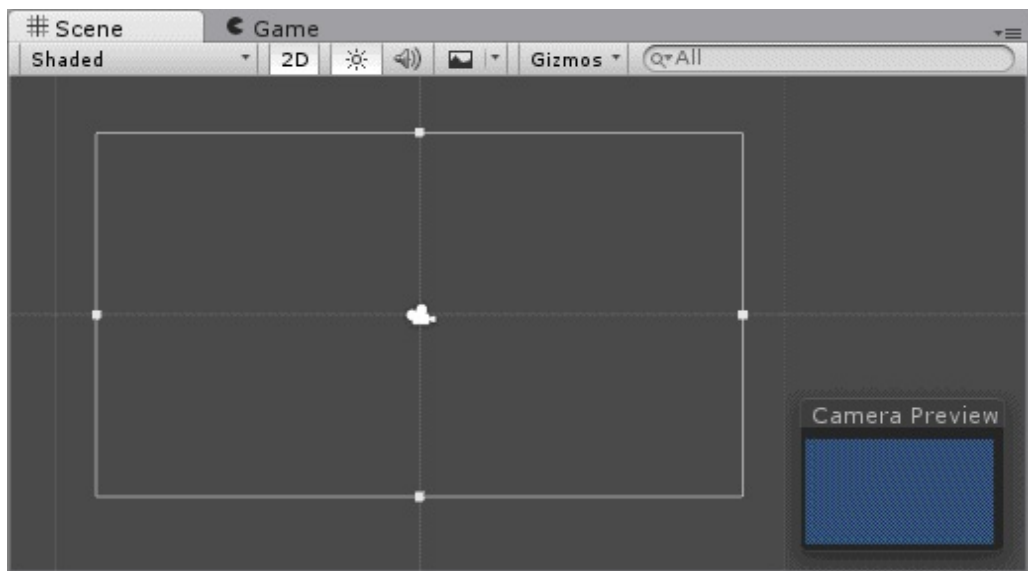
**// It calculates the largest size allowed by the camera at the time
of deportation**

// Depending on the scene's boundaries

```
float GetMaxSize () {  
    float maxHeight = sceneBounds.max.y - sceneBounds.min.y;  
    return maxHeight * 0.5f;
```

```
}  
}
```

The only general variable in this code is `minSize`, which specifies the minimum camera size when zooming. To explain the mechanism of approximation and distance, first note that the camera does not see the entire scene at all time, but it has a specific field of view in a rectangle. This rectangle appears when selecting the camera, as in the following figure:



The greater the size of this rectangle, the more space he sees from the scene, so it looks as if the camera is moving away, but in the case of reducing its size, the opposite happens, which is reducing the visible area so that it appears as if the camera is approaching. You can try this while running the game by adjusting the value of the variable in the camera component from the properties screen, where its default value is 5.

This means that we allow the camera to be rounded, i.e. reduce its size

until it reaches 2.5 and you can allow the camera to be zoomed closer if you reduce this number. The other two important variables are `camHeight` and `camWidth` and represent the length and width of the rectangle that defines the field of view, but they are calculated with Unity space values and we will see their importance in a little while. Also we need a reference for the camera component, which is the `cam`. Remember what I mentioned earlier, that it is the components that distinguish one object from another, and the existence of this component is what makes the object called Main Camera a camera and not something else. Finally we have a `Bounds` variable, which is `sceneBounds`, which will store the boundaries of the scene, which we will restrict the movement of the camera entered.

This time I will start with the `SceneBoundsChanged ()` function to complete the explanation of the idea of the messages. Going back to the `BackgroundManager` code you will see that it is sending a message named `SceneBoundsChanged` and attaching it to a `Bounds` variable. If you define a function with the same name as the message and take a variable of the same type as the attachment, then you have defined this function as one of the recipients of this message (the message can be received by an unlimited number of programs). Thus, the `CameraControl` code receives this message and stores the value of the new scene boundaries in the `sceneBounds` variable.

Returning to `Start ()`, what you are doing is storing the value of the camera component in the `cam` variable and then calling `UpdateCamDimensions ()`, which calculates the width and height of the camera in Unity spaces. Let's talk a little bit about how to calculate these values. Referring to Unity documentation, specifically this page, we will know that the value of the `cam.orthographicSize` variable, which is the size of the camera, is mid-height. So we can

multiply this value by 2 to get the camera height. As for its width it depends on the aspect ratio, which we can access via the `cam.aspect` variable, where we can multiply this variable in height to get the width of the camera.

By obtaining the boundaries of the scene and the height and width of the camera, we can calculate the limits within which the camera is allowed to move. The camera will be allowed to move right until the rectangular end of the right field of vision has reached the maximum right of the scene. Consequently, the rightmost camera location is the edge of the right view minus half of the camera's width. This value is calculated by the `GetRightLimit ()` function, and in a similar way we calculate other boundaries on the left, up, and down through `GetLeftLimit ()`, `GetUpperLimit ()`, and `GetLowerLimit ()`. Note that in these functions we use `sceneBounds.min` and `sceneBounds.max`, the two vectors that express the minimum (bottom and left) and the maximum (top and right) of the scene boundaries.

Before entering the main functions of the code, moving and rounding, let's get acquainted with the remaining function within the auxiliary functions, which is `GetMaxSize ()`. This function calculates the maximum permissible size of the camera when carrying out the dimensions, depending on the upper and lower limits of the scene in addition to the height of the camera. The maximum permissible height is the distance between the top and bottom scenery boundaries, and the permissible size is therefore half that distance, because the size of the camera is half the height of it as we have known.

Now we come to the main functions of this code, which is to move, zoom and move the camera. Beginning with the `move ()` function which takes a direction represents the amount of displacement we

want to move the camera with. The first step is to calculate the maximum and minimum permissible traffic values, and store them in the four variables `maxX`, `minX`, `maxY`, `minY`. Then we define a 3D vector, `dist3D`, because the location of the camera on the third axis of `z` is important unlike other objects, as this value must remain negative, that is, the camera is far from the scene outside and looks at it inside the screen. If we calculated its location using only the two-dimensional dimensions, this would result in the loss of the value of `z` for the location, and the camera would become in the location `z = 0`, and thus would not be able to see any object in the scene.

After calculating the new location, we store it in the `newPosition`, and all we have to do now is make sure that the value of `newPos.x` is between `minX` and `maxX`, as well as the value of `newPos.y` that must be between `minY` and `maxY`. Note here that you use the `Mathf.Clamp()` function, which verifies that the first number is between the second and third numbers and returns the maximum or minimum if the number exceeds these two limits. Finally we are changing the camera location for the newPOS site.

The second main function is to zoom in and out of the camera, which is done via the `Zoom` function, this function takes one number, which is the amount of rounding (if positive) or exclusion (if negative). The method of approximation and distance is accomplished by subtracting the amount from the current camera size, because the relationship between the size of the camera and its approximation is inverse, so increasing the size means reducing the approximation. Then the new value is confined between `minSize`, which is the variable that we can define as we like and `maxSize`, which is calculated by the `GetMaxSize()` function previously explained. After ensuring that the new size is within the limits, the `cam.orthogonalSize` value is changed to the new

new size. Finally we must recalculate the dimensions of the camera as its size has changed so we call `UpdateCamDimensions ()` in addition to its displacement by zero in order to preclude the possibility that the change in its size has led to exceeding its limits to the scene boundaries especially in the case of deportation where the size of the camera increases; The camera is always within the boundaries of the scene.

Thus, the camera control mechanism is ready, and it remains for us to link the call `(Move)` and `() Zoom` to the player inputs. The start will be with the mouse, because the engine runs on the computer, therefore it is easy to try it directly, while we will postpone the touch screen and smart phones for a later time. Moving the camera will be done by pressing the left mouse button on the background and then moving it in the four directions, while the right button is used for rounding and distance, where the player presses the right button and moves the mouse right to move and left to move away. These functions will be performed by the `CameraMouseInput` code that we must add to the `BGElement` background template, where all background images must be able to receive mouse inputs. Below is a listing with this code.

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class CameraMouseInput: MonoBehaviour {
```

```
// The movement speed of the camera
```

```
public float movementSpeed = 0.1f;
```

// Zoom speed and distance

public float zoomingSpeed = 0.01f;

// During rounding and distance true this value is variable

private bool zoomingActive = false;

// It stores the location of the mouse pointer in the previous window

private Vector2 lastPosition;

// Reference for camera control applet

private CameraControl camControl;

// This function is executed once upon startup

void Start () {

// Bring the camera control applet

camControl = FindObjectOfType <CameraControl> ();

}

// It is called when every frame is rendered, but late

void LateUpdate () {

UpdateZooming ();

}

// It reads the approximation and exclusion entries

void UpdateZooming () {

// Check the right mouse button

if (Input.GetMouseButtonDown (1)) {

zoomingActive = true;

lastPosition = Input.mousePosition;

}

// Update the zoom and distance based on the horizontal movement of the mouse

if (zoomingActive) {

float amount = Input.mousePosition.x - lastPosition.x;

camControl.Zoom (amount * zoomingSpeed);

lastPosition = Input.mousePosition;

}

// When decompressing the right mouse button false to zoomingActive, return a value

if (Input.GetMouseButtonUp (1)) {

zoomingActive = false;

}

}

// This function is called when the right mouse button is clicked on the object

```
void OnMouseDown () {
```

```
// Do not allow movement during rounding and deportation
```

```
if (! zoomingAcitve) {
```

```
    lastPosition = Input.mousePosition;
```

```
}
```

```
}
```

```
// This function is called when the left mouse button is over the  
object
```

```
void OnMouseDown () {
```

```
// Prevent movement during rounding and deportation
```

```
if (! zoomingAcitve) {
```

```
    Vector2 movement = lastPosition - (Vector2)
```

```
Input.mousePosition;
```

```
    camControl.Move (movement * movementSpeed);
```

```
    lastPosition = Input.mousePosition;
```

```
}
```

```
}
```

```
}
```

As I mentioned earlier, this applet must be added to the background image template, because the player will click on it with the mouse if he wants to move the camera. When the applet starts running, it searches in the scene for the applet that controls the camera, via Camera (FindObjectOfType), and since this applet is present once in the scene, then the search result will definitely be the applet on the camera. So now we can use the camController variable to control the camera.

When the player clicks the left mouse button on an object, the OnMouseDown message is sent once to that object, so here we receive this message via the function of the same name where we store the current mouse's location in the lastPosition variable. If the player presses the left button and moves the mouse, then the OnMouseDown message is sent to the object in every frame in which the cursor is moved while pressing. In the OnMouseDown () function, we do three things:

First we calculate the displacement amount from the pressure site by subtracting the current site from the previous site. In this way we make the movement of the camera opposite to the movement of the cursor, if the cursor moves to the right the camera moves left, which gives the player a feeling that he does not move the camera, but rather grabs the scene and moves it right and left and this method is easier for him to control, especially with the touch screen.

The second step is to call the Move () function and provide it with the calculated displacement multiplied by the speed of movement. Note that the movement speed is relatively low, because the value of mouse displacement is high compared to the amount of displacement required for the camera.

Finally, after the offset is executed, we store the current cursor

position in the `lastPosition` so that we are ready to calculate the next offset from the new position.

Note that all steps in the `OnMouseDown ()` and `OnMouseDown ()` functions are conditional on the `zoomingActive` value being false. This variable tells us whether the player is currently zooming or moving the camera using the right button, thus preventing movement and rounding at the same time. The `lateUpdate ()` function that we defined here instead of (`Update`) differs from the latter in one thing, and it is called late for it. When updating each window, Unity updates all programs by calling `Update ()` and then refreshes again using `LateUpdate ()`. Therefore, calling any steps during (`LateUpdate`) is guaranteed to be executed after updating all scene elements through (`Update`). The benefit of this is to ensure that all elements are ready for the new frame before we move the camera, so `LateUpdate ()` is usually used with anything related to camera control.

What we do in `LateUpdate ()` is to call `UpdateZooming ()`, which checks if the player presses the right mouse button via the `Input.GetMouseButtonDown ()` function, which checks the mouse buttons based on their numbers: the number 0 is for the left, 1 for the right and 2 for the middle. The first thing that we do when the right button is detected is to change the `zoomingActive` value to true which prevents the reception of inputs from moving. Then we store the cursor's current location in `lastPosition` just as we did with left-click state. Based on the `zoomingActive` value, we update the approximation and distance, so we calculate the horizontal offset by subtracting the coordinate x for the previous location from the coordinate x for the current location, so if the mouse pointer moves to the right, the result will be positive, which makes the value of `amount * zoomingSpeed` supplied to the function (`camControl.Zoom`

positive and results in the camera zooming , And vice versa when you move the mouse left. After updating the approximation value, we store the current location in `lastPosition` to calculate the next displacement. The last step in this function is to check if the player has removed the right mouse button and reset the `zoomingSpeed` value to false in this case.

Building a scene landscape

After we've finished preparing the scene background and we're able to move the camera to roam around freely, it's time to add the scene elements, which are the two-dimensional objects from which we will build the game. It will start with the floor on which the rest of the objects stand, and which we will build in a manner similar to the way the background is built. First we'll add an empty object as the name of the root object and call it `Ground`. This being will be the father of all terrestrial beings. Then we will create a template for the floor objects and add the `sprite renderer` and the `CameraMouseInput` component, which makes controlling the camera by pressing the floor possible just as with the background. We will call this template `GroundElement` and then we will write an applet that adds ground objects depending on the background and the boundaries of the scene. This code is `GroundManager` and shown in the following narration, note that this Code must also be added to the `SceneRoot` root object.

```
using UnityEngine;
using System.Collections;

public class GroundManager: MonoBehaviour {

    // Flooring Object Object Template
    public GameObject gePrefab;
```

// Matrix with available floor pictures

public Sprite [] selectionList;

// variable to store the boundaries of the current scene

private Bounds sceneBounds;

// Site background image currently displayed

private float selectedIndex = -1;

// called once upon start up

void Start () {

}

// It is called once when each frame is rendered

void Update () {

}

// It receives a message that changes the background image

void BackgroundChanged (int newIndex) {

ChangeGround (newIndex);

}

// It receives a message that changes the boundaries of the scene

void SceneBoundsChanged (Bounds newBounds) {

sceneBounds = newBounds;

}

// Changes the image of the displayed floor based on the number of the selected object

public void ChangeGround (int newIndex) {

// "Ground" Look for the named son being

Transform groundGameObject = transform.FindChild ("Ground");

// Delete the existing floor elements

```

for (int i = 0; i < groundGameObject.childCount; i++) {
    Transform ge = groundGameObject.GetChild (i);
    Destroy (ge.gameObject);
}

// Calculate the scene width
float sceneWidth = sceneBounds.max.x - sceneBounds.min.x;

// Calculate the point on the far left
float posX = sceneBounds.min.x;

// Grab the floor image from the array
Sprite newGround = selectionList [newIndex];

// Calculate the width and height of the floor image
float width = newGround.rect.width;
float height = newGround.rect.height;

// Calculate the width and height in units
width = width / newGround.pixelsPerUnit;
height = height / newGround.pixelsPerUnit;

// Calculate the vertical position of the floor
float posY = sceneBounds.min.y + height * 0.5f;

// How often should we repeat the floor image?
int repeats = Mathf.RoundToInt (sceneWidth / width) + 1;

for (int i = 0; i < repeats; i++) {
    // Add a new terrestrial object
    GameObject ge = (GameObject) Instantiate (gePrefab);
    ge.name = "GE_" + i;

    // Place the object in the correct position

```

```

        ge.transform.position = new Vector2 (posX, posY);

        // Select father
        ge.transform.parent = groundGameObject;

        // Place the image on the render object we added
        SpriteRenderer sr = ge.GetComponent <SpriteRenderer> ();
        sr.sprite = newGround;

        // Add a collision component
        ge.AddComponent <BoxCollider2D> ();

