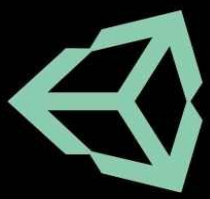
MEM LNC

LEARNING C# BY DEVELOPING GAMES WITH UNITY

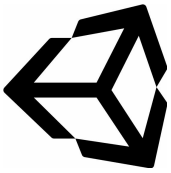
A BEGINNER'S GUIDE TO MAKING YOUR FIRST VIDEO GAME

BUILD, CUSTOMIZE, AND OPTIMIZE PROFESSIONAL GAMES USING UNITY



Claudia Alves

Learning C# by Developing Games with Unity



Build, customize, and optimize professional games using Unity

Claudia Alves

For information contact :
(alabamamond@gmail.com, memlnc)
http://www.memlnc.com

First Edition: 2020.11

Learning C# by Developing Games with Unity

"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.

I N T R O D U C T I O N

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 MonoDevelopor Visual Studio 2019 Communityfor 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.

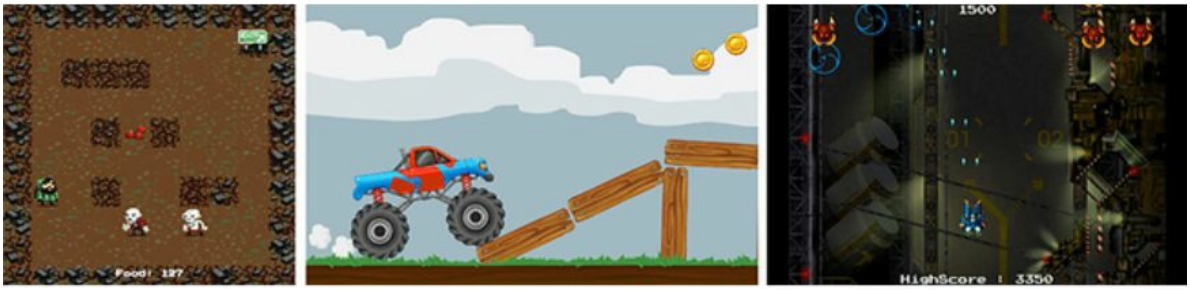
Full 3D

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 andaround 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 in3Dmode.

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 in3Dmode, because even though there is noperspective, you will still be working with 3D models and assets. You'll need to switch yourcameraandscene viewtoOrthographicthough.(scenes above fromSynty StudiosandBITGEM)

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 thegameplayto two dimensions. For example, the camera may show a “side scrolling view” and the player can only move intwo dimensions, but 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 isalsosometimes 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.

Fewer features, more promise

If you play Off-Road Velociraptor Safari and some of the other games on the Blurst site, you'll get a better sense of what you can do with Unity without a team of experienced Korean MMO developers. The game has 3D models, physics (code that controls how things move around somewhat realistically), collisions (code that detects when things hit each other), music, and sound effects. Just like Fusion Fall, the game can be played in the browser with the Unity Web Player plugin. Flashbang Studios also sells downloadable versions of its games, demonstrating that Unity can produce standalone executable game files too.

Maybe we should build Off-Road Velociraptor Safari?

Right then! We can't create Fusion Fall just yet, but we can surely create a tiny game like Off-Road Velociraptor Safari, right? Well... no. Again, this book isn't about crushing your game development dreams. But the fact remains that Off-Road Velociraptor Safari took five supremely talented and experienced guys eight weeks to build on full-time hours, and they've been tweaking and improving it ever since. Even a game like this, which may seem quite small in comparison to full-blown MMO like Fusion Fall, is a daunting challenge for a solo developer. Put it in a jar up on the shelf, and let's take a look at something you'll have more success with.

I bent my Wooglie

Wooglie.com is a Unity game portal hosted by M2H Game Studio in the Netherlands. One glance at the front page will tell you that it's a far different portal than Blurst.com. Many of the Wooglie games are rough around the edges, and lack the sophistication and the slick professional sheen of the games on Blurst. But here is where we'll make our start with Unity. This is exactly where you need to begin as a new game developer, or as someone approaching a new piece of technology like Unity.

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.Select3Dor2Dfor your project type. The default is 3D, coloured red to show it is selected. (The 2 Doption sets the Unity editor to display its 2D features, and the3Doption 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 selectAsset 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 onDone.

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:

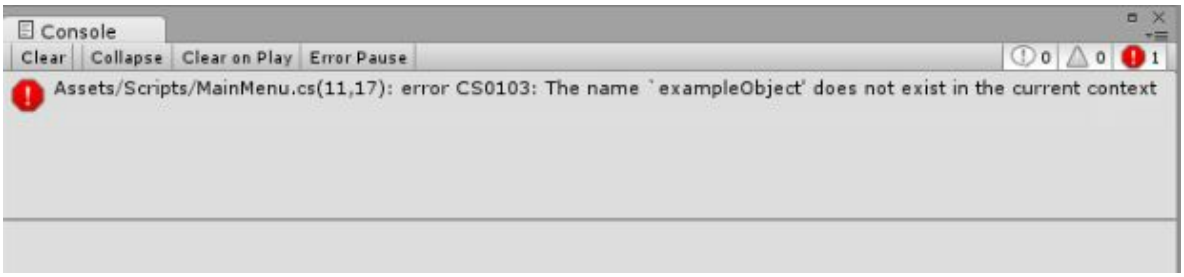
Scene Window: It you use to build the game scene, add different objects to it and distribute it in 2D space. Initially this window contains only one object which is the camera.

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.

Console : The console tab is used to display error and warning messages. If you have made programming errors, for example, Unity displays them here. But you, as a developer, can also issue reports about this.



Messages about errors that must be rectified (errors) are shown in red in the window. In contrast, warnings, i.e. non-critical errors, appear yellow. Normal information messages are displayed in white. You can also filter the messages according to these ratings using the three symbols on the right. In addition to the three filter options, the upper console bar also has a few buttons. These have the following meanings:

-Clear deletes all entries.

-Collapse suppresses duplicate messages.

-Clear on Play deletes all entries at the start of the game via the play button.

-Error Pause interrupts the game as soon as an entry with Debug.LogError ("error text") is executed.

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 1

C# and Unity

Even if Unity already supplies ready-made components for many standard tasks (for example for the areas of audio, physics, particle effects or even rendering), the actual, individual game logic must still be programmed by yourself. In Unity, this is done with scripts, individual text files that are translated by a program (called a compiler) into commands that are understandable to computers when the game is created. Scripts usually behave like components in Unity and are attached to GameObjects accordingly. As we are working with the C # programming language in this book, I would first like to introduce you to this language and how it is used in Unity. If you have already dealt with C #, I would still recommend that you read this chapter, because it also explains some special features that apply specifically in Unity.

The language C

C # was developed by Microsoft and belongs to the so-called object-oriented programming languages. It was developed to develop applications using the .NET Frame-work, a platform that provides a wide range of class libraries, programming interfaces and utilities. However, Unity uses the language in combination with the Mono Framework, an open source variant of the .NET Framework, which means that C # applications can also be operated on non-Microsoft systems (also known as a mono project) the language is very extensive, I will not introduce all of the possibilities of C # to you in the following and focus especially on the topics that are interesting for you as a Unity programmer. If you want more information about the C # language, you should find what you are looking for using the well-known search engines. There are also many good books, such as: For example, the Visual C # books by Walter Doberenz and Thomas Profitus, which deal with language in all its facets.

If you don't understand some of the topics right away, don't despair. You will surely have one or the other eye-opening effect if you have also worked through the other book chapters and read through these passages again later. Game will be programming. The script is used to manage the strength of life and is used by both the player and the opponents.

HealthController

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

```
public class HealthController : MonoBehaviour {
public float health = 5;
private bool isDead = false;
void ApplyDamage(float damage) {
        health -= damage;
         if(health <= 0 && !isDead) {
                isDead = true;
           Dying();
         }
        else {
      Damaging();
            }
        }
public virtual void Damaging() {
}
public virtual void Dying () {
}

}
```

syntax

First of all, a few words about basic programming: Program code (also called source code) consists of lines of code that are processed one after the other. So that the computer also understands these lines, they are translated into machine language with a program called compiler, the so-called compiling. So that the compiler also knows when the end of a single line of code is reached, a semicolon is added to the end in C #; written. Other important characters in C # are the curly braces {}. These are used to identify coherent blocks of code.

Comments

In addition to the actual code, you may also want to add annotations to your code that are not meant to be run by the computer. Such lines that are not to be executed are also called comments. Mark a comment with a double slash //. Anything behind it until the next line break is considered non-executable code. What is in this line before the double slash is considered executable!

One-line comment in number1; // This is a command line that is also compiled. Number1 = 2; if you want to mark multiple lines as comments, you can do this by writing / * at the beginning and * / at the end:

Multi-line comment

/ * This is old Codeint life

Points;

lifePoints = 2;

*** /**

int lifePoints = 2;

// This is the new code

Usually, comments from the development environment are highlighted in color, e.g. B. in green or red.

variables

First of all, variables are nothing more than placeholders for storing values. The name of a variable must always be unique and is assigned by you. Even if this doesn't matter at first, you have to make sure that it doesn't have any special characters, umlauts or spaces. In addition, variable names cannot begin with a number.

Naming conventions

To make program code easier to read, there are a few conventions that you should stick to. Firstly, the variable names should describe the content of the variables. It is best to use English terms, e.g. B. speed . In addition, variable names should always begin with a lowercase letter. For names that consist of several words, these should be separated from each other by capital letters, e.g. B. enemySpeed . This spelling is also called camel case.

Data types

So that the compiler also knows what values can be stored in a variable (numbers, letters ...), you have to tell the variable what type of data it is. This is called a variable declaration. Frequently used data types are in Unity game development: ƒ string for text ƒ int for integer ƒ float for floating point number ƒ bool for Boolean ƒ enum for enumeration In C # the declaration of a variable begins with the data type followed by the name of the variable. At the end of course follows the semicolon:

Variable declaration

int lifePoints;

float height;

string name;

To assign values to these variables, the equal sign is used in C #:

Assign values

lifePoints = 2;

name = "Carsten";

height = 1.5F;

As you can see, in addition to the text itself, I use quotes to tell the compiler where the assigned text begins and where it ends. When I assign commas (I need the float data type), I write a period instead of the comma. I also write an F after the value for float variables so that the compiler knows that the value is actually a floating point number of the float type. Alternatively, there is also the double type in C #, which, however, covers a larger number range than float and therefore does not "fit" into a float variable:

Difference float and double

float gravity;

gravity = 9.81F;

double gravity;

double = 9.81;

In Unity, however, float is more common than double, which is why I will only speak of float values in the future. To shorten the code above, you can also assign a value to the variable when you declare it (also called initialization):

Variable declaration with initialization

float gravity = 9.81F;

Another type that is used a lot in game programming is the data type Boolean (bool). This type can only assume two states: TRUE and FALSE, meaning true or false. A typical example of this is a checkbox, i.e. a checkmark in the GUI, for example to activate or deactivate a function. Depending on the state of this checkbox, the Boolean variable that stores the value in the program code has the value TRUE or FALSE.

Boolean variable

bool isAttacking = false;

Keyword var

In addition to specifying a data type with int, string, etc., there is also the option in C # to define a variable with var . In this case, the data type of this variable is only determined when the first value is assigned. Then the compiler decides which data type is the right one and defines it. Any future assignment that would require a different type then leads to an error.

var lifePoint; lifePoint = 5;

Data fields / array

If you need several variables of one data type, you can create them using an array definition. An array is not a data type in itself, but rather a variation of a data type.

Create arrays

You define an array with square brackets after the actual data type:

int [] speedLimits;

In contrast to a normal variable, the array does not yet exist through the sole definition of the variable. You have to do this again with the keyword new . This process is also called instantiation. In addition, when you instantiate the array, you must also specify how large it should be, i.e. how many integer variables the array should consist of. It looks like this:

speedLimits = new int [5];

Or a little shorter:

int [] speedLimits = new int [5];

When instantiating, you can also give each element of the array a start value. This can look like this:

int [] speedLimits = new int [5] {30,50,80,100,120};

It is important that you actually pass as many values in the curly brackets as there are elements in the array.

