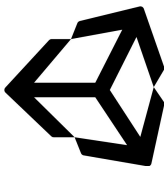
2ND edition - 2020

LEARNING C# BY DEVELOPING GAMES WITH UNITY

C# PROGRAMMING FOR UNITY GAME DEVELOPMENT

BY EMMA WILLIAM

Learning C# by Developing Games with Unity



C# Programming for Unity Game Development

by

Emma William

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

2 nd Edition: 2020

Learning C# by Developing Games with Unity

“The “joy of discovery” is one of the fundamental joys of play itself. Not just the joy of discovering secrets within the game, but also the joy of uncovering the creator’s vision. It’s that “Aha!” moment where it all makes sense, and behind the world the player

can feel the touch of another creative mind. In order for it to be truly joyful, however, it must remain hidden from plain view—not carved as commandments into stone tablets but revealed, piece by piece, through the player's exploration of the game's rules."

— Derek Yu, Spelunky

.

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.

memlnc

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 2015 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.

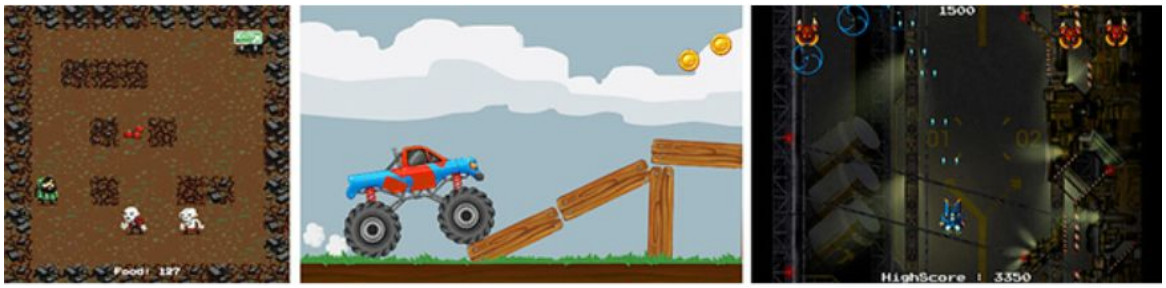
Full 3D

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.

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:

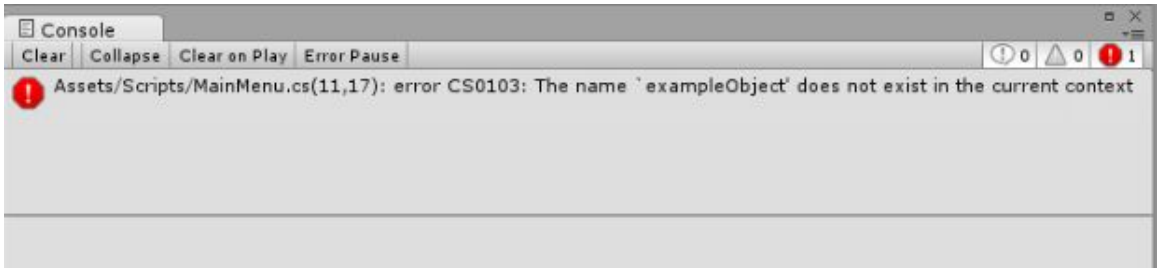
Scene Window : It you use to build the game scene, add different objects to it and distribute it in 2D space. Initially this window contains only one object which is the camera.

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 .

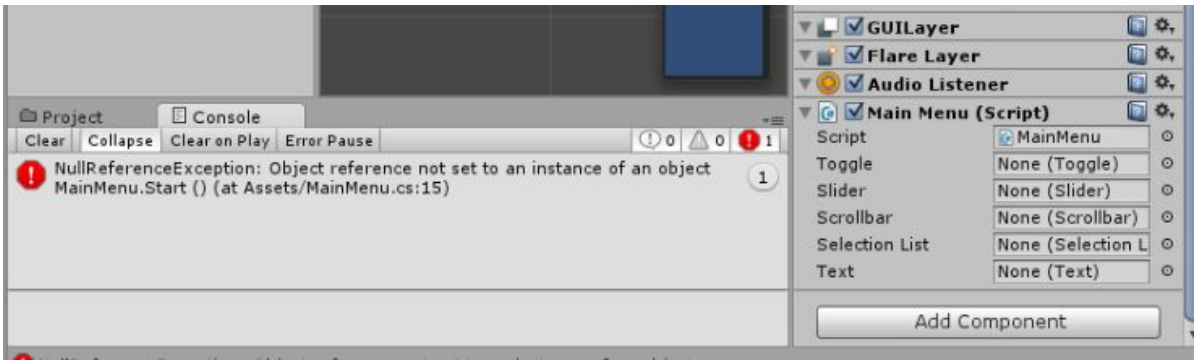
Help in MonoDevelop

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

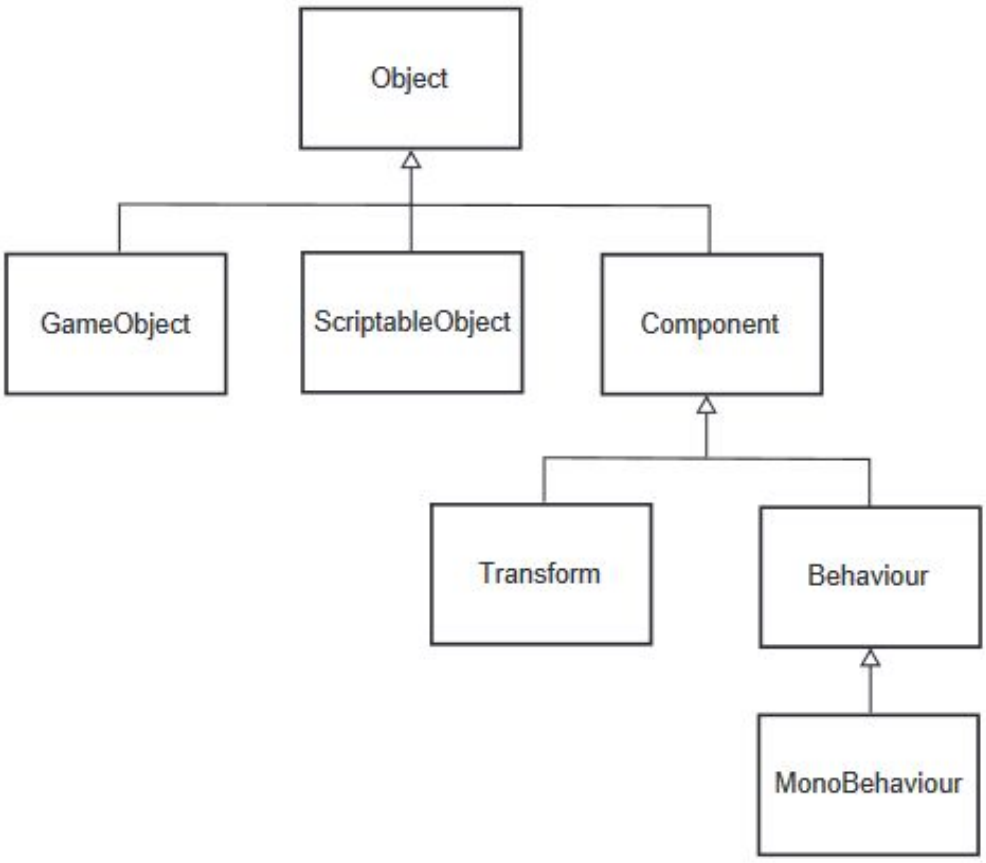
When programming scripts, you can choose between the programming languages JavaScript, Boo and C #. While you can mix the languages in a Unity project any amount (not within a script!), It is recommended, as far as it goes toward it verzichten.An itself is the mixing of language within a project not 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

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.

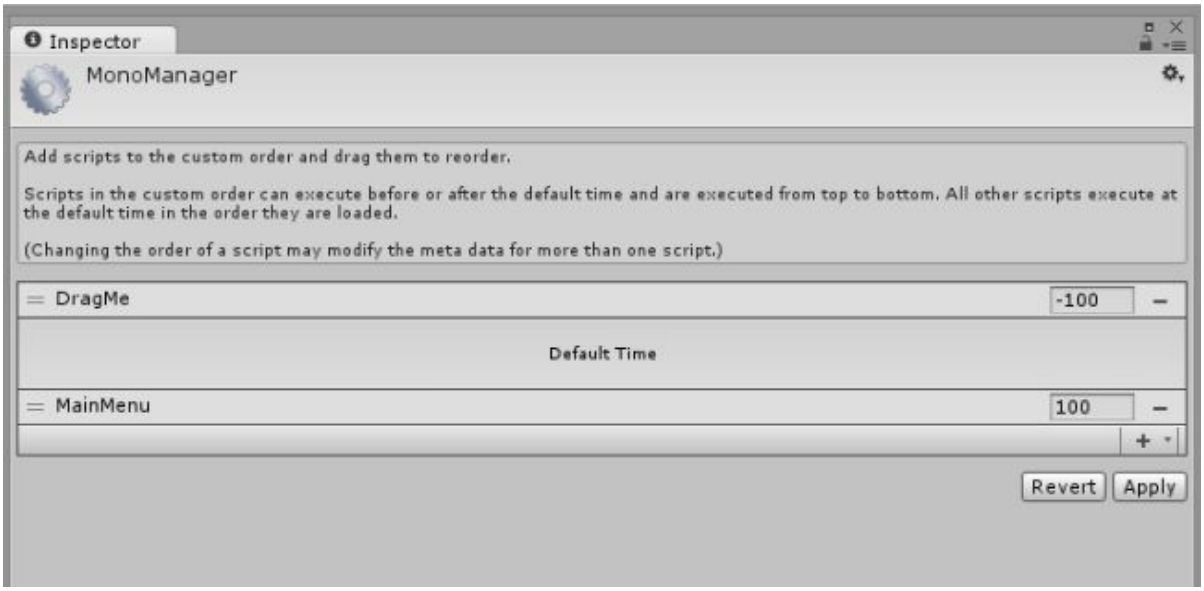
If you want to use your C # script to access a Boo or JavaScript class, the script you want to access must have been compiled in an earlier phase than itself. In this case, you could use the JS script in the Move the plugins folder and it is already accessible from your C # script.

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

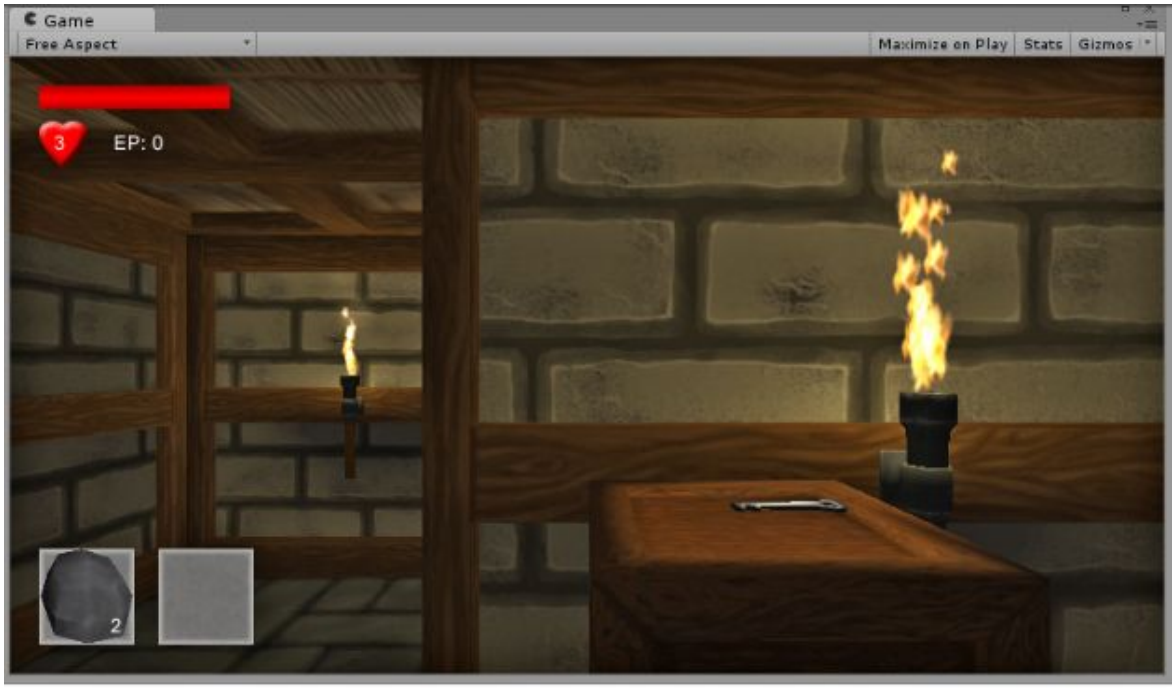
Create a simple 3d game

Now that you have learned the basics of Unity, let's put this knowledge into practice in a game. For this you will find an example game on the enclosed DVD, which should be available in three different versions:

- **Empty example game: This project only contains the various assets such as the 3D models, textures, sound files, etc.**
- **Template ExampleGame: In addition to the assets, this also has a scene with a predefined level design, to which the later descriptions in this chapter are tailored.**
- **Example game: This serves as a demonstration object and shows the example game how it should look in the end.**

In the context of this example game, we want to develop a small, role-playing 3D game from the dungeon crawler genre. This genre is mostly about solving puzzles in a wall (dungeon) and fighting hostile NPCs. Since a lot of different techniques can be used here, this is very good as an illustrative example. Developing a complete dungeon crawler would of course go a little beyond the scope, which is why we will only deal with a single quest here, i.e. with a task, that the player has to solve. In this case, the aim is to find a container to collect dripping drinking water. When searching for this container, the player will come across both enemy bats to fight and locked gates to be opened. that we will first implement the GUI

with GUIElements (GUITexture, GUIText) and OnGUI programming. At the end there is an additional section in which I go back to what needs to be changed in order to use the new uGUI controls that are available from version 4.6, because I only wrote beta versions of the new ones while writing the book uGUI objects and I would like to develop a working example with you, the example will initially be based on the conventional GUI techniques. As soon as the new uGUI system is officially available, I will offer you a second variant for download on my blog, which then uses the new uGUI controls. You can find more information about this in the "Introduction" of this book.



Backup copies and version management

When developing a game, you should of course save your project at regular intervals. It is also advisable to copy the intermediate status of your project again and to save it as a backup in another directory (or even external). It doesn't always have to be a disk crash, it's enough that z. B. a change does not do what you imagined. Then it is advantageous if you can easily switch back to a previous status . For this purpose there is also extra software, so-called version management software (Version Control System), which takes on this task. By copying your project, you can also do this manually in a simple way

Level design

Even if the other sections refer to the template project, I would like to explain briefly below how you can use the various 3D models to design a level yourself.

If you want to use your own designed 3D models, you must of course first import the FBX files, textures etc. into your project. This is very easy to do by dragging and dropping it into the project's Project Browser. It is best to use your own folder to sort the various asset types in the Project Browser and to keep track of things. B. "Materials", "Models", "Scenes" etc. You will also find these in the example game.

General import settings

With the supplied models and textures, you should always use the following import settings everywhere:

- Model / Scale Factor: 1
- Model / Generate Colliders: deactivate d
- Animations / Import animation: deactivate d

However, since some models require special settings, we will now adjust them later.

bat_03 Import Settings

The bat model "bat_03" has its own animations. Therefore, the following settings must also be made in the import settings of this model:

- Animations / Import Animations: activ e

Import settings

The model of the wooden box "crate_01" is larger in scale. So that we can also benefit from dynamic batching (see chapter "Troubleshooting and Performance") with this model , we are not allowed to scale the model in the scene afterwards. That's why we're already changing the scaling when importing. Otherwise, the remaining settings remain the same as for the "General Import Settings".

- Model / Scale Factor: 0.2 5

stone_01 Import Settings

We also want to adjust the size of the 3D model of the stone when importing it. In addition, we want to make the edges of the model a little softer, which is why we recalculate the normals within Unity and adjust them using Smoothing Angle. The rest is set as explained in the section "General Import Settings".

- Model / Scale Factor: 0. 1
- Model / Normals Calculate: activ e
- Model / Smoothing Angle: 10 0

floor_01 Import Settings

So that our player can walk on the floor, we want to automatically add a collider to the floor model "floor_01". Of course there are other options, but in this case we want to work with the automatically generated colliders.

- ModelGenerate Colliders: activ e

Assign material

When importing the FBX files, the associated material is usually created. You can also import textures and assign materials to them. You also have to select the shaders.

- Materials that have normal maps in addition to the textures can use the "Bumbed Diffuse" shader, to which you then assign both graphics accordingly .
- For materials, where you only have the actual texture, then select the "Dif-fuse" shader .

Create prefabs

Instead of simply dragging the models into the scene and creating the dungeon, we want to create our own prefabs from the models beforehand, which have additional components. Only from these prefabs will we then create the dungeon. If we did not proceed in this way, we would have to separately equip each model instance that we drag into the scene with these components, which of course would mean an immense additional effort.

Wall Prefab

For our walls we create a prefab called "Wall":

1. Drag the "wall_01" model into the scene.
2. Add a box collider to the model.
3. Drag the model into the “Prefabs” folder and rename it to “Wall”.

Torch Prefab

Next we want to create a prefab for the wall flares.

4. Drag the objects "torch_01" and "torch_holder_01" into the scene

5. Assign the torch "torch_01" to the torch holder "torch_holder_01" as a child object and position the torch on (0,0,0.13).

6. Now drag the "torch_holder_01" object into the "Prefabs" folder and rename it to "Torch".

Wall_Torch Prefab

Now we want to create a second wall prefab, which, however, also has the wall torch.

1. Copy the prefab "Wall" and rename it to "Wall_Torch".

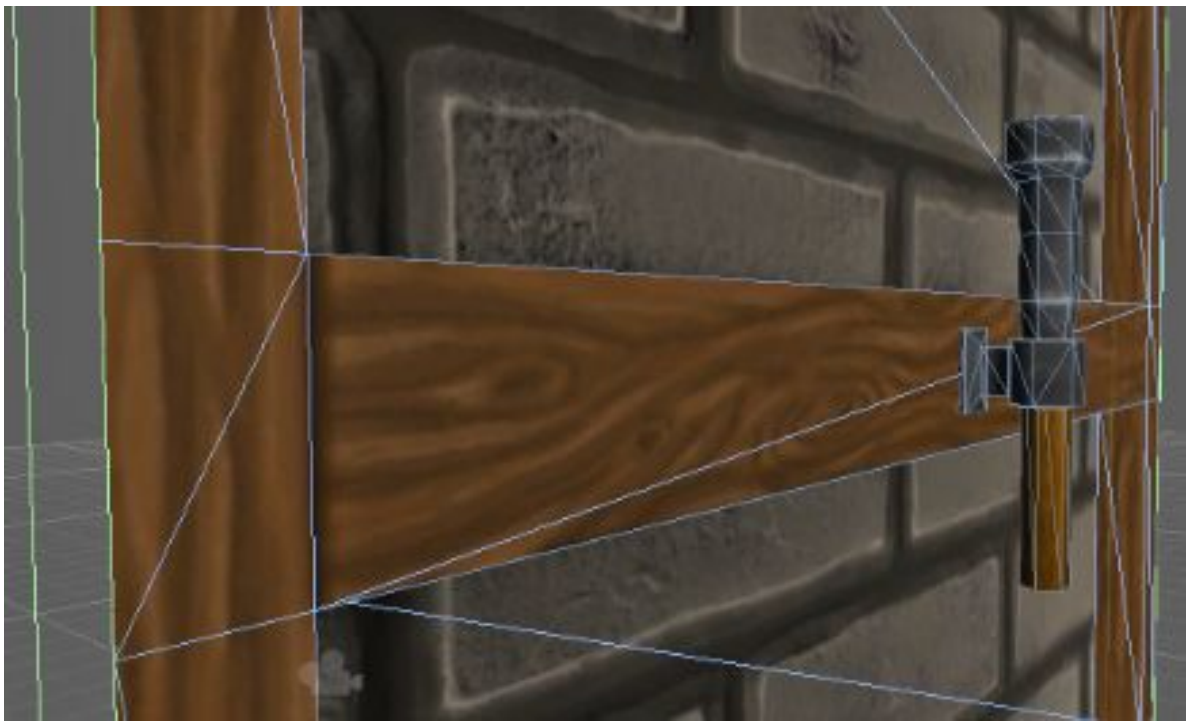
2. Now drag "Wall_Torch" and "Torch" into the scene and add "Torch" as a child object of "Wall_Torch".

Rotate Torch to (-90,90,0). Next we want to position the torch holder on the wooden surface using the vertex snapping:

4. To do this, mark "Wall_Torch" and press the [V] key . You can select a vertex of the object by moving the mouse.

5. If you now have a vertex on the outer edge of the holder, press the left mouse button and slide the torch towards the wall. The torch will jump to the wall and place it on a vertex of Wall. Presumably this will be somewhere on the edge of the wall. It is important here that this is a vertex on the outer edge, so that the torch is not in the wall, but outside. We don't want to achieve more with this and you can release the mouse button and [V] again. When you have positioned the torch holder on the outer surface, you only need to manually position “Torch” on the middle wooden beam. Therefore, change from “ Torch ”manually set the X position to“ 1 ”and the Y position to“ -1 ”.