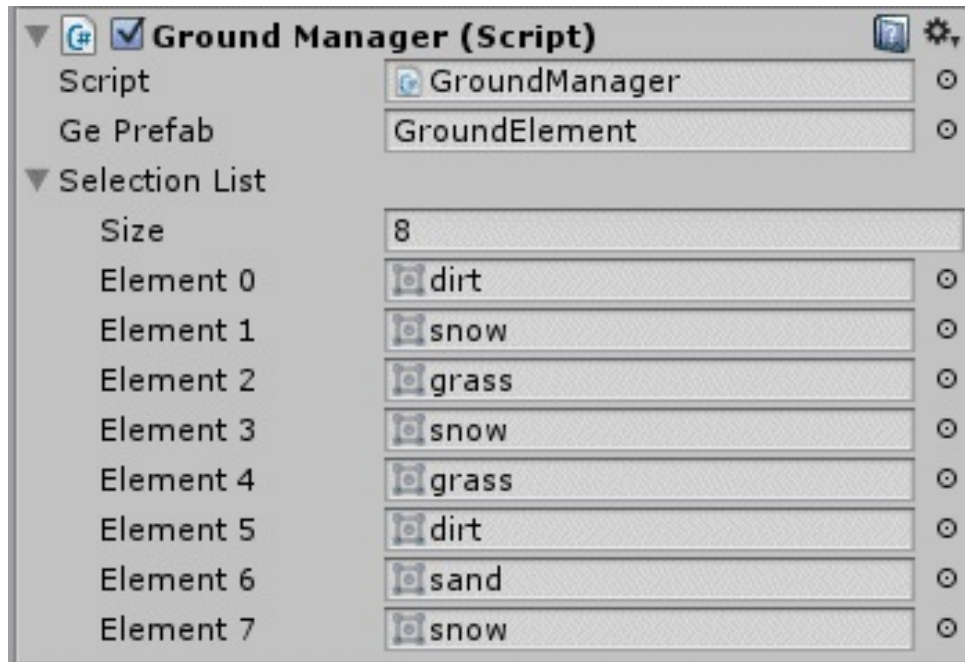
        // Increase the value of the horizontal position in preparation for the next item
        posX += width;
    }
}
}

```

What we notice at first glance is the very similarity between GroundManager and BackgroundManager in that each of them builds several objects using a specific template and then locks these objects side to side from left to right. The difference here is that this applet relies on its work on receiving the BackgroundChanged message, so that it chooses the image of the floor in the location equal to the background location chosen. This means that for the existing eight background images, we will need eight flooring images. The floor pictures are in the Other folder in the group of images and their number is 4 as you see in the following image, which means that the same floor can be repeated with more than one background:

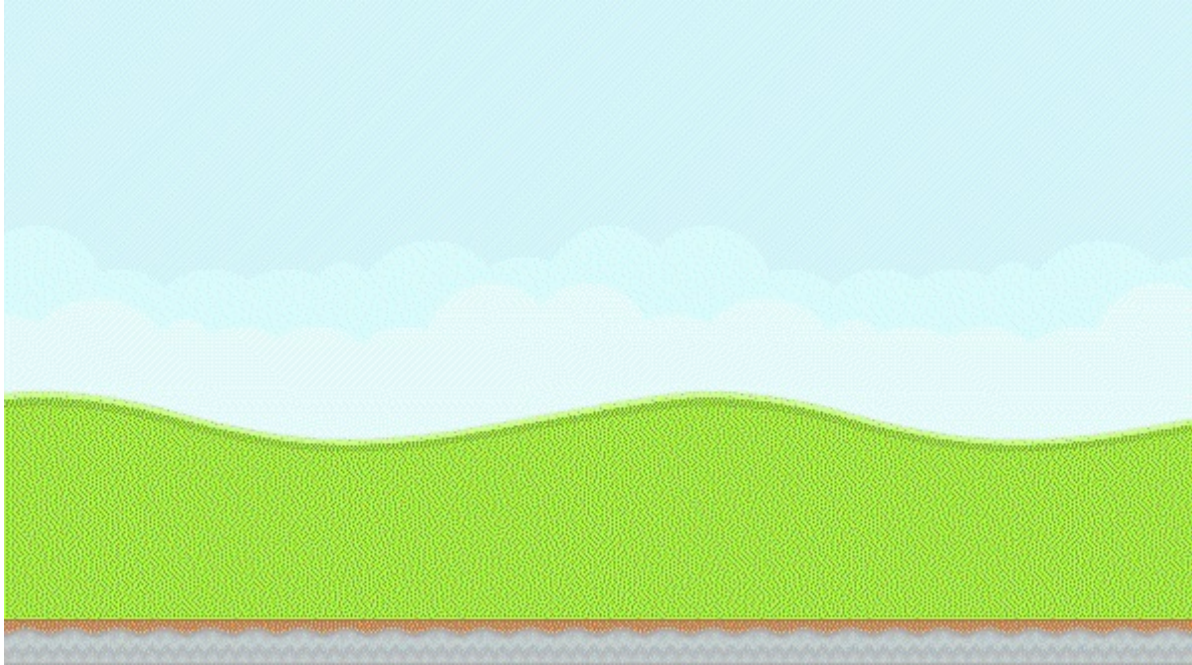


The first step we need to take is to change the anchor point of these four images to the left just as we did with the background images. And then we're going to drag it to the eight locations inside the `selectionList` array. While adding the background image corresponding to each background, I tried to choose a color that is distinct from the color of the lower part of the background, so that it is easy for the player to distinguish the floor of the scene from his background, and the result was as follows. Also, after adding the code to the `SceneRoot` object, don't forget to set the `GroundElement` template to be the source of building the ground objects.



Another difference between the two codes is observed in the location of the objects, where the background objects are placed vertically in the middle while the ground objects are placed at the bottom. Another important difference is that the number of repetitions in relation to the background is a variable whose value we can determine, while in the case of the floor it is calculated automatically by dividing the width of the scene by the width of the floor image, and then adding 1 to cover any remaining gap at the far right as a result of the fractures.

Up to this point we have finished building the background and floor, so if we try to run the game now we will get the following result:

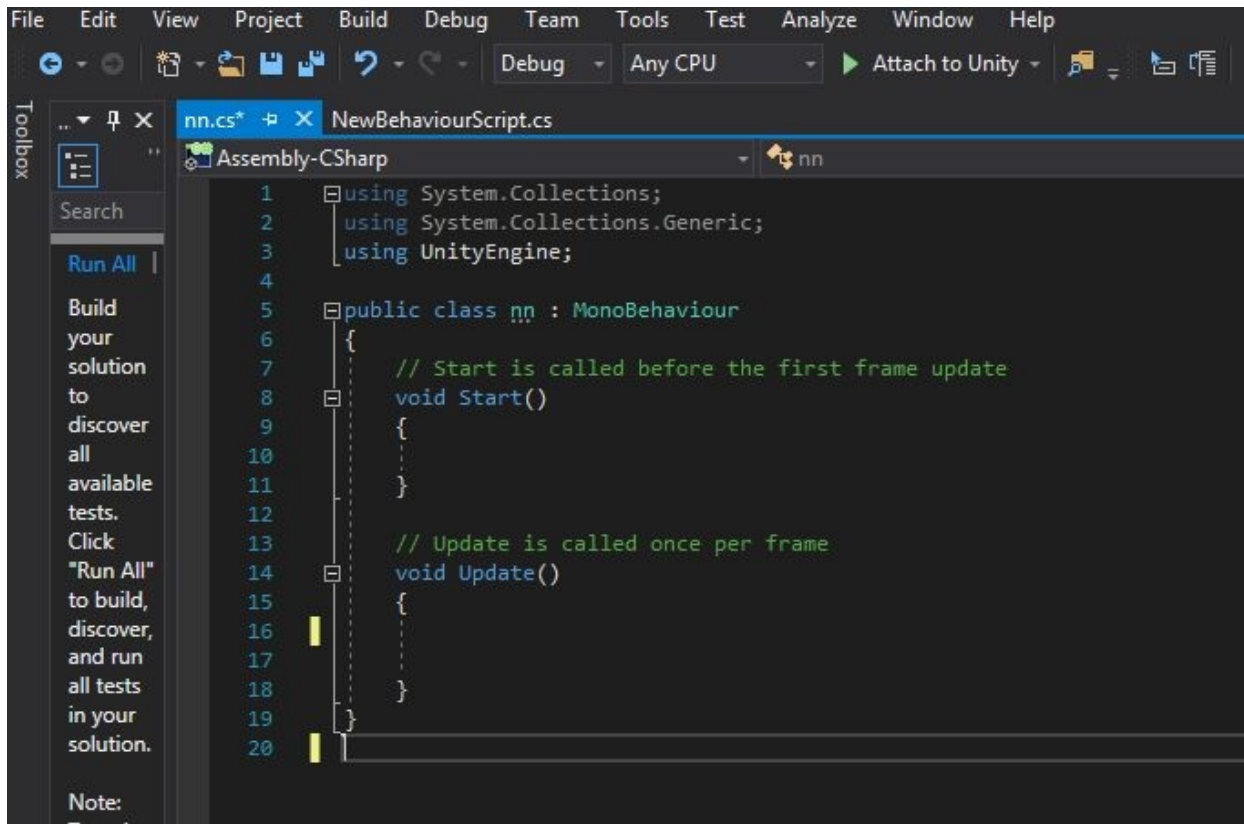


In this lesson, we prepared the background and floor of the game and also controlled the camera. In the next chapter we will create the game building blocks.

Chapter 2

Motion in programming

Open the game engine and create a new script by placing the mouse pointer in the assets location and right-click where you will see several options, choose create, then select c #, script and open it. The code will appear as follows:

A screenshot of the Unity IDE interface. The top menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, and Help. Below the menu is a toolbar with icons for running, saving, and other actions. The left sidebar shows the 'Toolbox' with a search bar and a list of actions like 'Run All', 'Build your solution', etc. The main editor window displays a new C# script named 'nn.cs' with the following code:

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class nn : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10         ...
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16         ...
17     }
18 }
19
20
```

These two arguments are hypothetical. The engine assumes two sets of commands and instructions:

- Void Start

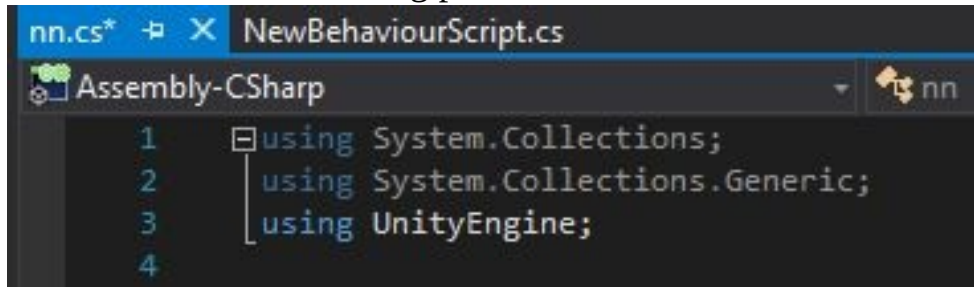
Instructions and commands that are executed when the user starts the program automatically.

- Void UpDate

This function means that the commands and instructions inside are automatically spoken in each frame and can be said to be executed loopwise and continuous running time of the program.

In general, when we want to write specific instructions, we need to call libraries for these instructions. In this context, there are public libraries and private libraries. When we want to move a particular object in a certain way, we use the public library `UnityEngine`.

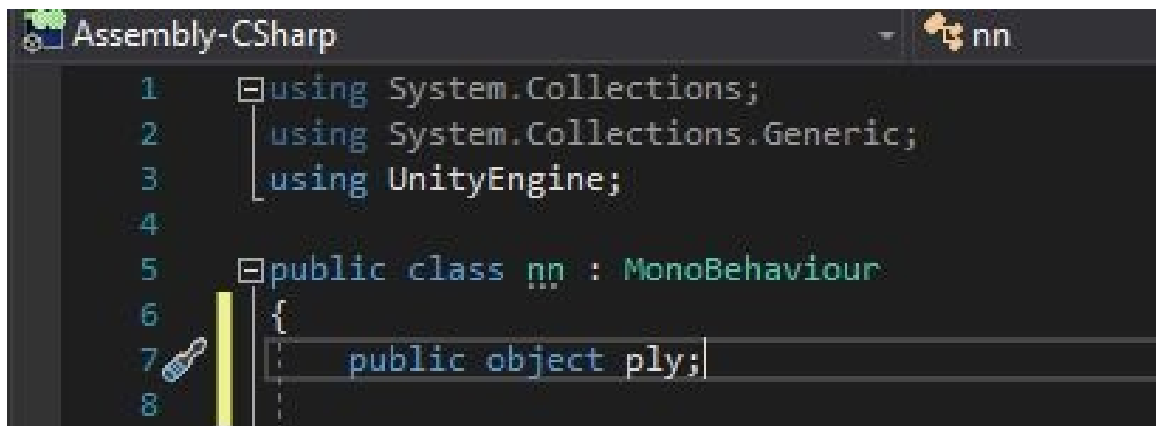
As follows in the following picture:



```
nn.cs* X NewBehaviourScript.cs
Assembly-CSharp
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
```

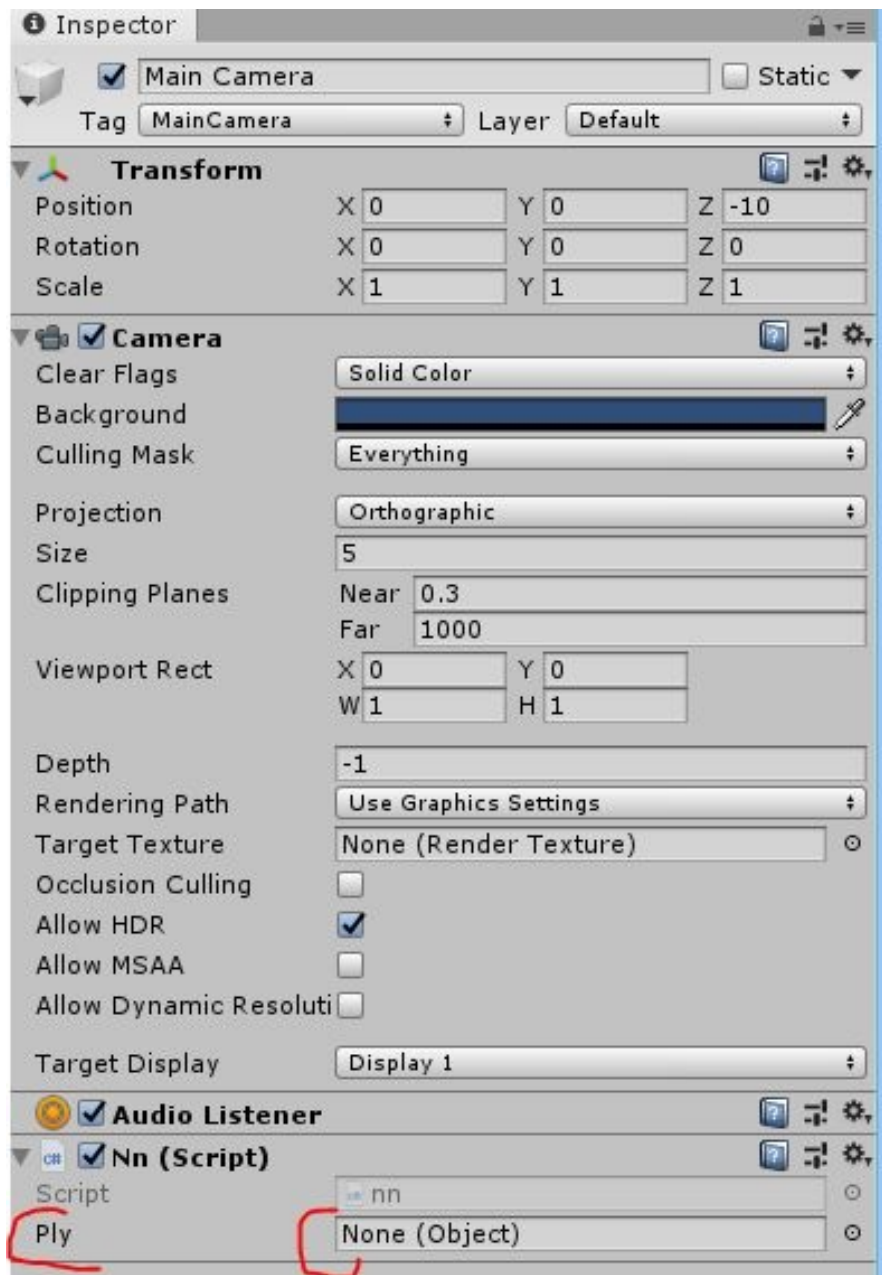
Programming, as the famous programmer Hussein Al-Rubaie says, depends primarily on your thinking and thinking. Think about how we can define the program. We want to define a specific object in the program to make it move in a certain way.