Access to an array element

If you now want to access a certain element of this array, you have to select the correct item using an index number. For inexperienced users, this will mean a change, because the first element of an array has the index 0, not 1! An array with five elements has the highest index 4!

int [] speedLimits = new int [5] {30,50,80,100,120};

int [0] = 20; // previously was 30

int [4] = 140; // previously was 120

Determine the number of all array items

Arrays may also be created automatically. If you want to know how many items the array now consists of, you can determine this using the Length property :

int len = speedLimits.Length;

Multidimensional arrays

You can also create an array with multiple dimensions, imagine you want to use an array to represent a square grid field (a chess board, for example). Then it is very confusing to create an array with 64 elements. It is easier to create an array with two dimensions, where the first index represents the X axis (in chess it would be the letters A to H) and the second the Y axis. Such a definition looks like this in C #:

[,] tile = new int [8,8];

Again, keep in mind that each index in the array begins with a 0. If you now encrypt the figures with numbers (0 means no figure, 1 represents a pawn, 2 is a tower), you can easily save the positions of the figures in this array:

[0.0] = 2; // There is a tower on A1.

tile [0,1] = 1; // There is a farmer on A2.

tile [1,1] = 1; // There is a farmer on B2.

tile [0.2] = 0; // There is no figure on A3.

Constants

There are always values that should not or should not be changed. You can define these as constants, which means that they can no longer be changed. To do this, you must write the keyword const before the data type . It is also necessary to initialize constants when declaring:

const float gravity = 9.81F;

Enumeration

You can use enumerations to create legible lists of constants. What sounds a bit cryptic is a very practical thing. Imagine you want to save the state of a figure. This has three states: standing, walking, jumping. In order to query the current state of the figure, you would like to save it in a variable. To do this, you first define an enumeration (short enum) that has these three values. We just want to state this enumeration:

enum state {idle, walk, jump}

You can then create a variable of this enumeration type and assign one of the three values to it.

myState State; myState = State.Idle;

Type conversion

It is also possible to convert a variable of one type into another type. The prerequisite is, of course, that it also fits in terms of content. An integer variable can always be converted into a float variable. The reverse way only works if the float number has no comma value. When converting a type, simply enter the target type in parentheses before the value to be converted:

int lifePoints; string lifePointsText = "2";

lifePoints = (int) lifePointsText;

Converting numbers to a string is even easier. They inherently have a method / function (I will explain what this is in a moment) that takes on this task. To do this, enter a point after the number variable and then the text ToString () . As soon as you tap the point, a selection should appear in MonoDevelop that makes this command available.

lifePointsText = lifePoints.ToString ();

This function is very useful, because in this way the integer variable can be treated like a string:

lifePointsText = "You own" + lifePoints.ToString () + "Life Points.";

As you can see, you can append strings in C # using the + sign. This procedure is also known as string concatenation and is very often used for text output.

Ramifications

Now that you have learned all the important previous knowledge for programming, we can finally start with the actual programming. Because this is not just about defining variables and filling them with values, rather it is about making decisions based on these values, which is where so-called ramifications come into play. You decide which code blocks are to be executed next on the basis of defined conditions.

if statements

The simplest branch is the so-called if statement. It works like a switch that can execute a code depending on a condition. In addition, an alternative code can be specified that is to be executed if the condition is not met, so that the entire code then behaves as follows: "If the condition applies, then do this, otherwise do it." An if statement is introduced with the signal word if , followed by the condition enclosed in parentheses. The actual code then follows in curly brackets. The optional alternative code is then with the word else initiated

```
if (lifePoints == 2) {

   message = "You have 2 lives";

    }

   else {

     message = "I don't know how many lives you have." + "But there are definitely no 2.";

  }
```

If the code block or else branch consists of only one line, the curly brackets can also be omitted.

```
if (lifePoints == 2) message = "You have 2 lives";
```

switch statement

With an if statement, you always have a maximum of two possible sections of code that can be executed. Although you can represent several states by nesting the if statement, this becomes very confusing at some point. The switch statement can help here. With this you can check any number of states and then execute the code that applies to the respective case.

```
int damage = 1;

string title;

switch (damage) {

case 0:

message = "Beside!";

case 1:

message = "Oh, just a scratch!";

break;

case 2:

message = "Damn it hurts!";
```

```
break;

default:

message = "Arrrgh!";

break;

}
```

It is important for the switch statement to end each case section with the break command . This ensures that the switch statement is exited after the code has been executed. If none of the defined case conditions apply, you can finally use the default signal word to define a standard case that is selected as soon as none of the previous cases matched. In Listing 3.34 this would always be the case if the damage was higher than two (we simply ignore a negative damage value).

grind

In addition to the branches, loops are another important element in programming. These ensure that certain code sections are repeated until a specified termination condition is met.

for loop

The classic loop is the for loop. It is used when it is known how often a code area should be repeated. With this type of loop, a count variable (loop counter) is assigned a start value and incremented as long as a specified running condition is fulfilled. In C # the first expression is the loop counter with the start value, the second represents the running condition, the third expression describes the counter increment.

```
int counter = 0;

for (counter = 0; counter <10; counter ++) {

//Code...

}
```

In the example above, the counter counter was set to the start value 0 and runs through the following code ten times. This is due to the counter increase ++ , which increases the value of counter by one after each run.

Negative step size

The next example shows a step size of –2 and runs through the code as long as the counter is greater than 0. In addition, here the loop counter is only defined in the for loop, which is also quite common in practice:

```
for (int counter = 10; counter> 0; counter- = 2)

{

//Code...

}
```

Break

A for loop does not always have to be run through to the end. It can also be canceled using the break keyword :

```
int [] speedLimits = new int [5] {30,50,80,100,120};

int currentSpeed = 100;

int currentGear;

   for (int counter = 0; counter <5; Counter ++) {

     //Code...

                if (speedLimits [counter]> = currentSpeed)

          {

            currentGear = counter +1; break;

                        }

       }
```

Foreach loop

The foreach loop is used to run through arrays. Here an additional run variable is used, which takes over the value of the current element of the array. Since this is a copy of the original, the contents of the array cannot be changed in this way. It is important to know that you only define the run variable in the loop.

```
foreach (int currentLimit in speedLimits) {

        if (currentLimit == 80) {

                // code ...}

    }
```

while loop

The while loop continues until an abort condition is met. The termination condition is queried at the beginning of the loop, so that the loop may not even be able to be executed, namely if the termination condition is met from the beginning.

```
int enemyIndex = 0;

while (enemyIndex <5) {

    if (enemyIndex == 3)

            break; //Code...

     enemyIndex ++;



}
```

This loop can also be simply ended with the break command without fulfilling the actual termination condition. In addition, there is also the continue command . This ensures that the loop behaves as if it had already arrived at the end of the loop code when the continue command is reached, and therefore begins with the next iteration of the loop.

do loop

In contrast to the while loop, where the termination condition is checked at the beginning, the condition in the do loop is only checked at the end. The result is that the loop makes at least one pass, regardless of whether the condition at the end of the loop was met from the beginning or not

```
int enemyIndex = 0;

        do {

          //Code...

          enemyIndex ++;

              }

  while (enemyIndex> 5);
```

With the do loop it is also possible to leave the loop with the break command .

Classes

As already mentioned at the beginning, C # is a so-called object-oriented programming. The aim is to bundle and encapsulate coherent functionalities and values in so-called objects (which are nothing more than variables). B. Everything that concerns the player is in an object called player . Everything that affects the enemy is saved by an object called enemy . What do we do if we want to have two or more opponents, not just one opponent? Do we have to program a completely new object for every opponent? No, of course not, because this is where classes come in. A class is a template for objects. It describes how objects in this class “look” and how they should work. A class begins with the word class , followed by the name of the class. To better distinguish it from normal variables, this begins with a capital letter in C #. Then the code of the class follows, which is enclosed in curly brackets:

```
class enemy {

    int health;

        }
```

In Unity, a script represents exactly one class. The name of the script must have the same name as the class in the script. If you now want to create an object from a script or a class, this is very easy in Unity. Since scripts are normally considered to be components (more on this in the chapter "Script programming"), you can easily drag them onto a respective GameObject, which automatically creates an object of this class. This process is also called instantiation. The result, i.e. the object, is therefore often referred to as an instance of this class.

Assign components by code

Instead of dragging such a component script onto a GameObject, you can alternatively assign a script or an object of your class to a GameObject using program code. For this there is the command AddComponent .

gameObject.AddComponent <MyScriptName> ();

Instantiation of non-components

Instances of non-components, such as For example, GameObjects or script classes that do not inherit from the MonoBehaviour class (more on this in the " Scripting " chapter) are usually generated using the typical C # path. To do this, first declare a variable of the type of the class. Next comes the instantiation of the object. This creates the new object, which you do with the keyword new and the class name.

Enemy enemy;

enemy = new enemy ();

The two lines above can also be made shorter, as follows:

Enemy enemy = new Enemy ();

The two brackets after new and the class name are important and must always be written, since this is not the class itself, but the constructor of this class, a method that is carried out when the object is created. You can find out more about the topic of constructor e.g. B. in the previously mentioned books by Wal-ter Doberenz and Thomas Profitus.

Methods / functions

Algorithms, i.e. lines of code that do not contain variable declarations or similar. within a class can only be written in methods. Methods are code units that are addressed by calling their method name and can have both transfer parameters and return values. Note that the term “method” is not normally used in JavaScript, instead the term “function” is used there . In both cases, however, the same is meant. And because, as already mentioned, Unity does not only support C # as a language, Unity developers often use both terms. For methods / functions there is a similar naming convention as for variables, except that they begin with a capital letter, e.g. B. SetPosition . A method definition has two important parts:

Transfer parameters, they are defined after the method name within parentheses with the variable type. If the method has no transfer parameters, the brackets are still written. Please note that when a variable is passed, a copy is created, which is then used in the method. So it is not the same variable that is passed to the method and that is used in the method itself.

Return value, it is defined in front of the method name in the form of a type definition. The return value is returned within the method with the keyword return . If the method should not have a return value, the key word void is written in front of the method. The following method checks whether the contents of two transferred string variables are the same.

bool IsEqual (string stringA, string stringB) {

bool result = false;

```
        if (stringA.ToLower () == stringB.ToLower ())

            result = true;

        return result;

    }
```

Note that the example also accesses methods, ToLower () , which provides each string variable. This returns the content of the variable stringB in lower case letters, so that "Test" becomes a "test". The following line now compares whether the two strings have the same letters and then gives a TRUE or a FALSE accordingly via the result variable.

```
//Code...

bool equal;

string myName = "Test";

string colliderName = "TEST"

equal = IsEqual (myName, colliderName); // Returns a TRUE
```

//Code...

Don't repeat yourself

An important task of methods is to avoid repeating sections of code. Multiple-use code is separated into its own methods, which means that the code only has to be maintained at one point. Do you need z. For example, if you compare two positions in a method, but you have to calculate them in the same way beforehand, you could split the logic of the calculation into one method. You then use the transfer parameters of the method to give it the initial values for the calculation.

Value types and reference types

Even if I don't want to delve too deeply into the subject, you should still know that there are two different data types of variables, the differences of which are particularly noticeable when you transfer them to methods.

Value types are data types that contain the value. Examples of this are the simple types int, bool or enumerations. If you pass a value type to a method, the value is also copied. In this case, you can change the variable within the method as desired, without this affecting the externally transferred variable.

Reference types are data types that do not contain the value, but only a pointer to another memory location (i.e. the memory address). The actual content of the variables is only stored in this other storage location. If you now pass a reference type of a method, it is not the content that is copied here, but only the pointer. Changing the variable data within the method also changes the data of the object that was transferred to the method from outside. A typical example of reference types are instances of classes.

To finally confuse you, I want to tell you that string variables are also reference types. But, and that's the good news, strings have a value semantics. This means that strings behave more or less like value types. Therefore, you can usually use strings in exactly the same way as int variables or Booleans. For the previous example method, this means that you can change the value of stringA within the method without affecting the passed value of the myName variable .

Overloaded methods

In C # there is also the possibility to define two methods in a class with the same name. This works if the transfer parameters differ. This technique is also known as overloading a method.

static bool IsEqual (string stringA, string stringB)

{// code ...}

static bool IsEqual (string stringA, string stringB, bool ignoreCase)

{// code ...}

Local and global variables

Depending on where you declare your own variables in classes, they only exist in one function or in the entire class. ƒ Local variables are declared within a method and only exist in this one method. ƒ Global variables are declared outside of methods and apply to the entire class.