6. Then confirm the changes with Apply so that the "Wall_Torch" prefab accepts all changes.



Floor Prefab

We also want to create a prefab for the flooring. Even if this does not differ in the first step from the normal model prefab, we do this in order to be more flexible in the future. If you want to add to the model you added later,

you can only transfer these adjustments to the other instances if you are using a prefab, but not if you drag the model prefab into the scene.

7. Drag “floor_01” into the scene.

8. Drag this directly into the “Prefabs” folder without making any adjustments and rename it to “Floor”.

Ceiling Prefab

Finally, we create a prefab for the ceiling model. We do this for the same flexibility reason as with flooring.

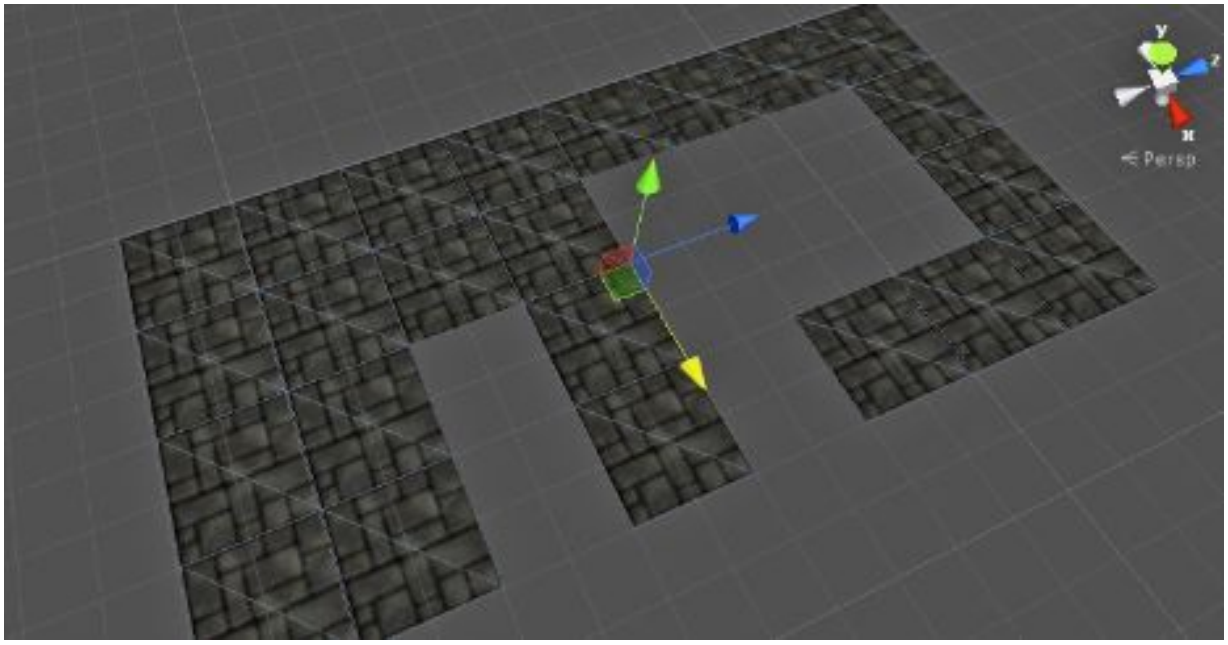
9. Drag “ceiling_01” into the scene. 10. Drag this directly into the "Prefabs" folder without changes and rename it to "Ceiling".

Create a dungeon

With the help of the various prefabs, we can now develop the dungeon in which the game is to take place. To do this, first create a new scene in the "Scenes" folder, e.g. B. with the name "Dungeon". So that we do not lose the overview later with the many GameObjects in the hierarchy, we first create three empty GameObjects in the scene via GameObject / Create Empty , which you can find in "Floors", "Walls" and rename "Ceilings". These will later serve as containers for the respective prefab instances, which you can simply fold in and out as you wish. You can reset all three container objects to their default values using the reset function in the Inspector .

Create floor

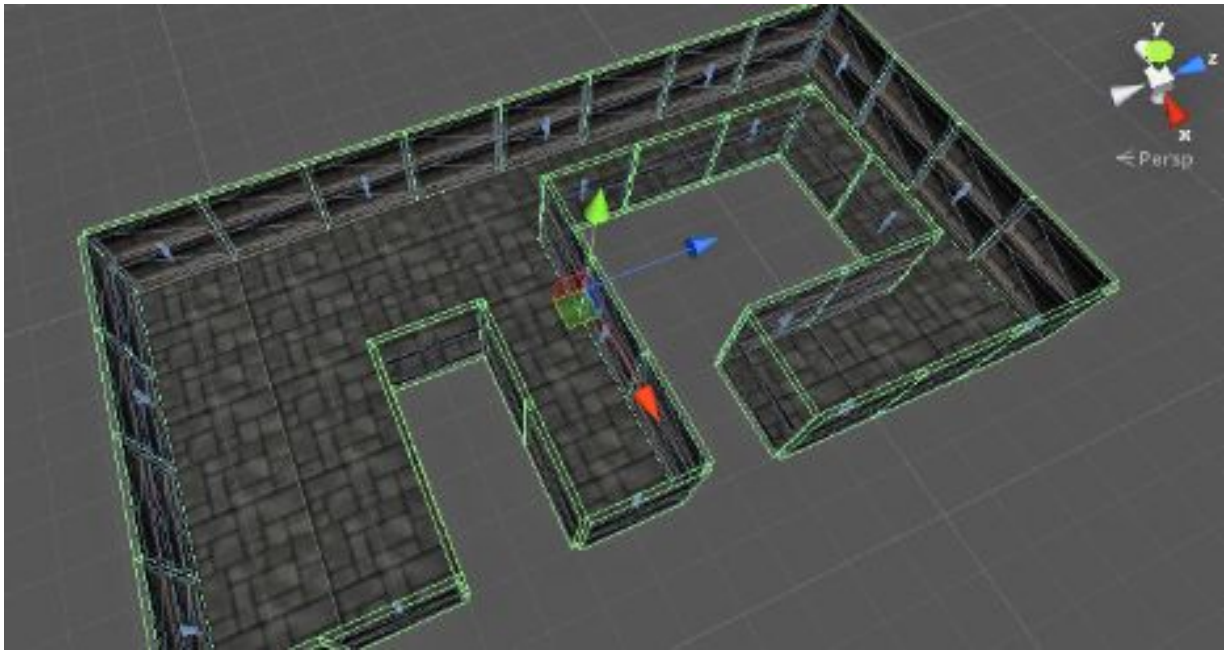
To create the floor area, first drag the floor prefab into the "Floors" container and reset it to the default values with Reset . Now rotate the Floor instance to (-90,0,0) so that it is positioned like a floor slab. And now we have the first base plate ready, copy this plate as often as you like and place it side by side. It is easiest to always copy a single disk, e.g. B. with [Ctrl] + [D] , and this immediately to the new position, z. B. (0,0,2), before you take on the next floor plate. I would always do the repositioning via the position parameters in the Inspector, as this is the fastest and most accurate. In this way, you can now gradually determine where the corridors and halls of your dungeon should be later



Create walls

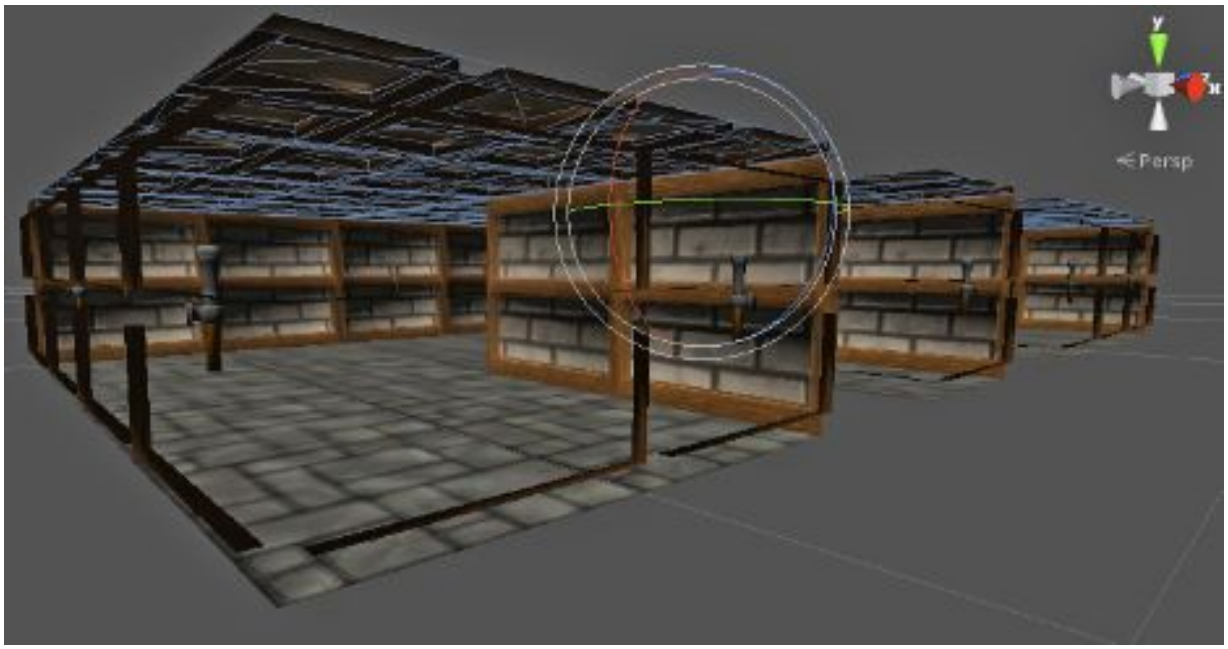
To create the walls, add both a “Wall” instance and a “Wall_Torch” instance to the “Walls” container and reset both to the standard values with Reset. Now position them upright on the outer ones Floor panels so that the visible surfaces point inwards towards the floor surfaces, e.g. B. on positions (0,2, -2) and (2,2, -2). Now copy these instances again as with the "Floor" instances and arrange them as you wish on the outer edges of the floor slabs so that at the end the entire floor area is framed. Since the torches of the "Wall_Torch" instances are responsible for the lighting later you should

make sure that enough instances are distributed in the level so that all corners are illuminated.



Create ceiling

Finally the ceiling of the dungeon has to be built. The procedure is almost identical to that of the floor surfaces, except that the visible side is naturally facing down. Add a "ceiling" instance to the "Ceilings" container and reset it with a reset . Now move it to position (0,2,0) and rotate it with the rotation parameter to (90,0,0) so that it is exactly above a "Floor" instance at a distance of the height of a Wall instance Now copy this instance and move it around until finally the entire dungeon is completely enclosed by the floor, wall and ceiling elements.



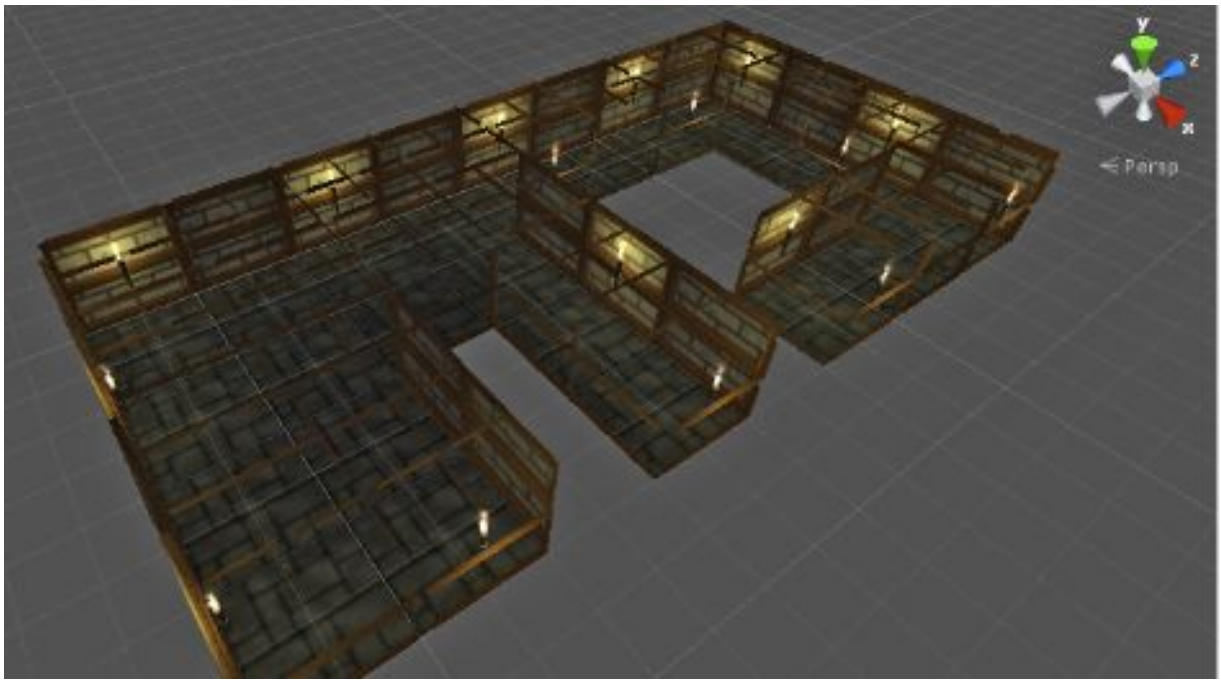
Add fire and light

Now create the particle effect of a torch fire as described in the chapter "Particle Effects with Shuriken" and create a prefab called "TorchFlames" from it. Here it is advantageous if you create a new test scene in which you develop this particle effect. If you have now created the particle effect, drag another “Wall_Torch” prefab into the scene. Now drag the “TorchFlames” object onto the “Torch” object of “Wall_Torch” so that the flame becomes a child of the torch. Position the flame directly above the torch (-0.1,0,0.35). Now mark “Wall_Torch” in the scene and press Apply in its Inspector so that the adjustments are transferred to the prefab. You can now save the test scene and open your “Dungeon” scene. As you will see, all “Wall_Torch” instances of your dungeons scene have also received the flames. Switch back to the previous test scene in which you created the particle effects and complete the torch. To do this, add a new PointLight to the scene via GameObject / Create Other / Point Light . You also drag this to "Torch" from "Wall_Torch" and position it there on (0,0,0.4) so that the PointLight is directly in the flame. Now you also define the following parameter values of the Light component:

- Color: R 254, G 254, B 139, A 25 5

- Range: 3
- Intensity: 1

Finally select "Wall_Torch" again and press Apply in the Inspector of the Prefab instance. Now you can switch back to your "dungeon", where the walls are illuminated by the PointLights. Basically we are now finished. However, I would like to increase the ambient light of this scene at this point so that we get a certain basic brightness in the entire dungeon. To do this, go to Edit / Render Settings in the main menu and click on the color bar of the Ambient Light parameter. There you set the R, G and B values to 100 each. Leave the alpha channel at 255.



Create decoration

A scene doesn't necessarily only include game-relevant objects. Usually a lot of objects are placed there for purely decorative purposes. In our game we also have some objects for this purpose that we can place in the dungeon. In order to always have an overview, we also want to group all decorative objects in a common container object. Again create an empty GameObject and this time name it “Deco”. Then reset these container values with Reset .

Crate Prefab

First of all we want to create a prefab for the already imported wooden box.

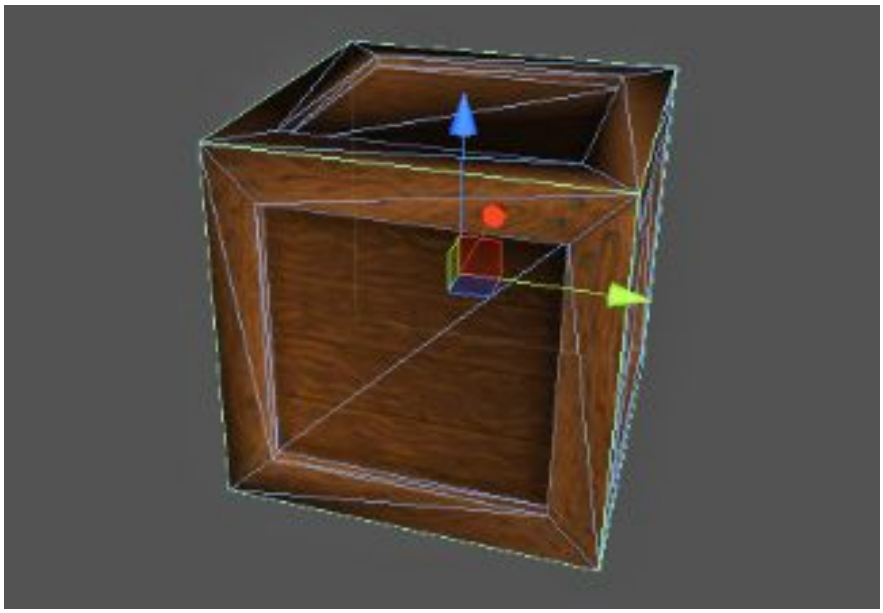
1. Drag the “crate_01” model into the scene and rename it to “Crate”.

2. Check the scaling that it is (1,1,1).

3. Check the assigned material. This should be “placeables_01_diff”, which uses the texture of the same name and has set the “Diffuse” shader.

4. Add a Box Collider to the GameObject.

5. Now create a prefab from this by dragging the GameObject into the "Prefabs" folder.



Barrel Prefabs

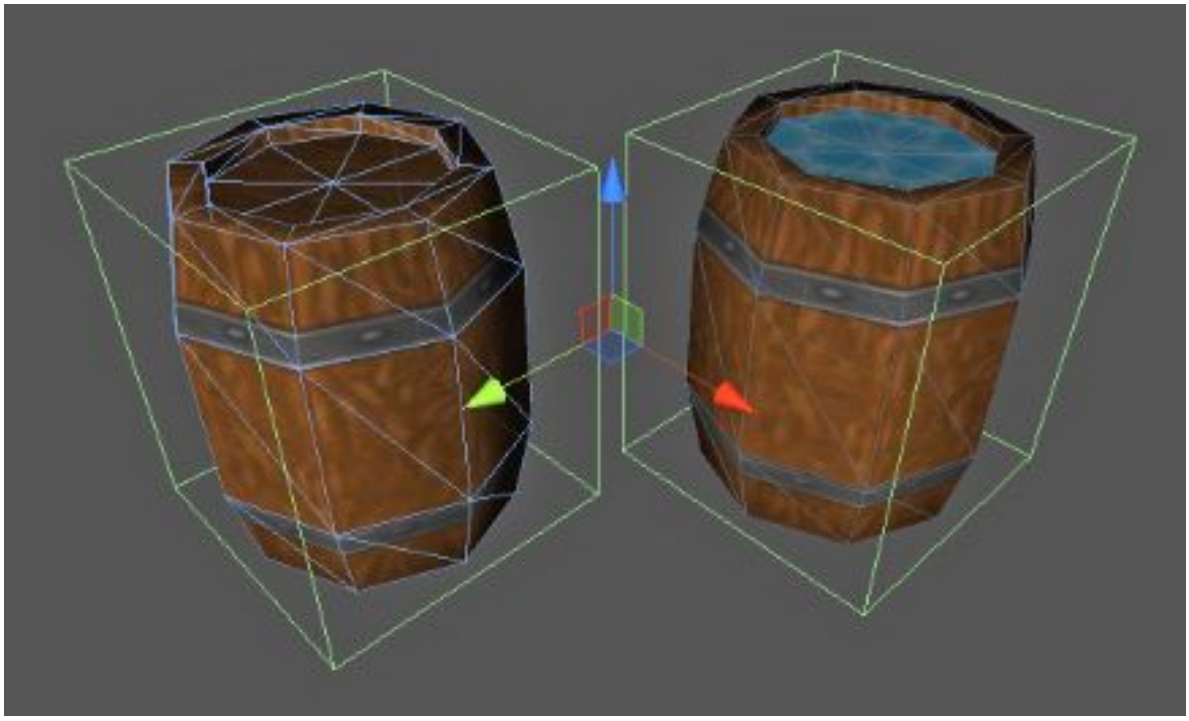
Now we want to create our own prefabs from the imported barrels. Here we have two different ones: a closed barrel and one that is filled with water.

1. Drag the model "barrel_01" into the scene and rename it there "Barrel".

2. Check the assigned material here too. This should also be "placeables_01_diff", which uses the texture of the same name and has set the "Diffuse" shader.

3. To simplify matters, add a Box Collider to the GameObject as well. To make it more precise, you can instead activate Generate Collider in the import settings of this model.

4. Now create a prefab from it by dragging the GameObject into the “Prefabs” folder. Now do the same with the model “barrel_01_water” and simply name it “Barrel_Water”.



Create an inventory system

A core element of most role-playing games is an inventory system that is used to manage items that the hero finds in the course of the game. Most of the time it also plays an important role in solving quests, where it is often about finding certain items and taking them to a person or a specific place. In our example game, too, the player will have to find various items that will ultimately be managed with this inventory system.