We will certainly insert a variable of type `Object` in the editor, to try to do this :



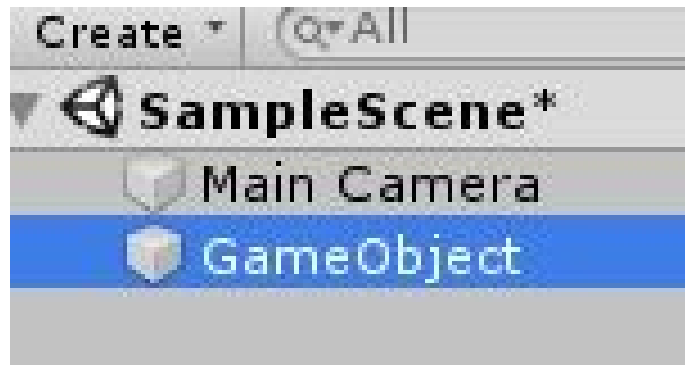
```
Assembly-CSharp nn
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class nn : MonoBehaviour
6  {
7      public object ply;
8
```

:



- Urge be variable space in the main camera

Now we have to create a new object in the program and drag it into the space of the variable in order to identify it and to put the code inside it through clouds and

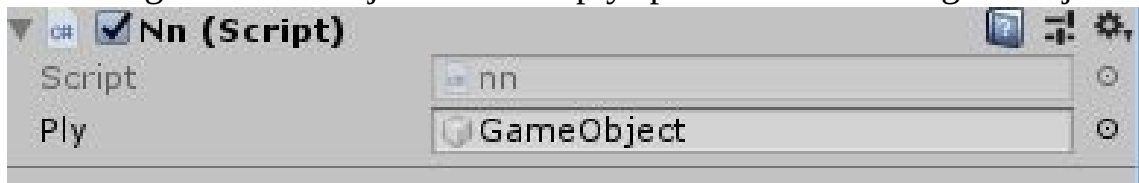


vacations ..

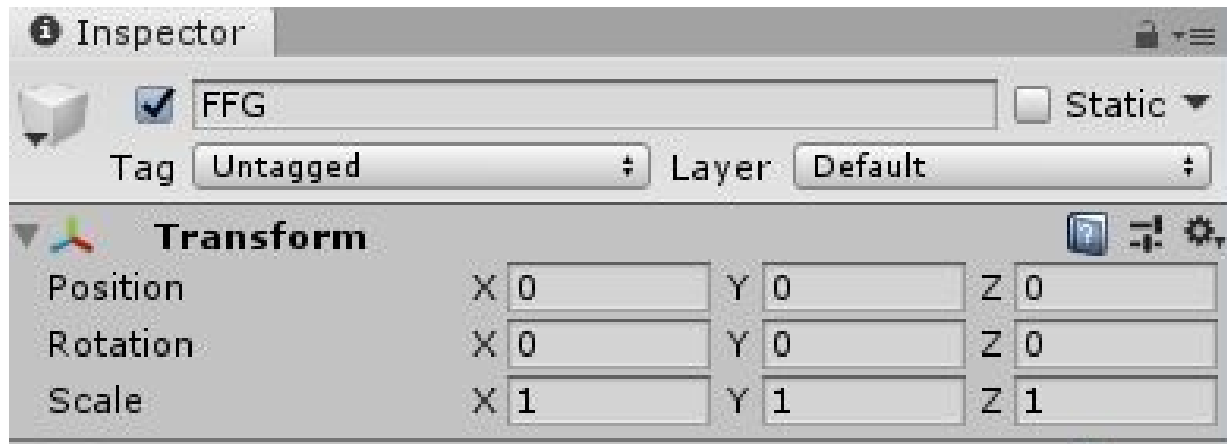
Insert any image you like in the editor and link it to "GameObject" as you follow :



Then drag the GameObject to the empty space "none" in the gameobject :



Now we will go back to Visual Studio after we have defined any variable we want to specify in the program, and before I start I want you to see this list in the inspector :



The transform means (variable transformation) and is used to store and manipulate the position of the object and its rotation and size to urge it contains 3 different areas:

- Position

Is the position of the body according to the three axes x, y, z Change the values in one of the axes You will find the pattern of change is vertical and horizontal (top, bottom, right, left)

- Rotation

Is the amount of rotation or rotation in the position of the body according to the three axes try to change the values or rotate the body and will notice the change of numbers according to the amount of rotation.

Scale -

It is a fixed body measurement (width and height) by default.

Our goal will be to rotate this image attached to the object, which will rotate it completely

How to write a transform, a comma, and a rotation where the rotation is within the sub-list of the transform

Submitting the following image:

```
nn.cs* X
Assembly-CSharp nn
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class nn : MonoBehaviour
6 {
7
8     // Start is called before the first frame update
9     void Start()
10    {
11    }
12
13    // Update is called once per frame
14    void Update()
15    {
16        transform.Rotate(0, 0, 6);
17    }
18 }
19
20
```

Now the body that put this script will rotate but follow the course of any axis?
As previously explained



The value that changes is the value of the z axis and therefore the body will rotate around this axis only

To preach that the script works the work of the mind mind of the object object without a script can not show any movement will be like the statue is

frozen

When you see the documentation of Unity and what programmers write of codes in the development of games are likely to think that it is a real curse and will not understand what is happening .. Do not worry it's like: (Default)

Milky Way. Solar Group. Planet Earth. Iraq. Karbala

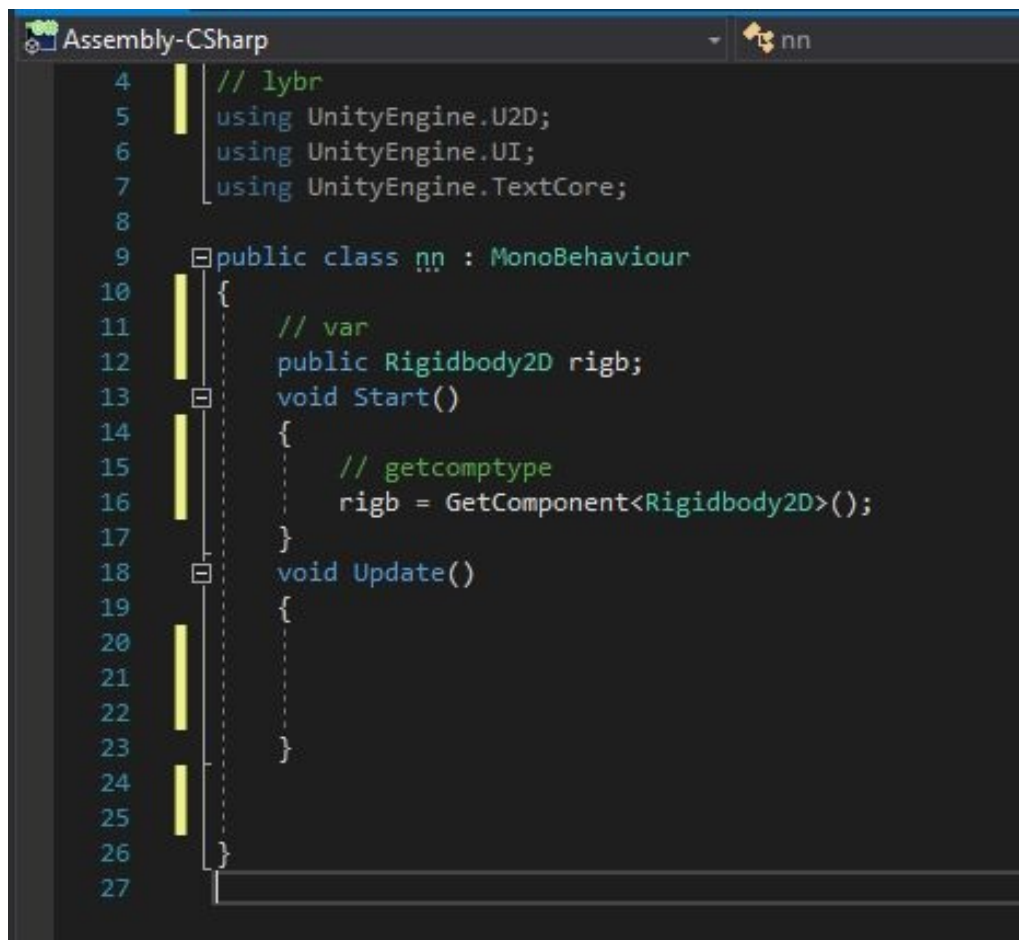
If you see this address remotely, it will definitely make fun of me, but when you read it close you will know what it means. There are basic sentences, branches, and branching branches, so when a program code or a class is attached to a GameObject, it becomes a component. Dot Syntax is similar to the title to locate GameObjects and its components.

With these concepts within your thoughts, we can go ahead and see the details of the structure of the sentence and the rules and syntax used to work with variables and functions is very simple now just think about how to move an object on the axis in the engine engine space using the keyboard buttons What sub-sentences will we use? Of course, you should use the if command in this operation with Position. Before writing the code, you should know that we can add the Rigidbody property, which means the movement of the body, which is cohesive with gravity. Each object added to this feature has real torque and movement. Most often according to the experts, in order to get to this property programmatically by the editor must be written a variable type Rigidbody2D in the case of a two-dimensional game either in the games of three games to write only 2D, and to adjust this property in order to deal with it and you can move the body through the keyboard must also Asthhar and call the property In the wild function Or "start" function where we will write:

- The gravitational variable in the beginning
- equal sign = (set)
- GetComponent instruction that means "Get the component" where you can access both embedded components or scripts using this instruction.
- The value of the instruction between the two largest and smallest marks <2> where the value would be Rigidbody2D

- closing the instruction that will be ();

See the following image:



```
4 // lybr
5 using UnityEngine.U2D;
6 using UnityEngine.UI;
7 using UnityEngine.TextCore;
8
9 public class nn : MonoBehaviour
10 {
11     // var
12     public Rigidbody2D rigb;
13     void Start()
14     {
15         // getcomptype
16         rigb = GetComponent<Rigidbody2D>();
17     }
18     void Update()
19     {
20
21
22
23     }
24
25
26 }
27
```

Well this is good, now with the use of conditional statement (IF)

As you know, conditional statements in C # language take the condition and assume that if the condition in the body area is achieved in parentheses, do the specified event or activate the specified property. So, now think with me how to write the condition inside the brackets? Our goal is to make the object move in a certain direction if the w (for example) button is activated, ie the user has pressed it and the state of the condition in this case is true,

Yes, we must write "input" which means the input of the user in the game and want the latter to restrict the input and take only from the keyboard in this instruction, try typing the following code (code is incomplete) in the space in the update function (update):

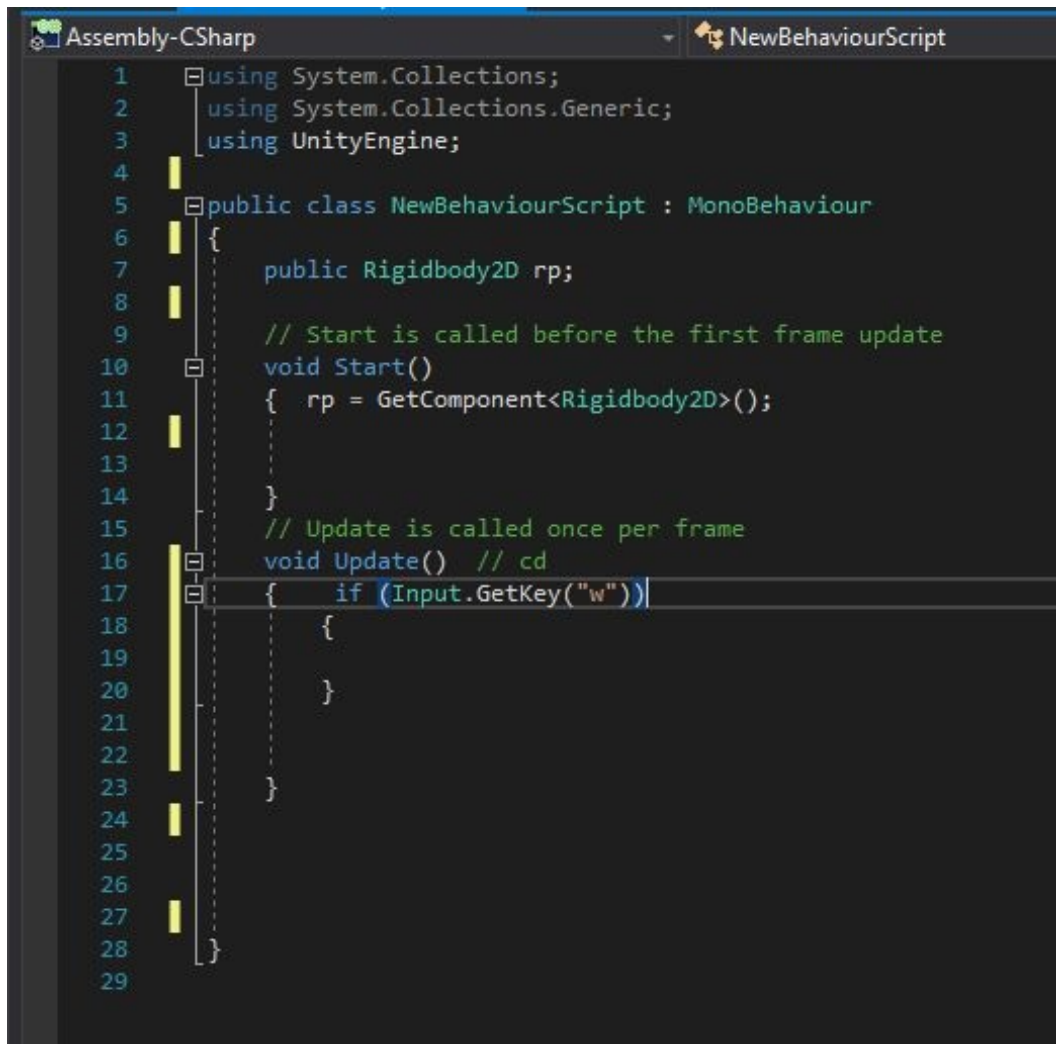
```
If (Input.) {}
```

Many months of input submenu will be followed by GetMouseButtonUp and GetMouseButtonDw

And anykey

Now you can definitely use it for the same purpose but for simplicity and ease

Use Getkey where "key access" means this instruction takes (string) and (CHAR) characters and strings of characters in the following image:



```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NewBehaviourScript : MonoBehaviour
6  {
7      public Rigidbody2D rp;
8
9      // Start is called before the first frame update
10     void Start()
11     { rp = GetComponent<Rigidbody2D>();
12
13     }
14
15     // Update is called once per frame
16     void Update() // cd
17     { if (Input.GetKey("w"))
18
19     {
20
21     }
22
23     }
24
25
26
27
28 }
29
```

Code format:

```
If (input.GetKey ( "w" ))
{

}
```

Do you now think like me how to move the body up in the axis (jump)?

Well, do you see the variable rb, whose task is to set the gravity of the body in the editor, why not try it?