Prevent confusion with this

Due to the different areas of validity, it can happen that a local variable and a global variable have the same name. To avoid misunderstandings, C # offers the keyword this , which refers to your own class instance. If you want to access the local variable in this case , simply write the name of the variable. If you want to access the global variable, write this first , then a period and then the name.

```
int quantity = 0;

  void Add (int quantity) {

     this.quantity + = quantity;

}
```

Access and visibility

Global variables as well as all methods can be expanded with additional access modifiers. These determine whether they are only visible within the class and can therefore also be used, or whether they can also be accessed by another class or instance.

public enables access from anywhere. Public variables are also displayed in Unity in the Inspector.

private limits access to your own class.

protected limits access to your own class and all other classes that inherit from it.

internal limits access to all classes in your own assembly (important for DLLs).

If you do not specify any of these keywords, variables and methods in C # are automatically considered private. By the way, classes can only be awarded public and internal . If the keyword is missing, it is considered internal.

```
public class test {

private int health;

public bool IsEqual (string stringA, string stringB) {

  //Code...
```

```
    }

}
```

Static classes and class members

There are situations where you B. Programming methods or variables that should work independently of a created instance, e.g. B. a counter of all previously created instances of this class. In this case, of course, not every instance may have its own counting mechanism, the results of which may even differ. So-called static methods / variables / are used for such applications. . . or also called general class members. (In contrast, the conventional methods / variables /... Are referred to as instance members.) They are declared with the keyword static and are not accessible via an instance, but directly via the class name. Unity uses this e.g. B. in his math class Mathf.

int myInteger = Mathf.RoundToInt (1.5f);

The declaration of a static method then looks like this:

public static void SetLevelSettings (int levelIndex) {

//Code...

}

The Mathf class goes even further. Here the whole class was declared static, so that the compiler only allows class members. Member is the overall term for all “things” that can be in a class, for example: B. methods, variables, etc.

A static class is then defined as follows:

```
public static class GameSettings {

    public static string playerName = "";

    public static void SetLevelSettings (int levelIndex) {

      //Code...

    }

}
```

Parameter modifier out / ref

Usually, a variable that is passed to a function is copied, as mentioned above. This means that a change in value type variables (or with a value type semantics) within a function has no influence on the variable passed from outside. However, this may be desirable in certain cases by using the parameter modifiers ref and out , which pass the variables (regardless of whether they are value types or reference types) to a method by reference and not by copying the content. Both the transfer parameter in the method must be identified with the modifier and the variable that is transferred to the method.

```
public bool IsEqual (ref myName, colliderName) {

bool result = false;

    if (stringA.ToLower () == stringB.ToLower ()) {

            result = true; stringA = stringB;

      }

      return result;

  }
```

Call a method with the ref modifier

string myName = "Player";

string colliderName = "player";

equal = IsEqual (ref myName, colliderName);

In the above case, myName initially has the value "Player" and colliderName has the value "player". After calling the method, both variables have the value "player". out is used the same way as ref . The difference between these two parameters is as follows:

- ref requests an already initialized variable. The transferred variable must therefore already have a value before it is transferred to the method.
- out does not need an initialized variable. To do this, however, it must be ensured in the method itself that a valid value is assigned in any case.

A great advantage is that you now have the option of returning multiple values from one method. For example, the Unity class Physics uses an out parameter in the Raycast method . The method sends a test beam from a starting point to check whether there are other objects in the direction. The actual return value of the function is only a Boolean and only says whether the virtual test beam has hit an object. If something has been hit, you can get further information about the hit object via the out parameter.

```
RaycastHit hit;

    if (Physics.Raycast (transform.position, Vector3.down, out hit))

        float distanceToGround = hit.distance;
```

Array passing with params

In addition to the above parameter modifiers, there is also the params modifier . This simplifies the passing of arrays to a method, so that you do not necessarily have to define an array of a type when passing one or more values. You can simply pass a comma-separated list of variables.

```
public int DamageAddition (params int [] damageValues) {

        int val = 0;

                foreach (int current in damageValues) {

            val + = current;

                                }

    return val;

}
```

In the method, the transfer parameter damageValues is treated like a normal array of the type int . This is nothing special at first. What is special only becomes apparent when the array is passed to the method.

```
int sum;
```

```
int damage1 = 5;

int damage2 = 3;

sum = DamageAddition (damage1, damage2);

int [] damageArray = new int [2];

damageArray [0] = 5;

damageArray [1] = 3;

sum = DamageAddition (damageArray);
```

Properties and property methods

Imagine that you have an NPC class that has a public variable health . You can of course assign any value to objects of this class, even if it is not allowed that objects have a negative health value. Instead of taking this into account wherever you process this value, it would make sense to ensure from the outset that this cannot happen. For this there are so-called property methods in C #, which can check and change the value as soon as the value is transferred. A property method, called property for short, consists of two sub-methods called get and set accessors.

Get is used to query the value.

Set is used to assign the value.

The actual variable, in our case it is health , is declared private and is therefore only accessible via the property methods. This procedure is also called data encapsulation. The class NPC would then look like this with these accessors:

```
public class NPC {

  private int health;

  public int health {
```

```
    get {

      return health;

    }

    set {

   health = value;

        if (health <0)

     health = 0;

  }

}

}
```

Pay attention to upper and lower case: The property methods are capitalized and are public. The variables are in lower case and are private.

```
NPC npc = new NPC ();

int h = 3;
```

npc.Health = 3; // sets the private variable health to 3

npc.Health - = 5; // subtracts 5 from the current value, so health would actually be -2

h = npc.Health; // Health returns 0

Because the Inspector only displays public variables in Unity, but no property methods, property methods are often not used in Unity.

Inheritance

An important feature of object-oriented programming is the possibility of inheritance. That means nothing more than that as a programmer you can develop a new class that can use another class with all its methods, variables etc. as a basis without having to rewrite the code. This new class is also called the derived class. The class from which it inherits is called the base class. If a class should now inherit from another, a colon is written after the class name, followed by the name of the base class.

public class MyScriptName: MonoBehaviour {

//Code

}

Even if you may not make much use of inheritance yourself, this is a very important issue for you as a Unity programmer. Every class that is created in Unity and is to be used as a component must inherit from the Unity class MonoBehaviour . This includes everything a class needs to be able to be used as a component. You can find out more about this in Chapter 4, “Script Programming”.

Base class and derived classes

If a class is to inherit from another in C #, a colon must be written after the class name, followed by the name of the base class. Suppose we have a class person with a public variable name and a private variable points .

```
public class person {

public string name = "";

private int points = 0;

}
```

The derived class NPC , which should have an additional variable health , would then look like this:

```
public class NPC: person {

   public int health;

}
```

Now you can create an object of the class NPC and use both public variables:

```
NPC npc = new NPC ();

npc.name = "Legolas";

npc.health = 10;
```

Inheritance and visibility

Even if the NPC class inherits from the base class Person in the example above , the rules of visibility that I explained earlier still apply. You cannot access a private variable or a private method of the base class in a derived class. If you still want to allow access, you can define these members with the keyword protected instead of private .

```
public class person {

  public string name = "";

  protected int points = 0;

}
```

This means that the variable points is still not visible from the outside, but can be accessed in the derived class NPC .

```
public class NPC: person {

   public int health;
```

```
    public void DoSomething () {

     points = 100;

    }

}
```

Override inherited method

Imagine that you have a basic class person who, in addition to various variables and methods, also has a method called Shoot . To display this functionality, a short text should first be output in the console (console tab). You can do this with the Debug.Log command . While the class Person is provided with a standard text "Peng!", The class NPC should now say something else. For this purpose in C # there is the possibility to overwrite methods. To do this, the method to be overwritten must first be declared as overwritable in the base class. This is done with the keyword virtual .

```
public class person {

   public virtual void Shoot () {

     Debug.Log ("Peng!");

                          }

}
```

The derived class can (but does not have to) override this method. The key word here is override .

```
public class NPC: person {

   public override void Shoot () {

  Debug.Log ("Tie!");

}

}
```

Access to the base class

If you have now overwritten a method of the base class in the derived class, but now want to access the method of the base class, the question naturally arises how to do it now. This is made possible by the base keyword . You access methods, variables etc. of the base class via base .

```
public class NPC: person {

public override void shoot ()

        {

            base.Shoot ();

                    Debug.Log ("Tie!");

        }

}
```

Seal classes

Another way of inheritance is by sealing. This means preventing the inheritance ability of a class. In other words, if you prepend the sealed keyword in the class definition , no other class can inherit from it.

public sealed class NPC: person {// code ...}

Polymorphism

Object-oriented programming languages offer another useful skill called polymorphism. You can treat all objects / instances that have a common base class in the same way. The following method triggers the Shoot function from a passed object . The script does not care whether the object is actually of the Person type or a class that was derived from Person (e.g. NPC from the previous example).

void LetsShoot (Person person) {

person.Shoot ();

}

Delivery of various objects to LetsShoot

Person person = new person ();

NPC npc = new NPC ();

LetsShoot (person);

LetsShoot (npc);

Interfaces

Another feature of the object orientation of C # are interfaces. With these you define common features of different classes, via which you can address them again. In this way you can treat different classes equally, which can be very useful in certain situations. There are general interfaces that have already been predefined by Microsoft, which are also important for certain functionalities, and you can define your own interfaces.

Define interface

To define an interface, the keyword is interface used. By convention, the name of an interface always begins with an I , so in the example above it would be IShootable . The rest of the interface is similar to a class, with the difference that the methods are not programmed. Because this only happens in the classes that implement this interface. The IShootable interface could then look like this:

public interface IShootable {

public void shoot ();

}

An important difference between classes and interfaces concerns access modifiers. While everything in classes is considered private that has no access modifier, interfaces are considered internal. After all, an interface is also used to describe the external impact of a class. All other access modifiers, apart from public and internal , are even prohibited for interfaces.

Implement interfaces

If a class is to implement an interface, this is done in exactly the same way as inheriting a class, i.e. with a colon. If an inheritance is inherited from a class as well as one (or more) interfaces are implemented, these are listed separated by commas after the name of the base class. The implementation of the IShootable interface could then look like this. Please note above all that the script has the Shoot method , which is enforced by the implementation of the interface.

```
using UnityEngine;

using System.Collections;

    public class gun: MonoBehaviour, IShootable {

          private WeaponController weaponController;

            void Awake () {

            weaponController = gameObject.GetComponent
<WeaponController> ();

                }

       public void UseMe () {

        weaponController.Switch (this);
```

```
        }

    public void Shoot () {

    Debug.Log ("Peng!");

}

}
```

The individual areas of the script are not so important at the moment and will be dealt with in more detail in the other chapters. Nevertheless, I would like to anticipate what is actually happening here. As soon as the Awake method is called, the script component WeaponController (whose code will be treated in a moment) is searched for and assigned to a private variable. If the UseMe method is now called, Gun's own script instance is transferred to the switch method of the WeaponController . Finally, the Shoot method only writes the text "Peng!" Into the Unity console for demonstration purposes. Later the code would be programmed here, which would then e.g. B. fires a projectile.

Support from MonoDevelop

The programming environment Mono-Develop helps you to implement an interface. If you have written the implementation IShootable behind MonoBehaviour in the example above , move your mouse pointer to IShootable and press the right mouse button. In the context menu that appears, go to Refactor / Implement implicit . Then all properties, methods, etc. are automatically added to your class. All you have to do now is fill it with life. Of course, you can then remove the exception (error message) that is inserted by default.

Access via an interface

To access an interface, you can use an interface like a variable type. In the following script, the interface is therefore used for the parameter definition of the Switch method as well as for the public variable definition. This means that any class can be transferred to Switch as long as it has the IShootable interface and thus also the shoot function.

```
using UnityEngine;

using System.Collections;


  public class WeaponController: MonoBehaviour {

        public IShootable weapon;

        public void switch (IShootable newWeapon) {

          weapon = newWeapon;

             }

public void ShootCurrentWeapon () {
```

```
    weapon.Shoot ();

}

}
```

Namespaces

If we develop the demo game at the end of this book, you will find that quite a few classes are created during a Unity project. In addition, the framework that Unity uses for programming in the background also provides many classes. So that you do not lose the overview and there is confusion, there are so-called namespaces.Namespaces are organizational structures that structure related types (classes etc.) and group them into logical units. You can think of this as a folder on a file system that contains various files. If you now want to use a type in Unity, the namespace in the project must be referenced and integrated into the class. Usually Unity takes care of all of this. You only have to do it yourself if you want to use special types.