Management logic

An object of the generic class Dictionary <TKey, TValue> , which you already got to know in the chapter "C # and Unity" , should serve as the core element of our inventory system. A dictionary manages so-called key-value pairs. Each data record has a unique identifier, the value of which can be changed as required. In our case, this means that the name of the item could be used as the key and the number of individual items is taken as the value. If a new inventory item is now added to the dictionary, you simply need to increase the value of the respective key by one. I already presented such an example in the “C # and Unity” chapter.

ItemProperties

In our case, we want to go a little further. Because we don't just want to count the items, we also want to display them in the GUI of our game. For this we need another value that must be saved in the dictionary, namely the graphic associated with the object. For this reason, we cannot simply take the number as a value, but we have to create our own type that contains both the number and the graphic of the item. For this we create a small class that we call ItemProperties . It only has a Texture2D variable and an int variable that saves the two parameters mentioned for an inventory item. Since we won't be using the class as a component, it doesn't inherit from MonoBehaviour . The content of this class can be found in the listing, the complete script can be found in the section “ItemProperties.cs”.

```
public class ItemProperties {
   public Texture2D texture;
  public int quantity;
}
```

Inventory

With the help of the ItemProperties class , we can now create the dictionary object for our inventory management , which we will call items in our case . In addition, we need two array objects with which we can display the contents of the inventory in the GUI. The first array is used to display the graphics, the second shows the quantities. Thanks to the arrays, you will be more flexible later on how large your inventory should actually be or how much the GUI can display.

```
public GUITexture [] guiItemTextures;
public GUIText [] guiItemQuantities;
private Dictionary <string, ItemProperties> items = new Dictionary <string,
ItemProperties> ();
```

Please note that in order to use the Dictionary class you first have to include the associated namespace with using System.Collections.Generic. To display the content now, we will develop a small method that will later be called every time the Inventory has changed something and this should be updated in the GUI. This runs through all the items of the arrays and fills them with the respective contents after all have first been initialized with blank values.

```
void UpdateView () {
   int index = 0;
   int guiCount = guiItemTextures.Length;
    for (int i = 0; i <guiCount; i ++) {
   guiItemTextures [i] .texture = null;
        guiItemQuantities [i] .text = "";
  }
  foreach (KeyValuePair <string, ItemProperties> current in items) {
```

```
  guiItemTextures [index] .texture = current.Value.texture;
     guiItemQuantities [index] .text = current.Value.quantity.ToString ();
      index ++;
  }
}
```

This is also the first thing to do once the script starts.

```
void start () {
   UpdateView ();
   }
```

Now come the actual methods you use to fill and remove the inventory. We want to create two methods for this. The AddItem method is used to add a new item. The name, i.e. the key of the data record, as well as the texture through which the item is to be displayed in the GUI are transferred to it.

The method then checks whether the item is already in the dictionary and then counts the amount up. To ensure that we do not include more items in our inventory than we can display, a check is made before creating a new dictionary data record to determine whether all GUI objects are already occupied by item graphics. You can find out more about this point in the section "Interface of the inventory system".

```
public bool AddItem (string itemName, Texture2D texture) {
    if (! items.ContainsKey (itemName)) {
      if (items.Count <guiItemTextures.Length) {
       ItemProperties ip = new ItemProperties ();
      ip.texture = texture; ip.quantity = 1;
      items.Add (itemName, ip);
```

```
    UpdateView ();
  return true;
}
 else {
      return false;
 }
  } else {
 items [itemName] .quantity + = 1;
    UpdateView ();
return true;
   }
  }
```

Finally, there is the RemoveItem method, which ensures that an item is removed from the inventory. We use a Boolean return value to signal whether this was successful or whether the requested item was still in the inventory at all.

```
bool RemoveItem (string itemName) {
   if (items.ContainsKey (itemName)) {
   if (items [itemName] .quantity == 1)
         items.Remove (itemName);
    else items [itemName] .quantity - = 1;
     UpdateView ();
  return true;
}
else return false;
}
```

The complete script can be found in the “Inventory.cs” section.

InventoryItem

Finally, we want to develop a script that is attached to every item that should be added to the inventory. Here we define three public variables for specifying the item name, the texture that is to be displayed in the GUI, and an audio clip that is to be played when it is picked. We also need two private variables that buffer the inventory script and the player character (or its transform).

```
string itemName = "";
public Texture2D texture;
public AudioClip picSound; private Inventory inventory;
private transform player;
```

The two private variables are then assigned in the Start method

```
void start () {
  player = GameObject.FindGameObjectWithTag ("Player"). transform;
   inventory = GameObject.FindGameObjectWithTag ("Inventory").
GetComponent <Inventory> ();
}
```

Please note that we later have to assign the GameObject with the inventory script the tag "Inventory" and the player the tag "Player". In the next step, we want the GameObject to which this script is attached to also be assigned to the inventory is added. We want to do this with a click. As soon as someone clicks on this item with the mouse, the item should be removed from the game world and moved to the inventory. To underline the process, the picSound is played.

```
void OnMouseDown () {
   if (inventory.AddItem (itemName, texture)) {
```

```
        if (picSound! = null) AudioSource.PlayClipAtPoint (picSound,
player.position);
        Destroy (gameObject);
    }
  }
```

ItemProperties .cs

The ItemProperties script developed in the previous section then looks like the listing below.

```
using UnityEngine;
using System.Collections;
   public class ItemProperties {
        public Texture2D texture;
   public int quantity;
}
```

Inventory .cs

The script inventory is the heart of this inventory system and looks like the following listing.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
  public class inventory: MonoBehaviour {
  public GUITexture [] guiItemTextures;
      public GUIText [] guiItemQuantities;
      private Dictionary <string, ItemProperties>
      items = new Dictionary <string, ItemProperties> ();
void Start () {
```

```
        UpdateView ();
}
       public bool AddItem (string itemName, Texture2D texture) {
        if (! items.ContainsKey (itemName)) {
  if (items.Count <guiItemTextures.Length) {
    ItemProperties ip = new ItemProperties ();
           ip.texture = texture;
           ip.quantity = 1;
           items.Add (itemName, ip);
     UpdateView ();
  return true;
  }
       else {
  return false;
   }
  }
  else { i
      tems [itemName] .quantity + = 1;
      UpdateView ();
   return true;
   }
}
ublic bool RemoveItem (string itemName) {
  if (items.ContainsKey (itemName)) {
   if (items [itemName] .quantity == 1) items.Remove (itemName);
  else items [itemName] .quantity - = 1;
   UpdateView ();
return true;
```

```
}
   else return false;
}
void UpdateView () {
   int index = 0;
         int guiCount = guiItemTextures.Length;
  for (int i = 0; i <guiCount; i ++) {
   guiItemTextures [i] .texture = null;
        guiItemQuantities [i] .text = "";
  }
    foreach (KeyValuePair <string, ItemProperties> current in items) {
        guiItemTextures [index] .texture = current.Value.texture;
                    guiItemQuantities [index] .text =
current.Value.quantity.ToString ();
      index ++;
  }
}
}
```

InventoryItem .cs

In the next listing you can see the script last developed for adding objects to the game.

```
using UnityEngine;
using System.Collections;
    public class InventoryItem: MonoBehaviour {
         public string itemName = "";
         public Texture2D texture;
```

```
    public AudioClip picSound;
    private inventory inventory;
    private transform player;
  void start () {
   player = GameObject.FindGameObjectWithTag ("Player"). transform;
        inventory = GameObject.FindGameObjectWithTag ("Inventory").
GetComponent <Inventory> ();
}
     void OnMouseDown () {
        if (inventory.AddItem (itemName, texture)) {
       if (picSound! = null) AudioSource.PlayClipAtPoint (picSound,
player.position);
                  Destroy (gameObject);
          }
   }
 }
```

Interface of the inventory system

To display the contents of this inventory system, you need two GUI objects per item: a graphic GUI object to display the item graphic and a text-oriented GUI object to display the quantity. In our case, we want to keep it simple and the inventory system at the bottom left show in game. The only thing that is added to the above objects is a background graphic for each item, which remains permanently displayed to show the location of the inventory system. Overall, we will have to do with three different item types in the game (see section "Inventory Items"), whereby due to the course of the game at most two types can be in the inventory at the same time. Since we therefore only need two item slots, we only need it accordingly

- two texture elements for the background s

- **two texture elements for the item graphic s**
- **two text elements to represent the quantitie s**

If you want to expand the game later, you will need more seats. But since we are working with arrays in the inventory script, nothing stands in the way of expanding the inventory GUI. In the import settings of these item graphics ("key", "stone" and "bucket") you should now add the texture Set Type to GUI and Max Size to 64. But before we start with the actual GUI objects, create an empty GameObject beforehand, which you can reset to the default values and rename to "Inventory". Now add the Inventory script of the same name that we created above. Now create a new tag "Inventory" in the tag list and assign it to the GameObject "Inventory". First, create two GUIText objects and four GUITexture objects and add them to the GameObject "Inventory" as child objects Rename the four GUITexture objects "Tile1-Bg", "Tile2-Bg", "Tile1-Value" and "Tile2-Value" . Rename the two GUIText objects to “Tile1-Quantity” and “Tile2-Quan-tity”. Make sure that the GUITexture objects have the scaling (0,0,1).

Assign the texture "inventory-tile" to the texture properties of "Tile1-Bg" and "Tile2-Bg". Set the Texture Type to GUI and the Max Size to 64. Then set the position of both objects to (0,0, -1) so that they are arranged further back. Set the position for “Tile1-Value” and “Tile2-Value” to (0,0,0) so that they appear in front of the two background textures. Now set the pixel inset values of these four objects. For “Tile1-Bg” and “Tile1-Value”, set X 20, Y 20, W 64, H 64. For “Tile2-Bg” and “Tile2-Value”, set X 100, Y 20, W 64, H 64. Finally, the two text objects "Tile1-Quantity" and "Tile2-Quantity" must be equipped with the correct parameters. The quantities of the respective items should always be shown at the bottom right of the respective graphic. Therefore set the following values for both objects:

- **Position: (0,0,1) .**
- **Anchor: upper righ t**
- **Alignment: righ t**

Finally, set the pixel offset values that determine the positions of the texts. For "Tile1-Quantity" set X 75, Y 40 and for "Tile2-Quantity" X 155, Y 40. Now set the size of guiItemTextures and guiItemQuantities from the inventory script to 2. Then add "Tile1-Value" and "Tile2-Value" in the order corresponding to the guiItemTextures -Array and "Tile1-Quantity" and "Tile2-Quantity" to the guiItemQuantities -Array. You can now test the text property of the two "Quantity" objects assign different numerical values and assign item graphics (eg the "key" texture) to the texture properties of the "Value" objects. If you look at the Game View without starting the game, you should now see the inventory at the bottom left.

Inventory items

In our game we are dealing with three different item types. We want to create these in this section.

HoverEffects

Before that, we want to program a small script that should display an information text as soon as the player points to an object with the mouse. In addition, the player should be informed whether he can pick up this object and add it to his inventory. We want to achieve this by changing the cursor.

Since the information text should always appear directly with the respective item , there are now two different versions of the implementation. Either we place the text object directly in the three-dimensional world with the respective object (e.g. with a 3D text component), or we use a normal GUI object, which is then positioned in the two-dimensional pixel coordinate system of the GUI The advantage of the latter method is that the font would always be the same size, regardless of how far away an object is. And that's why we want to take this procedure too. So first we will create a general GUIText object that we will always want to place later at the respective position where an information text is to be displayed.

1. First create a GUIText object via GameObject / Create Other / GUI Text , which you call “TooltipText”.

2. Position the GameObject on (0,0,0).

3. Now create a new “Tooltip” tag and assign it to the GameObject “Tooltip Text”.

4. Set Anchor to "middle center" and Alignment to "center".

5. Remove the assigned default text.

6. If you want, you can also assign a different TrueType font or OpenType font to the Font parameter and use the Font Size, Font Style and Color properties to adjust it as you wish.

Now we create a new script called "HoverEffects". First, we need a reference to the GUIText object or the component and a variable for the information text. We also want to change the cursor. Since we want to use a different cursor than the standard operating system pointer in our game anyway, we are now using two variables: standardCursor and clickableCursor . With an additional variable hotSpot we also specify the position of the cursor, which however simply remains at zero.

public Texture2D standardCursor;

```
public Texture2D clickableCursor;
public string tooltipText;
private Vector2 hotSpot = Vector2.zero;
  private GUIText msgBox;
     void start () {
   msgBox = GameObject.FindGameObjectWithTag ("Tooltip").
GetComponent <GUIText> ();
        msgBox.transform.position = Vector3.zero;
  }
```

The MouseEnter method, which is triggered as soon as the mouse is over this object, looks like this.

```
void OnMouseEnter () {
   if (clickableCursor! = null)
         Cursor.SetCursor (clickableCursor, hotSpot, CursorMode.Auto);
        msgBox.text = tooltipText;
        Vector2 pos;
        pos.x = Input.mousePosition.x;
        pos.y = Input.mousePosition.y + 20; // we put the text a little over the pointer.
        msgBox.pixelOffset = pos;
}
```

Exit the effect we are spinning off into a separate method because we the code to reset the cursor in both OnMouseExit and in OnDestroy want to call

```
void OnMouseExit () {
   ExitEffect ();
  }
void OnDestroy () {
        ExitEffect ();
```

```
}
void ExitEffect () {
	if (standardCursor! = null) Cursor.SetCursor (standardCursor,
hotSpot, CursorMode.Auto);
	if (msgBox! = null) msgBox.text = "";
}
```

Default cursor

And while we're at the cursor, let's change the game's general default cursor. Instead of the normal cursor, we want to display our own pointer in the form of a sword in the game. We will later assign this image to the standard cursor variable of the HoverEffects script. To assign the default cursor, go to Edit / Project Settings / Player and assign the graphic "icon_01_32x32" to the parameter "Default Cursor"

key

Now we can take care of our items. First, we want to create a key that we will need later to open a locked gate.

1. Drag the model "iron_key_01" into your scene and rename the GameObject to "IronKey".

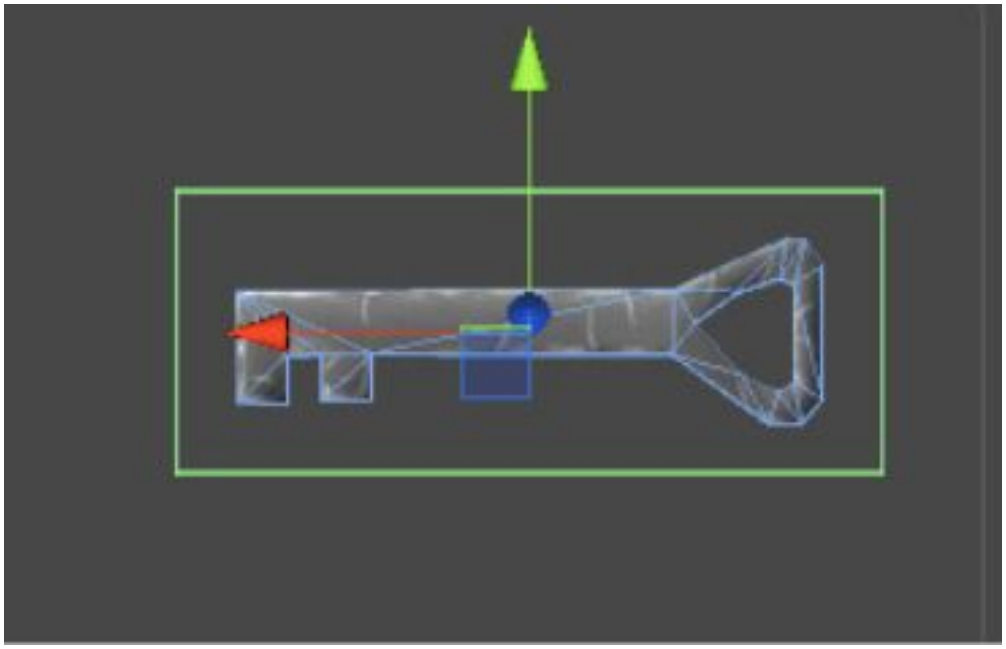
2. Assign it the material "placeables_01_diff", which uses the texture of the same name with the shader "Diffuse".

3. Now add a box collider to it and slightly enlarge this collider (not the GameObject!) Over its size in all directions, e.g. B. on (0.2,0.08,0.02). Since the key is quite small, we increase the contact area of the key so that the player can later select it more easily.

4. Now add the HoverEffects script to the key. Assign standard cursor "icon_01_32x32" clickable cursor "icon_02_32x32" and toolTipText "Iron Key" to. Check the import settings for both textures. The Texture Type should be on "Cursor", the Wrap Mode on "Clamp" and the Max Size on 32.

5. Finally add the InventoryItem script to the key. itemName is “Key”, texture assign the texture “key” and as picSound take “picUp”. You should also check the import settings for the “key” texture. Here the Texture Type should be "GUI" and the Max Size should be 64.

6. Finally, drag the completed GameObject into the “Prefab” folder to create a prefab.

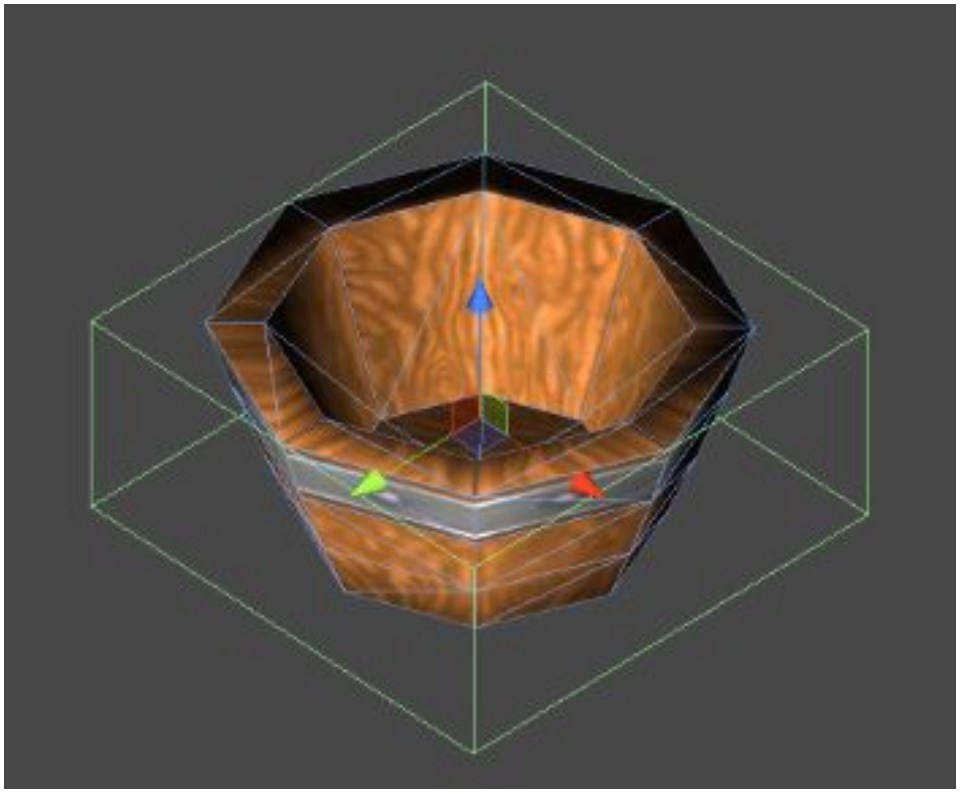


Water tank

Do the same with some changes now with the "bucket_01_empty" model. I want to briefly explain these changes:

- The name of the GameObject is changed to "BucketEmpty" .
- With HoverEffects you assign “water tank ” as toolTipText .
- For InventoryItem, assign “Bucket” as the itemName and “bucket” as the texture .

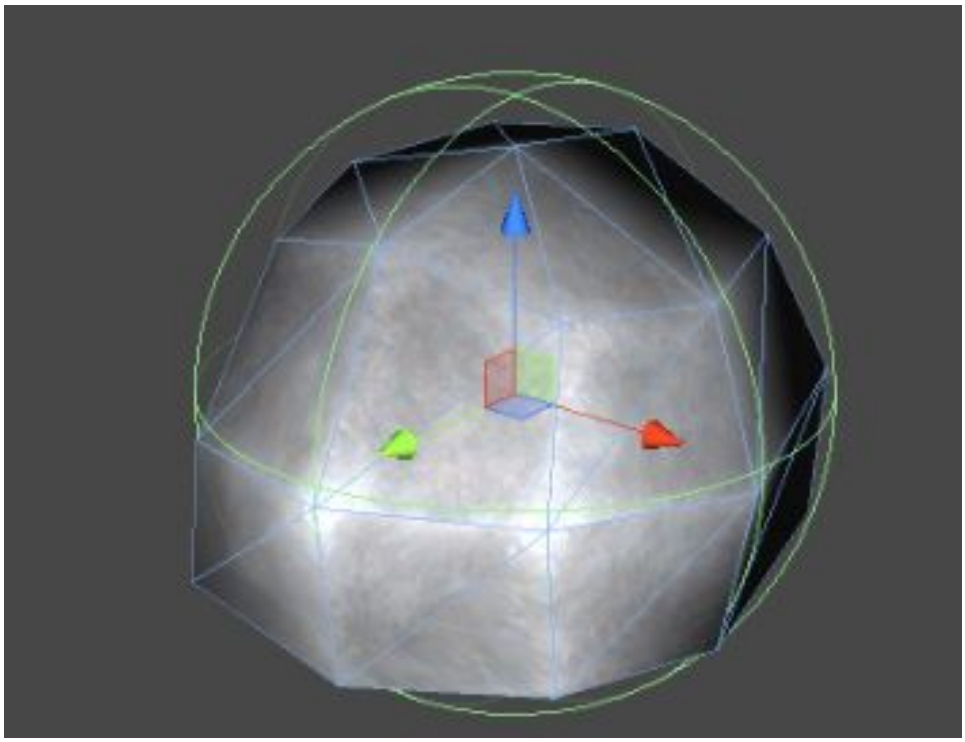
- Since we need this container for other purposes, we add a rigid body to the GameObject .
- Do not increase the collider size on this object !



stone

And finally, do this with the "stone_01" model. However, there are also some differences that I would like to briefly explain:

- The name is changed to "Stone" .
- Instead of the Box Collider, this time add a Sphere Collider and reduce its radius to 0.08 .
- With HoverEffects you assign toolTipText the text " Throwing stone " .
- With InventoryItem you assign "Stone" as itemName and "stone" as texture .
- You do not need to change the collider size here either .



Remember that you always create a prefab from each of the three inventory items.

HoverEffects .cs

You can assign the HoverEffects script to any object for which information is to be displayed in the GUI, including GameObjects that cannot be added to the inventory. Assign this text information to the tooltipText variable. You can also use the two Texture2D variables to signal that an object is clickable and e.g. B. can be added to the inventory. For this you should assign a cursor-compatible texture with a hand or a similar symbol to clickableCursore.