Always write, try and think logically, and do not forget that you are using Visual Studio. This integrated development environment will not leave you alone in the code but will try to understand exactly what you want and help you view suggestions and solutions

When you write the variable of gravity within the event area of the instruction if you follow several sub-instructions for it, choose the velocity instruction, which means "speed" to urge that we want to reach something. The code will be as follows:

```
If (input.GetKey ("w"))
```

```
}
```

```
Rb. Velocity
```

```
{
```

- The code is incomplete

As you know, the use of a single equals sign = in c # means "mapping" mostly. We will assign a value to the gravitational speed that is run through the three axes x, y, z

We will tell the editor that we want to assign a new value to the current gravity value. What will we use? Certainly new is the instruction for changing from one state to another for values.

```
rb.velocity = new
```

- The code is incomplete

We want to (turn) the body to the top by pressing the key

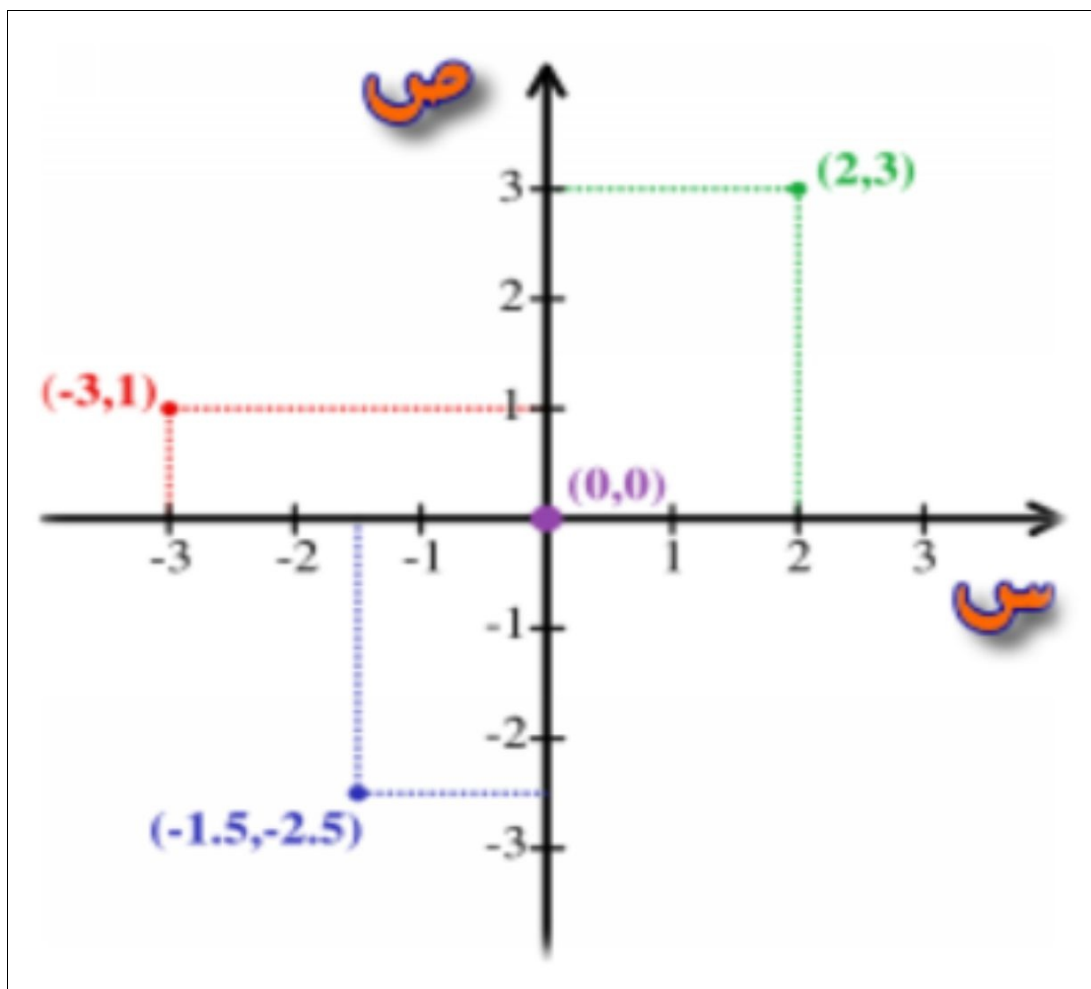
We will write vector instruction which means (vector) since we are writing in a 2D game we have to use 2vector

This instruction takes only two vectors x, y only, unlike vector3

Which takes three vectors x, y, z in the 3D games :

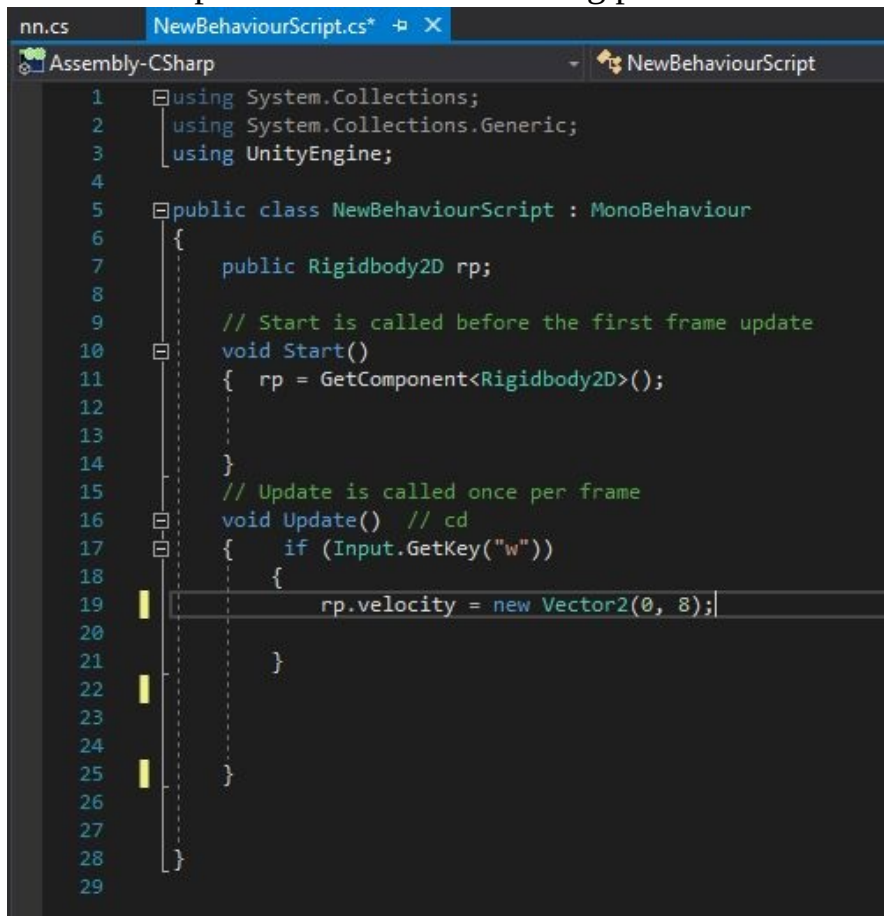
rb.velocity = new Vector2(x, y);

x, y is the amount of change in the value of the vector velocity within the discrete coordinate in the game engine space :



You can set the motion value to higher by putting the amount of movement speed in the right axis position on the right. For example, type 8 and in the horizontal x axis make it 0

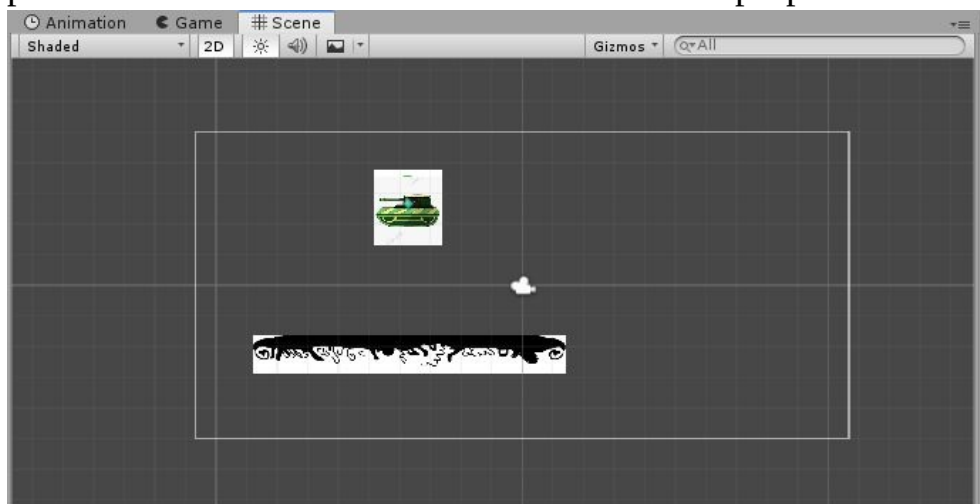
See the complete code in the following picture:

A screenshot of a C# script named 'NewBehaviourScript.cs' in the Unity IDE. The script is a MonoBehaviour class. It includes using statements for System.Collections, System.Collections.Generic, and UnityEngine. It has a public Rigidbody2D variable 'rp'. The Start method calls GetComponent<Rigidbody2D>(). The Update method checks if the 'w' key is pressed and sets the velocity of 'rp' to a new Vector2(0, 8).

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class NewBehaviourScript : MonoBehaviour
6 {
7     public Rigidbody2D rp;
8
9     // Start is called before the first frame update
10    void Start()
11    {
12        rp = GetComponent<Rigidbody2D>();
13    }
14
15    // Update is called once per frame
16    void Update() // cd
17    {
18        if (Input.GetKey("w"))
19        {
20            rp.velocity = new Vector2(0, 8);
21        }
22    }
23
24
25 }
26
27
28
29
```

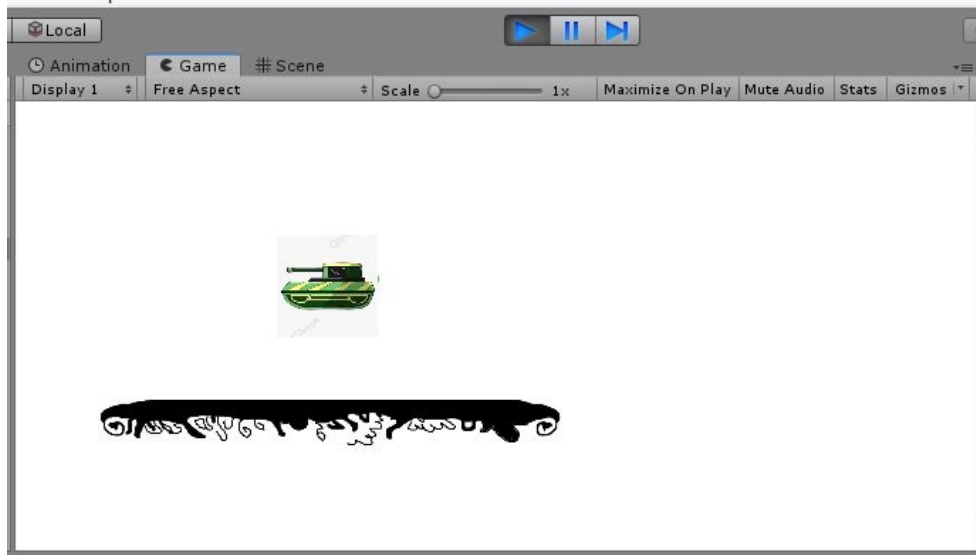
This is great, now we're saving the code from the file menu we choose to save and get back to unity

Add some plug-in to the scene for the experiment only, I've downloaded some pictures from the Internet and I have added the properties of the objects:



- I added the rigidbody property to the tank, the Box Collider, the jump code,
- The Box Collider has been added to the floor, without gravity because we do not want it to fall in the scene from running the game

Now run the game and press the (w)



This is great. I've sent the code spirit in the Game Object to interact with input, but is that enough? Do not want to move the body to the four directions in order to move freely in space. Think now about how to do this with the same rules you have learned. Always remember that programming depends on your logical thinking. Think about the dimensions of this code.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class NewBehaviourScript : MonoBehaviour
6  {
7      public Rigidbody2D rp;
8      // Start is called before the first frame update
9      void Start()
10     { rp = GetComponent<Rigidbody2D>();
11     }
12     // Update is called once per frame
13     void Update() // cd
14     { if (Input.GetKey("w"))
15       {
16           rp.velocity = new Vector2(0, 8);
17       }
18       else if (Input.GetKey("a"))
19       {
20           rp.velocity = new Vector2(-8, 0);
21       }
22       else if (Input.GetKey("d"))
23       {
24           rp.velocity = new Vector2(8, 0);
25       }
26       else if (Input.GetKey("s"))
27       {
28           rp.velocity = new Vector2(0, -8);
29       }
30     }
31 }

```

- I noticed that I used serialized if and else if

Chapter III

Create and launch projectiles

So far we have come a long way towards completing the mechanics of the game, where we have a scene where we can wander and zoom in and out of the camera, in addition to the possibility of building a stage within this scene using the building blocks and forms of monsters that we made. The next step is to make a slingshot to launch projectiles, in addition to the projectiles themselves for which we will use the animal images in the Kenney.nl \ AnimalPack folder. These animals are shown in the following picture:



We have to build a extruded mold for each of these four. This template will initially contain the Sprite Renderer component that has become known to us, as well as the Rigid Body 2D and Circle Collider 2D components. These components can transform each image into a physically active object. All we need to do is adjust the mass values of the solid body component to 5 for each of these images. In addition, to activate the Is Kinematic option, which prevents the solid body from responding to external forces, which we will need to change. Later. The large mass is necessary to make these projectiles have a noticeable

effect when they hit the building blocks or the opponents upon launch. After that we start to write and add the necessary applets for these projectiles. The beginning will be with the main applet and most important is Projectile. The following narrative is described:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class Projectile: MonoBehaviour {
```

```
// The number of seconds the projectile will live in the scene after its launch
```

```
public float lifeSpan = 7.5f;
```

```
// Is the player allowed to control this projectile right now?
```

```
private bool controllable = false;
```

```
// Has the player grabbed this projectile and prepared it for launch?
```

```
private bool held = false;
```

```
// Is this projectile already launched?
```

```
private bool launched = false;
```

```
// Was the projectile attack carried out?
```

```
private bool attackPerformed = false;
```

```
// Variable to store the projectile location the moment the player grabs him  
and before pulling it in preparation for launch
```



```
private Vector2 holdPosition;
```

```
// It is called once at startup
```

```
void Start () {
```

```
}
```

```
// It is called once when each frame is rendered
```

```
void Update () {
```

```
}
```

```
// Allows the player to control this projectile provided that it has not already  
been fired
```

```
public void AllowControl ()
```

```
{
```

```
    if (! launched)
```

```
    {
```

```
        controllable = true;
```

```
    }
```

```
}
```

```
// Called at the beginning of the player to hold the projectile in preparation  
for launch
```

```
public void Hold ()
```

```
{
```

```
    if (controllable &&! held &&! launched)
```

```

{
    held = true;
    holdPosition = transform.position;
    // Send a message telling the player to catch the projectile
    SendMessage ("ProjectileHeld");
}
}

// Launches the projectile using the supplied force
public void Launch (float forceMultiplier)
{
    if (controllable && held &&! launched)
    {
        // Calculate the launch vector
        Vector2 launchPos = transform.position;
        Vector2 launchForce = (holdPosition - launchPos) * forceMultiplier;
        // Add the calculated release force to the solid body
        Rigidbody2D myRB = GetComponent <Rigidbody2D> ();
        myRB.isKinematic = false;
        myRB.AddForce (launchForce, ForceMode2D.Impulse);
        // Set the new status variables
        launched = true;
        held = false;
        controllable = false;

        // Destroy the projectile after the specified seconds have elapsed to stay
        in the scene
        Destroy (gameObject, lifeSpan);
    }
}

```

```
// Send a message telling the launch  
SendMessage ("ProjectileLaunched");  
}  
}
```

// Carry out the special attack of this projectile after its launch

```
public void PerformSpecialAttack ()  
{  
    if (! attackPerformed && launched)  
    {  
        // Allow your attack only once  
        attackPerformed = true;  
        SendMessage ("DoSpecialAttack",  
SendMessageOptions.DontRequireReceiver);  
    }  
}
```

// Drag the projectile to the specified location provided it is manageable by the player

```
public void Drag (Vector2 position)  
{  
    if (controllable && held &&! launched)  
    {  
        transform.position = position;  
    }  
}
```

// Tell if the player is currently holding the projectile in preparation for launch