You include a namespace at the beginning of a script file with the signal word using . The following example class includes the System.Collections.Generic namespace in order to use the List type.

```
using System.Collections.Generic;

public class TestClass

                {

        private list <string> myStringList;

                }
```

If you now look at the interface example again, you will notice that namespaces have already been integrated in this way, namely the namespaces UnityEngine and System.Collections . You can find out more about these two namespaces in the "Script programming" chapter.

Generic classes and methods

The term generic in object-oriented programming actually only means that e.g. B. Methods or classes in general, that is, be designed regardless of type. The type (or method) to be used at the end is then passed to the class (or method) using angle brackets. In the case of C #, this type to be assigned is usually symbolized by the letter T in the generic class (or method).

```
using UnityEngine;

using System.Collections;

public class MyGeneric <T> {

    public T current;

}
```

Wherever the parameter T was used in the generic class, the transferred type is used later. The variable current will therefore have the data type that you specify when declaring the instance.

```
MyGeneric <string> test = new MyGeneric <string> ();
```

```
test.current = "ddd";

MyGeneric <int> test2 = new MyGeneric <int> ();

test2.current = 2;
```

Cunning

A generic type that is often used in Unity projects is the List <T> type . This works similar to an array, except that objects of this type offer significantly more functions than normal arrays. You can add as many new elements to these objects using the Add method , search them with various commands, or delete specific elements using Remove . You only have to enter the type that the object should contain when creating a list object. In the following example, you should therefore pay particular attention to the variable declaration of myNumbers , but also the inclusion of the System.Collections.Generic namespace , to which the List class belongs.

```
using UnityEngine;

using System.Collections;

using System.Collections.Generic;

public class GenericTest: MonoBehaviour {

    private list <int> myNumbers = new list <int> ();

    void Start () {

        myNumbers.Add (2);
```

```
                myNumbers.Add (33);

                myNumbers.Add (17);

               if (myNumbers.IndexOf (33)> = 0)

              myNumbers.Remove (33);

            Debug.Log (myNumbers.Count.ToString ());

// prints a 2

}
```

Sort list objects

Another strength of the List class is the sorting of the different elements using the Sort method . However, you should note that not every type can be sorted. This will only work if the latter has implemented the IComparable interface from the system namespace . You can find out more about this topic on the Internet (keyword “IComparable interface”).

Dictionary

Another class from the generic namespace that you should know is the Dictionary <TKey, TValue> class, or Dictionary for short. It even has two generic parameters. A dictionary stores so-called key-value pairs. The first value represents a unique identifier (key) and the second the value of this key. Each key may only appear once in the dictionary. The next example shows some basic functions of this class using a simple inventory management. This shows how you can use ContainsKey to determine whether this key already exists in the Dictionary, how you can use the Add method to create a new key-value pair, and how you use the key to access the associated value to change it .

```
using System.Collections;

using System.Collections.Generic;

public static class SimpleInventory {

        private static dictionary <string, int> items = new dictionary <string, int>
();

                public static void AddItems (string key, int val) {

                        if (items.ContainsKey (key))
```

```
                items [key] + = val;

        else

                items.Add (key, val);

}

        public static bool RemoveItems (string key, int val) {

                if (items.ContainsKey (key)) {

                        if (items [key]> = val) {

                                items [key] - = val;

                        return true;

                                }

                else

        return false;

                        }

                         else
```

```
                return false;

        }

}
```

-

CHAPTER 2

Script programming

Script programming

After getting to know the most important C # basics for Unity in the previous chapter, we now come to the actual programming of the scripts. Each script usually corresponds to exactly one class, which is why the terms script and class are often used synonymously in Unity.

MonoDevelop

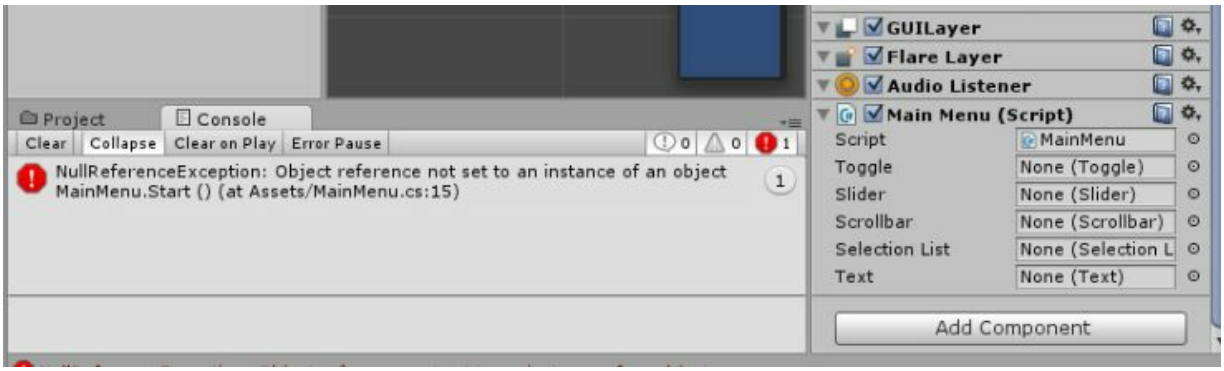
Unity provides the programming environment MonoDevelopmit for programming the scripts. MonoDevelop is an open source development environment for software developers, which was developed as part of the Mono project, an open source alternative to the Microsoft .NET framework. Unity delivers a specially adapted version that, among other things, enables easy debugging in conjunction with Unity . Double-clicking a script in Unity automatically opens MonoDevelop with the respective script. If MonoDevelop is already open with another script, an additional tab is created with the respective script. The other script remains open in MonoDevelop, so it is possible to switch between the scriptsAs a Unity developer, the main window of MonoDevelop is particularly important, in whose tabs the code of the various scripts is displayed. The Solution Explorer, which you can find on the left, is also helpful. This shows you all code files of the Unity project sorted according to programming languages and enables the opening of further script files. The storage function of MonoDevelop is particularly elementary, which you can use File / Save or the key combination [Ctrl] + Reach [S] . Adjustments that should be available in Unity must always be saved in MonoDevelop first. We will talk about the debugging functionalities, which are of course also very important, in the chapter "Troubleshooting and Performance". If you already have experience with other development environments and prefer a special one (e.g. Visual Studio), you can use this instead of MonoDeve-lop. In this case you can save an alternative IDE in the External Tools area as External Script Editor under Edit / Preferences .

Help in MonoDevelop

The version of MonoDevelop that comes with Unity has some special modifications, including an additional help function that leads the developer directly to the scripting reference of Unity. To do this, select the command for which you want more information and press On the German PC keyboard [Ctrl] + [Shift] + ['] . Alternatively, you can do this via the MonoDevelop menu Help / Unity API Reference . If you only have one class name in Mono-Develop, such as For example, entering the application and calling the help will give you all the information about methods and variables that this class provides.

Syntax error

If you have a syntax error in a script, e.g. For example, if you have forgotten a semicolon ("parsing error") or use a non-instantiated variable, Unity will display an error message after you save this incorrect script. This appears both at the bottom of the Unity window and in the console (console tab). If you click on this message, MonoDevelop will open and you will get to where the error occurs.



Forwarding of error messages

Clicking on the error message in the console does not always lead to the location where the developer made the programming error. Should you e.g. For example, if you forget a semicolon at the end of a line, the compiler may only recognize the next line as faulty, since the (previous) line of code has not yet been completed and this new command must not appear at this point. Especially for beginners, this sometimes leads to long error searches because the error can be found somewhere else than it is displayed.

Usable programming languages

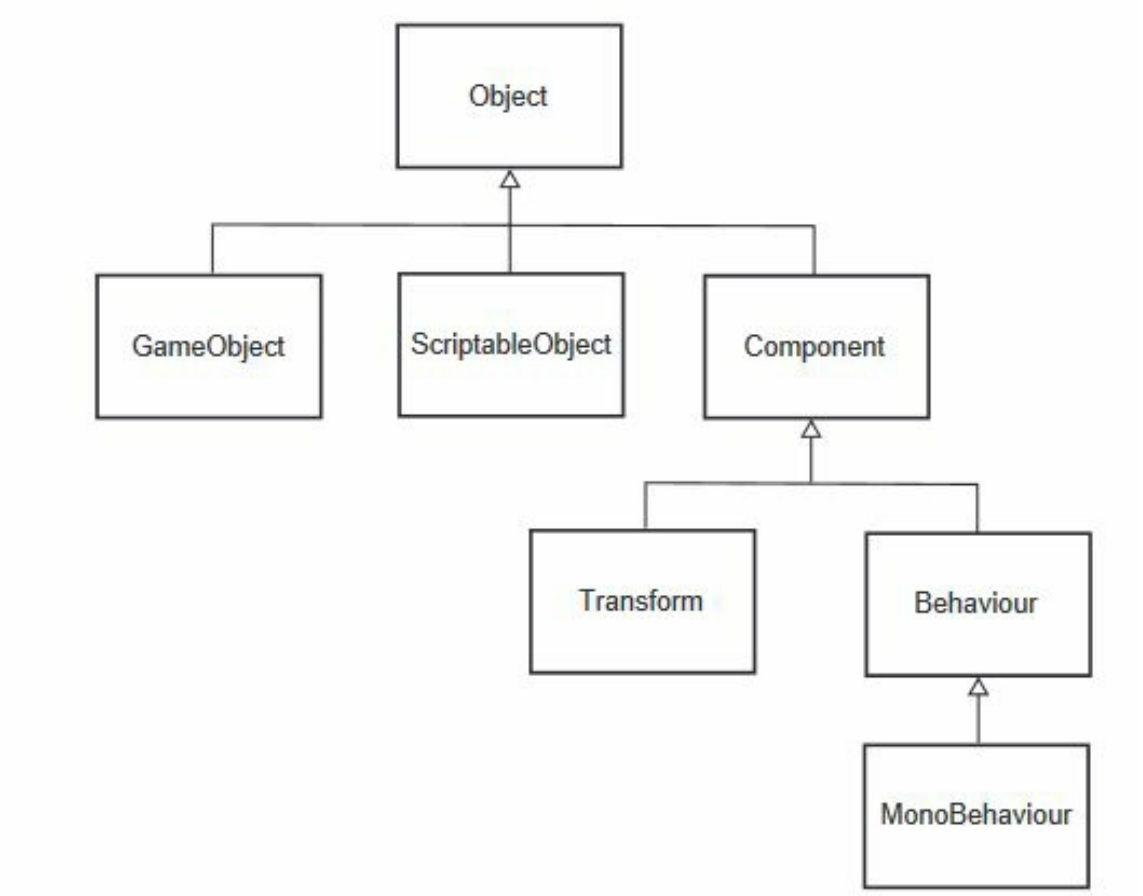
When programming scripts, you can choose between the programming languages JavaScript, Boo and C #. While you can mix the languages in a Unity project any amount (not within a script!), It is recommended, as far as it goes toward it verzichten.An itself is the mixing of language within a project not a problem. But once you try to access another from one script, it can become problematic. I'll get to the details in Section 4.14, “ Compilation Order” . Therefore, as far as possible, I recommend restricting yourself to one language within a project.

Why C #?

As you will have noticed, I use the language C # in this book. Initially, Unity focused on the JavaScript language. However, this has changed significantly over time, so that C # is now the focus. C # offers more functionalities and is also more widespread than the JavaScript branch specially adapted for Unity, also often called UnityScript

Unity's inheritance structure

As I mentioned in the previous chapter "C # and Unity" , all scripts that are to be used as components must inherit from the MonoBehaviour class . However, this class inherits from another class. In the end, components and GameObjects even have the same origin, namely the Object class from the NamespaceUnityEngine. Figure 4.3 shows the relationships between the most important classes of the Namespace Unity engine in a class diagram. The arrows always point from the derived class to the respective base class.


Object
GameObject
ScriptableObject
Component
Transform
Behaviour
MonoBehaviour

Object

Object is the parent base class of all objects in Unity. It provides some fundamental properties and methods, such as For example, the static method Destroy to destroy an object or the method DontDestroyOnLoad , which I will explain in this chapter. The classes GameObject, Component and ScriptableObject are derived from this class. For more experienced C # developers, it should be pointed out again that this is a Unity object class, not the one provided by the framework. NET class.

GameObject

GameObject inherits directly from Object is the base class of all objects in a scene Unity befinden.Meistens GameObjects be via the main menu GameObject created in the editor. If you want to create GameObject instances by code, they are instantiated like normal objects of a class with the signal word new .