```
using UnityEngine;
using System.Collections;
  public class HoverEffects: MonoBehaviour {
```

```
    public Texture2D standardCursor;
    public Texture2D clickableCursor;
    public string tooltipText;
    private Vector2 hotSpot = Vector2.zero;
    private GUIText msgBox;
void Start () {
        msgBox = GameObject.FindGameObjectWithTag ("Tooltip").
GetComponent <GUIText> ();
        msgBox.transform.position = Vector3.zero;
  }
 void OnMouseEnter () {
      if (clickableCursor! = null) Cursor.SetCursor (clickableCursor,
hotSpot, CursorMode.Auto);
                msgBox.text = tooltipText;
                Vector2 pos;
                pos.x = Input.mousePosition.x;
                pos.y = Input.mousePosition.y + 20; // to display text a little
higher.
                msgBox.pixelOffset = pos;
   }
  void OnMouseExit () {
       ExitEffect ();
  }

void OnDestroy () {
      ExitEffect ();
}
  void ExitEffect () {
```

```
  if (standardCursor! = null)
crsor.SetCursor (standardCursor, hotSpot, CursorMode.Auto);
if (msgBox! = null) msgBox.text = "";
}
}
```

Game controller

In a game there are always functionalities that cannot be specifically assigned to a game object, but relate to the entire game or at least to the current scene. A good example of this is a pause menu or background music. Such components are usually assigned to a GameObject called "Game Controller". "Game controllers" are typical components of a game, which is why Unity already provides a "GameController" tag by default in order to be able to identify them more quickly. Since we also need a "Game Controller" in our game, create one now New GameObject with the name "Game Controller" and assign the tag "GameController" to it. First, you transfer an AudioSource to it, in which you activate both Play On Awak and Loop. Now assign the "backgroundMusic" file to the AudioClip property and set the volume to 0.3 for the time being, which you can later adjust to suit your taste. So that the background music is the same volume everywhere always evenly distributed over both speakers, we deactivate the parameter 3D sound in the import settings of "back-groundMusic".

Create players

Next, let's take care of the player. Because of the size, I'd like to break this down into three parts. First, we're going to equip the player with its main components. Then we will develop the management of life, which will also be important for our future opponent. Only then will we take care of the actual player controls and the attack.

1. First of all we add a new Empty GameObject to our scene and call it “Player”. So that we can find it more easily later by code, we also give it

the tag "Player". Unity provides this tag by default, so we don't need to worry about creating this tag any further.

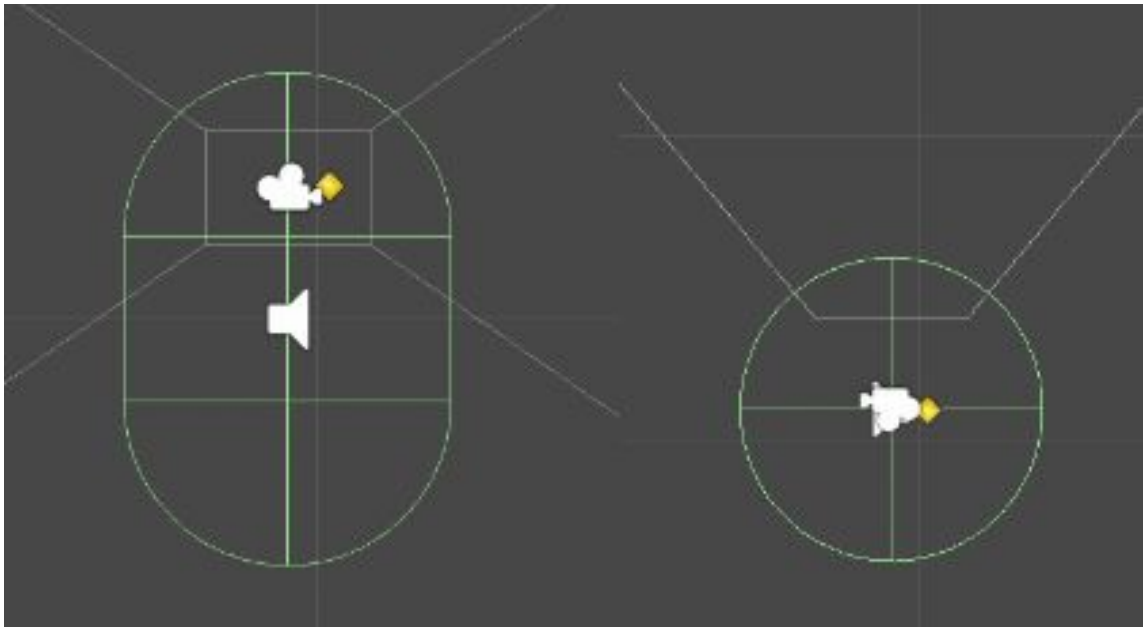
2. Now we add a character controller component to the “player” via Component / Physics / Character Controller , which we will control later using a separate script. Set the radius to 0.5, height to 1.5.

3. Then add an AudioSource to the GameObject, via which we will later play all voice samples of the player. You can then adjust the volume according to your needs. Make sure that Play On Awake and Loop are deactivated.

4. Now add the "Main Camera" to the "Player" as a child object and reset it to the default values with Reset . To do this, select “Main Camera” and press Reset in the transform context menu. Then adjust the position to position (0,0,4,0).

5. Now create another Empty GameObject and name it “SpawnPoint”. This object will later be used as a position for throwing weapons. For a better representation, you can give the "SpawnPoint" its own icon in the Inspector of the GameObject, e.g. B. a yellow diamond.

6. Add the "SpawnPoint" object to the "Player" as a child object. Reset the “SpawnPoint” with Reset and then move it to the position (0.125,0.4,0) so that it is right next to the camera.



7. Now turn the “SpawnPoint” very slightly with a rotation of (0,357.5,0) to the left. Since the objects to be thrown will later orientate themselves towards the direction of this object, we enable the player to throw them more easily and precisely.

8. Finally add an AudioSource to the “SpawnPoint”. This will be responsible for generating the throwing noise. Deactivate Play On Awake and Loop again and add the AudioClip “throwing” to the AudioSource.

9. So that the player can still see something in the darker corners of the dungeon, we now add a separate light source to the "player". To do this, create a Point Light Object and add this to the “Player” as a child object.

10. Center the point light with the reset function and reduce the intensity to 0.5.

That was it with the configuration of the "Player" GameObject. Now place your hero in the dungeon and start the game once to test the functions. If everything works, you can drag your player prototype into the "Prefabs" folder to create a prefab. If you later want to add more levels / scenes to the game, this will make it easier for you to reuse your hero. Next, we want to take care of programming life management and player controls. Once you have developed this, you should of course also add it to the "Player" prefab

Life management

In this game I would like to create a life management system with you, which does not only manage the simple health of the player. The player should also have several life points, so that the game is only over when all lives have been used up.

GUI elements

But first we want to create the GUI elements for this life management.

1. First we create a new Empty GameObject, which we simply call "GUI". This should serve as a container object for the GUI objects.

2. Reset the GameObject "GUI" to the default values with the Reset function .

3. Immediately create another GUIText object that you name "LpText". This should serve to display the life points. Add “LpText” as a child object to the “GUI” container object. Now position “LpText” on (0,1,0) and a pixel offset (33, -49).

4. To make the value in the GUI a little more appealing, we want to underlay the value with a more appealing graphic. To do this, create a GUITexture object and name it “LpBg”. Set position to (0,1, -1) and scale to (0,0,1). Set Pixel Offset to (20, -75,32,32), with the last values being W and H. Assign texture to the graphic "lifepointHeart", whose texture type should be "GUI" and Max Size 32.

5. In addition to the life points, we also want to display the current state of health. To do this, create a GUITexture object that you call “HealthBar”, set Position to (0,1,0) and Scale to (0,0,1). Since this also belongs to the GUI, add this GameObject to the "GUI" container as well. Set the pixel offset of "HealthBar" to (20, -35,128,16) and assign the graphic "healthbar" to Texture. Set the Texture Type from "healthbar" back to "GUI" and the Max Size to 128.

MessageText

In order to be able to implement text outputs such as "Game Over" or the like later, we need an additional GUIText object. For better identification, we also want to equip this with the "Message" tag

1. Create a GUIText GameObject via GameObject / Create Other / GUI Text (or GameObject / Create General / GUI Text) and name it “MessageText”.

2. Drag it onto the “GUI” container object to make “MessageText” a child object of “GUI”.

3. Now center the "MessageText" using the position (0.5,0.5,0).

4. Now center the text yourself within the GameObject with the anchor “middle center” and the alignment “center”.

5. Remove the default text.

6. Finally, create a new "Message" tag that you assign to the GameObject "MessageText".

LifePointController

But let's get back to the actual management of life. For this we first need a new script called LifePointController , which is to manage the life points. Since this life management is a higher level, this script should not be added directly to the player, but to the game controller. If you have added the script to the "Game Controller", we now come to the programming of the LifePointController script. The only variables we need are a private variable for managing the life points and a public variable for access to a GUIText object, to which we then assign the "LpText" object.

```
public GUIText lpText;
 private int lifePoints = 0;
```

Since we are going to use property methods to access the lifePoints variable, the int variable is declared private. We do this for the reason that when assigning the value we want to ensure that lifePoints can never have a value less than 0. We also want to ensure that the GUI is updated every time a new value is assigned. For this we will again implement an UpdateView method that does this.

```
public int LifePoints {
  get {
    return lifePoints;
     } set {
    lifePoints = value;
    if (lifePoints <0) lifePoints = 0; UpdateView ();
}
}
```

In the UpdateView method, only the new lifePoints value is then assigned to the GUI object

```
void UpdateView () {
   lpText.text = lifePoints.ToString ();
  }
```

The last thing we have to do now is to assign the current number of life points to the controller at the beginning of a scene. We will do this with the PlayerPrefs class and the "LPs" parameter. At the beginning of the game we will save the parameter "LPs" with a starting value in an opening scene. If the "Dungeon" scene is started with the LifePointController , the latter loads the value into the lifePoints variable.

```
void Awake () {
   LifePoints = PlayerPrefs.GetInt ("LPs");
   }
```

At the end of the scene, the script then writes its current value away again. This is done using the OnDestroy method, which is executed when the script is destroyed (which is the case when a scene is terminated).

```
void OnDestroy () {
  PlayerPrefs.SetInt ("LPs", lifePoints);
}
```

If the scene ends because the player has died, a lower value is written in "LPs" than he had at the beginning. As soon as the player dies, one life point is deducted from lifePoints in the PlayerHealth script, which we will be programming in a moment . And so a lower value is loaded when the scene is restarted. The complete LifePointController script can be found at the end of this section.

HealthController

Now we come to the actual management of the player's health. But since our opponents also have a state of health, we now want to work with heredity. We will create a base class HealthController , on which the scripts of the player and the opponent will be based. The class HealthController only consists of the public variable health , which stores the state of health , and the private variable isDead .

```
public float health = 3;
private bool isDead = false;
```

The class also has the ApplyDamage method, which causes damage to the owner. Here, health reduced by the amount of damage and it is a method Damaging called. However, if the damage brings health to 0 or even negative, a method called Dying is called instead and the variable isDead is set to TRUE.

```
void ApplyDamage (float damage) {
    health - = damage;
```

```
    if (health <= 0 &&! isDead) {
        isDead = true; Dying ();
  }
else {
   Damaging ();
  }
}
```

Since the two methods of damaging and dying differ greatly between the player and the opponent, these are only declared as empty methods that can be overwritten.

```
public virtual void damaging () {

}
  public virtual void Dying () {

}
```

PlayerHealth

Now we can create the PlayerHealth script based on the HealthController class. This script will also have an UpdateView method, in which this time the player's health value is to be displayed graphically. However, in this case we do not want to simply display a number, we want to display this value in the form of a bar. For this bar we use the GUITexture object "HealthBar", which we have already created. Finally, we assign the object to the public variable healthGui . In the UpdateView method, the length of this graphic object is finally calculated depending on the health value and a specified maximum length.

```
void UpdateView () {
    if (healthGui! = null) {
    guiRect.width = guiMaxWidth * health / maxHealth;
    healthGui.pixelInset = guiRect;
  }
}
```

Note the various variables in the listing that were not declared there. These are public and private variables that have been declared in the class header and are assigned their values in the Start method. guiRect receives the original values here, which are then immediately passed on to guiMaxWidth . And maxHealth also gets its value from the start value of health . In addition to these there are two more variables that are assigned in Start . The variable life point controller is assigned to the particular instance of the game controller and the variable message text is assigned a GUIText component that belongs to a GameObject tagged "message" that we have further created above. Finally, UpdateView is called again in the Start method so that the correct bar length is displayed in the GUI.

```
public AudioClip hurtClip;
public AudioClip deathClip;
public GUITexture healthGui;
private LifePointController lifePointController;
private float maxHealth;
private GUIText messageText;
private Rect guiRect; private float guiMaxWidth;
void Start () {
  messageText = GameObject.FindGameObjectWithTag ("Message").
GetComponent <GUIText> (); lifePointController =
```

```
GameObject.FindGameObjectWithTag ("GameController").
GetComponent <LifePointController> ();
guiRect = new Rect (healthGui.pixelInset);
guiMaxWidth = guiRect.width; maxHealth = health;
UpdateView ();
}
```

Now we only need to program the two overridable methods Damaging and Dying , which are called in the base class, but are empty. In the event of a violation, an audio clip is to be played whose public variable hurtClip was already defined in Listing 21.26. The clip is assigned to the player's AudioSource component, via which all of the player's speech samples are to be played. Be While this does not necessarily done, but we want machen.Außerdem must be in the overriding method for a consistent listening experience, in this case update View to be called so that the GUI is updated.

```
public override void Damaging () {
  audio.clip = hurtClip;
    audio.Play ();
     UpdateView ();
}
```

And in the Dying method, the associated AudioClip (deathClip) is assigned to your own AudioSource and played, and the GUI is updated, but now there is also the update of life points, which must be reduced by 1. If there are now still life points, the level or the scene is restarted after a short pause. If all life points up, to "Game Over" appears werden.Für the fade of "Game Over" we use here again the GUIText component, we their object

and component already in the Start using method of finding Game ObjectWithTag the variable message text assigned .

```
public override void Dying () {
    audio.clip = deathClip;
    audio.Play ();
    UpdateView ();
    lifePointController.LifePoints - = 1;
    if (lifePointController.LifePoints> 0)
    Invoke ("Restart", 1);
else
    messageText.text = "Game Over";
}
```

As you can see in Listing 21.28, a restart method is called with a one-second delay to restart the game . This last method of the script does nothing other than restart the current scene with LoadLevel .

```
void Restart () {
   Application.LoadLevel (Application.loadedLevel);
}
```

If you have now finished the PlayerHealth script, you can also attach this to your "Player". Assign the audio clip “hit” to the hurtClip variable and the “death” file as the “death clip”. We assign the already created GUITexture object “HealthBar” to healthGui

LifePointController .cs

The LifePointController script that you add to the GameObject “Game Controller” looks summarized as in the next listing. You add the script to the “Game Controller”.

```
using UnityEngine;
using System.Collections;
   public class LifePointController: MonoBehaviour {
        public GUIText lpText;
        private int lifePoints = 0;
        public int LifePoints {
 get {
    return lifePoints; }
 set {
    lifePoints = value;
        if (lifePoints <0) lifePoints = 0;
        UpdateView ();
   }
}

void Awake () {
    LifePoints = PlayerPrefs.GetInt ("LPs");
}

void UpdateView () {
   lpText.text = lifePoints.ToString ();
}
void OnDestroy () {
        PlayerPrefs.SetInt ("LPs", lifePoints);
 }
```

```
}
```

The base class HealthController , from which the PlayerHealth class also inherits, looks like the following listing

```
using UnityEngine;
using System.Collections;
  public class HealthController: MonoBehaviour {
       public float health = 3;
      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 () {

  }
```

}

PlayerHealth .cs

The actual management of the player's health is done by the PlayerHealth script, which inherits from the HealthController class . Add this to the player directly. Then assign the audio clips "hit" and "death" to the two AudioClip variables hurtClip and deathClip

```
using UnityEngine;
using System.Collections;
    public class PlayerHealth: HealthController {
        public AudioClip hurtClip;
    public AudioClip deathClip;
     public GUITexture healthGui;
            private LifePointController lifePointController;
            private float maxHealth;
       private GUIText messageText;
       private rect guiRect;
      private float guiMaxWidth;
void start () {
      messageText = GameObject.FindGameObjectWithTag ("Message").
GetComponent <GUIText> ();
              lifePointController = GameObject. FindGameObjectWithTag
("GameController").
    etComponent <LifePointController> ();
    guiRect = new Rect (healthGui.pixelInset);
    guiMaxWidth = guiRect.width;
    maxHealth = health;
```

```
    UpdateView ();
  }
public override void Damaging () {
            audio.clip = hurtClip;
            audio.Play ();
            UpdateView ();
   }
public override void Dying () {
    audio.clip = deathClip;
     audio.Play ();
        UpdateView ();
        lifePointController.LifePoints - = 1;
         if (lifePointController.LifePoints> 0)
  Invoke ("Restart", 1);
       Else
    messageText.text = "Game Over";
   }
  void restart () {
    Application.LoadLevel (Application.loadedLevel);
  }

void UpdateView () {
     if (healthGui! = null) {
                guiRect.width = guiMaxWidth * health / maxHealth;
                healthGui.pixelInset = guiRect;
       }
  }
}
```

Player controls

There are innumerable ways to control the hero of a game. To make things easier for the player, however, it usually makes sense to use known mechanisms that are already used in similar games. For this reason I would like to use the traditional WASD key control for our example game. Since our player control does not have to follow any physical laws, we can implement the control with a character controller. We had already added the right component to the "player".

PlayerController

In the chapter "Physics in Unity" I showed you in the section "Simple First Person Controller" what a controller with a character controller could look like. There you will also find a detailed description of the development of this script. Program this according to the description and add the PlayerController script to the "Player". Please also note the creation of the two additional virtual axes “Left” and “Right” (key [Q] for “Left” and [E] for “Right”). At the end of the section “Simple First Person Controller” you will finally find the complete script. In the following we want to extend this script called “PlayerController” with game-specific functionalities. This also includes preventing the controls as soon as the player has won the game or the game is dead. First, we want to create two variables: the Boolean variable gameEnded and the variable playerHealth of type PlayerHealth . Since gameEnded is later to be set by another script that determines the completion of the quest, this variable must be declared with public . But since we don't want to set the variable in the Inspector, we can provide it with a so-called attribute, which ensures that the variable cannot be seen in the Inspector despite the public declaration. This attribute is HideInInspector and is specified in square brackets above the variable.

public float moveSpeed = 5.0F;

public float rotationSpeed = 300.0F;

[HideInInspector]

```
public bool gameEnded = false;
private Vector3 moveDirection = Vector3.zero;
private CharacterController controller;
private playerHealth playerHealth;
private quaternion dest rotation;
```

In the Start method, in addition to the assignments that have already been made, the PlayerHealth script is now assigned to the associated variables.