```
public bool IsHeld ()  
{  
    return held;  
}
```

// Tell if the projectile is already launched

```
public bool IsLaunched ()  
{  
    return launched;  
}  
}
```

Note that the only general variable in this applet is `lifeSpan`, which determines how long the projectile stays in the scene after it is launched. Otherwise, we have state variables `launched`, `held`, `controllable`, and `attackPerformed`, all of which are private and can only be controlled by sending messages or calling functions. In addition to this general variable we have four special variables reflect the different situations in which the projectile from the beginning of the game until it is launched until it finally disappears from the scene. These variables are `controllable`, `held`, `launched`, and `attackPerformed` and their initial value is false. The different projectile modes come in the following sequence:

At the beginning of the game, the projectile is placed on the ground next to the slingshot, and in the meantime remains static and the player can not control until the turn comes into play. In this case the value of the `controllable` variable is false, which prevents the player from controlling the projectile.

As soon as the projectile turns into the launcher and is placed on the ejector, the `AllowControl ()` function is called, which changes the `controllable` value to true and allows the player to control the projectile. The other three variables remain false.

Once the player clicks the projectile, the Hold () function is called, which assumes that the value of the held variable is changed to true. Since this variable indicates that the player is holding the projectile, it must first make sure that the player is allowed to control it by checking the controllable value, it should also make sure that it is not held by checking the value held itself, and finally must check the value of launched to be sure That the projectile is not released yet, because the projectile cannot be held after its release. After these three conditions are verified, the location where the projected player is held is stored in the holdPosition variable and the ProjectileHeld message is sent to report that the projectile was caught.

After grabbing the player begins to move the projectile to prepare it for launch, where he pulls back and down in preparation for launch. While holding the projectile, the player is allowed to call the Drag () function, which moves the projectile to a specific location. As you can see, the process of moving through this function depends on the fact that the projectile is manageable and currently held, and it should not have been launched.

When the player drops the projectile, the Launch () function is called, which can be given a numeric value representing the firing force coefficient. This coefficient enables us to make more than one slingshot with different firing forces. When called, this function verifies that the projectile is under the control of the player and that the player is currently holding it, and that it has not yet been launched. After these conditions are met, the firing force is calculated by the distance between the projectile's holding position and its dropping position and multiplied by the function-supplied operator, since pulling the projectile further would result in greater firing force. The solid object is then activated again by setting the isKinematic variable to false and thus reactivating the solid body's response to external forces before adding its firing force. After the launch is executed the status variables are updated; the player is prevented from controlling the projectile by changing the controllable to false and the launch state is activated by changing launched to true and is also re-held to false since the player is no longer holding the projectile. Finally the ProjectileLaunched message is sent in order to inform other applets that the projectile has been launched.

After launching the projectile, one last step the player can take is to carry out the projectile attack, such as splitting into three smaller, double-speed or other projectiles. A player can perform this attack by calling the PerformSpecialAttack

() function, which makes sure that the attackPerformed value is false; this attack is allowed only once. In addition to this condition, it must be ensured that the projectile has already been launched by checking the launched variable; this attack can be carried out only after the projectile is launched. As you can see, this function does not actually execute the attack, instead it sends a DoSpecialAttack message that another applet will receive and execute the actual attack accordingly. By separating the recall from the attack, we continue to work on the principle of separation of interests and enable ourselves to program more than one type of attack without affecting the basic program structure.

Unlike these phases, the IsHeld () and IsLaunched () functions enable other applets to read the values of special variables but not change their value. Reading these two variables will be of interest to applets whose work depends on the projectile applet as we will see shortly. Another note is to use the SendMessageOptions.DontRequireReceiver option when you send the DoSpecialAttack message, so we do not require a message receiver. The reason is that this attack is optional and it is OK to have projectiles that have no special attack.

With this we have identified the basic programmable ballistics, and we have some small auxiliary programs for secondary functions.

The first applet is ProjectileSounds and is responsible for ballistics sounds. What this applet simply does is receive the ProjectileHeld constipation messages and launch ProjectileLaunched and play the selected audio file for each process. The following narrative illustrates this applet:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class ProjectileSounds: MonoBehaviour {
```

```
// Audio file for launch
```

```

    public AudioClip launchSound;

    // Audio file for constipation
    public AudioClip holdSound;

    // Called once at startup
    void Start () {

    }

    // Called once when rendering each frame
    void Update () {

    }

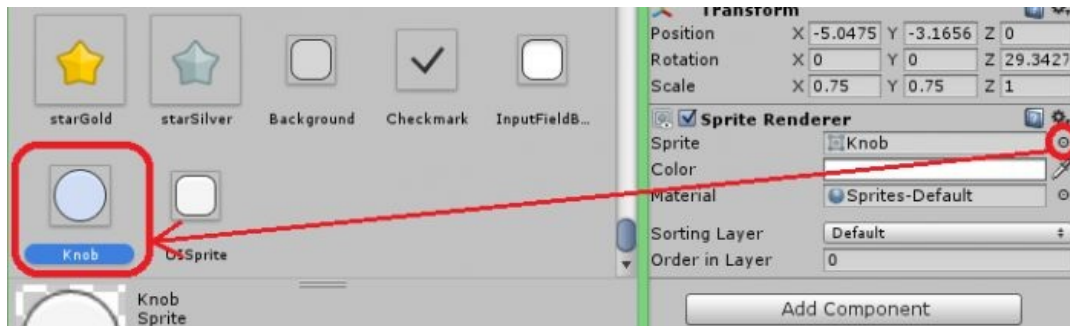
    void ProjectileHeld ()
    {
        AudioSource.PlayClipAtPoint (holdSound, transform.position);
    }

    void ProjectileLaunched ()
    {
        AudioSource.PlayClipAtPoint (launchSound, transform.position);
    }
}

```

The second applet that we will be discussing from the projectile template applet is the applet for drawing the projectile motion path after launch. The drawn path

will be points, including fixed distances, along the path that the projectile traveled from the moment it dropped at the launch slingshot to its last point. Before explaining the applet we will build a template that represents the point object that we will use to draw the path. To build the template, just add a new blank object to the scene and then add the SpriteRenderer component to it. Then click on the browse button for the Sprite cell in the component as in the image, and scroll down to the bottom of the window where you will find below a set of default images that Unity uses to build the user interface. Select the Knob image and then close the window. Then name the new template PathPoint and reduce its size on the x and y axes to 0.75:



Now we can write the path drawing applet and add it to the projectile template. This applet is described in the following narrative:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class PathDrawer: MonoBehaviour {
```

```
    // The template used to draw points
```

```
    public GameObject pathPointPrefab;
```

```
    // Distance between two consecutive points
```

```
    public float pointDistance = 0.75f;
```



```

// Parent object for path point objects
Transform pathParent;

// Variable to store the location of the last point added
Vector2 lastPointPosition;

// Internal variable to see if the projectile was launched or not
bool launched = false;

// Called once at startup
void Start () {
// Path Find the parent object of the named path points
    pathParent = GameObject.Find ("Path"). transform;
}

// Called once when rendering each frame
void Update () {
    if (launched)
    {
        float dist = Vector2.Distance (transform.position, lastPointPosition);
        if (dist>= pointDistance)
        {
            // It's time to add a new point
            AddPathPoint ();
        }
    }
}

void ProjectileLaunched ()
{
    // The projectile is just launched so delete the previous track
    for (int i = 0; i <pathParent.childCount; i ++)

```

```

{
    Destroy (pathParent.GetChild (i) .gameObject);
}

AddPathPoint ();

// Update the variable value as the projectile has been launched
launched = true;
}

// Adds a new point to the path
void AddPathPoint ()
{
    // Create a new point using the template
    GameObject newPoint = (GameObject) Instantiate (pathPointPrefab);
    // Place the point in the current projectile location
    newPoint.transform.position = transform.position;
    // Set the parent object to the point
    newPoint.transform.parent = pathParent;
    // Store the location of the added point
    lastPointPosition = transform.position;
}
}

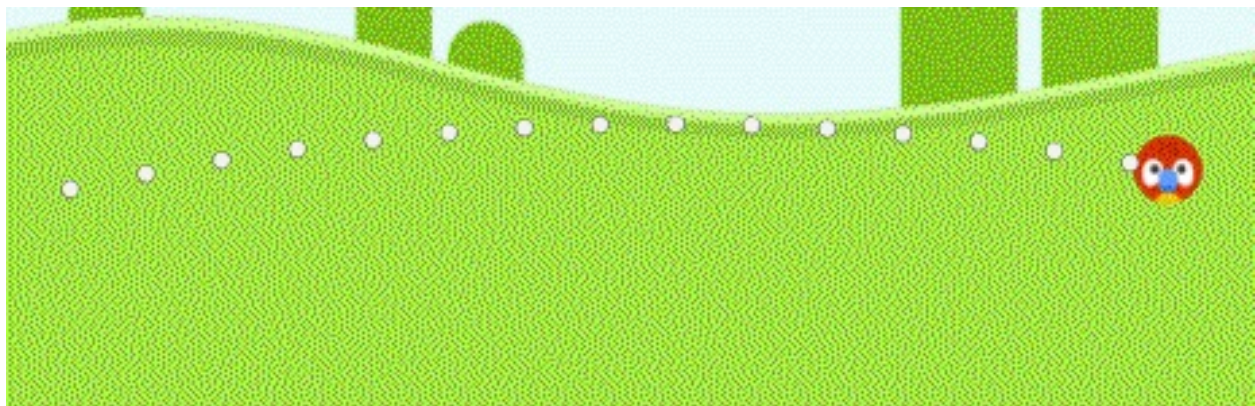
```

This applet uses the template we just created in order to draw points along the path, so we'll need to define this template via the `pathPointPrefab` variable. Then, with the `pointDistance` variable, we can adjust the distance we want between two consecutive points. Next we need a reference to `Path`, an empty object that we have to add to the hierarchy of the scene as the root object. This object will be the father of all the track points, and it helps us access them at once to delete them while drawing a new path as we'll see shortly. Since we will calculate the distance between each two consecutive points as the projectile moves to draw the path, we always have to keep the location of the last point drawn. This location is stored in the `lastPointPosition` variable. Finally, we know that the path

should be drawn only after the projectile is launched, so we use the launched variable to know whether or not it was launched.

Remember that when the projectile is launched, the Projectile applet sends the ProjectileLaunched message, which the PathDrawer receives through the function of the same name. Once the message arrives, the previously drawn path (if any) is deleted by deleting all the sons of the empty object, which we keep a reference to in the pathParent variable. After the deletion is finished, we draw a point at the launch location by calling the AddPathPoint () function, and then the launched value is changed to true.

What the AddPathPoint () function does is to create a new point in the current projectile location by using the pathPointPrefab template, add it as a son of the Path object, and then store its location in the lastPointPosition variable. As long as the projectile object is in the scene, the Update () function will be called in each frame, but it will not do anything until the launched value changes to true. If this condition is met, it means that the projectile has been released and therefore the path must be plotted as it moves; therefore, we calculate the distance between the current projectile location of transform.position and the location of the lastPointPosition. If this distance increases or is equal to pointDistance, then it is time to add a new point to this and AddNewPoint () is called. The following image represents the process of drawing the projectile path as it moves:



Special attacks for projectiles

To complete the projectiles we manufacture a special attack that the player can perform after launching the projectile. This attack has multiple images in the original game Angry Birds from which we quote in this series of lessons. We will be satisfied with two examples to illustrate how these attacks are built. The first is the velocity attack, which we will adopt for bird-shaped projectiles, which doubles the speed of the projectile, making its impact even greater when it hits building blocks or opponents. The second attack we will adopt for giraffe and elephant projectiles is the fissile attack, where the original projectile is divided into a number of smaller projectiles that can hit more than one target in different places.

Let's start with the easiest attack, a speed attack. Since the logic of attacks is quite different from one attack to another, we have to separate each attack into a separate applet. The only common factor between these attacks is that they will receive the DoSpecialAttack message that the Projectile projectile sends when the PerformSpecialAttack () function is called and the conditions necessary to perform this attack are checked. A speed attack implementation applet is called SpeedAttack, and what it does is bring in the solid object component and then double its speed by a certain amount without changing its direction. This applet is described in the following narrative. Remember that attack applets should be added to projectile templates.

```
using UnityEngine;
using System.Collections;

public class SpeedAttack: MonoBehaviour {

    // Multiply the current velocity of the projectile by this
    // Amount when carrying out the attack
    public float speedFactor = 1.5f;

    // It is called once at startup
    void Start () {

    }
```

```

// It is called once when each frame is rendered
void Update () {

}

// Consequently, it performs the DoSpecialAttack speed attack that receives the message
public void DoSpecialAttack ()
{
    // Bring the rigid body component of the projectile object
    Rigidbody2D myRB = GetComponent <Rigidbody2D> ();

    // Multiply the speed by multiplying and then set the speed of the object to the new output
    myRB.velocity = myRB.velocity * speedFactor;
}
}

```

The second type of special attacks as we mentioned is a fissile attack, which leads to the fragmentation of the projectile into smaller projectiles (fragments), which in turn scatter over a relatively large area. Before moving to the programmer for this attack, we note that its implementation will need to create new objects, fragments that will be scattered by the implementation of the attack. This means that we will need to build molds for these fragments, and there will be two molds specifically: one for elephant extruded fragments and the other for giraffe extruded fragments. I will call these two templates ElephantCluster and GiraffeCluster, which are virtually the same in everything except the image shown. These two molds are simple, each carrying the original projectile image with the object scaled down to 0.75 on the x and y axes to make the fragments smaller than the original projectile. In addition, we will add a Rigid Body 2D rigid component and a Circle Collider 2D collision component, making the fragment molds ready.

The applet that will implement this type of attack is called ClusterAttack and is described in the following narrative:

```

using UnityEngine;
using System.Collections;

public class ClusterAttack: MonoBehaviour {

    // template that will be used to create fragments
    public GameObject clusterPrefab;

    // The number of seconds each fragment will live before being destroyed and removed from the scene
    public float clusterLife = 4.0f;

    // How many fragments will be produced from this projectile
    public int clusterCount = 3;

    // Called once at startup
    void Start () {

    }

    // Called once when rendering each frame
    void Update () {

    }

    // Consequently implements the DoSpecialAttack fission attack to receive the message
    public void DoSpecialAttack ()
    {
        // Bring the current speed of the original projectile
        Rigidbody2D myRB = GetComponent <Rigidbody2D> ();
        float originalVelocity = myRB.velocity.magnitude;
    }
}

```

```

// Store all the collision components of the fragments in this array
Collider2D [] colliders = new Collider2D [clusterCount];
Collider2D myCollider = GetComponent <Collider2D> ();

for (int i = 0; i <clusterCount; i ++)
{
    // Create a new fragment
    GameObject cluster = (GameObject) Instantiate (clusterPrefab);
    // Set the location, name, and father of the splint object
    cluster.transform.parent = transform.parent;
    cluster.name = name + "_cluster_" + i;
    cluster.transform.position = transform.position;
    // Store the fragment collision component at the current location in the array
    colliders [i] = cluster.GetComponent <Collider2D> ();
    // Neglect the collision that can occur between this fragment and the fragments created before
it
    // In addition to the collision that can occur between the fragment and the original object
    Physics2D.IgnoreCollision (colliders [i], myCollider);
    for (int a = 0; a <i; a ++)
    {
        Physics2D.IgnoreCollision (colliders [i], colliders [a]);
    }

    Vector2 clusterVelocity;

    // With each new fragment we reduce the vehicle speed horizontally and increase it vertically
in order to ensure fragmentation
    clusterVelocity.x = (originalVelocity / clusterCount) * (clusterCount - i);
    clusterVelocity.y = (originalVelocity / clusterCount) * -i;

    // Bring a solid body object to the new splinter
    Rigidbody2D clusterRB = cluster.GetComponent <Rigidbody2D> ();
    clusterRB.velocity = clusterVelocity;
    // Select the fragment block to equal the mass of the original object

```