ScriptableObject

ScriptableObject also inherits directly from Object. First of all, ScriptableObjects are common scripts that inherit from ScriptableObject and are usually used to store large amounts of data. Instances of these classes are created with the static method CreateInstance of the ScriptableObject class and can be created at runtime as well as at development time. ScriptableObjects are used more in special situations, e.g. B. to create data containers that can be stored outside of a scene, configured and read in if required.

Component

The Component class also inherits from Object and is the basis of all classes that can be attached to a GameObject. That is why they are created by adding them to the respective instance using the GameObject method AddComponent. Other important classes inherit from Component, of which I would like to give the Transform and Behavior classes as examples. Because in Unity all of them are derived from Component Classes referred to as components are often referred to in German as "components".

Transform

The Transform class inherits from Component and is added to every GameObject in a scene by default. Only if you create a GameObject with code does it not automatically have a transform component. In turn, the class RectTransform inherits from Transform, which is used by the uGUI system in the GUI design.

Behavior

The Behavior class inherits from the Component class. Classes that inherit from Behavior are also known as Behaviors or also generally as Components / Components and can be switched on and off via the enabled property . Examples of classes that are derived from Behaviors are Light or NavMeshAgent which I will talk about in the other chapters. Finally, the MonoBehaviour class also inherited from Behavior.

MonoBehaviour

The class MonoBehaviour is the base class of all script classes in Unity that are to be used as components and can be attached to a GameObject. Even if MonoBehaviour inherits from Behavior, you have to keep in mind that with a deactivated enabled property only that Execution of the methods Start , Awake , Update and OnGUI is prevented. Other methods can still be called and executed. B. the physics methods OnTriggerEnter or OnCollisionEnter .

Create scripts

In Unity, each script contains exactly one class. The name of the script is always the same as that of the class. Otherwise there will be an error later if you want to use this. Therefore, when choosing the name of the script, the following also applies: do not use spaces or special characters. To create a script, do one of the following:

- You use the main menu Assets / Create and select the desired script type (in our case C # Script).

- You use the Create menu of the Project Browser (located above), where you can also find the script types listed.

- You use the context menu of the Project Browser, which you can access with the right mouse button. You can also find the script types there via the Create menu branch .

- You use the New Script option of the Add Component button in the Inspector. As soon as you select a GameObject in the hierarchy, you will find it among all added components of the GameObject. The script is created by default in the root of the project browser and attached to the selected GameObject.

Double-clicking on the created script opens it in the standard development environment MonoDevelop.

Rename scripts

If you subsequently change the name of a script in the project browser or in the script itself, the counterpart must always be changed at the same time. If they differ, be it only case-sensitive, there will be an error later or you simply cannot add it to a GameObject. Therefore, use the rename functionality in MonoDevelop at best when renaming. This not only renames the class, but also the script and all integrations of the script in the project. You can access this function with the right mouse button Refactor / Rename as soon as you have positioned the cursor in the class name. The Rename function also makes sense if you want to rename related variables. In this case, too, all positions where you have already used them will be renamed.

Rename scripts

If you rename a script that has already been attached to a GameObject in a scene, you may find an entry "Missing (MonoBehaviour)" instead of the script after renaming it . In such a situation, delete this entry and add your renamed script again. The easiest way to delete this entry is to use the Remove Component function in the context menu of the component (symbolized by the gear in the Inspector).

The script framework

If you have created a new C # script as described above and now open it with a double click, you will see the following basic structure:

```
using UnityEngine;

using System.Collections;

    public class MyScriptName: MonoBehaviour {

        // Use this for initialization

          void Start () {

                }

  // Update is called once per frame

    void Update () {



                }
```

```
}
```

In the first two lines, the command using two namespaces is integrated. The UnityEngine namespace in particular is mandatory for a MonoBehaviour script and must not be removed. If necessary, additional namespaces can of course be added above. After the namespaces, the class definition follows with the MonoBehaviour inheritance. By default, each script inherits from this class, which also provides the start and update methods. You can also delete these methods if necessary. You can also change or completely remove the inheritance. Only then are these classes logically no longer usable as components.

Unity's event methods

You now know how to script in Unity. Now the question remains when the code in these scripts will actually be executed. The procedure of Unity is the following: MonoBehaviour provides several methods that are triggered by certain events or that are called at regular intervals by Unity. Here Unity runs through all GameObjects and their script instances and executes the corresponding methods there . I would like to briefly introduce some of these important methods below. Keep in mind that every script can / may have these methods, but they do not have to be implemented. On the contrary, it is better to delete methods if they are not filled with code. Because each of the following methods is called according to the definitions and accordingly costs performance, even if they are empty.

Update

The most important method in Unity is the Update method . Most of the code is programmed here. Update runs throughout the game, each time before a frame is rendered, that is, before a new frame is drawn on the monitor. Note that the rendering times of the individual frames can vary. If you want to program in update code that should be independent of frames, the values should be multiplied again with the time value deltaTime of the class Time . This value represents the time between the last and the current frame. For example, if you add motion directly to a GameObject using the Transform component. Each time Update is called, the object is moved a little bit further. You can find out more about Transform in Chapter 5, "Objects in the Third Dimension" .

```
void Update ( ) {

float speed = 20.0F;

float distance = speed * Time.deltaTime

; transform.Translate (Vector3.forward * distance);

}
```

FixedUpdate

The FixedUpdate method is called in defined time intervals and should be used if you Rigidbodys want to access (Rigidbodys are responsible for Simu lose physical behavior). The background is that the Unity physics engine does not work frame-based, but works in discrete time intervals, i.e. in fixed time periods. Therefore, before each calculation of the physics in Unity, all FixedUpdate methods are executed in the MonoBehaviour scripts. You can find out more about physics in the chapter “Physics in Unity”. The following example adds a torque to the rigid body of the object in every FixedUpdate call so that it rotates around the Y axis.

```
void FixedUpdate ( ) {

rigidbody.AddTorque (0, 2, 0);

}
```

Change time interval

By default, the time interval in which FixedUpdate is called is 0.2 seconds. You can change this via Edit / Project Settings / Time / Fixed Timestep .

Awake

This method is executed once, when the script instance is loaded. When loading a scene, all objects in the scene are initially initialized. Then the Awake methods of the individual GameObjects are called in a random order . Since all GameObjects were initially initialized, it is already possible to find other GameObjects and assign variables.

```
private GameObject cam;

    void Awake ( ) {

    cam = GameObject.FindWithTag ("MainCamera");

}
```

begin

The Start method behaves similarly to Awake . However, unlike Awake , this method is only carried out on active instances. If a script instance is deactivated at the beginning and is later set to enabled , Start is only called at that moment. In addition, the Start methods are only called after all Awake methods have been executed, which makes it easy to control the order of initialization and to delay certain executions. The following example uses the FindWithTag method to find the player character for theirs Assign a script instance of the HealthController class to a temporary variable. The next step is to query the current value of health and assign it to its own private variable.

```
private int playerHealth;

  void Start ( ) {

    HealthController hc = GameObject.FindWithTag ("Player").
GetComponent <HealthController > ( ); playerHealth = hc.health;

}
```

OnGUI

The OnGUI method is part of Unity's own programming- oriented GUI system. In this method, the controls for this system are programmed. Since the system does not work frame-based, but rather event-based, where the events are triggered by user input or internal Unity rendering processes, this method is also called accordingly. For this reason, OnGUI can also be called several times per frame. You can find out more about GUI in the chapter of the same name.

```
void OnGUI ( ) {

    GUI.Label (new Rect ( 0,0,100,30), "Hello World!");

}
```

LateUpdate

Another method that is not used as often, but is still important, is the LateUpdate method. This is carried out after all update methods have been called, but before rendering. This method is therefore often used, especially for camera scripts. If a camera tracks a target that is moved in the update method, the camera can also be repositioned in LateUpdate before it is finally rendered . The following example could be added to one camera so that it always points in the direction of another Object is directed, more precisely in the direction of its trans-form component. For example, this could be for the control of a surveillance camera, which is permanently installed in one place, but keeps track of the player in the room all the time.

```
public transform target;

    void LateUpdate () {

      transform.LookAt (target.position);

}
```

Component programming

An important principle of Unity is that every GameObject has its own components. So you are not programming a life management system that manages the life strength of all opponents, but a script that only manages the health of an individual opponent. This script is then added to each opponent again, so that each opponent has its own administration, and this principle can (and should) sometimes not be adhered to in certain situations. But first of all, you should make sure that each GameObject has its own components. Incidentally, this also makes a big difference in access to the components, since Unity often offers simplifications for access to its own components, which you will also see below.

Access GameObjects

Accessing a component's own GameObject is very easy. Simply use the lowercase variable gameObject for this . This makes it very easy to access components of your own GameObjects, which I will talk about in a moment.

OtherScript otherScript = gameObject.GetComponent <OtherScript > ( );

To access other GameObjects, there are several methods that you can use both at development time and at runtime. The first method, which is also the most gentle on performance, is the Inspector assignment, in which you create a public variable to which you can drag and drop the other GameObject during development.

```
public GameObject player;

        void Update ( ) {

    //Code...

        if ( player! = null) {

    //Code...
```

```
            }

}
```

But you can also access other GameObjects at runtime. If you want to access them more often, make sure that you save them in variables. This brings an enormous performance advantage, since Unity has to run through all GameObjects of a scene in the worst case to find the right one. If you then also make this access in an update method, this would happen in every frame.

FindWithTag

You can find a GameObject by its tag. To this end, the class GameObjectdie method FindWithTag available. If there are several GameObjects with this tag in the scene, one is selected at random.

GameObject cam; void Awake () {

cam = GameObject.FindWithTag ("MainCamera");

}

FindGameObjectsWithTag

If there are several objects with the same tag, but you also want to collect them all in an array, you should use the FindGameObjectsWithTag method . As a return value, you will receive a GameObject array of all objects found with this tag.

```
GameObject [ ] enemies;

    void Awake ( ) {

     enemies = GameObject.FindGameObjectsWithTag ("Enemy");

}
```

Find

You can find a GameObject by its name. The static method Find of the class GameObject is suitable for this.

GameObject leftArm;

leftArm = GameObject.Find ("Left Arm");

You can also further specify the search query by also entering the parent objects of the GameObject you are looking for. Separate the different GameObjects with a slash character [/] . To show that an object is the last one and does not itself have a parent object (root object), you can do this with a preceding slash.

leftArm = GameObject.Find ("Bob / Left Arm"); // Left Arm has a parent Bob

leftArm = GameObject.Find ("/ Bob / Left Arm"); // Bob is the root object

Activate and deactivate GameObjects

Each GameObject offers the SetActive method . You can use this to activate and deactivate a complete GameObject. Give the method the state you want to achieve.

```
void Start ( ) {

    gameObject.SetActive (false);

}
```

Note that you can only activate a deactivated object if you already have access to it (e.g. via an Inspector assignment). With the find methods presented above, you have no chance of finding a deactivated object

Destroy GameObjects

To completely destroy a GameObject, you have to pass the object to the Destroy method .

```
void DestroyPlayer ( ) {

    GameObject player;

        player = GameObject.FindWithTag ("Player");

      Destroy (player);

  }
```

You can also give the Destroy method a float parameter that delays the destruction by the respective seconds.

```
Destroy (player, 2.5F);
```

Create GameObjects

Most of the time you will create GameObjects via the main menu or with prefabs (see chapter " Prefabs"). But of course you can also create completely new GameObjects at runtime. You do this by creating a new instance of the GameObject class in the normal C # way. It does not matter whether the instance is a local or global variable or which access modifier it has. Since each GameObjectper Default initially has the name "New GameObject" , you can also give your GameObject a name when you instantiate it.

GameObject go = new GameObject ( );

GameObject go2 = new GameObject ("My Name");

Access components

There are several ways to access components. The easiest way to access components of other GameObjects is to declare a public variable of the type of the component to which you drag the component's GameObject during development. The variable then does not refer to the GameObject, but to the component of the GameObject. If the GameObject does not have this component, the content of the variable is null, i.e. empty.

```
public transform player;

    void Start ( ) {

        if ( player! = null)

          player.position = Vector3.zero;

}
```

GetComponent

Each GameObject has the GetComponent method , with which you can access all components of the respective GameObject. If you write this call without a GameObject, the method looks for this component in your own GameObject. The following example searches for the HealthController of the player as well as its own and assigns two different variables in order to use them later.

```
HealthController playerHc;

HealthController myHc;

    void Awake ( ) {

        GameObject player;

        player = GameObject.FindWithTag ("Player");

        playerHc = player.GetComponent <HealthController> ();

       myHc = GetComponent <HealthController > ( );

}

   void Update () {

         if (playerHc.health> 0) {
```

```
//Code...

            }

    }
```

Reduce traffic

Be sure to reduce GetComponent accesses as much as possible. For more frequent access, save the components in variables instead.