```
void Start () {
    playerHealth = GetComponent <PlayerHealth> ();
    controller = GetComponent <CharacterController> ();
    destRotation = transform.rotation;
 }
```

The only real change to the original script from the "Physics in Unity" chapter is now querying the Boolean variable gameEnded and the current health status before the control is executed. However, if the health variable of playerHealth is "0" or gameEnded TRUE, the control is not executed and the SimpleMove method is passed the value Vector3.zero , which means nothing other than that the character controller should stop. You can see how this looks in the code in the complete script, which you can find in the following section “PlayerController.cs”.

Footsteps

Most of the time, it is not enough to just control it, not even in our case. Since our hero should also create step noises as soon as he moves, we now create a new script called Footsteps. This requires five variables: one for the AudioClip, to which we assign the sound of a single step. Then we need a float variable with which we define the interval at which the steps can be heard, as well as a float variable for the volume of the step noise. We also

need a private variable in which we store the reference to the character controller in the Start method, as well as a last float variable which stores the time elapsed since the last step sound.

Variables of footsteps

```
public AudioClip audioClip;
public float stepLength = 0.4F;
public float volume = 0.7F;
private CharacterController controller;
private float delay = 0;
```

The footstep sounds should now be played whenever the player moves on the floor. To do this, we first check with sqrMagnitude with a certain tolerance whether the player is moving, and with isGrounded we test whether he is touching the ground.

```
if condition to control movement of the player
  if (controller.velocity.sqrMagnitude> 0.2F && controller.isGrounded) {
    // ...
}
```

If this is the case, we now play the audio clip with PlayClipAtPoint at a time interval of stepLength . We determine the elapsed time with the float variable delay , the value of which we increase by deltaTime in each frame . If delay has now reached stepLength , the AudioClip is played and delay is set to 0 again. For this, of course, the code has to be called in each frame, which is why we put the procedure appropriately in the update method.

Play footsteps in update

```
void Update () {
    if (controller.velocity.sqrMagnitude> 0.2F && controller.isGrounded) {
            if (delay> = stepLength) {
               AudioSource.PlayClipAtPoint (audioClip, transform.position,
volume);
               delay = 0;
             }
    }
     delay + = Time.deltaTime;
}
```

Now add the finished footstep script to your “player” and assign the audio file “footstep” to the audioClip variable . Please note that you deactivate the 3D Sound property in the Import Settings for this file. This is important because the AudioSources temporarily generated by PlayClipAtPoint do not move with the player, but remain at the point where they were generated. Shooting Finally, we want to make sure that our hero can fight. He should be able to use stones as a weapon, which he can collect during the game and then throw. Of course, interaction with the inventory system is important for this mechanism, which is why we first create a variable of the Inventory type in our variable declarations . We also need a reference to the PlayerHealth script of our player so that we can determine whether we are able to throw at all, then we need a GameObject variable for our prefab. If throwing stones is too boring for you you can do this with other 3D models such as B. Replace with knives or arrows of a bow or a crossbow. Since the principle is the same, it doesn't really matter anymore. That's why I'll call the script “Shooting” in general and call the prefab variable projectile .

```
public GameObject projectile;
private playerHealth playerHealth;
```

```
private inventory inventory;
  private GUIText messageText;
     private string info = "To attack you need something to throw";
     private bool messageShown = false;
```

As you can see from the listing , we need three more variables in addition to the variables already mentioned, but I will come to them later. First of all, it is important that the messageText variable in the Start method refers to the GUIText component of the GameObject with the tag "Message", which we have already used in the Player Health script. If our hero now throws a stone, The question now arises as to where the prefa should actually be generated. There are so-called spawn points for defining such points. A spawn point is usually just an empty object that determines its position with the help of its transform component. In our case, the spawn point will be a child object of our player, since it logically always moves with the player (see introduction of this section). The special thing about this script is that it is not added to the "player", but is attached directly to the GameObject "SpawnPoint". The charm of this is that we don't have to tell the script where the prefabs should be created. We also use the rotation of this GameObject to take over the direction of the throw from the spawn point.

However, we also have to consider this when accessing other components. Since we refer to the Inventory and PlayerHealth in the Start method, we now have to access the parent object for the latter. We do this with transform. parent .

```
void start () {
   playerHealth = transform.parent.GetComponent <PlayerHealth> ();
        inventory = GameObject.FindGameObjectWithTag ("Inventory").
GetComponent <Inventory> ();
        messageText = GameObject.FindGameObjectWithTag ("Message").
GetComponent <GUIText> ();
```

```
}
```

We now want to trigger the throwing or shooting with a shoot method . This simply instantiates a prefab instance of projectile at its own position with the same rotation as the "SpawnPoint". In addition, a throwing noise should be heard that we want to play back at the same time. For this purpose, we added a separate audio source to the “SpawnPoint” at the beginning, to which a suitable sound was assigned.

```
void Shoot () {
        Instantiate (projectile, transform.position, transform.rotation);
        audio.Play ();
  }
```

The last thing we have to do is trigger Shoot . We do this in the update method, where we will use the virtual button "Fire2". For this I would perhaps change the key from Positive Button from “left alt” to “space” in the input settings. But you can decide for yourself which key you want to use for throwing, depending on your preferences. For each throw, an item with the tag "Stone" is removed from the inventory. If there is none left, it cannot be thrown. An important point that will be noticed at the latest when the first tester is going to play the game is an indication that the player must first collect the objects that he wants want to throw. Otherwise, he presses the attack button and is surprised that nothing happens, for this we want to display a small info text that should appear when the player tries for the first time without attacking a stone in the inventory. We want to show this in a coroutine that displays the text for two seconds. In doing so, we note in the variable messageShown when this information was shown. In the update method, as soon as there is no stone in the inventory, we check the status of these variables and then start the coroutine accordingly.

```
void Update () {
```

```
    if (playerHealth.health> 0) {
     if (Input.GetButtonDown ("Fire2")) {
      if (inventory.RemoveItem ("Stone"))
        Shoot ();
      else if (! messageShown)
                    StartCoroutine (ShowInfoText ());
                   }
     }
   }
  IEnumerator ShowInfoText () {
                messageText.text = info;
                messageShown = true;
                yield return new WaitForSeconds (2);
                messageText.text = "";
   }
```

As already mentioned, this script is now attached to the “SpawnPoint”. In the section “Develop throwing stone” we will also create the prefab that we still have to assign to the script parameter projectile .

PlayerController .cs

Below you can see the complete PlayerController script. Please note that you have to create the virtual buttons "Left" and "Right" in order to use this script. You define these in the input settings and best assign them with the buttons [Q] (for "Left") and [E] (for "Right").

```
sing UnityEngine;
```

```
using System.Collections;
  public class PlayerController: MonoBehaviour {
      public float moveSpeed = 5.0F;
      public float rotationSpeed = 300.0F;
   [HideInInspector]
      public bool gameEnded = false;
       private Vector3 moveDirection = Vector3.zero;
  private CharacterController controller;
 private playerHealth playerHealth;
      private quaternion dest rotation;
   void Start () {
            playerHealth = GetComponent <PlayerHealth> ();
            controller = GetComponent <CharacterController> ();
            destRotation = transform.rotation;
   }
      void Update () {
        if (playerHealth.health> 0 &&! gameEnded) {
         moveDirection = new Vector3 (Input.GetAxis ("Horizontal"), 0,
Input.GetAxis ("Vertical"));
                    moveDirection = transform.TransformDirection
(moveDirection);
                moveDirection = moveDirection * moveSpeed;
   controller.SimpleMove (moveDirection);
     if (Input.GetButtonDown ("Left")) destRotation.eulerAngles =
destRotation.eulerAngles -
           new Vector3 (0.90.0);
            if (Input.GetButtonDown ("Right")) destRotation.eulerAngles =
destRotation.eulerAngles +
```

```
            new Vector3 (0.90.0);
              float step = rotationSpeed * Time.deltaTime;
            transform.rotation = Quaternion.RotateTowards
(transform.rotation, destRotation, step);
      }
   else {
        controller.SimpleMove (Vector3.zero);
       }
  }
 }
```

Footsteps .cs

The Footsteps script responsible for the footsteps is shown in Listing 21.43. Please add this script to the "Player" and assign the sound file "footstep" to the audioClip variable . Since the audio sources temporarily generated by the script do not move with the player, you should deactivate the 3D Sound property for this file in the import settings. Use stepLength to define the interval at which the steps can be heard. The lower you set this value, the more often the steps can be heard.

```
using UnityEngine;
using System.Collections;
 public class footsteps: MonoBehaviour {
   public AudioClip audioClip;
        public float stepLength = 0.4F;
        public float volume = 0.7F;
      private CharacterController controller;
  private float delay = 0;
```

```
    void start () {
        controller = GetComponent <CharacterController> ();
    }
   void Update () {
if (controller.velocity.sqrMagnitude> 0.2F && controller.isGrounded) {
  if (delay> = stepLength) {
    AudioSource.PlayClipAtPoint (audioClip, transform.position, volume);
          delay = 0;
     }
  }
delay + = Time.deltaTime;
}
}
```

Shooting .cs

Add the following script to the "SpawnPoint" object. Later, you have the VARIAB-len projectile the throwing stone Prefab to which we are still in the "throw stones develop" create. Depending on your preference, you can define a different key for throwing in the input settings. By default, the "left alt" button is defined as a positive button. You could change this "space" and thus the space bar [Space] to use for throwing.

```
using UnityEngine;
using System.Collections;
 public class shooting: MonoBehaviour {
    public GameObject projectile;
    private playerHealth playerHealth;
  private inventory inventory;
 private GUIText messageText;
```

```
    private string info = "To attack you need something to throw";
    private bool messageShown = false;
    void start () {
            playerHealth = transform.parent.GetComponent <PlayerHealth>
();
   inventory = GameObject.FindGameObjectWithTag ("Inventory").
GetComponent <Inventory> ();
  messageText = GameObject.FindGameObjectWithTag ("Message").
GetComponent <GUIText> ();
     }
 void Update () {
     if (playerHealth.health> 0) {
      if (Input.GetButtonDown ("Fire2")) {
        if (inventory.RemoveItem ("Stone"))
                    Shoot ();
          else if (! messageShown)
              StartCoroutine (ShowInfoText ());
           }
    }
  }
void Shoot () {
   GameObject go = (GameObject) Instantiate (projectile, transform.position,
transform.rotation);
          audio.Play ();
    }
  IEnumerator ShowInfoText () {
        messageText.text = info;
        messageShown = true;
```

```
        yield return new WaitForSeconds (2);
    messageText.text = "";
  }
}
```

Develop throwing stone

We had already created a prefab for the throwing stones when creating the inventory items. However, this only serves to collect the throwing stones. However, this cannot be used as a weapon, as the object requires a few other components and settings for this. For this reason, we will next create another stone prefab, which we will then call a “thro-wing stone”.

1. Drag the "stone_01" model into the scene again.

2. Rename the GameObject in the scene to “Throwing Stone”.

3. Add a sphere collider to it and activate Is Trigger.

4. Since the thrown stones will be quite small, but also fast, we now slightly increase the radius of the collider z. B. to 0.12 to make collision detection easier.

5. Now add a Rigidbody to the GameObject and activate Is Kinematic. Instead, disable Use Gravity. In addition, set the parameter Collision Detection to "Continue Dynamic" for better collision detection.

6. Drag the GameObject "Throwing Stone" into the "Prefab" folder to create a prefab.

7. Now assign this prefab to the projectile parameter of the shooting script of the "SpawnPoint".

As you may have already noticed, the shooting script will now create the throwing stone, but it will not move.

We now want to program the ability for the stone to fly in the correct direction in a separate script. This script will not only take care of the movement, but also cause the damage and destroy the stone itself if it collides with another object. In short: This script will take care of the whole behavior of the stone, which is why we simply call it "Stone Behavior". So now create a new “StoneBehaviour” script and add it to the “ThrowingStone” prefab . To do this, select the prefab in the Project Browser and add the script to the prefab using the Add Component button in the Inspector. In order to get the best control over the throw or flight path of the stone, we will translate the stone using the translate method move the transform component. When creating the throwing stone prefab, we therefore already activated the Is Kinematic parameter of the rigid body. We control the speed of the stone using a variable called speed .

```
void Update () {
    transform.Translate (transform.forward * speed * Time.deltaTime,
Space.World);
  }
```

To get the fun out of fighting, aiming with the stones should be relatively easy. Of course, the easiest way to do this is if the spawn point is quite far in the middle of the player. So that the throwing stone does not already collide with the collider of the character controller, we use the Is Trigger parameter on the collider of the stone. This also means that we can evaluate and inflict damage in an OnTrigger method, preferably in the OnTriggerEnter method need to run. However, this method is also carried out if the object collides with another trigger collider, e.g. For example, with that of our future quest giver (more on this in the section “Creating a quest giver”). Therefore, in this case we want to work with a layer mask, through which we define which layers the following code should be executed on and which should not. For this we first generate via

Edit / Project Settings / Tags and Layers

a new layer with the name "IgnoreStoneDetection". Now we declare a public layer mask variable with the name acceptableLayers as well as the already mentioned speed variable and a damage variable that determines the amount of damage. In addition, a small particle effect is to be created if the stone breaks due to a collision with another GameObject. For this we need a GameObject variable with the name destroyGo . We will create the particle effect itself immediately afterwards.

```
public LayerMask acceptableLayers;
public GameObject destroyedGo;
public float damage = 1;
public float speed = 10;
```

Now note that you have deactivated the new layer "IgnoreStoneDetection" in the Inspector of "ThrowingStone" at acceptableLayers and the others are checked. In the OnTriggerEnter method, we now evaluate the layer with which the stone is currently colliding (see Section “Working with layer masks” in the chapter “Physics in Unity”). If the layer of the other object is available in acceptableLayers , it is then checked whether the object has the tag "Enemy". If this is also the case is with SendMessage method ApplyDamage called and the variable damage over-geben.Aber regardless of whether it is now a "Enemy" not when the object a Layerbesitzt, located in acceptableLayers located is Finally destroy your own GameObject and create an instance of destroyGo . However, this is also destroyed immediately after a second. For this purpose, Instantiate is immediately called as part of a time-delayed Destroy command.

```
void OnTriggerEnter (Collider other) {
        if ((acceptableLayers.value & 1 << other.gameObject.layer) == 1 <<
other.gameObject.layer) {
      if (other.gameObject.CompareTag ("Enemy")) {
```

```
        other.gameObject.SendMessage ("ApplyDamage", damage,
SendMessageOptions.DontRequireReceiver);
    }
  Destroy (gameObject);
Destroy (Instantiate (destroyedGo, transform.position, Quaternion.identity),
1);
  }
}
```

So that the throwing stone does not collide with the Collider of the CharacterController when it is created, we set the "Player" layer to "IgnoreStoneDetection" right at the beginning. Confirm the following query that you want to apply the setting of the layer to all child objects.

StoneBehaviour .cs

The complete StoneBehaviour script that you add to the throwing stone prefab "ThrowingStone" can be found in the following listing.

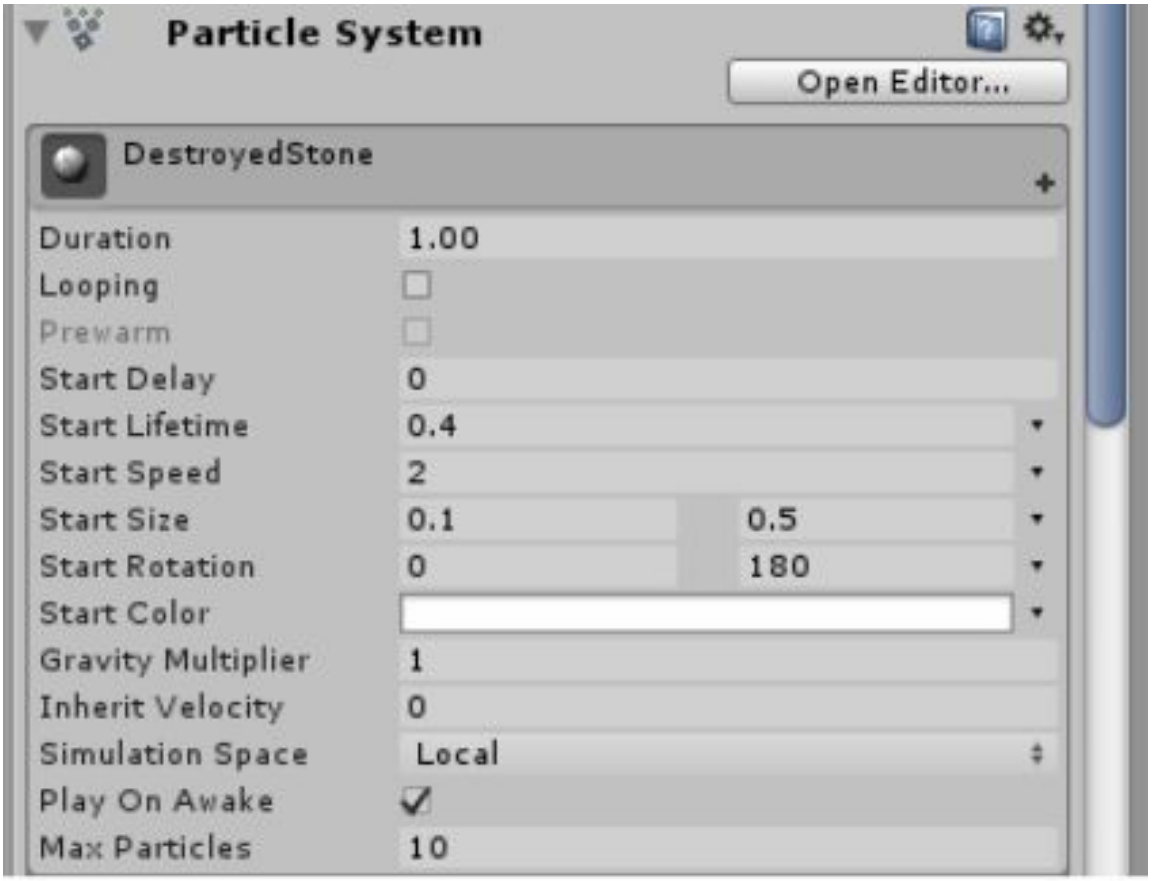
```
using UnityEngine;
using System.Collections;
    public class StoneBehaviour: MonoBehaviour {
            public LayerMask acceptableLayers;
            public GameObject destroyedGo;
            public float damage = 1;
            public float speed = 10;
             void Update () {
```