```

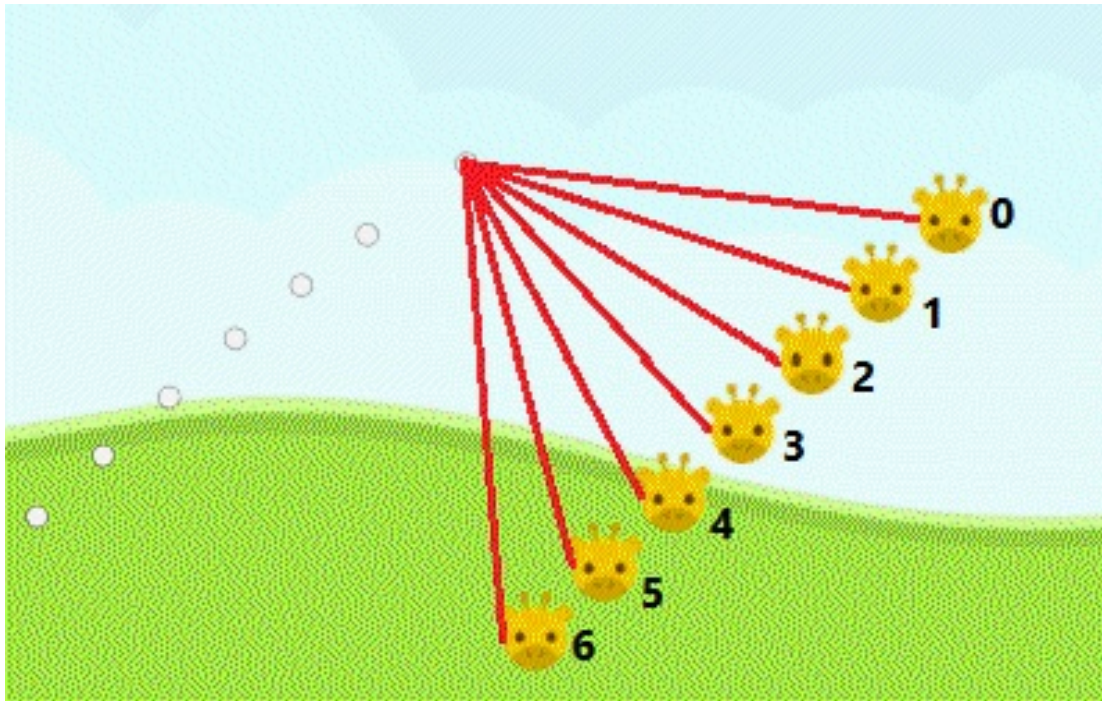
        clusterRB.mass = myRB.mass;

        // Destroy the splinter after its age has passed
        Destroy (cluster, clusterLife);
    }

    // Finally destroy the original projectile object
    Destroy (gameObject);
}
}

```

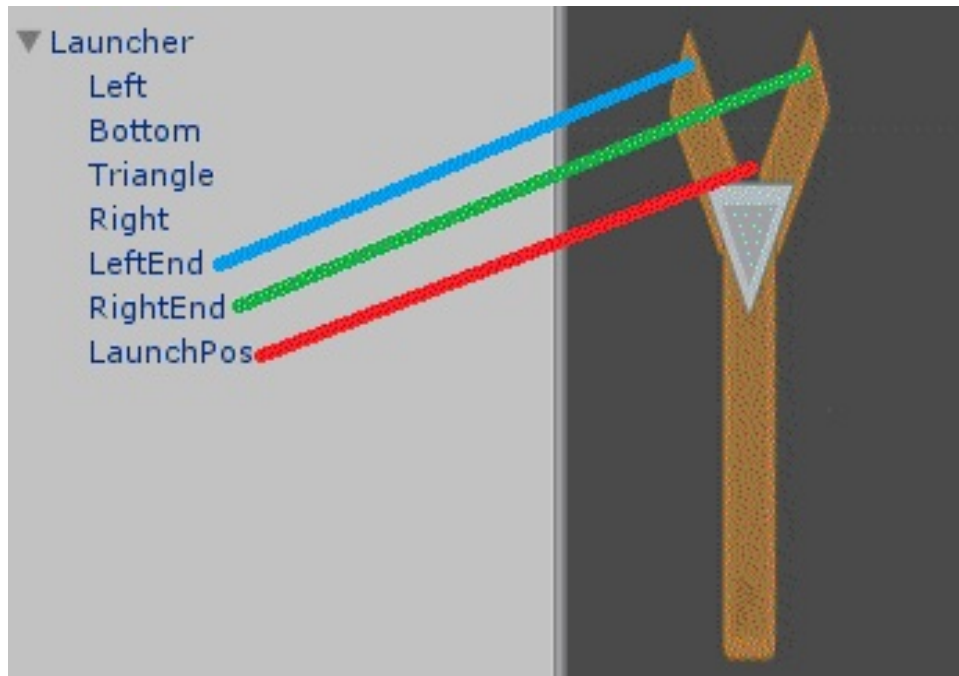
The idea of doing this attack is to receive the DoSpecialAttack message and then create the specified number of fragments using the specified template. In order to prevent collisions between fragments and some of them and also between fragments and the original projectile - where a collision can occur the moment before it is deleted from the scene - we use the Physics2D.IgnoreCollision () function and provide it with the two collision components that we want to ignore collisions between. Note that we knew a matrix of collision components in order to store the components of all fragments, and when creating a new fragment we pass on the components of the collisions of the previous fragments and call the function mentioned between the old and new components in order to negate the collisions. The next step is the speed of the splinter motion, where we take the amount of the original projectile's velocity and multiply it each time by a different value to get the horizontal and vertical components of the new velocity. These two components change from fragment to another, where the first fragment begins with a high horizontal and low vertical velocity, and then these values begin to change as the vertical value increases gradually downward and the horizontal decreases, resulting in the dispersion of projectiles in a manner similar to what you see in the picture below (I have in this picture Increase the number of fragments to clarify the idea):



Note that the fragments spread apart from the original projectile fission site. Then we adjust the mass of each fragment to equal the mass of the original projectile. While it is logical to divide the mass by the number of fragments in order to distribute them evenly, copying the original mass of all fragments will give them greater destruction power, giving the special attack its preferred advantage.

Ejector industry

Let us now turn to the ejector, a slingshot that will launch these projectiles towards its targets. Unfortunately, our graphics package does not contain a slingshot image, so we will try to use some wooden and metal shapes to create a simple shape that looks like it. Here I will use three wooden rectangles and a stone triangle to make the shape you see in the following picture. These objects must be placed as sons of one empty object containing all of them, and the Order in Layer value in the Sprite Renderer component of the triangle must also be adjusted and made 1, so that it appears in front of the pieces as shown in the picture:



In addition to the four images we have scaled and rotated to make a slingshot, there are 3 blank objects that indicate colored lines to their locations. These empty objects have a software utility that we will see shortly. The LaunchPos object represents where the projectile is placed before it is launched, and the RightEnd and LeftEnd objects represent the locations of the ends of the rubber band that will push the projectiles when they are launched. The slingshot in its current form is ready to make the initial version of the mold, to which we will add some applets and other components to it.

The first of these is the main applet that launches projectiles at targets. Let's get to know this applet called Launcher in the following narrative and then discuss the details of its functions:

```
using UnityEngine;
using System.Collections;
public class Launcher: MonoBehaviour {
    // The launch force coefficient of this ejector
    public float launchForce = 1.0f;
```

```

    // Maximum length the rubber ejector rope can extend to
    public float maxStretch = 1.0f;
    // The location where the current projectile will be placed before being held by the player
    public Transform launchPosition;
    // Current projectile subject on the ejector
    public Projectile currentProjectile;
    // Have all the projectiles in the scene been fired?
    private bool projectilesConsumed = false;
    // Called once at startup
    void Start () {

    }

    // Called once when rendering each frame
    void Update () {
        if (projectilesConsumed)
        {
            // There's nothing to do
            return;
        }
        if (currentProjectile != null)
        {
            // If the projectile was not fired, it was also not caught by the player
            // Then bring the projectile to the launch site
            if (! currentProjectile.IsHeld () &&! currentProjectile.IsLaunched ())
            {
                BringCurrentProjectile ();
            }
        }
        else
        {
            // There is currently no projectile on the ejector
            // Find the nearest projectile and bring it to the launch site
            currentProjectile = GetNearestProjectile ();
            if (currentProjectile == null)
            {

```

// All the projectiles were consumed, send a message telling them

```
    projectilesConsumed = true;
    SendMessageUpwards ("ProjectilesConsumed");
}
}
```

// Finds the nearest projectile and returns it

Projectile GetNearestProjectile ()

```
{
    Projectile [] allProjectiles = FindObjectsOfType <Projectile> ();

    if (allProjectiles.Length == 0)
    {
        // There are no longer any projectiles
        return null;
    }
}
```

// Search for the nearest projectile and return it

```
Projectile nearest = allProjectiles [0];
float minDist = Vector2.Distance (nearest.transform.position, transform.position);

for (int i = 1; i <allProjectiles.Length; i ++)
{
    float dist = Vector2.Distance (allProjectiles [i] .transform.position, transform.position);
    if (dist <minDist)
    {
        minDist = dist;
        nearest = allProjectiles [i];
    }
}

return nearest;
}
```

```

// You move the current projectile one step smoothly towards the launch site
void BringCurrentProjectile ()
{
    // Bring locations where the projectile will move between them
    Vector2 projectilePos = currentProjectile.transform.position;
    Vector2 launcherPos = launchPosition.transform.position;

    if (projectilePos == launcherPos)
    {
        // Extruded at the launch site actually, no need to move it
        return;
    }

    // Use linear interpolation with elapsed time between frames for smooth movement
    projectilePos = Vector2.Lerp (projectilePos, launcherPos, Time.deltaTime * 5.0f);
    // Put the projectile in its new position
    currentProjectile.transform.position = projectilePos;

    if (Vector2.Distance (launcherPos, projectilePos) <0.1f)
    {
        // The projectile became very close, place it directly at the launch site
        currentProjectile.transform.position = launcherPos;
        currentProjectile.AllowControl ();
    }
}

// Holds the current projectile
public void HoldProjectile ()
{
    if (currentProjectile != null)
    {
        currentProjectile.Hold ();
    }
}

```

```

// Drags the current projectile to a new location
public void DragProjectile (Vector2 newPosition)
{

    if (currentProjectile != null)
    {
        // Make sure not to exceed the maximum elastic cord tension
        float currentDist = Vector2.Distance (newPosition, launchPosition.position);

        if (currentDist > maxStretch)
        {
            // Change the location provided to the furthest allowed point
            float lerpAmount = maxStretch / currentDist;
            newPosition = Vector2.Lerp (launchPosition.position, newPosition, lerpAmount);
        }

        // Place the projectile in the new location
        currentProjectile.Drag (newPosition);
    }

}

// Drops the current projectile and launches it if the player is holding it
public void ReleaseProjectile ()
{
    if (currentProjectile != null)
    {
        currentProjectile.Launch (launchForce);
    }
}

```

The general variables in this applet are launchForce, which represents the launch force, maxStretch, which is the maximum permissible distance between the

projectile and the firing position during the tension (i.e., the maximum extension of the rubber thread) and launchPosition, a variable to store the launch site object named LaunchPos, which we added to the slingshot template when we created it. Finally we have a reference to the current projectile located on the ejector which is currentProjectile.

In each update cycle, the Update () function checks whether an existing projectile is present, and if it does not, it calls the GetNearestProjectile () function that searches for the nearest projectile and returns it. If this function does not find any projectiles in the scene, it returns the value null, in which case the applet sends the ProjectilesConsumed message to the top of the scene hierarchy in order to notify the game controllers that the player has exhausted all their projectiles at this point. When all projectiles are exhausted, the value of the projectilesConsumed variable is changed to true, which means that the Update () function will not do anything anymore. In the case of an existing projectile that the player has not yet grabbed or launched, Update () calls the BringCurrentProjectile () function which moves the current projectile towards the launch site smoothly (remember that the original projectile position is on the ground next to the slingshot). When the projectile arrives at launchPosition, this function will automatically stop moving it even if it is still called by Update ().

Here I will talk a little bit more about the BringCurrentProjectile () function to explain the mechanism you use to achieve the smooth movement of the projectile from its current position towards the launch site. The smooth movement in game engines depends on the elapsed time factor between each two consecutive frames, which is in the Unity variable Time.deltaTime. In addition to this variable we will need a function that calculates the Linear Interpolation between two different values. But what is a linear interpolation? It is simply a value between two lower and higher limits. This value may be an abstract number between two numbers, a position between two locations, a color between two colors, etc. But where exactly is this value between the two limits? What determines this location is the interpolation value, which is a fractional value between zero and one. For example, if we want to calculate interpolation between the numbers 0 and 10, and the interpolation value is 0.6, the result will be 6, and if the interpolation value is 0.45, the result will be 4.5 and so on.

We calculate the new projectile position as it moves towards the launch site using this technique, in which case the interpolation takes place between the minimum projectilePos current and the maximum target we want to reach, the launcherPos. The value of the interpolation is relatively low ie it is closer to the target, which is the time elapsed since the previous tire was rendered multiplied by 5. The importance of using the time value here lies in the fact that not all tires are rendered at the same speed. Not fixed and may increase and decrease. Therefore, to maintain a constant movement speed, we have to multiply by the amount of time which increases as the number of frames per second decreases and vice versa, making the movement speed that the player sees constant regardless of the number of frames per second or less.

The three functions HoldProjectile (), DragProjectile (), and ReleaseProjectile () call the Hold (), Drag (), and Launch () functions of the current projectile currentProjectile. The function that needs some explanation here is DragProjectile () because it contains an additional step that is not present in the Projectile, which is to verify that the distance of the projectile from the initial launch site while pulling back does not exceed the allowable length of the rubber slingshot stretch. This stretch is defined in the maxStretch variable. The way we are going to impose this maximum length should take into account the ease of control as well, if the player pulls the cursor beyond the allowed length should not withdraw with the projectile, but at the same time must remain able to change the angle of launch. In order to achieve this mechanism we will use linear interpolation again and this usage is described in lines 131 and 132.

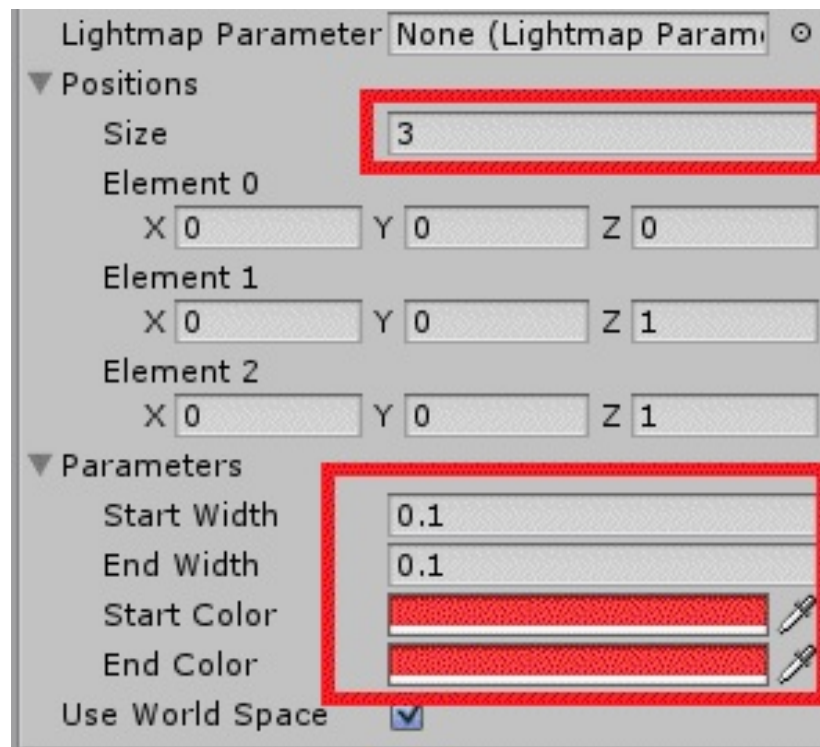
The idea is to calculate the distance between the launch location and the current location of the currentDist and compare it to the maximum expansion, maxStretch. If this distance exceeds the limit, we will divide maxStretch by currentDist, thus obtaining the required interpolation between the original launchPosition.position and the current location of the newPosition. This value will naturally decrease with increasing distance from the cursor to the launch site, thus maintaining a constant distance from the launch site, which is the maxStretch distance. By completing the interpolation, we get the correct newPosition without affecting the smooth motion, and then use the Drag ()

function to move the projectile. It is necessary to use this function and not to move the projectile directly because it verifies the conditions in terms of the fact that the projectile is held by the player and has not been launched, which is the movement conditions according to the rules of the game.

After writing the applet we have to add it to the empty Object Launcher, which is the root of all the slash objects that make up the slingshot. The next applet we will add will draw the rubber cord between the ends of the slingshot and the projectile. But before moving on to the applet we have to add the component responsible for drawing the line that will represent this rope. The component we will add is called Line Renderer and can be added as usual from the Add Component button and then write the component name as in the following picture. This component draws a solid line between a set of points assigned to it via the positions array, starting with the first point in the array to the last point:



After adding the component we have to adjust some of its values: first we have to change the number of points that draw the positions line to 3 and then make it thinner by changing both Start Width and End Width to 0.1, and finally we'll change its color to red at the beginning and end (you can of course choose Any other color). These settings are shown in the following image:



Now let's launch the LauncherRope, which is responsible for drawing this line between the ends of the slingshot and the projectile. This applet is described in the following narrative:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class LauncherRope: MonoBehaviour {
```

```
// Location of the left end of the rope
```

```
public Transform leftEnd;
```

```
// Location of the right end of the rope
```

```
public Transform rightEnd;
```

```
// Reference for Ejector Applet
```

```
Launcher launcher;
```

```
// Reference to the object rendering component added
```

```
LineRenderer line;
```

```
// Called once at startup
```

```
void Start () {
```

```
    launcher = GetComponent <Launcher> ();
```

```
    line = GetComponent <LineRenderer> ();
```

```
// Hide the font at first by disabling its component
```

```
    line.enabled = false;
```

```
}
```

```
// Called once when rendering each frame
```

```
void Update () {
```

```
    // Show the line only if the projectile is held by the player
```

```
    if (launcher.currentProjectile != null &&
```

```
        launcher.currentProjectile.IsHeld ())
```

```
    {
```