SendMessage

A very interesting way to call a method is the SendMessage method . SendMessage is provided by the GameObject class and runs through all MonoBehaviour scripts of the GameObject, calling every method with this name. So you don't even need to know which script it is in. Optionally, you can also give SendMessage a transfer parameter as well as an option parameter of the type SendMessageOptions . The latter determines whether a recipient method is absolutely necessary or not. If one is necessary, but none is found, an error is triggered.

The following example script could be attached to any weapon or trap with a collider to cause damage on contact. The script tries to call a method called ApplyDamage in the other GameObject and transfer the damage value 1 to it. For this purpose, the GameObject of the passed collision parameter is accessed and SendMessage is called.

```
using UnityEngine;

using System.Collections;

    public class damage: MonoBehaviour {

        public float amount = 1;

        void OnCollisionEnter ( Collision collision) {
collision.gameObject.SendMessage ("ApplyDamage", amount,
SendMessageOptions.DontRequireReceiver);     }
```

}

The DontRequireReceiver parameter states that no receiver is necessary. This means that no error is triggered if the target object does not have a script using this method. You can find out more about collisions in the chapter “Physics in Unity” .

Variable access

If you want to access components of your own GameObjects, Unity offers access via variables for some, which considerably simplifies the programming effort. This includes, for example, the AudioSource component, the rigid body component or the transform component.

audio.Play ();

rigidbody.AddForce (0, 1, 0);

transform.Translate (0, 10, 0);

Add components

Want to add a new component at runtime a GameObject, you can run the method AddComponent use of each GameObject offering. The following example adds a script component own GameObject Health controllers and a Rigidbody to.

gameObject.AddComponent <HealthController > ( );

gameObject.AddComponent <Rigidbody > ( );

Remove components

To remove a component from a GameObject, use the Destroy method as if you were destroying a GameObject . If you want to destroy your own script, use the keyword this . Here, too, you can incorporate time delays using a second parameter.

Destroy (rigidbody); // destroy your own rigid body

Destroy (this, 2); // destroy your script instance with a 2 sec delay

Activate and deactivate components

Instead of removing a component completely straight away, it can also make sense to just deactivate it. The advantage is that you can reactivate it later. A classic example is light. The following example accesses the Light component of your own GameObject and uses the logical negation operator (the exclamation mark) to reverse the state of enabled. The effect is that the switched on light is switched off and the switched off light is switched on.

void Switch ( ) {

light.enabled =! light.enabled;

}

Random values

Random values are an important element in making games interesting. The application options range from assigning a random parameter value such as strength to generating and placing GameObjects to creating entire levels. For this, Unity offers the Random class , which offers several interesting methods for generating random values. The most common is the Range method . Please note that there are two different versions.

- Range for float values generates a random value that lies between the start and the end value. The start and end values can also be returned as results.

- Range for int values generates a random value that ranges from the start value, but must be smaller (!) Than the end value.

```
using UnityEngine;

using System.Collections;

    public class RandomXPosition: MonoBehaviour {

            void Start () {

              float xPos = Random.Range (0.0F, 5.0F);

              Vector3 pos = new Vector3 (xPos, 10.0);

              transform.position = pos;
```

```
        }

}
```

The class offers a few more variants of the random calculation. How to achieve a random rotation of the type Quaternion via the static variable rotation or with the help of insideUnitSphere you get a random value in Vector3 format, in which x , y and z each have a random value between –1 and 1.

```
using UnityEngine;

using System.Collections;

        public class randomize: MonoBehaviour {

                public float radius = 3;

              void Start () {

      transform.position = Random.insideUnitSphere * radius;
transform.rotation = random.rotation; }

}
```

Execute code in parallel

With Coroutines, you have the option of executing methods that are executed in parallel to the actual Update and LateUpdate calls. While normal code, which is in the update method, runs in every frame, you can program code blocks with Coroutines that extend over any number of frames.

```
string message = "";

IEnumerator Countdown ( ) {

    yield return new WaitForSeconds ( 1);

message = "3";

yield return new WaitForSeconds ( 1);

message = "2";

yield return new WaitForSeconds ( 1);

message = "1";

yield return new WaitForSeconds ( 1);

message = "Go!";

}
```

The example in Listing shows a countdown that waits four times a second and then assigns new content to the message variable (which is then visualized in the GUI, for example). To start such a coroutine, use the command StartCoroutine .

```
void Start () {

    StartCoroutine ( Countdown ());

 }
```

The key keywords of a coroutine are the IEnumerator interface definition , which enables the method to be run through several frames, and the yield return command (see listing). The latter saves the current status of the routine so that it can continue with the next frame at the same point. To do this, specify yield return null . The following coroutine runs through ten frames and writes the current playing time into the console in each frame.

```
IEnumerator FrameTest ( ) {

        for (int i = 0; i <10; i ++) {

      Debug.Log (Time.time);
```

```
            yield return null;

        }

}
```

The WaitForSeconds method used above extends the code suspension so that it does not continue in the next frame, but only after a second.

WaitForSeconds

With yield return WaitForSeconds (2.5F) you can interrupt the execution of a coroutine for a certain time. The passed float parameter specifies the waiting time. Please note that an interruption with WaitForSeconds can only be as precise as the frame duration. So if every frame lasts 0.02 seconds, the accuracy of WaitForSeconds cannot be better.

Delayed and repeating function calls with Invoke

Unity provides you with options so that you can delay other methods within a MonoBehaviour class and call them repeatedly. For this Unity offers the so-called Invoke commands.

Invoke

The Invoke method expects the name of a method to start and a second value that indicates the delay. The code continues to run normally after the command has been executed. The specified method is only triggered after the time has been reached. The specified method must be in the same script. You can use IsInvoking to find out whether a method has already been called with a delay via Invoke (more on this in a moment). The following example executes the DelayedMessage method with a delay of 2.5 seconds after execution of the Start method. This assigns a welcome text to a variable, the content of which could in turn be displayed in the GUI.

```
public string welcomeMessage = "";

    void Start () {

        Invoke ("DelayedMessage", 2.5F);

        }

void DelayedMessage ( ) {

      welcomeMessage = "Welcome, Master!";

}
```

InvokeRepeating, IsInvoking and CancelInvoke

InvokeRepeat works in a similar way to Invoke , except that this method also calls the specified method not just once, but as often as you like. You pass the interval of the repetition as the third parameter. The second one also determines the delay of the first execution. You can also use the IsInvoking method to check whether a method has already been initiated with an Invoke command. You can then cancel these statements with CancelInvoke . You pass the name of the method you want to cancel. If you don't pass any, all methods that you started with an Invoke command are aborted. The following example could come from a Spawner script that creates a new object at a random position every second three seconds after the game starts.

```
public GameObject go;

public bool isGameOver = false;

    void Start () {

        // Starts the delayed retry of RandomInstantiation InvokeRepeating
("RandomInstantiation", 3.1);

}

   void Update () {
```

```
// Checks whether RandomInstantiation is still running

                if (IsInvoking ("RandomInstantiation")) {

                 // When the game is over

                // RandomInstantiation is to be canceled

               if (isGameOver) CancelInvoke ("RandomInstantiation");

              }

       }

          void RandomInstantiation ( ) {

     // position where the prefab should be instantiated

                  Vector3 pos = new Vector3 ();

                   pos = Random.insideUnitSphere * 10;

                     // random values for all axes pos.y = 0;

                // Y is set to 0. X and Z are between -10 and 10 // instantiate
prefab

Instantiate ( go, pos , Quaternion.identity);
```

}

Save and load data

With the PlayerPrefs class, Unity offers you a great way to save values regardless of the system. Regardless of whether Windows, Mac OSX, the web player or Android, you simply transfer the values to be saved to the PlayerPrefs methods. Unity will take care of where they actually end up. In the following section there is a small application example.

PlayerPrefs preferences

Even if the paths differ on all platforms, they all use the names for Company Name and Product Name stored in Unity. On Windows systems, the registry path HKCU \ Software \ [company name is used for stand-alone applications, for example] \ [product name] taken, web player applications write to the directory % APPDATA% \ Unity \ WebPlayerPrefs on Windows systems .

Change the company name and product name in the Player Settings (Edit / Project Settings / Player). By default, Company Name is set to " DefaultCompany" and Product Name to the name of your project.

save data

PlayerPrefs provides three static methods to save values as int, float or string. All three methods expect two parameters. Pass the first parameter a string that represents a unique identifier, i.e. the key (the ID or the name of the value). The second parameter is then the actual value in the particular data type.

PlayerPrefs.SetInt ("Lifepoints", 4);

PlayerPrefs.SetFloat ("Speed", 2.5F);

string playerName = "Carsten";

PlayerPrefs.SetString ("Name", playerName);

There are also three static methods for loading the data. You also have to hand over the key here.

int myLifepoints = PlayerPrefs.GetInt ("Lifepoints");

float speed = PlayerPrefs.GetFloat ("Speed");

string playerName = PlayerPrefs.GetString ("Name");

Check the key

Before loading data, i.e. accessing a key, it can often make sense to first check whether it actually exists. Even if Unity does not cause an error here, it can still make sense to assign default values to the variable. You do this with the HasKey command .

if (! PlayerPrefs.HasKey ("Lifepoints"))

PlayerPrefs.SetInt ("Lifepoints", 5);

Clear

Of course, you can not only create new entries, you can also delete them. Here you can use the static method DeleteKey . To delete all saved values at once, you can use DeleteAll .

PlayerPrefs.DeleteKey ("Lifepoints");

PlayerPrefs.DeleteAll ();

Save

On some platforms, such as the web player, Unity does not write the values to the hard drive when calling the methods SetInt , SetFloat and SetString . Instead, Unity temporarily saves them temporarily. Only when you exit the application, the web player that would be the proper closing the tab or the browser, these values are only to disk geschrieben.Da to Unity in the background taking care of everything that makes this behavior for First no difference. Only in the event of an error, e.g. For example, if the browser crashes, the values may no longer be able to be written back in time. For this, Unity now provides the Save command, which you can use to force the actual saving.

PlayerPrefs.Save ();

Cross-scene data

GameObjects actually only exist within a scene. If a new scene is started, all GameObjects of the previous scene are destroyed, loading a new scene is done with the LoadLevel method of the Application class. You pass this either the name or the level index of the respective scene. You define these in the Build Settings, which you can access via File / Build Settings . You can find out more about this in the chapter “Creating and publishing games ” .

Application.LoadLevel (1);

However, if all GameObjects are now destroyed, this of course has the disadvantage that data such. B. Experience points, strength of life, etc. are lost when the scene changes. After all, they are saved in scripts that are attached to a GameObject. There are, of course, solutions to this dilemma, even several. I would like to introduce two of these approaches to you.

Passing values with PlayerPrefs

You have already got to know the first approach: PlayerPrefs. You can easily all the values that you want to pass, they simply load when exiting a scene with PlayerPrefs abspei-manuals and at the start of a new scene and the variables again zuweisen.Hierfür one option would be storing the values in the OnDestroy - method provides every MonoBehaviour script. If the objects and components are then destroyed when changing to a new scene, the data of a script is written away here. In the start - or Awake method they can then read back werden.Hierzu a small example: The following script has a variable health whose aktuel-ler value is stored methods PlayerPrefs-on exit with. If a new scene is then started, the script reads this value back into the variable from the PlayerPrefs in the start method . Because the key will not yet exist when the script is called up for the first time, a check is carried out to prevent errors before reading in whether this key actually exists.

```
using UnityEngine;

using System.Collections;

    public class HealthValue: MonoBehaviour {

        public int health = 2;

      void Start () {

       if (PlayerPrefs.HasKey ("Health"))

                health = PlayerPrefs.GetInt ("Health");
```

```
    }

        void OnDestroy () {

         PlayerPrefs.SetInt ("Health ", health );

  }

}
```

Use start menus for initialization

There are a few points to consider when using the above script. For example, when the game is ended and restarted, the variable is always assigned the value of the last game. This may be desirable when continuing an old game, but certainly not when restarting the game. To prevent this from happening, there is a simple solution: You can do this in an upstream scene, e.g. B. a start menu scene, the keys with the initial values. This allows you to use the above script even without the HasKey query. You can also use the start menu to ask the player whether they want to continue an old game or start a new one. Such a script could then look like this:

```
using UnityEngine;

using System.Collections;

      public class InitValues: MonoBehaviour {

         public int health = 2;

   void OnGUI ( ) {

          if ( GUILayout.Button ("New")) {

    PlayerPrefs.SetInt ("Health", health);

        Application.LoadLevel (1);

          }
```

```
    if (PlayerPrefs.HasKey ("Health")) {

     if (GUILayout.Button ("Continue")) {

       Application.LoadLevel (1);

      }

     }

  }

}
```

For example, the above script could be used in the scene with index 0, with the previous HealthValue script being used accordingly in a scene with index 1.