```
        transform.Translate (transform.forward * speed * Time.deltaTime,
Space.World);
  }
   void OnTriggerEnter (Collider other) {
// Converts an integer into a bit number by bit-wise shifting.
//Debug.Log(1 << other.gameObject.layer);
// The result of the bitwise comparison (&),
// between layer and mask, is compared with the layer as a bit number.
     if ((acceptableLayers.value & 1 << other.gameObject.layer) == 1 <<
other.gameObject.layer)
    {
            if (other.gameObject.CompareTag ("Enemy")) {
    other.gameObject.SendMessage ("ApplyDamage", damage,
SendMessageOptions.DontRequireReceiver); }
    Destroy (gameObject);
          Destroy (Instantiate (destroyedGo, transform.position,
Quaternion.identity), 1);
   }
  }
}
```

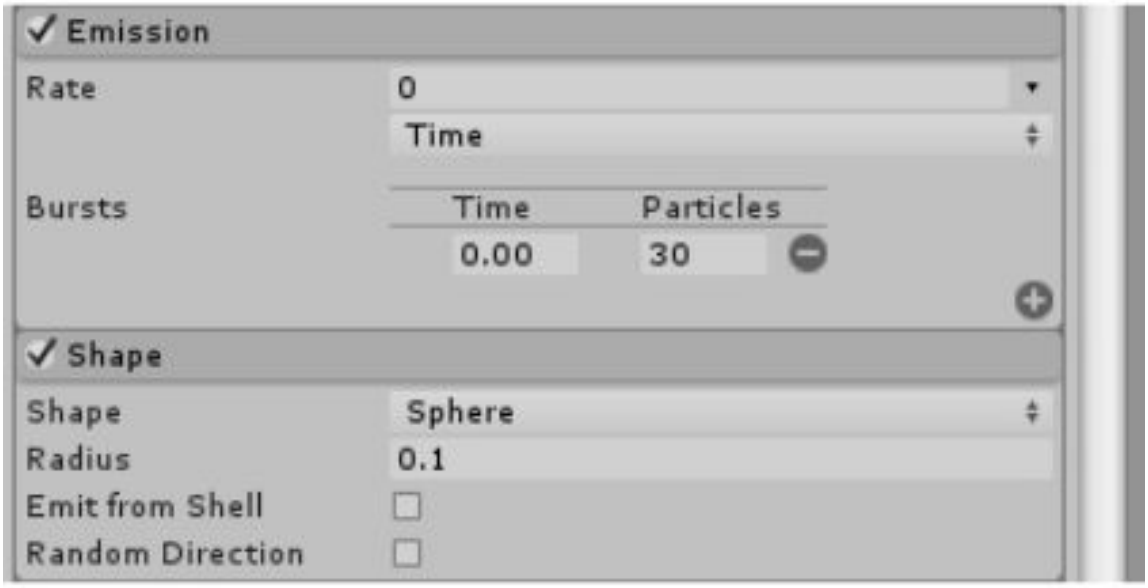
Configure bursting stone

If a throwing stone hits a wall or another object, it should burst. For this we want to create a small particle effect that does not create conventional particles, but three-dimensional particles in the form of the stone mesh. Create a new particle system for this and name it "DestroyedStone". Then drag it into the "Prefab" folder to create a prefab. Then we configure the individual modules of the particle system. First, we parameterize the default

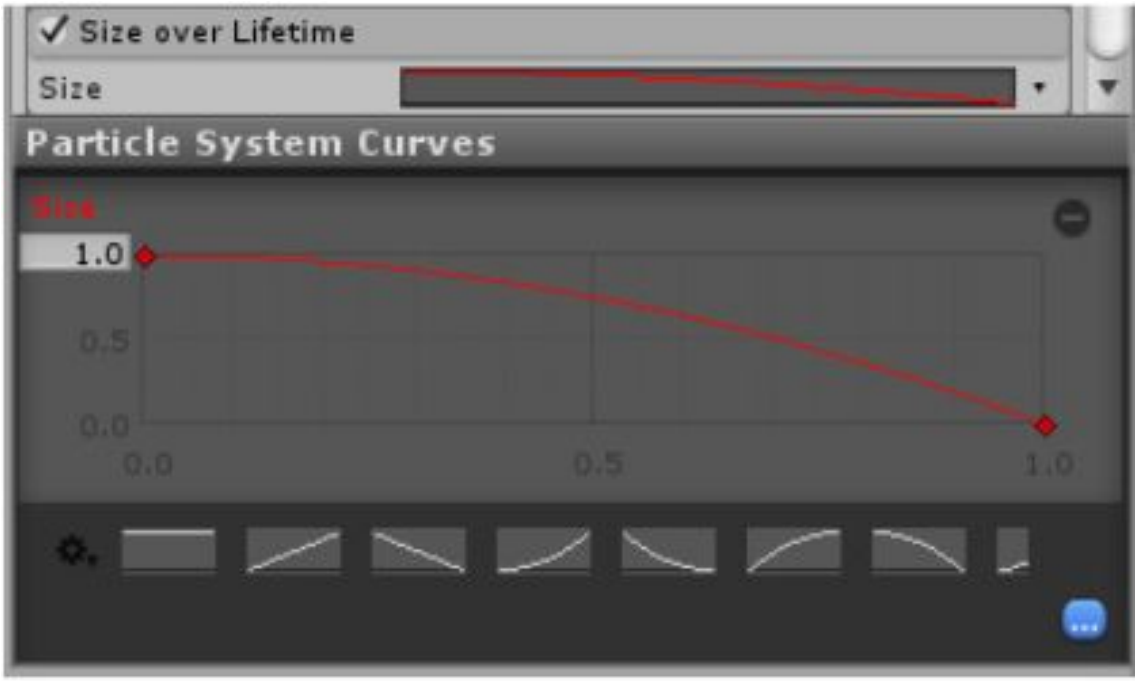
module with the following settings. Both Start Size and Start Rotation are set to Random Between Two Constants.



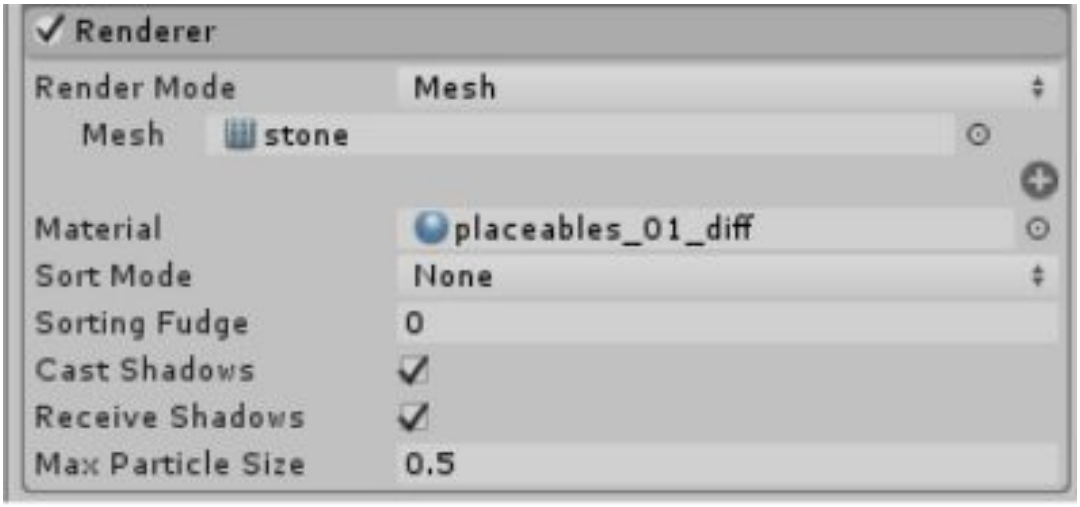
This is followed by the emission module and the shape module. Since the effect should only appear once, we have already deactivated looping above. In the emission module we now define a single burst. Due to the Max Particles value of 10 in the default module, it does not matter whether we set the burst to 10, 30 or 100, it will only be up to 10



generated. Finally, we set the shape as Sphere with a radius of 0.1, because the particles should fly from one point in all directions. Then we use the Size over Lifetime module to define the size of each particle. Here the particles should be seen for a long time, but then also disappear smoothly. For this we use one of the standard curves that Unity provides.



Finally, the renderer module follows, where we set the render mode to mesh. Now we select the mesh of the stone model under Mesh ("stone"). It is now important that you also select the associated material.



And the particle effect is ready. Now you only need to transfer the settings made to the prefab with Apply in the inspector of the particle system. Finally, the stone should not only burst visually, but also reproduce this acoustically. To do this, add your own AudioSource to the GameObject, where you activate Play On Awake and deactivate Loop. Now assign the audio clip "stoneCrash" to this. In order to use the particle system GameObject, assign this prefab to the destroyedGo variable from the StoneBehaviour script of the "ThrowingStone" prefab.

Create a quest

This section is about the actual task the player has to solve. In our game, our player should find a water container with which he can catch dripping water. To solve this task, also known as a quest, he still has to complete sub-quests and fight opponents.

Manage experience points

For overpowering an opponent and solving the quest, the player should also get experience points at the end. Therefore we want to first create a small script called EPController that manages these experience points and displays them in the GUI. You assign this EPController script to the "Game Controller", which already has the LifePoint controller. For the functionalities we need a total of two variables: one variable that stores the EP value and another that contains a reference to a GUI Object to display the value there.

```
public GUIText epText;
private int eps = 0;
```

Since we are still missing an object for the GUI, we want to create this next.

1. Create a new GUIText object and name it "EpText".

2. Add “EpText” as a child object to the “GUI” container object that has already been created.

3. Position this on (0,1,0) and a pixel offset (70, -49).

4. Assign this object to the script variable epText from EPController. For the actual logic, we again need an UpdateView method, which takes over the display, and a method AddPoints , with which we can transfer new EPs to the script from outside. The UpdateView method is also called again in the Start method for the first time to assign an initial value to the GUI.

```
void start () {
  UpdateView ();
 }
 public void AddPoints (int points) {
     eps + = points;
     UpdateView ();
}
```

```
void UpdateView () {
    epText.text = "EP:" + eps.ToString ();
}
```

That was the complete content of EPController. Now you only need to add the script to the “Game Controller” and assign the GUIText object “EpText” to the epText variable

EPController .cs

In the following listing you can see the finished EPController script that manages the player's experience points. Add the finished script to the "Game Controller".

```
using UnityEngine;
using System.Collections;
  public class EPController: MonoBehaviour {
      public GUIText epText;
      private int eps = 0;
       void Start () {
            UpdateView ();
  }
public void AddPoints (int points) {
        eps + = points;
  UpdateView ();
  }
 void UpdateView () {
  epText.text = "EP:" + eps.ToString ();
  }
```

}

Create quest givers

Now we come to the actual quest, which I will simply call the Water Quest. As mentioned at the beginning, this should consist of finding an object to catch water; in the end, it does not matter which object to look for. The task might as well be looking for a spell scroll, magical sword, or piece of gold. The principle is always the same: as soon as the player enters a trigger collider, it is checked whether it has the object it is looking for. If he has this in his inventory system, it will be removed there and the quest will be marked as "fulfilled". If he does not have this item, the player is informed about his task. Alternative quest giver texts Texts from quest givers are often structured in such a way that different texts appear when the quest giver is visited several times. All you need is a Boolean variable that is set to TRUE as soon as the player learns his task. If this variable is now TRUE when entering the trigger collider, this means that the player has already been there and another text can be displayed. Such a trigger collider is usually positioned next to a person, an object or a location that thematically fits the task at hand. Together, the overall structure is also referred to as the "quest giver", as this person / object / location communicates this task to the hero.

Quest giver configuration

As a quest giver, I would like to use the particle system "Waterdrops", which I already introduced in Section 11.10, "Creating Water Drops", of the chapter "Particle Effects with Shuriken". So that the player actually pays attention to this particle system, we are adding this to it the WaterdropSound script, which we also developed in “Create water drops”. Use the "waterdrop" file as an audio clip. Additionally, you can also

increase the amount of water droplets created by the particle system. To do this, set the rate value in the emission module slightly higher, e.g. E.g. on 0.8. Position this particle system in one corner of the dungeon on the ceiling. It is best to place it in a place that the player can see when the game starts. This will lead the player to the quest immediately and find out what to do there. Under this particle system we will now place our trigger collider with the following quest script, which we will still develop.

1. To do this, create a cube that you call "QuestTrigger".

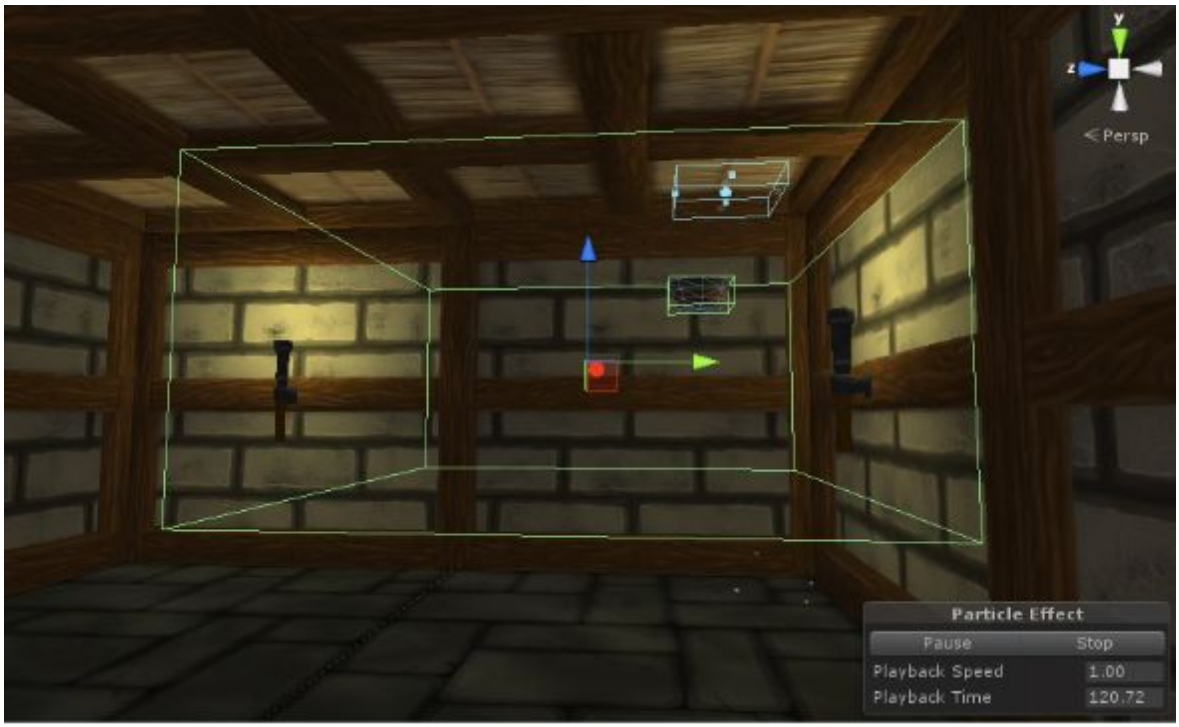
2. Scale the cube to (2,1,2) and position it directly under the water droplet particle system at player height

3. Set the layer from "QuestTrigger" to "IgnoreStoneDetection".

4. Deactivate the mesh renderer of the cube.

5. Activate the Is Trigger parameter of the Box Collider

6. Now create a new script called "WaterQuest" and add it to the cube as well. Since the player should look for a water container and bring it here, we want to symbolize the successful delivery of the container by the appearance and dropping of a container So add an instance of the “BucketEmpty” prefab to the scene and place it directly under the water droplet particle system at the level of the camera.



WaterQuest

For the actual quest we now need the already created WaterQuest script with the following variables, which I will explain to you in more detail in a moment.

```
public int eps = 20;
public GameObject finalBucket;
private GUIText messageText;
private string questMessage = "Find a container" + "to drink this water.";
private string questMessage2 = "You need an empty container.";
private string questEndedMessage = "<size = 20> Congratulations! </size> \
n" + "You have solved the task.";
private bool gotQuest = false;
private GameObject player;
private inventory inventory;
```

```
private PlayerController playerController;
private EPController epController;
```

The variable eps is later used to add experience points when the quest has been solved. The GameObject variable finalBucket is assigned the "BucketEmpty" instance that we have just added to the scene. With the Boolean variable gotQuest, we remember when the player has received the quest, and then the alternative text the second time questMessage2 . The string variables questEndedMessage , questMessage and questMessage2 serve as storage for the texts to be displayed. The command "\ n" creates a line break within the string. Also note the HTML tags in the content of questEndedMessage , with which we want to highlight the size of a single word. Unity supports some HTML format tags here, which are excellent for highlighting certain terms within a string. The next thing is the Start method, in which we let the remaining variables refer to the corresponding components. We also deactivate the GameObject finalBucket so that we can reactivate it later if successful. In this case, the object then appears and, thanks to the rigid body component that has already been assigned, it falls to the floor.

```
void start () {
        finalBucket.SetActive (false);
       player = GameObject.FindGameObjectWithTag ("Player");
        playerController = player.GetComponent <PlayerController> ();
         inventory = GameObject.FindGameObjectWithTag ("Inventory").
GetComponent <Inventory> ();
         messageText = GameObject.FindGameObjectWithTag ("Message").
GetComponent <GUIText> ();
          epController = GameObject.FindGameObjectWithTag
("GameController"). GetComponent <EPController> ();
  }
```

Note the variable messageText , which again refers to the GUIText GameObject with the tag "Message". So here we use the same object that we already used in the PlayerHealth script to display the "Game Over" text. Next we want to program the core of this quest giver: the OnTrigger Enter method. This is executed as soon as another GameObject with a rigid body and a collider enters this trigger collider. To do this, we first check whether the object is the player. If this is the case, we try to get an item with the ID "Bucket" from the inventory system. If we get one back, we activate finalBucket , which makes the GameObject visible, show questEndedMessage in the GUI and finally assign the EPs to the EPController . Since our game only consists of a single quest, we finally set the variable gameEnded from the PlayerController to TRUE.

However, if the player does not have an object with the ID "Bucket", the quest text is displayed. If the player has already read this and repeatedly triggers this trigger, the alternative text questMessage2 is displayed.

```
void OnTriggerEnter (Collider other) {
    if (other.gameObject == player) {
   if (inventory.RemoveItem ("Bucket")) {
    finalBucket.SetActive (true);
            playerController.gameEnded = true;
             messageText.text = questEndedMessage;
             epController.AddPoints (eps);
      }
  else {
    if (gotQuest) messageText.text = questMessage2;
  else messageText.text = questMessage;
       gotQuest = true;
         }
  }
 }
```

The last thing we want to do now is to ensure that the questMessage or questMessage2 disappears again as soon as the hero leaves the trigger again. For this we use the OnTriggerExit method .

```
void OnTriggerExit (Collider other) {
  if (other.gameObject == player) {
      messageText.text = "";
      }
 }
```

InGameMenu

With that we have the actual quest complete. But what should happen now when the player has played the game through? At the moment the hero is no longer controllable and the text of questEndedMessage is displayed in the GUI. So that it goes on somehow, the player should now have the opportunity to restart the game or go to a start menu. For this we want to create a small script called "InGameMenu", which we also the "Game Controller “Want to add. This now requires references to the components PlayerController and LifePointController .

```
private PlayerController playerController;
private LifePointController lifePointController;
void start () {
      playerController = GameObject.FindGameObjectWithTag ("Player").
GetComponent <PlayerController> (); lifePointController =
GameObject.FindGameObjectWithTag ("GameController").
GetComponent <LifePointController> ();
 }
```

Now we check in each frame whether the hero has died or the game is over. If one of the two is the case, the opening scene of the game (index 0) is started as soon as the [Escape] , [Space] or [Return] key has been pressed. Alternatively, you could of course check all keys at once via Input.anyKeyDown .

```
void Update () {
   if (lifePointController.LifePoints == 0 || playerController.gameEnded) {
          if (Input.GetKeyDown (KeyCode.Escape) || Input.GetKeyDown
(KeyCode.Space) || Input.GetKeyDown (KeyCode.Return))
  {
   Application.LoadLevel (0);
  }
   }
}
```

WaterQuest .cs

The WaterQuest script has the logic of the quest and is added directly to a GameObject with a trigger collider.

```
using UnityEngine;
using System.Collections;
  public class WaterQuest: MonoBehaviour {
     public int eps = 20;
     public GameObject finalBucket;
     private GUIText messageText;
     private string questMessage = "Find a container" + "to drink this
water.";
     private string questMessage2 = "You need an empty container.";
```

```
    private string questEndedMessage = "<size = 20> Congratulations!
</size> \ n" + "You have solved the task.";
    private bool gotQuest = false;
    private GameObject player;
   private inventory inventory;
  private PlayerController playerController;
     private EPController epController;
void start () {
   finalBucket.SetActive (false);
         player = GameObject.FindGameObjectWithTag ("Player");
        playerController = player.GetComponent <PlayerController> ();
        inventory = GameObject.FindGameObjectWithTag ("Inventory").
GetComponent <Inventory> ();
        messageText = GameObject.FindGameObjectWithTag ("Message").
GetComponent <GUIText> (); epController =
GameObject.FindGameObjectWithTag ("GameController").
GetComponent <EPController> ();
 }
  void OnTriggerEnter (Collider other) {
         if (other.gameObject == player) {
            if (inventory.RemoveItem ("Bucket")) {
      finalBucket.SetActive (true);
                playerController.gameEnded = true;
               messageText.text = questEndedMessage;
                 epController.AddPoints (eps);
            }
    else {
       if (gotQuest) messageText.text = questMessage2;
```

```
   else messageText.text = questMessage;
       gotQuest = true;
  }
 }
 }
void OnTriggerExit (Collider other) {
 if (other.gameObject == player) { messageText.text = "";
   }
 }
}
```

WaterdropSound .cs

This script creates a sound as soon as a particle of a particle system collides. Add this script to the "Waterdrops" particle system that creates the water that drips down. Assign the file “waterdrop” to the AudioClip variable clip .

```
using UnityEngine;
using System.Collections;
   public class WaterdropSound: MonoBehaviour {
       public AudioClip clip;
    v oid OnParticleCollision (GameObject other) {
ParticleSystem.CollisionEvent [] collisionEvents = new
ParticleSystem.CollisionEvent [5];
int quantityCollisionEvents = particleSystem.GetCollisionEvents (other,
collisionEvents);
int i = 0;
while (i <quantityCollisionEvents) {
Vector3 pos = collisionEvents [i] .intersection;
AudioSource.PlayClipAtPoint (clip, pos, 0.3F);
```

```
i ++;
}
}
}
```

InGameMenu .cs

The InGameMenu script is used to end the game or restart it. Add the script from the following listing to the "Game Controller".