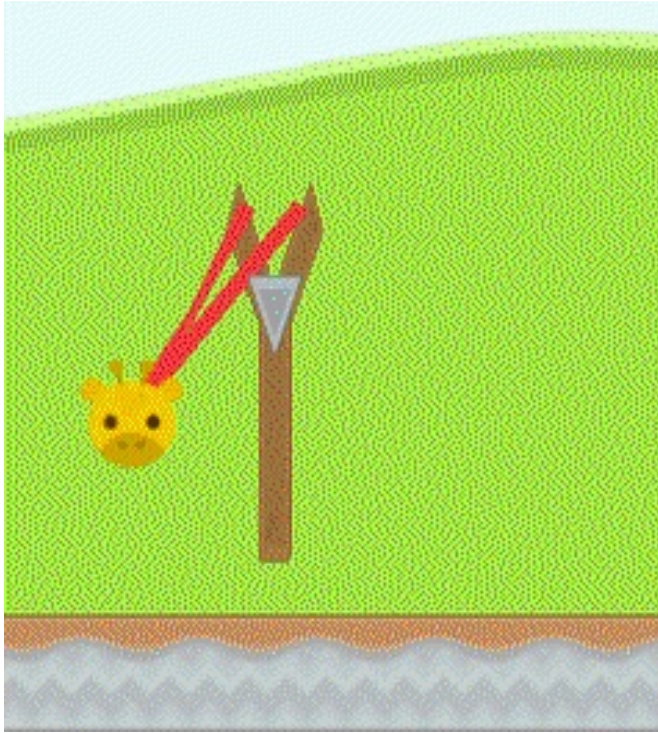
```
        if (! line.enabled)
```

```

    {
        line.enabled = true;
    }
    // Draw the line starting from the left end, the projectile is the right
end
    line.SetPosition (0, leftEnd.position);
    line.SetPosition (1, launcher.currentProjectile.transform.position);
    line.SetPosition (2, rightEnd.position);
}
else
{
    line.enabled = false;
}
}
}

```

You can see how simple this applet is. All it does is disable the line drawing component at first, and then check the status of the current projectile (if any). If this projectile is held by the player, the LineRenderer component is activated making the line visible, and then adjusts the line drawing positions. Remember the two empty objects that we added as slingshot sons, RightEnd and LeftEnd. We will use the leftEnd and rightEnd references defined in the applet and link them via the browser to these two objects. Thus, we have located the first and last point of the line drawn. It remains to determine the location of the middle point, which is, of course, the location of the projectile. Note that we use the SetPosition () function and give it the order of the location in the array followed by the point where we want this location to be. When you play the game and hold the projectile, this line will look like this:



Thus, the tasks of the required slingshot are completed, and we have to add a program to read the player's input so that he can use the mouse to launch the projectiles. This applet is called LauncherMouseInput and its task is to read the mouse input from the player and turn it into commands for the Launcher applet. The following narrative illustrates this applet:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class LauncherMouseInput: MonoBehaviour {
```

```
// Reference for launch applet
```

```
private Launcher launcher;
```

```
// Called once at startup
```

```
void Start () {  
    launcher = GetComponent <Launcher> ();  
}
```

// Called once when rendering each frame

```
void Update () {  
    CheckButtonDown ();  
    CheckDragging ();  
    CheckButtonUp ();  
}
```

```
void CheckButtonDown ()  
{  
    if (Input.GetMouseButtonDown (0))  
    {  
        // The left mouse button has just been pressed  
        // Is there an existing projectile?  
        if (launcher.currentProjectile != null)  
        {  
            // Switch the cursor position from the screen coordinates to the scene space coordinates  
            Vector2 mouseWorldPos = Camera.main.ScreenToWorldPoint  
(Input.mousePosition);  
            // Extract the collision component from the object  
            Collider2D projectileCol =  
launcher.currentProjectile.GetComponent <Collider2D> ();  
            // Is your mouse within the range of the projectile's collision  
component?  
            if (projectileCol.bounds.Contains (mouseWorldPos))
```

```

    {
        // Yes, that is, the mouse button was pressed over the projectile
        // Hold the projectile
        launcher.HoldProjectile ();
    }
}
}
}

```

// Checks whether the player pulls the mouse using the left button

```

void CheckDragging ()
{
    if (Input.GetMouseButton (0))
    {
        Vector2 mouseWorldPos = Camera.main.ScreenToWorldPoint
(Input.mousePosition);
        launcher.DragProjectile (mouseWorldPos);
    }
}

```

// Check if the left mouse button has been depressed

```

void CheckButtonUp ()
{
    if (Input.GetMouseButtonUp (0))
    {
        // The left button is depressed
    }
}

```

```
// Launch the projectile
```

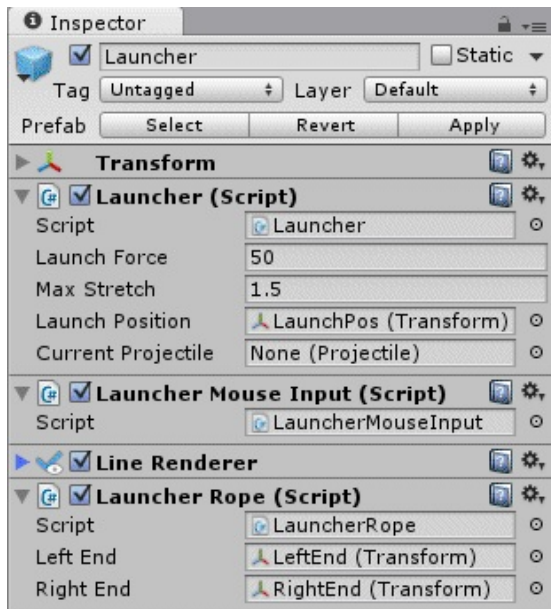
```
launcher.ReleaseProjectile ();
```

```
}
```

```
}
```

```
}
```

The Update () function in this program calls three other functions, respectively: CheckButtonDown (), CheckDragging (), and CheckButtonUp (). In the CheckButtonDown () function, it is first checked that a projectile is present on the slingshot. If found, the mouse pointer is converted from the screen coordinates to the scene coordinates, and then examines whether the site falls within the boundaries of the projectile's collision component. This condition means that the player has pressed the left mouse button on the projectile and therefore grabs it, so the Hold () function is called from the launcher. In the CheckDragging () function, the left mouse button is checked, and in this case the projectile is moved to the cursor position by calling the DragProjectile () function. Remember that this function prevents the projector from exceeding the maximum stretch of the rubber cord, so regardless of the position of the indicator the projectile will remain within that limit. Finally, the CheckButtonUp () function checks whether the player has dropped the left mouse button, in which case the ReleaseProjectile () function is called from the launcher applet until the projectile is launched. The following figure represents the final components of the launch slingshot template:



Well, we now have a background, floor, building blocks, opponents, ejector and projectiles, that is, all elements of the game are ready, and we can try building a scene and playing with it. The following picture shows the advanced stage we have reached after this painstaking effort!



•

Chapter IV

Touch screen input in Unity3D and export for phones

Receiver touch screen input:

Receiving input from touch screen is different from receiving mouse or keyboard input. You are dealing with a screen that receives several touches, dealing with events such as finger positioning, moving and lifting, or placing two fingers and rounding them apart. All these types of input will have to be recognized and executed by the appropriate commands.

Let's start first with the input of UI elements such as buttons. What should we do to receive input on these elements? nothing! This command does the Unity engine automatically. That is, when you touch a button on the screen the application will recognize this command as if you pressed the button and execute the command or commands associated with it.

Let us now turn to the input for the game. Remember that we have written 3 applets to receive mouse input, and we will now need to type 3 against it to receive touch screen input. These three programs are:

CameraMouseInput, which we added to the background and floor elements templates in order to move the camera and control the zoom and zoom.

LauncherMouseInput which we added to the slingshot template so that it receives the player's input on the projectiles and determines the direction and intensity of the aiming and releases the projectile when it drops.

SpecialAttackMouseInput which we added to the projectile templates in order to

perform your attack when you click the mouse button after launch.

The subject is very simply that for each input applet from the mouse we will write an input applet from the touch screen and add it to the same template. The common denominator of touch input receivers is that they will check the game's operating environment. If you find that the operating system is Android, it will destroy the mouse input receiver applets with it on the same object. The importance of this step is that Unity tries to simulate the presence of a mouse on touch screens by turning touches into mouse inputs. This leads to undesirable behavior that is not useful in our game, so we avoid it by deleting any applet read from the mouse.

Let's start with the CameraTouchInput applet control applet that we will add to my background and ground elements where CameraMouseInput is also located. This applet is described in the following narrative:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class CameraTouchInput: MonoBehaviour
```

```
{
```

```
    // Speed to move the camera
```

```
    public float movementSpeed = 0.0075f;
```

```
    // Speed of rounding and deportation
```

```
    public float zoomingSpeed = 0.00075f;
```

```
    // Is the player currently touching the screen with one finger?
```

```
    private bool singleTouch = false;
```

```
    // Does the player touch the screen currently with two fingers
```

```
private bool doubleTouch = false;
```

```
// Reference for camera control applet
```

```
CameraControl camControl;
```

```
// Called once at startup
```

```
void Start ()
```

```
{
```

```
    // Destroy mouse input applet if smartphone operating system is detected
```

```
    if (Application.platform == RuntimePlatform.Android)
```

```
    {
```

```
        CameraMouseInput mouseInput = GetComponent <CameraMouseInput> ();
```

```
        Destroy (mouseInput);
```

```
    }
```

```
    camControl = FindObjectOfType <CameraControl> ();
```

```
}
```

```
// It is called once when each frame is rendered late
```

```
void LateUpdate ()
```

```
{
```

```
    UpdateSingleTouch ();
```

```
    UpdateDragging ();
```

```
    UpdateDoubleTouch ();
```

```
    UpdateZooming ();
```

```
}
```

```
// Make sure the player touches the screen with one finger
```

```
void UpdateSingleTouch ()
```

```
{
```

```
    if (Input.touchCount == 1)
```

```
    {
```

```

Touch playerTouch = Input.touches [0];
if (playerTouch.phase == TouchPhase.Began)
{
    // The player just put his finger on the screen
    // Was the finger placed on this particular object?
    Vector2 touchPos = Camera.main.ScreenToWorldPoint (playerTouch.position);

    // Bring the crash component of this object
    Collider2D myCollider = GetComponent <Collider2D> ();

    // Generate a very short beam starting from the touch position and facing up and right
    // Then verify that this beam has hit the collision component of this object
    RaycastHit2D hit = Physics2D.Raycast (touchPos, Vector2.one, 0.1f);
    if (hit.collider == myCollider)
    {
        // Yes, the player touched this object with one finger
        singleTouch = true;
    }
}
else if (playerTouch.phase == TouchPhase.Ended ||
playerTouch.phase == TouchPhase.Canceled)
{
    // The player just lifted his finger off the screen
    singleTouch = false;
}
else
{
    // The number of fingers on the screen does not equal 1 ie it
    // There is no touch with one finger
    singleTouch = false;
}

```

```
}
```

```
// Scans the player's move to one finger on the screen
```

```
void UpdateDragging ()
```

```
{
```

```
    if (singleTouch)
```

```
    {
```

```
        Touch playerTouch = Input.touches [0];
```

```
        camControl.Move (playerTouch.deltaPosition * -movementSpeed);
```

```
    }
```

```
}
```

```
// Checks the player's touch with two fingers
```

```
void UpdateDoubleTouch ()
```

```
{
```

```
    // Make sure there is no one-finger touch currently
```

```
    if (! singleTouch)
```

```
    {
```

```
        if (Input.touchCount == 2)
```

```
        {
```

```
            doubleTouch = true;
```

```
        }
```

```
    else
```

```
    {
```

```
        doubleTouch = false;
```

```
    }
```

```
}
```

```
}
```

```
// Updates zoom and zoom using two fingers on the screen
```

```
void UpdateZooming ()
```

```
{
```

```

if (doubleTouch)
{
    // On the screen A, B positions the two fingers named
    // in both the current frame 2 and the previous frame 1
    Vector2 posA1, posA2, posB1, posB2;
    Touch a, b;
    a = Input.touches [0];
    b = Input.touches [1];

    posA2 = a.position;
    posB2 = b.position;

    posA1 = a.position - a.deltaPosition;
    posB1 = b.position - b.deltaPosition;

    // Make sure the distance between the fingers has increased or decreased since
    // previous frame
    float currentDist = Vector2.Distance (posA2, posB2);
    float prevDist = Vector2.Distance (posA1, posB1);

    // Subtract the previous distance from the current will give us
    // Correct reference to zoom or deportation
    camControl.Zoom ((currentDist - prevDist) * zoomingSpeed);
}
}
}

```

How this program works depends on four main steps:

- Check the touch screen with one finger.

- **Then check the camera move using finger.**
- **Then check the touch screen with two fingers.**
- **Finally make sure to perform rounding and deportation using two fingers.**

Reading the touch screen input is as follows: First we recognize the number of touches on the screen by the variable `Input.touchCount`, if the number of touches is equal to one, this means that the player put a finger on the screen and the possibility of moving the camera in box. Here we use the `singleTouch` and `doubleTouch` variables to store the number of touches we detected. These touches are stored in the `Input.touches` matrix and are an array containing `Touch` elements. This type of variant contains information for each touch on the screen. The first of these information we will deal with is `Touch.phase`, which represents the stage the touch passes through. Once the player puts his finger on the screen, the stage will be `TouchPhase.Began`. The first moment to touch the screen is very important, as it is the moment that we must check that the player has placed his finger on the background or floor element, and therefore determine whether we will allow him to move the camera. We do not want to move the camera if the player has put his finger on the projectile or the exit button, for example.