Prevent destruction

The above procedure has some disadvantages. So it can be quite expensive if you want to save all parameters of a customizable GUI, a player or an inventory. But at the latest when you want to transfer textures and materials from one scene to another, you have a problem here. For textures you can not handle the PlayerPrefs abspeichern.Hierfür the parent Object class has the static method DontDestroyOnLoad which prevents the destruction of any object during charging. Since the transfer parameter is also of the Object type , you can transfer it to any object (see "Unity's inheritance structure"), even if a GameObject is usually passed here. The following example script prevents the destruction of the GameObject, which the following script is appended, and all of its components. If the scene is ended and a new one is started, the entire GameObject is transferred to the next scene. This course of Wertder variables remain life points received.

```
using UnityEngine;

 using System.Collections;

   public class LifePointController: MonoBehaviour {

        public int lifePoints = 0;

        void Awake ( ) {

        DontDestroyOnLoad (gameObject);

       }
```

//Code...

}

Note here that at the start of the next scene, the Awake - as well as the start will not run method again. If you use this, you can also use the OnLevelWasLoaded method , which also receives the current level index.

void OnLevelWasLoaded (int level) {

//Code...

}

DontDestroyOnLoad as a singleton

With DontDestroyOnLoad you prevent the passed object from being destroyed when the scene changes. However, if you do not switch to a new scene at all, but maybe just start the same scene again (e.g. because the player died), the object suddenly exists twice. To solve this problem, the so-called singleton design pattern can be used be used. This ensures that there can only be one instance of a class. A static variable is used for this, which stores the first instance that is generated by the class. If the variable is already assigned to another instance, it is simply destroyed (in our case, we want to make sure that the entire GameObject is destroyed). In addition , either the variable itself or a property is often made publicly available in Unity easier to access this instance from the outside. The above MonoBehaviour script could then look like this as a singleton:

```
using UnityEngine;
using System.Collections;
public class LifePointController : MonoBehaviour {
  public int lifePoints = 0;
  //Statische Variable vom Typ der eigenen Klasse
  private static LifePointController instance;
  //Statische Eigenschaft fuer den einfachen Zugriff auf die Instanzen.
  public static LifePointController Instance {
    get{
      return instance;
    }
  }
  void Awake() {
    //Ist die statische Variable noch nicht gefuellt?
    if (instance == null) {
      //Weise diese Instanz der Variablen instance zu.
      instance = this;
      DontDestroyOnLoad(gameObject);
    }
    else {
      //Zerstoere dieses GameObject
      Destroy(gameObject);
    }
  }
}
```

Debug class

Unity has a Console window that displays Unity error messages and warnings. Unity offers the possibility to output informational texts and similar ones there. The Debug class offers some useful functions for this, the main function of which is the Log method . This outputs any text in the console.

```
void Start ( ) {

    Debug.Log ("test output");

    }
```

In addition to the normal log method, there are the variants LogWarning and LogError , which output the messages as warning and error messages, but the Debug class offers other functions for debugging. You can also use the class to draw lines in Scene View (DrawLine and DrawRay) or use the Break method to stop the editor. You can find out more about this class in the Scripting Reference that is supplied with Unity.

```
void Update ( ) {

    Debug.DrawLine (Vector3.zero, new Vector3 (2, 0, 0), Color.red);

}
```

Compilation order

A special feature of Unity is the script compilation order, which is divided into four different phases. This is important because from a script you can only access others that are in the same or an earlier phase , the question remains how to determine which script is compiled in which phase. This is controlled in Unity via the folder names in which you store your scripts in the Project Browser. The names don't really matter, but some of them are reserved for special purposes, for example Editor. Folders with this name should only store scripts that also inherit from the class Editor, not from MonoBe-haviour. You can use these to expand Unity with your own functionalities , for example. Unity proceeds with the compilation in four different steps.

1. All scripts are compiled which are located in folders with the names Standard Assets, Pro Standard Assets and Plugins.

2. All editor scripts that are located in folders named Editor and in the above folders Standard Assets, Pro Standard Assets and Plugins are compiled.

3. All other scripts that are outside of folders named Editor are compiled.

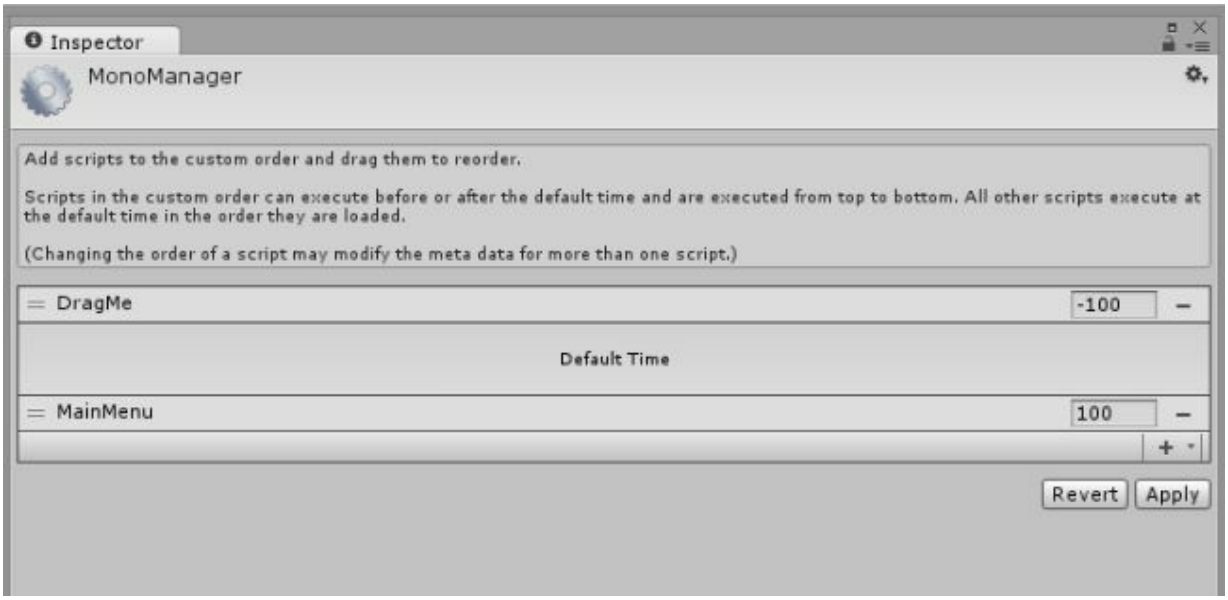
4. Finally, any remaining scripts (in folders named editor) kompi- liert.Weiter , there is the reserved folder names WebPlayerTemplates whose scripts contained are not compiled.

If you want to use your C # script to access a Boo or JavaScript class, the script you want to access must have been compiled in an earlier phase than itself. In this case, you could use the JS script in the Move the plugins folder and it is already accessible from your C # script.

However, this creates a problem: you can only access in one direction, it is not possible to access in both directions. You should keep this in mind when working with several programming languages.

Execution order

Sometimes it is important to make sure that one script is executed before the other, with two scripts you can often still use Update and LateUpdate , but with three scripts this is no longer possible. For this there are the Script Execution Order Settings, which you can find under Edit / Project Settings / Script Execution Order .

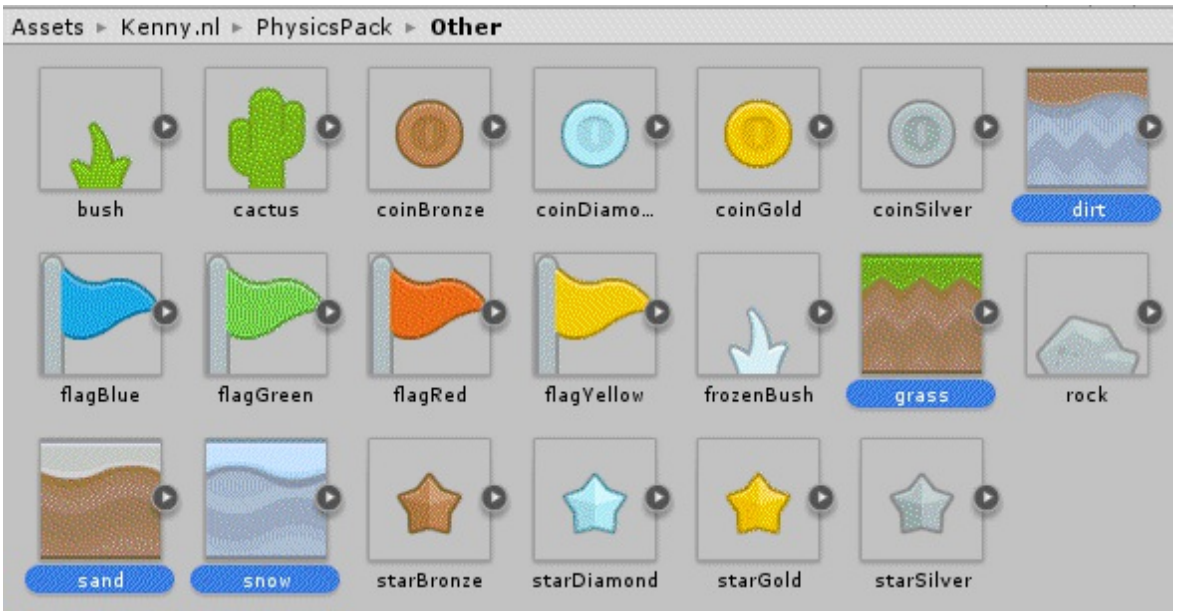


You can add a script by clicking the plus sign. You can then leave the script there or drag it up and change the time value. The scripts are then processed from top to bottom. All scripts not defined here are called in the Default Time area in a random order.

CHAPTER 3

Create a simple game

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.

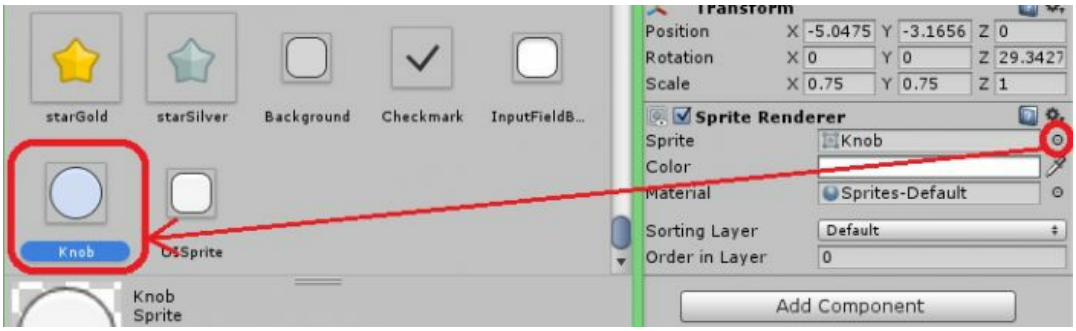
With this we have identified the basic programmable ballistics, and we have some small auxiliary programs for secondary functions.