```
using UnityEngine;
using System.Collections;
  public class InGameMenu: MonoBehaviour {
    private PlayerController playerController;
          private LifePointController lifePointController;
          // Use this for initialization
      void start () {
      playerController = GameObject.FindGameObjectWithTag ("Player").
GetComponent <PlayerController> (); lifePointController =
GameObject.FindGameObjectWithTag ("GameController").
GetComponent <LifePointController> ();
}

void Update () {
    if (lifePointController.LifePoints == 0 || playerController.gameEnded)
{
if (Input.GetKeyDown (KeyCode.Escape) || Input.GetKeyDown
(KeyCode.Space) || Input.GetKeyDown (KeyCode.Return))
{
```

Application.LoadLevel (0);

}

}

}

}

Create sub-quest

In order for the player to complete his quest, a prefab instance of the container with the respective inventory ID must be placed in the game, which he can also find. However, it makes little sense if the solution to the quest can be found right next to the quest giver and the player simply needs to pick up this item. To make solving the task a little more difficult for the player, there are so-called sub-quests that have to be solved beforehand. These can be puzzles or simply opponents that have to be defeated.We want to use both in our game and start by giving the player a little search task. You already got to know an animated portcullis in the “Animations” chapter that we now want to use in our game. This can e.g. B. block a corridor at the end of which the water tank can be found. So that we get a real task here, the gate should be locked so that the player must first find the right key. So now create this porthole as described in the chapter "Animation" with its animations, the Animator Controller and the Controller script including the Animation Events and the DoorMovingState script. Create a new “Door” tag and assign it to the child object “gate_01”. 3. Now add the HoverEffects script with the two textures “icon_01_32x32” and “icon_02_32x32” to the child object “gate_01” (see “Inventory Items”). As toolTipText assign "iron gate" zu.4. Then drag the entire Fallgatter GameObject into the "Prefab" folder of the Project Browser to create a prefab called "Gate".

Lock the gate

In order to "lock" the gate, you do not need to do anything other than add a Boolean variable to the AnimateDoor script that symbolizes the lock. If the

gate is open, this variable is TRUE, if the gate is closed, it is FALSE. That's why we simply call this variable isLocked. In addition, a reaction should come when the gate is locked. Instead of displaying a text message again, we want our heroes to say something that tells the player to look for a key. For this we need a second AudioClip variable lockedDoorMonologueClip . This audio clip is to be played back via the player audio source, which is why we also need a variable for the player. We also need a reference to our inventory to get the key from there.

```
public AudioClip doorClip;
  public AudioClip lockedDoorMonologueClip;
    public bool isLocked = true;
    private animator anim;
private int switchTrigger;
private DoorMovingState doorMovingState;
private GameObject player;
private inventory inventory;
  void start () {
        switchTrigger = Animator.StringToHash ("Switch");
        anim = transform.parent.GetComponent <Animator> ();
       doorMovingState = transform.parent.GetComponent
<DoorMovingState> ();
      player = GameObject.FindGameObjectWithTag ("Player");
      inventory = GameObject.FindGameObjectWithTag ("Inventory").
GetComponent <Inventory> ();
  }
```

Since we declare the variable isLocked as public, we can use this script for gates that are not locked. You just need to set the value to TRUE. Now we have to implement the lock mechanism in the OnMouseDown method. To

do this, we first check whether the gate is locked. If this is the case, we try to get an item with the ID "Key" from the inventory system. If this is successful, we set isLocked to TRUE

```
if (isLocked) {
   if (inventory.RemoveItem ("Key")) {
    isLocked = false;
  }
}
```

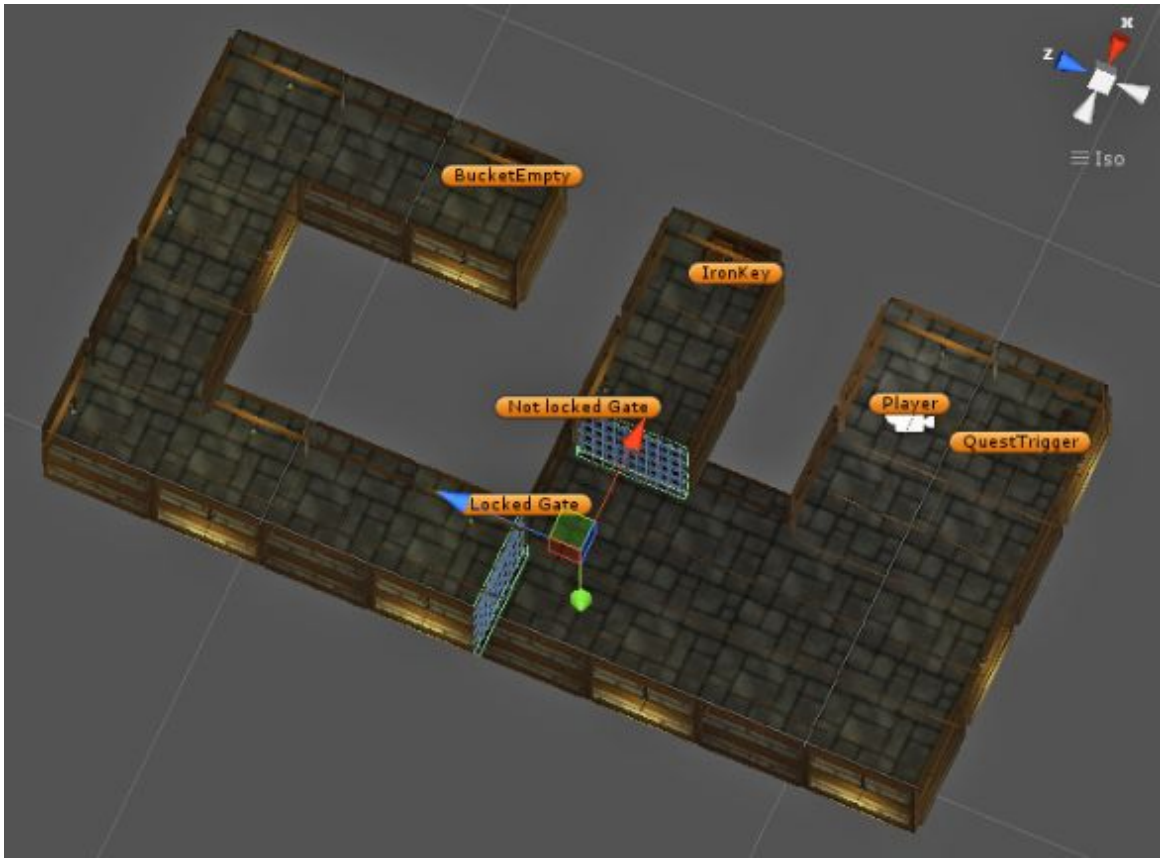
Next we check if isLocked is FALSE. If this is the case, we execute the code we have already developed to animate the gate from the "Animations" chapter. But if the value is still TRUE, we output the AudioClip lockedDoorMonologueClip via the audio source of the player.

```
 if (! isLocked) {
  if (! doorMovingState.isMoving) {
      anim.SetTrigger (switchTrigger);
              AudioSource.PlayClipAtPoint (doorClip, transform.position);
       }
    }
   else {
   player.audio.clip = lockedDoorMonologueClip;
        player.audio.Play ();
}
```

Now assign the file "chains" and lockedDoor MonologueClip to the audio clip "locked-door-monologue" in the inspector of the doorClip variable and press Apply so that these adjustments are transferred to the prefab.

Place and configure objects

Now you can define the starting point of the hero in your dungeon and place all objects of the quest and sub-quest in such a way that a playable plot arises from it (see Figure 21.21). In addition to the “BucketEmpty” water container prefab, this also includes the “IronKey” key prefab. Both should already have the necessary InventoryItem scripts with the respective IDs. You also need an instance of the “Gate” prefab to display the locked gate (isLocked is TRUE). You can also place multiple gates in your dungeon, e.g. B. are not locked. Set isLocked to FALSE for these gates .



When you have done this, you can now start your game and test whether all objects actually react the way you want them to. Make sure that the game-relevant objects are all clearly visible and clickable in the game. With a rigid camera like the one we are using here, it is best to position such objects at eye level or camera level. Use decorative objects such as wooden boxes or barrels to place the game-relevant objects on them, or you can

make these objects more conspicuous so that they can be seen more easily. So you can e.g. B. through the use of special shaders or cleverly placed light sources, the objects present and more eye-catching.

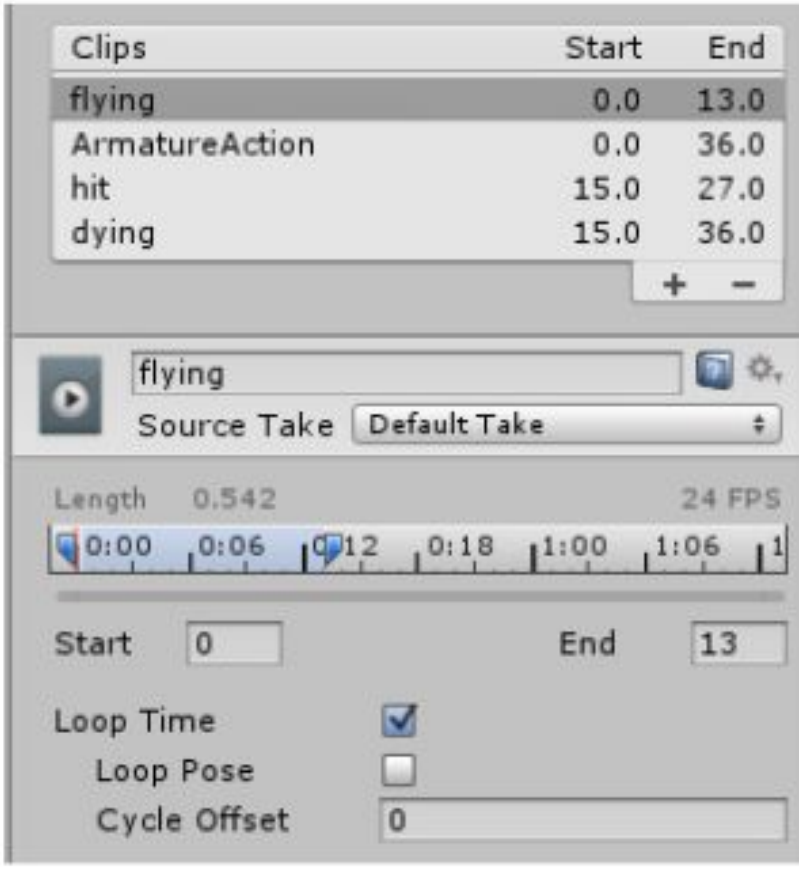
For this reason, in Figure 21.23, a Point Lightals child object was added to the key on the left motif, which was positioned just above the key. So that this does not illuminate the surroundings too much, the radius was reduced to 0.1 and the intensity increased to 8. On the right motif you can see the same key without the Point Light. Depending on the demands of the game, you can give the player a hand and help to solve tasks faster. In the end, you should always make something like this dependent on the target group and check it again with the help of test players. This is the only way to really find out in the end whether the respective tasks are realistic, too difficult or perhaps too easy

Create opponents

In addition to puzzles, a real adventure game also includes opponents who attack the hero. In our case, we want to integrate bats into our dungeon that should attack the player, but for this we need not only a bat model, but also animations that make the model move accordingly. We also need an artificial intelligence that controls the bat with the help of the path-finding functions already presented (see chapter "Artificial Intelligence") and also lets our hero attack.

Model, rig and animation import

Since the bat model "bat_03" already comes with its own animations, we no longer need to worry about creating the animations. However, these animations are often packed into a single large animation when exporting from the actual modeling program. This is also the case in our case, which is why we now have to take them apart again in the import settings and distribute them to various animation clips. To do this, go to the animations in the import settings of the bat model "bat_03" -Equestrian. You can now add new clips to the list using the plus sign below the animation clips list. You then specify the section of the associated animation using the timeline or the two input fields below. You can enter the name directly in the text field and overwrite the suggested name. In total, we now create three animations in this way: "flying", "hit" and "dying" (see Figure 21.24).



Note the Loop Time parameter, which is responsible for repeating the animation clips. Activate this parameter only for "flying"; it remains deactivated for the other animation clips. You should also set the scale factor to 0.1 on the Model tab of the Import Settings and check that the animation is also on the Rig tab Type "Generic" was selected. All adjustments must of course be confirmed with Apply and transferred to the model.

Configure components and prefab

First of all, we want to equip the GameObject for our opponent with all the necessary components that it needs.

1. Drag the object "bat_03" into the scene and rename it to "Bat".

2. Create a new "Enemy" tag and assign it to the GameObject.

3. Now add a ridigbody to the GameObject. Because we are going to control the GameObject via a NavMeshAgent, deactivate Use Gravity and activate Is Kine-matic.

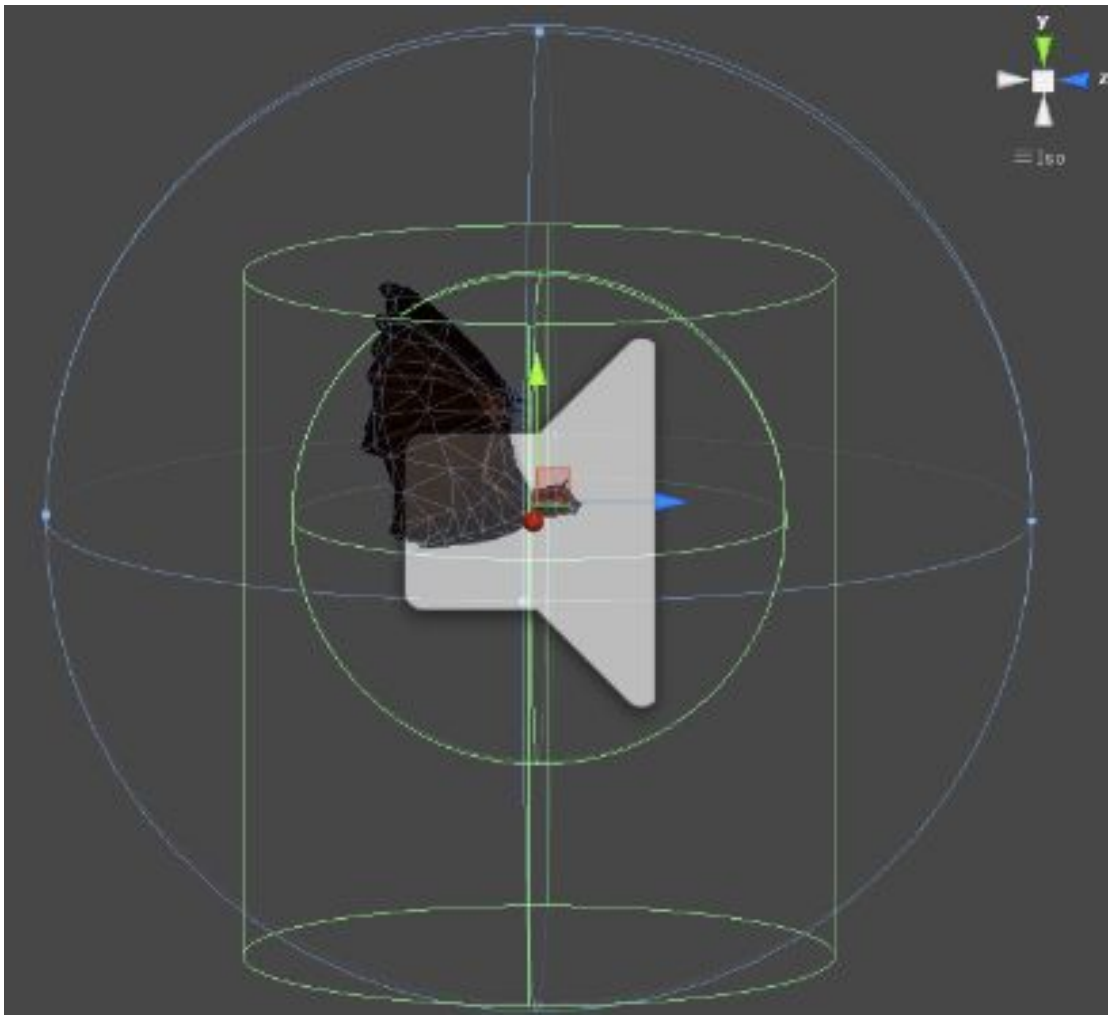
4. Enter “Bat” a sphere collider with a radius of 0.5 and activate the Is Trigger parameter.

5. Give the GameObject an AudioSource for which you deactivate Loop and Play On Awake. Assign the file "batCry" to the AudioSource as an AudioClip.

6. For the path finding , we finally add a NavMeshAgent component to “Bat” via Component / Navigation / Nav Mesh Agent . Please take the settings of this component from Figure 21.24.

7. Finally, drag the entire GameObject into the “Prefabs” folder to create a prefab out of it

When configuring the NavMeshAgent, the main thing to keep in mind is that our NPC is flying and not moving on the ground. But we don't want to make it too complicated and simply move the bat object within the NavMeshAgent to the upper end with a base offset and reduce the height to the upper end of the bat or its wing flapping. Since our bat initially only looks like a species If the watch is only to fly a predefined route slowly, we set the parameters Speed and Acceleration to relatively low values. Later, when attacking, we will increase the speed via a script.



Create Animator Controller

So that we can now animate our bat, we first need an animator controller. To do this, select the "Animations" folder in the Project Browser and create e.g. B. Via Assets / Create / Animator Controller an Animator Controller with the name "BatController". Now we can take care of the various animation states that our NPC should have. To do this, open the Animator window via Window / Animator and select the “BatController” that you just created. Our bat will have a total of four different animation states, which you can create using the context menu of the Animator window with Create State / Empty (accessible via the right mouse button). Now name and configure them as follows:

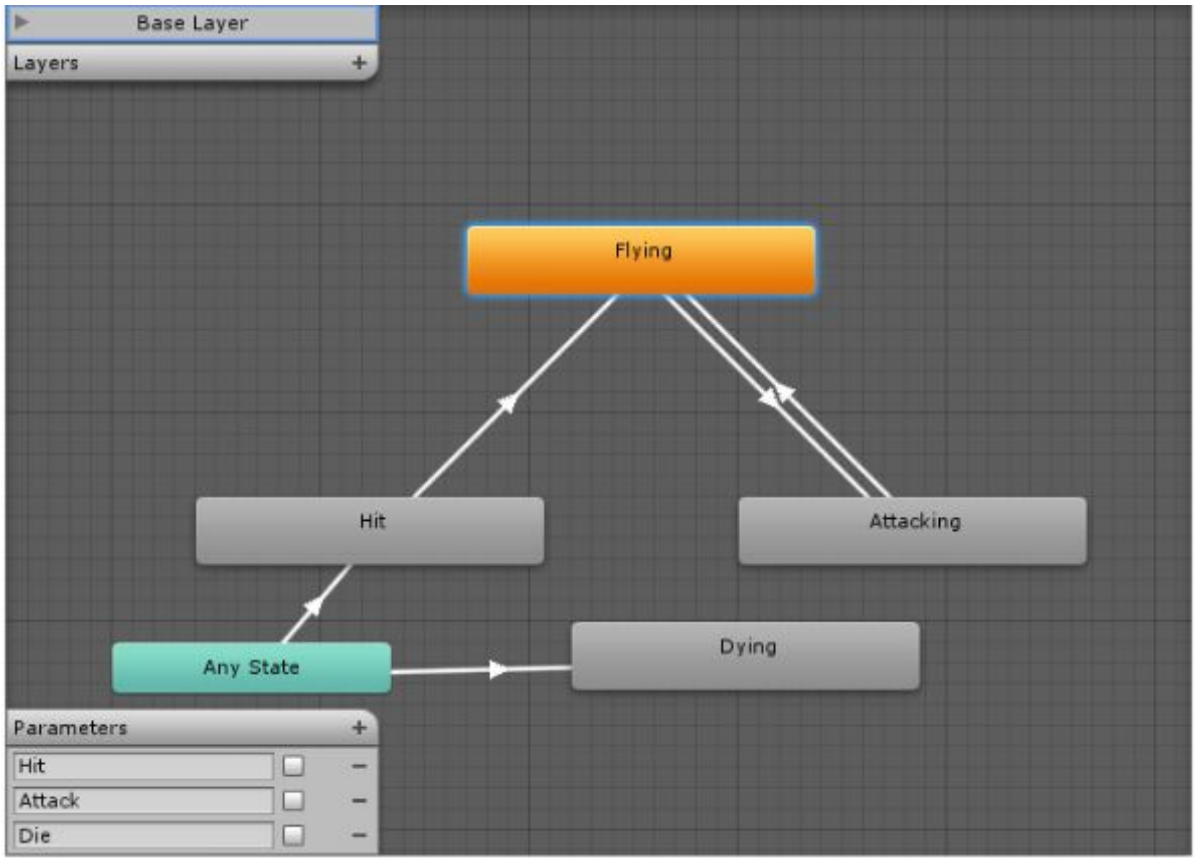
- Flying is set with "Set as Default" as the default animation state and should be played during normal flying. Assign the animation clip to

the animation clip "flying". If you cannot find the clip in the selection, make sure that you look in the selection window in the "Assets" area, not in the "Scene" tab.

- Attacking will be active when the bat deals damage to the hero.

Here, too, assign the normal "flying" animation clip. However, this should be played at five times the speed. That's why you set Speed to 5.

- Hit assign the animation clip “hit”. This should be played if the bat is damaged.
- Assign the animation clip “dying” to Dying. This animation state becomes active when the bat dies. To control these animation states, we now need a total of three different parameters, which you have to create in the parameter list:
- Hit as a trigger parameter
- Attack as a bool parameter
- You can now define the transitions that are responsible for switching between the different animation states (see Figure 21.26). To do this, right-click on the Start Animation State and select Make Transition . Then click with the left mouse button on the target animation State



In the Inspector of the respective transition, the conditions are now stored, for which fulfilled conditions should be changed from one state to the other:

- Any State -> Hit: Hit; The = FALSE.
- Any State -> Dying: Hit; The = TRUE.
- Hit -> Flying: Exit Time = 0.80.
- Flying -> Attacking: Attack = TRUE.
- Attacking -> Flying: Attack = FALSE.

Now you can assign the Animator Controller to the Animator component of our “Bat” prefab controller. You can use the "Bat" instance in the scene for this and, after assigning the controller with Apply, transfer these

adjustments back to the prefab. Since the animations do not transfer any movements to the model, you can also deactivate Apply Root Motion there.

Create a NavMesh

Next we want to create the NavMesh for the pathfinding function. To do this, open the navigation window via Window / Navigation. Now mark all static objects of your scene with the navigation static expert in the navigation window. Or set the static tick in the inspector of the respective GameObjects completely. In our case, this should be all “Wall”, “Floor” and “Ceiling” objects as well as the decorative objects “Create” and “Barrel”.

Make sure that all "Gate" objects and their child objects are not declared as static. Otherwise these will always flow into the route calculations as

obstacles, regardless of whether the gate is open or closed. Now we have to adjust the settings in the beacon tab of the navigation window on the NavMeshAgent.

- Set the radius to 0.6.
- Set Height to 1.4.

Now you can create the NavMesh using the Bake button. Make sure that you start the beacon again after moving, removing or adding new decorative items so that the NavMesh is also recalculated. After creating, you will see the NavMesh shown in blue. Now you can check whether you might have forgotten to call an obstacle static or mistakenly define a moving part as such.

Recognize surroundings and enemies

As soon as the bat notices our hero, it should attack him. To do this, we want to develop a script called EnemySonar that looks for the player, but we also want this script to ensure that the bat doesn't hit the portcullis. After all, we did not call the fall gate (“gate”) static, which means that the entire object is not considered an obstacle for the path calculation. If you use Unity Pro, you can simply equip the child object "gate_01" with a NavMeshObstacle component and neglect this part of the script. Unity then takes care of the fall gate independently. But if you're using the free version, I'll show you how you can do it without this component. For this script we now need four variables. The first variable fieldOfView indicates the field of view of the NPC, the second obstacleDetected indicates whether an obstacle was detected. The third variable, playerDetected, stores whether the player has been detected, and the last thing we need is a variable in which we store the “Player” GameObject. After all, the script is supposed to find exactly this object.

public float fieldOfView = 120;

```
public bool obstacleDetected = false;
public bool playerDetected = false;
private GameObject player;
```

For our search process, we need exactly one method that we want to call Searching . In this we want to search for the player using a raycast. However, since this is very computationally intensive and could sometimes affect the performance of many opponents in a game, we only want to carry out this process every half a second.

To do this, we start this method in the Start method with the InvokeRepeating command , where we also define the repetition rhythm . If you want to change the interval, you can easily set it here.