Verifying that the player has placed his or her hand on the background or ground element is done via the `UpdateSingleTouch ()` function, where we first have to shift the finger position from the screen coordinates to the scene coordinates exactly as we did with the mouse pointer. Next we extract the collision component of this element and then use the Ray Casting. Radiology is a method used by the physics engine to detect the intersection of a straight line with an object in the scene, and we will use it here to draw a very short line from the touch site to a point very close to it, and see if this line intersects with the collision component we extracted. The radiation transmission is performed by the `Physics2D.Raycast ()` function, which takes three variables: the first variable is the starting position of the beam, the second variable is the direction the beam will travel, while the third variable, a numeric value, determines the maximum distance that the beam can travel. Here we notice our use of a very short distance, which is sufficient when the starting position is already inside the

collision component with which the collision is checked. The value returned by this function is of type RaycastHit2D, and contains a variable to store the collision component that the beam has hit.

We then examine whether the component that the beam has hit is the same as the collision component that we extracted from the current element. If this is achieved, we rely on this player's touch on the background or ground and allow it to move the camera by changing the value of singleTouch to true. If the touch is at another stage such as TouchPhase.Ended or TouchPhase.Canceled, this means that the player has lifted his finger off the screen, thus returning the singleTouch value to false. The same will happen if we detect that the number of fingers on the Input.touchCount screen is not equal to one, so we will not allow the player to move the camera.

The UpdateDragging () function is responsible for moving the camera, so it first has to check the value of singleTouch and then move the camera by the amount of playerTouch.deltaPosition multiplied by the speed of movement. Note that here we do not need to store the previous touch location as the offset value comes directly opposite to what was the case when dealing with the mouse pointer.

The UpdateDoubleTouch () function then verifies that the player has two fingers on the screen, in which case it immediately changes the value of doubleTouch to true. Note that two-finger touch has no meaning in the game except rounding and divergence other than one-touch that can be used for more than one purpose. For this reason we do not need to verify the locations of the two fingers, but their presence is sufficient to prevent any input other than rounding and deportation.

Finally, the UpdateZooming () function makes sure that there are two fingers on the screen via the doubleTouch variable and therefore calculates 4 locations as follows:

posA1: location of the first finger during the previous frame.

posA2: location of the first finger through the current frame.

posB1: location of the second finger during the previous frame.

posB2: location of the second finger through the current frame.

Note that we subtracted the displacement from the current finger positions so that we get the finger positions in the previous frame, because we did not store these positions at all. We then calculate the distance between the two fingers in the prevDist frame and the currentDist frame. Remember that the Zoom () function in the CameraControl applet rounds if we give it a negative value and dimension if we give it a positive value. To do this we subtract the previous distance from the current distance and multiply the result by zooming and spacing. This way we ensure a positive value if the player moves his fingers away from each other, which leads to rounding and a negative value if the player closes his fingers to each other which leads to deportation, and this behavior is of course typical for users of smartphones and tablets.

The second applet that we will discuss about receiving touchscreen input is the projectile launcher. Remember that we have added an applet called LauncherMouseInput on a slingshot template. The new applet is called LauncherTouchInput and we will add it to the same template in order to enable the player to launch projectiles by touch. The following narrative illustrates this applet

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class LauncherTouchInput: MonoBehaviour {
```

```
// Reference for Ballistics Launch Applet
```

```
private Launcher launcher;
```

```
// Called once at startup
```

```

void Start () {
    // Destroy applet read mouse input if phone operating system is detected
    if (Application.platform == RuntimePlatform.Android)
    {
        LauncherMouseInput mouseInput = GetComponent <LauncherMouseInput> ();
        Destroy (mouseInput);
    }

    launcher = GetComponent <Launcher> ();
}

```

// Called once when rendering each frame

```

void Update () {
    UpdateTouchStart ();
    UpdateDragging ();
    UpdateRelease ();
}

```

// Checks whether the player has put one finger

// On the current projectile

```

void UpdateTouchStart ()
{
    Projectile currentProj = launcher.currentProjectile;
    if (currentProj == null)
    {
        // There's nothing to do
        return;
    }

    if (Input.touchCount == 1)
    {
        Touch playerTouch = Input.touches [0];

```

```

if (playerTouch.phase == TouchPhase.Began)
{
    // The player just touched the screen
    // Verify that the touch operation is within the projectile boundaries
    Vector2 touchPos = Camera.main.ScreenToWorldPoint (playerTouch.position);

    // Extract the collision component of the current projectile
    Collider2D projectileCollider = currentProj.GetComponent <Collider2D> ();

    if (projectileCollider.bounds.Contains (touchPos))
    {
        // The touch operation is within the projectile boundaries so it must be grasped
        launcher.HoldProjectile ();
    }
}

}

// Checks the player's finger move on the screen while holding the projectile
void UpdateDragging ()
{
    if (Input.touchCount == 1)
    {
        Vector2 touchWorldPos = Camera.main.ScreenToWorldPoint (Input.touches [0] .position);
        launcher.DragProjectile (touchWorldPos);
    }
}

// Checks the player raised his finger off the screen
void UpdateRelease ()
{
    if (Input.touchCount == 1)

```

```

{
    Touch playerTouch = Input.touches [0];
    if (playerTouch.phase == TouchPhase.Ended ||
        playerTouch.phase == TouchPhase.Canceled)
    {
        launcher.ReleaseProjectile ();
    }
}
}
}

```

This program performs three steps in each update:

The first step is through the `UpdateTouchStart ()` function, which verifies that the player has just touched the screen with one finger. In this case, the function calculates the location of the touch in the scene space and then checks whether that location is within the limits of the collision component of the current projectile on the launch slingshot. If this touch is actually within the bounds of the projectile, it means that the player wants to hold it, so the function holds the projectile by calling `HoldProjectile ()` from the Launcher.

The second step through `UpdateDragging ()` involves having one finger on the screen, calculating the location of the finger in the scene space and asking the Launcher to move the current projectile to that location. Keep in mind that other things like making sure that you have a projectile or if it is held or launched is done by the Launcher itself so you don't need to check it here, and you may remember that we also didn't check it in the mouse input reading applet.

Finally, the `UpdateRelease ()` function verifies that the player has lifted one finger off the screen, by checking the `playerTouch.phase` touch stage if it is equal to `Ended` or `Canceled`, which is the expected state when the finger is raised. When you confirm this, you call the `ReleaseProjectile ()` function.

The last applet that reads the touchscreen input is the applet that performs the projectile's special attack after launch. This attack is done by touching the screen once at any location after launching the projectile. This applet is called `SpecialAttackTouchInput` and should be added to all the projectile templates that

we have created. This applet is described in the following narrative:

```
using UnityEngine;
```

```
using System.Collections;
```

```
public class SpecialAttackTouchInput: MonoBehaviour {
```

```
    // Called once at startup
```

```
    void Start () {
```

```
        // Destroy applet read mouse input if phone operating system is detected
```

```
        if (Application.platform == RuntimePlatform.Android)
```

```
        {
```

```
            SpecialAttackMouseInput mouseInput = GetComponent <SpecialAttackMouseInput> ();
```

```
            Destroy (mouseInput);
```

```
        }
```

```
    }
```

```
    // Called once when rendering each frame
```

```
    void Update () {
```

```
        if (Input.touchCount == 1)
```

```
        {
```

```
            Touch playerTouch = Input.touches [0];
```

```
            if (playerTouch.phase == TouchPhase.Began)
```

```
            {
```

```
                SendMessage ("PerformSpecialAttack");
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

You can see how simple this applet is, as all it does is send the PerformSpecialAttack message if the player touches the screen once.

The game is complete for computers and smartphones

Tips for Designer Games

The role of the Game Designer is how the game works correctly during its design process. It sets goals, rules and procedures, sets the story and gives life, and is also responsible for planning everything that makes the game acceptable. Level Design Whether it is based on architecture or sketching on blueprint, it also coordinates the story (the overall scenario) of the game. Anyone who has dreamed of a distinguished Hollywood job can now do what he wants by writing the story script and releasing it completely. The previous disciplines are now fully divided, each person responsible for one of them, but the successful game

developer must be familiar with the details of each specialty. Don't learn something about everything, but what are things needed to become a game designer? What talents and abilities are required? What is the best way to design a game? This is all we will discover through this book .

The game designer puts himself in the player's place

The role of the game designer is first and foremost to act like the player. The designer should see his game from another angle. This is the angle in which Player sees this game. Although this is easy but unfortunately overlooked, this may result in a big gap in the game. This is a simple error. It is simple to make sure that the graphics, story and programming design are going well, but we overlook the most important feature of the game, which is the excitement that combines all of the above to come up with something special. As a Game Designer, most of your role is to focus on and maintain the User Experience experience and observe the things you will gain from the gameplay, and also not to make technical matters (graphics, effects, programming) overwhelm this aspect. Let the person who cares about his specialty, let the creative director imaginative, let the producer pay attention to the budget, and let the technical director care about the game engine and its problems. Your main task is that when the game is delivered to the player you should be surprised by the game play and simply say: Wow does not let the game in a short time. When we mention the term game play, you must first come to mind what the player will gain through a game and what are the experiences and basics that will be found in the game.

Game Testers

Any game that must contain game testers, they are the people who play the game in full and extract the feed, whether by increasing things .. Or correcting other things, and also there is a different perspective when you see someone else is playing, will generate things were not Calculate it and whose purpose is to

enrich your game. You will notice the player in his actions, what he focuses on, and what things may cause him boredom, everything that he says should be taken and taken into account, it will be your guide to highlight these aspects. If you want to be on the site of the game labs, you have to set the main objectives you will do, and finally you can take all the features of the laboratory to become a designer and laboratory at the same time.

There are many designers who do not include Game Tester in the game design process, perhaps because of tight work time, or does not want the laboratory to remove things that the designer thinks are positive, or they think it will cost them a lot of money, That the testing process comes after marketing and through players across the world but they are actually wasting more time and more cost and also a lot of creativity expected from the laboratory of the game, because the game is not one way to communicate One-way that you do not have to put the game and then Waiting for the money to come !!, not the order To make sure that each part of the functional part of the game is working successfully, it is to find the features that the player likes to have in his or her ideal Game Play game. This will only be discovered by the Game Tester.

The game designer can be likened to being the executive of a particular party. His primary task is to make sure that everything is ready: food, juice, decorations, and the atmosphere. Finally, he opens the doors to receive the guests to see what is inside, and the results are mostly unpredictable or unimaginable. , The game is like a concert, but what kind of party? Will the guests really enjoy this concert or will they try to escape from the party? Will they watch and exchange conversations among themselves and wish that this night will not end?

Inviting your friends to experience your game, knowing the reactions and moments of silence and matching them on the ground is really the best way to understand how your game is going and to become a professional designer. Once you are ready to hear from others and learn from their criticism, this will help your game to grow quickly and positively. The process of designing games is very similar to the life cycle, it is in the development stage, there are no rules and absolute techniques, and as a designer, the option is open to the inventor to create your own ways. This does not mean that you take all the tips of your

friends seriously, they have different tastes, and you should often take into account your rules and future aspirations for your game.

Skills

What do you need to become a game designer? Of course there is not one answer, and there is no single way to success. There are some common advantages and skills in the great game designer. The first is the love of the games themselves. If you do not like it, you will never succeed in this because you will not sit for long hours. Creativity and invention of the game and give it the right.

As an ordinary person who has nothing to do with it, it seems a bit unimportant. Whoever looks at the game labs thinks they are playing and wasting the long time and do not realize that this process is necessary and an integral part of the game design phase. As a game designer you have to be always busy. Your project should be the heart of the project, and do not forget that it will pass you hours of pressure, especially at the end of the project requires patience and waiting for the expected results, based on the former, you must develop certain skills, Vbk games is not the only skill required of you there are other factors :

the ability to communicate with different people

One of the most important features of a game designer is that he has the ability to communicate with everyone you think will give success to your game, so you have to be positive to deal with the degree of flexibility with them as a whole, because you will present the idea of the game in different kinds of ways of thinking, because you will show your game on: , Management, investors, and possibly your friends and family. To confront this, you must be persuasive and clear language interpretation, that is to be interpreted logical logical and clear and simple according to the nature of the thinking of the opposite person, and be a successful provider and exhibitor of the project. It is the only thing whose goal is to provide you with adequate coverage of the subject with the material support of the parties concerned, and you must realize that everyone looks at your game from a different perspective, the donor is different from your friend!

And the skill of communication is not limited to writing and public speaking, but it also means listening and preparing to discuss in a logical and logical manner. Your willingness to listen to the Game Tester and your teammates in Teammate will produce new ideas and new directions. What is your reaction when you hear something you do not like to hear? We know that it is difficult to make some concessions, but here it is different and you have to concede in this matter because it is intended to increase productivity. In fact, most game designers believe that the opinions and ideas they have collected should never be waived. If the new opinion is good it will add ideas from its goal to give creativity to the game. For example, you may see some of the effects of their goal adding useful things to the game that make them beautiful, but Technically, you can not do that because it is expensive on the type of device the game will get on, and maybe the time is not enough to work, but what if The programmer provided a secondary way of the effects you want so that they can be programmed in a short time and at a lower cost? Will you keep your first opinion under these circumstances? In the end, your goal is to find the right idea that can be adapted from all the requirements of the work, you must listen to any new opinion of the purpose of the success of the project.

The skill of working within a team

The game industry is one of the most important areas through which to understand the meaning and usefulness of work within an integrated team, where everyone is responsible for the field is specialized, it is specialized science in the field of computer there will be programming graphics, and there will be the design, And to lay the foundations of artificial intelligence and who will manufacture the characters and move them, and there will support the game and ensure the physical aspect of them, and there will be marketed, and there will be handed over to the hand of the hand! There is a series of disciplines, all of which are one body complementing each other.

And because you are a game designer, you will deal with most of the previous disciplines and you will understand that each character has a different view from the previous one and a way of thinking fits his perspective. When you talk to a programmer in a case, you will find a difference of opinion between him and the

producer who cares about securing the material aspect. To be the link between all characters and the mediator between them to make sure that this group will all play the game as expected to the end.

Able to handle matters when necessary

Often, when you have a timeline, you will have to make a decision that sometimes negatively affects the quality of the game as a whole or the process of development, and will generate future problems continue to appear one by one until the game as a whole set of problems can not be solved! This is a serious issue especially in the final stages of the design of the game. When the time runs out, some problems will arise. You will be able to solve it and you will have to delete some Stages and some elements to keep items that are designed properly, it's a real disaster, but ultimately help us to understand why contain a lot of games on the contradictions that often deadens the game before the arrival of the market (Dead On Arrival (D.O.A shortly.

Capable of creativity

Creativity factor is the most difficult factors in comparison, but in the end you need this factor in your game to get a beautiful idea, and each person innovated in a way different from the other, we can not say that this is a creative person and this is not, some people come with many ideas and in fact did not search And some take the same idea with a group to get things unified among everyone. Some of them are looking for new experiences to develop his imagination, and some of them do not do so, The game designer is the one who takes advantage of a His happy and bad dreams, his illusions and his fears are presented as an interactive experience for others.

One of the biggest and most famous designers of Nintendo's Shigeru Miyamoto says:

I always look at my childhood and my inspirational hobbies and hobbies when I " was a child. I went to a park and found a lake that was nice and amazing for me to find. When I wandered through the countries without a map, I tried to break

my own way, found amazing things and realized how happy I was with such an adventure."

When I was a child, I went hiking and found a lake. It was quite a surprise for "me to stumble upon it. When I traveled around the country without a map, trying to find my way, stumbling on amazing things as I went, I realized how it felt to go on an adventure like this "

In many of his games he took advantage of what he dreamed of and saw in his childhood.

Think now about your childhood and the experiences you gained. Will you find an idea of your memories suitable for the game? One of the best inspirations for new ideas for game designers is to remember the unique experiences they have experienced in their lives, especially since their childhood has seen the longest period of play. The way children interact is done by playing. This makes toys the most important social factor in children. If you try to remember your childhood and look at what you have enjoyed, you will discover a new idea that no one else has ever discovered. It is also creative ways to put two things are contradictory and have nothing to do with some such as Shakespeare and Omar Mukhtar and find what are the factors common among them !!