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 ();
```

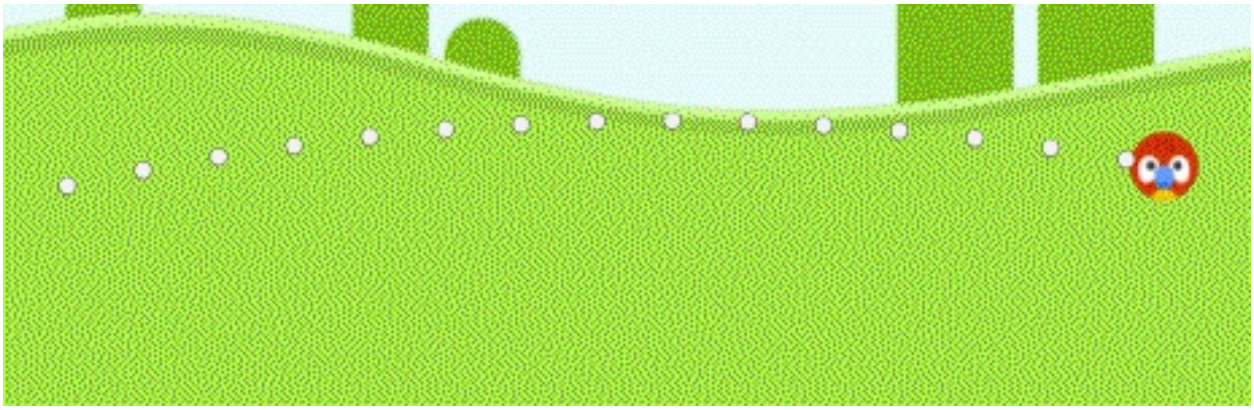
```
        // Update the variable value as the projectile has been launched
        launched = true;
    }

    // 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.

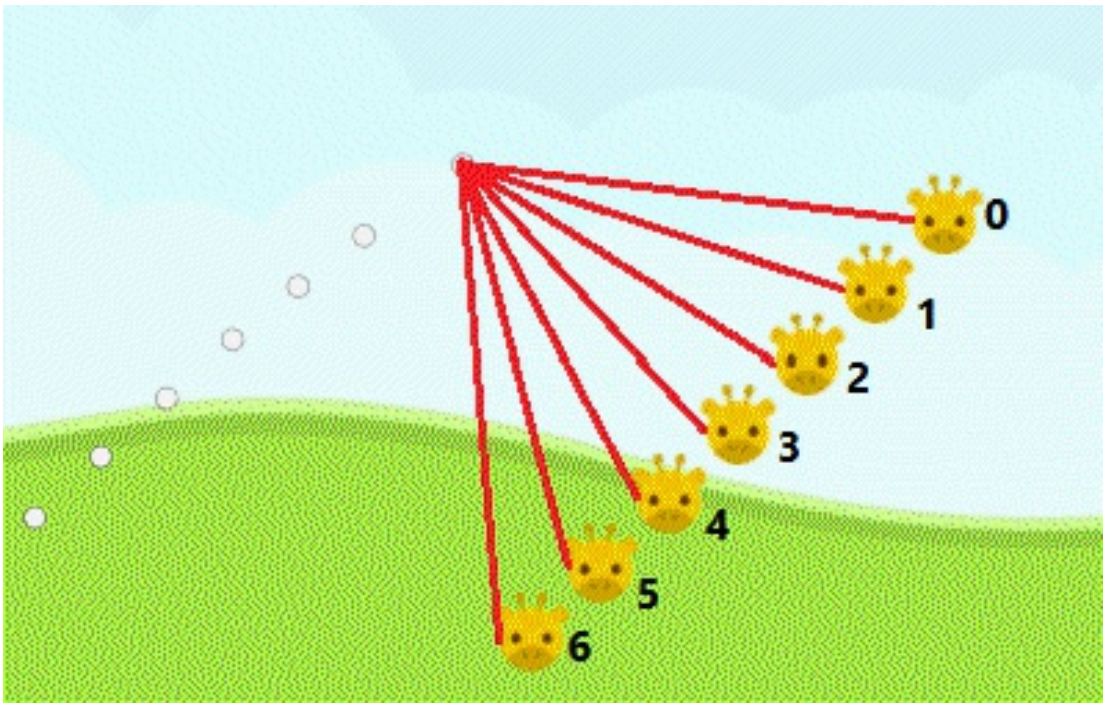
The applet that will implement this type of attack is called ClusterAttack and is described in the following narrative:

```
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);
    }
}
```

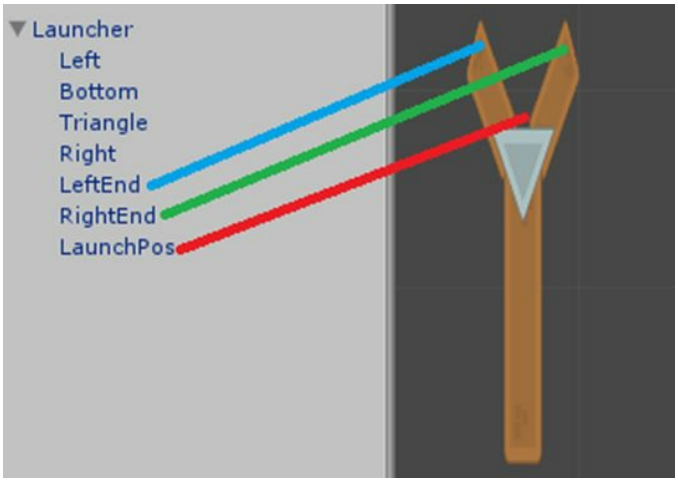
The idea of doing this attack is to receive the DoSpecialAttack message and then create the specified number of fragments using the specified template. In order to prevent collisions between fragments and some of them and also between fragments and the original projectile - where a collision can occur 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 vehicle, 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 ().

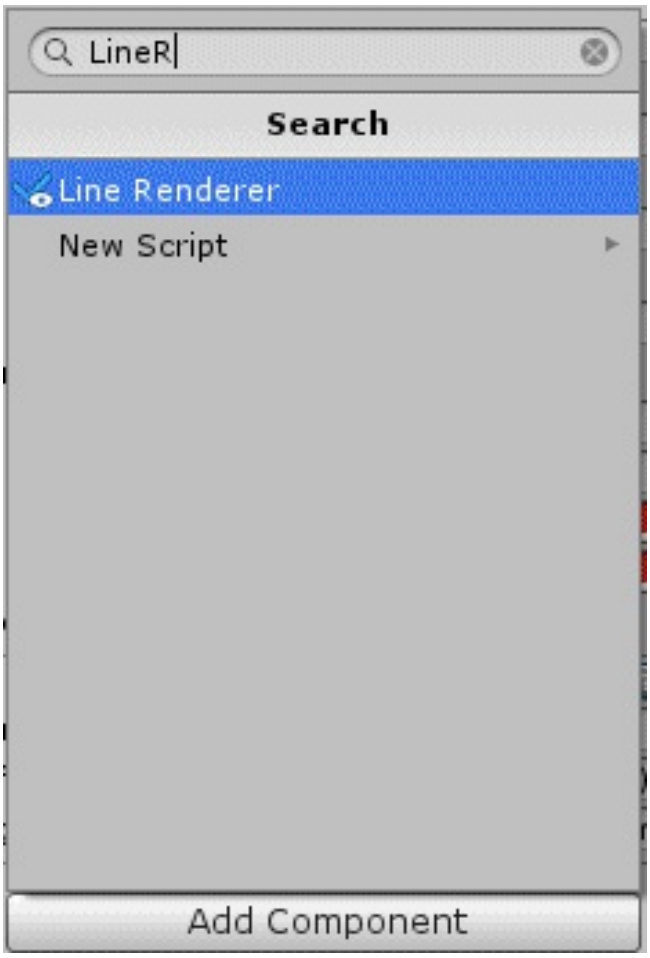
Here I will talk a little bit more about the BringCurrentProjectile () function to explain the mechanism you use to achieve the smooth movement of the projectile from its current position towards the launch site. The smooth movement in game engines depends on the elapsed time factor between each two 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.

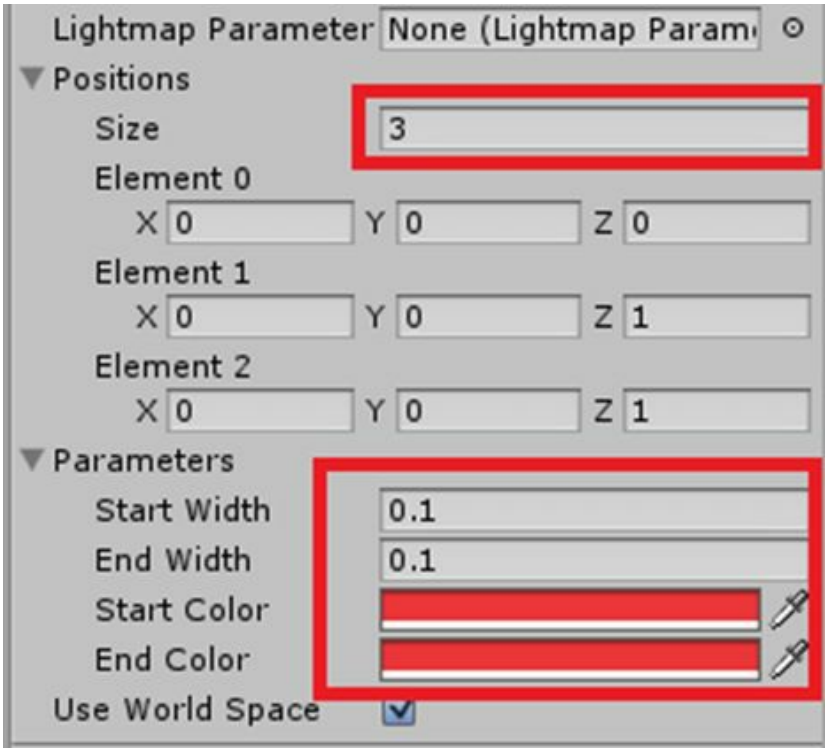
The idea is to calculate the distance between the launch location and the current location of the currentDist and compare it to the maximum expansion, maxStretch. If this distance exceeds the limit, we will divide maxStretch by currentDist, thus obtaining the required interpolation between the original 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:

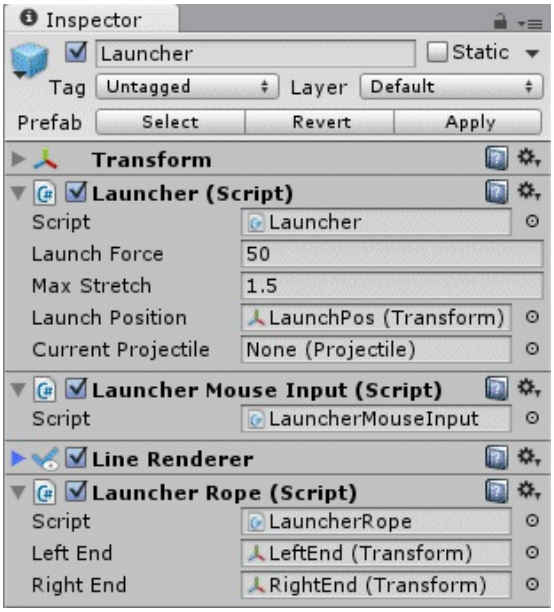
```
using UnityEngine;
using System.Collections;

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 ();
        }
```

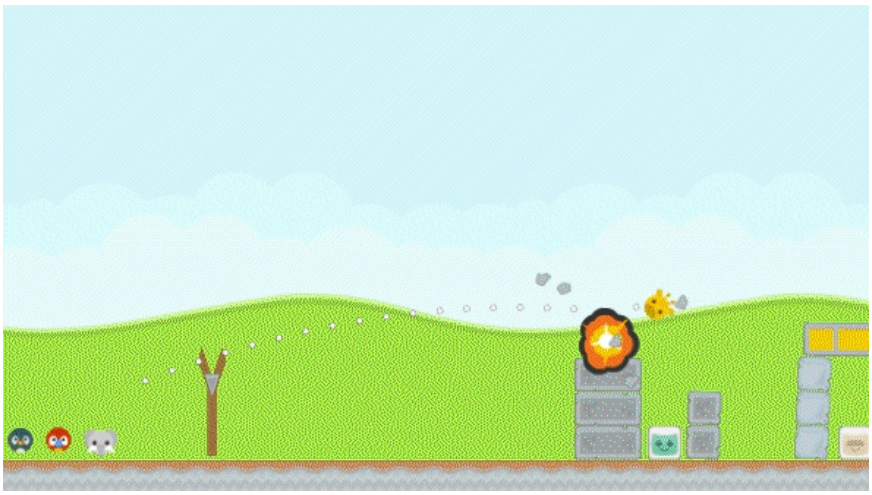
```
    }
}
```

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!



*

***