```
void start () {
    player = GameObject.FindGameObjectWithTag ("Player");
    InvokeRepeating ("Searching", 0.5F, 0.5F);
  }
```

In the Searching method we now send a raycast from our own position to the position of the player (we had already made the variable assignment in the Start method). If there is no obstacle in between, we now check whether the angle moves within the frame of fieldOfView. If this is the case, we call the StopSearching method , which sets the playerDetected variable to TRUE and stops the Searching method . Because as soon as the opponent has discovered the "player", he should follow it, even if it disappears around a corner and is no longer seen by the NPC. So that we can interrupt the search process from another script, we create StopSearching immediately as a publicly visible method.

```
void Searching () {
  Vector3 direction = player.transform.position - transform.position;
```

```
        Ray ray = new Ray (transform.position, direction.normalized);
      RaycastHit hit;
        if (Physics.Raycast (ray, out hit, 5)) {
      if (hit.collider.gameObject == player) {
                Vector3 dir = hit.transform.position - transform.position;
               float angle = Vector3.Angle (dir, transform.forward);
                if (angle <fieldOfView * 0.5f)
          StopSearching ();
   }
  }
 }
  public void StopSearching () {
        playerDetected = true;
       CancelInvoke ("Searching");
  }
```

Now let's get to the part we need to identify the closed portcullis. Since we have defined the sphere collider as a trigger collider, we use the OnTriggerEnter method to check whether we collide with the fall gate. If this is the case, we set obstacleDetected to TRUE. If we now step out of a collider with the tag "Door", e.g. B. because the gate is moving up, we set the variable to FALSE. We do this with the OnTriggerExit method.

```
 void OnTriggerEnter (Collider other) {
        if (other.gameObject.CompareTag ("Door")) {
     obstacleDetected = true;
 }
}
 void OnTriggerExit (Collider other) {
   if (other.gameObject.CompareTag ("Door")) {
           obstacleDetected = false;
  }
```

```
}
```

Of course, this solution also has some weaknesses, but for our purpose it should be enough for now. It is of course most elegant to use Unity Pro with the NavMeshObstacle component or another / own pathfinding system. This is of course also possible, but less suitable for beginners. When you have finished the script, add it to the “Bat” prefab of our bat.

EnemySonar .cs

The following listing shows the complete EnemySonar script that you have to assign to the bat prefab.

```
using UnityEngine;
using System.Collections;
  public class EnemySonar: MonoBehaviour {
    public float fieldOfView = 120;
            public bool obstacleDetected = false;
           public bool playerDetected = false;
          private GameObject player;
     void start () {
         player = GameObject.FindGameObjectWithTag ("Player");
         InvokeRepeating ("Searching", 0.5F, 0.5F);
   }
  void Searching () {
        Vector3 direction = player.transform.position - transform.position;
         Ray ray = new Ray (transform.position, direction.normalized);
         RaycastHit hit;
    if (Physics.Raycast (ray, out hit, 5)) {
      if (hit.collider.gameObject == player) {
```

```
    Vector3 dir = hit.transform.position - transform.position;
          float angle = Vector3.Angle (dir, transform.forward);
         if (angle <fieldOfView * 0.5f)
       StopSearching ();
   }
  }
 }

public void StopSearching () {
       playerDetected = true; CancelInvoke ("Searching");
        }
  void OnTriggerEnter (Collider other) {
    if (other.gameObject.CompareTag ("Door")) {
            obstacleDetected = true;
     }
   }
void OnTriggerExit (Collider other) {
     if (other.gameObject.CompareTag ("Door")) {
    obstacleDetected = false;
      }
  }
 }
```

Manage health status

Next we want to take care of the life management and the health of our NPC. Like the PlayerHealth script, this script is supposed to be built on top of the HealthController class, so we'll create a new script called EnemyHealth that inherits from HealthController and has the following variables (see Listing 21.69):

```
using UnityEngine;
using System.Collections;
  public class EnemyHealth: HealthController {
        public bool isShocked = false;
         public float shockedTime = 0.5F;
         public int eps = 1;
       private EnemySonar enemySonar;
  private EPController epController;
    private animator anim;
          private int hitTrigger;
        private int dieBool;
     void Start () {
       enemySonar = GetComponent <EnemySonar> ();
           epController = GameObject.FindGameObjectWithTag
("GameController"). GetComponent <EPController> (); hitTrigger =
Animator.StringToHash ("Hit");
  dieBool = Animator.StringToHash ("Die");
anim = transform.GetComponent <Animator> ();
}
```

As you can see from the listing, we need references to the previously programmed EnemySonar script, to the EPController script of the "Game Controller" and to our own animator component. We also want to address the “Hit” and “Die” variables from the animator later, which is why we save their hash values in variables. Of course there should also be experience points for killing an opponent, which is why we still need a variable eps . Finally, we need two variables for the state of shock that the opponent should fall into if he was injured. For this we roaring-chen a Boolean variable that saves the state, and a variable that we want the two methods

rewritable duration of this condition vorgibt.Im next step Damaging and Dying program out that in the base class Health controllers empty sind.Als call First we set the trigger parameter "Hit" in the Damaging method so that the Animator Controller switches to the "Hit" state. In addition, the sound of your own audio source should be played and the isShocked variable set to TRUE. Of course, this should be reset after a time specified by the variable shockedTime . We do this in a separate ResetShocked method .

```
public override void Damaging () {
   anim.SetTrigger (hitTrigger);
        audio.Play ();
      isShocked = true;
         if (! enemySonar.playerDetected)
        enemySonar.StopSearching ();
         Invoke ("ResetShocked", shockedTime);
   }
void ResetShocked () {
     isShocked = false;
}
```

We also want the player to be identified as soon as they injure their opponent. That is why we also call StopSearching from the EnemySonar script if it has not yet been detected. In the Dying method, we set the “Die” parameter to TRUE in addition to the “Hit” trigger. We also play the AudioClip here and set isShocked to TRUE. However, we now call the DestroyMe method instead with a second delay, which destroys the entire GameObject and assigns the experience points to the EPController.

```
public override void Dying () {
    anim.SetBool (dieBool, true); anim.SetTrigger (hitTrigger);
```

```
        audio.Play ();
        isShocked = true;
        Invoke ("DestroyMe", 1);
    }
void DestroyMe () {
    Destroy (gameObject);
        epController.AddPoints (eps);
 }
```

Now assign this script to the “Bat” prefab as well. Actually, that's it. Just make sure that you place enough “Stone” items in the dungeon so that the player can kill the bats later.

If you want to choose a structure as shown in Figure 21.28, note that the prefab does not have a rigid body. If the player first removes a lower stone, the others remain in the air. In this case, you should make the following changes in the scene for all "Stone" instances that are not at the bottom:

- Highlight these GameObjects in the scene.

- Add a rigid body to these.
- Enable Freeze Rotation for X, Y and Z.

As a result, the stone pyramid remains standing, but as soon as the lower stones are removed, the upper ones fall down.

EnemyHealth .cs

The following EnemyHealth script manages the health of an enemy NPC. Since it accesses the previous EnemySonar script, you need both scripts if you want to assign this to the GameObject.

```
using UnityEngine;
using System.Collections;
 public class EnemyHealth: HealthController {
  public bool isShocked = false;
      public float shockedTime = 0.5F;
       public int eps = 1;
     private EnemySonar enemySonar;
   private EPController epController;
  private animator anim;
  private int hitTrigger;
   private int dieBool;
    void Start () {
        enemySonar = GetComponent <EnemySonar> ();
        epController = GameObject.FindGameObjectWithTag ("GameController"). GetComponent
<EPController> (); hitTrigger = Animator.StringToHash ("Hit");
   dieBool = Animator.StringToHash ("Die");
          anim = transform.GetComponent <Animator> ();
    }
       public override void Damaging () {
        anim.SetTrigger (hitTrigger);
                    audio.Play ();
                    isShocked = true;
                if (! enemySonar.playerDetected)
            enemySonar.StopSearching ();
   Invoke ("ResetShocked", shockedTime);
```

```
    }
public override void Dying () {
        anim.SetBool (dieBool, true);
                    anim.SetTrigger (hitTrigger);
                    audio.Play ();
                      isShocked = true;
                     Invoke ("DestroyMe", 1);
        }
   void ResetShocked () {
       isShocked = false;
        }

void DestroyMe () {
        Destroy (gameObject);
           epController.AddPoints (eps);
    }
 }
```

Develop artificial intelligence

Now we come to the actual core of the NPC, artificial intelligence, or AI for short. Of course, this AI also includes the already configured path finding or enemy detection that we have already programmed. But these different elements still have to be merged and packed into a common logical structure. For this we are now creating a script called EnemyAI, which you should also add to the "Bat" prefab. Our hostile NPC now knows three different states that we now have to take care of:

- a standard state in which the NPC as a guard departs a predefined route and stops after the player out,
- an attack state in which he attacks the player,
- a state of emergency in which the NPC no longer moves,

because he z. For example, we are in a state of shock or the player is dead. To attack, we now need a damageValue variable to which we assign the damage value that the NPC can inflict on the player. In addition, we need a variable attackingSpeed , which determines the speed at which the bat attacks the player. After all, it should fly faster when attacking than when patrolling. We also need a Boolean variable that indicates whether the bat is currently able to attack or cause damage.

public float damageValue = 1;

public float attackingSpeed = 4;

private bool readyToHit = true;

For patrolling we need a transform array in which we save the waypoints between which the NPC should fly back and forth, as well as an index variable that saves the current index. We also need a time variable waypointPauseTime , which indicates how long the bat should wait at one waypoint before moving on to the next.

public Transform [] waypoints;

public float waypointPauseTime = 2;

private int currentWaypointIndex;

Finally, we need references to the player and its PlayerHealth script, to its own NavMeshAgent component and the EnemySonar and Enemy Health scripts. We also need access to our own animator and the hash value of the "Attack" parameter. For this we define further variables and assign the values in the Start method to them.

private GameObject player;

private playerHealth playerHealth;

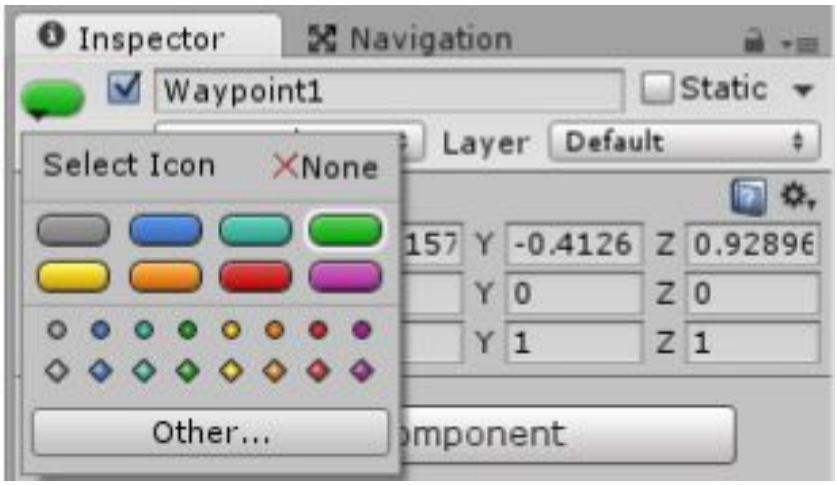
private NavMeshAgent agent;

```
private EnemySonar enemySonar;
private EnemyHealth enemyHealth;
private animator anim;
private int attackBool;
    void start () {
     agent = GetComponent <NavMeshAgent> ();
                player = GameObject.FindGameObjectWithTag ("Player");
                 playerHealth = player.GetComponent <PlayerHealth> ();
                 enemySonar = GetComponent <EnemySonar> ();
                  enemyHealth = GetComponent <EnemyHealth> ();
                anim = transform.GetComponent <Animator> ();
                attackBool = Animator.StringToHash ("Attack");
}
```

First of all, let's take care of patrolling. The main task, controlling and moving the NPC, is already taken over by Unity's Pathfinding module. All we have to do is specify the target position where it should fly next, so now we always want to set two waypoints for each bat, between which the NPC should now fly back and forth. If you want, you can of course take several. We define the positions via Empty GameObjects, which we then assign to the waypoints array.

1. The best thing to do is to add a new, empty GameObject to the scene and name it “Waypoint1”.

2. Now assign it its own icon in the Inspector.

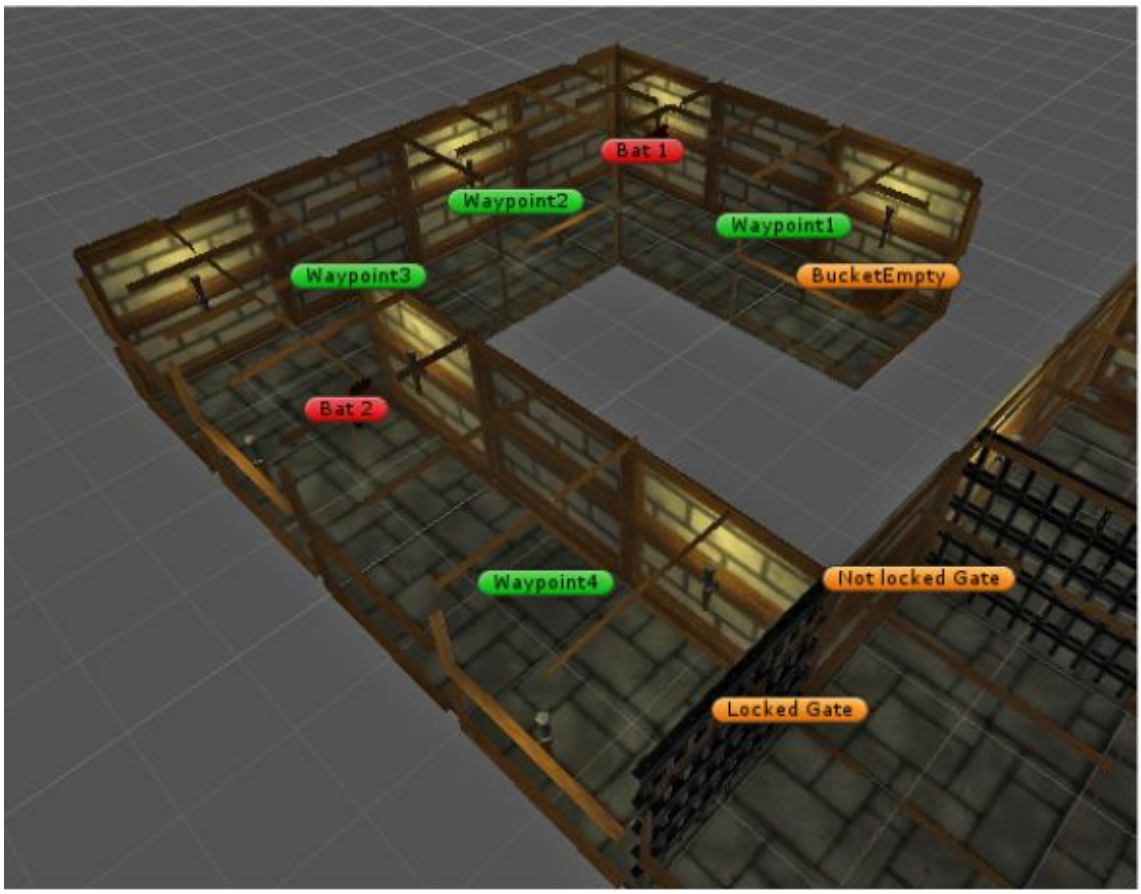


3. Place the waypoint at an end position of the route that the bat should fly, e.g. B. at position (7,1,10).

4. Copy the waypoint and rename it to “Waypoint2”.

5. Position "Waypoint2" at the second of the two positions between which the bat should fly back and forth, eg. B. to position (5,1,13).

6. Now drag a bat instance into your game and place it somewhere between the two waypoints and drag both waypoints onto the waypoints -Ar ray. If you want to place a second or third bat in the dungeon, repeat the procedure.



Now we come to the actual logic that controls the flying back and forth. We program this in a method called WaypointWalk . In this we check whether the distance to the target is less than or equal to the distance at which we want to stop. If this is the case, we change the target by incrementing currentWaypointIndex or, if we are already at the highest index, setting it to 0. Then we start the WaypointPause method , which lowers the speed property of the NavMeshAgent for a short time so that the NPC does not immediately fly to the next destination.

```
void WaypointWalk () {
        if (agent.remainingDistance <= agent.stoppingDistance) {
           if (currentWaypointIndex == waypoints.Length - 1)
                        currentWaypointIndex = 0;
    else
```

```
        currentWaypointIndex ++;
                    StartCoroutine ("WaypointPause", waypointPauseTime);
          }
       agent.SetDestination (waypoints [currentWaypointIndex] .position);
    }

IEnumerator WaypointPause (float seconds) {
        float oldSpeed = agent.speed;
                    agent.speed = 0.1F;
                    yield return new WaitForSeconds (seconds);
                     agent.speed = oldSpeed;
}
```

In the next step we want to take care of inflicting the damage. We want to do this quite simply. The bat immediately damages the player if they collide. In the "OnCollision Methods" section of the "Physics in Unity" chapter, I explained that these methods are only triggered if at least one of the two colliders colliding Objects has a rigid body with the Is Kinematic parameter deactivated. Since unfortunately neither the bat (due to the NavMeshAgent) nor our “player” or the movable “gate” owns it, we cannot work with it. Alternatively, however, we can use the OnTriggerStay method, which does not require this. When calling OnTriggerStay , we now check whether the NPC is still alive and whether the colliding object is the "Player" object, of course should still be alive. If all of this is the case, the damage will now be done, but this method will be called in every physics cycle as long as there is another collider in it. So that the player is not dead after a second stay in the collider, we also introduce the readyToHit variable, which only allows damage to be inflicted once per second, while the variable is set to FALSE immediately as soon as damage is inflicted , this is only set back to TRUE with a time delay using the SetReadyToHit method . In addition, we also set the Boolean parameter "Attack" of the

animator in this method in order to switch to the "Attacking" status for half a second.

```
void OnTriggerStay (Collider other) {
          if (enemyHealth.health> 0) {
    if (other.gameObject == player && playerHealth.health> 0) {
             if (readyToHit) {
      readyToHit = false;
         if (! enemySonar.playerDetected)
         enemySonar.StopSearching ();
    other.gameObject.SendMessage ("ApplyDamage", damageValue,
SendMessageOptions.DontRequireReceiver);
     StartCoroutine ("SetReadyToHit");
     }
    }
   }
  }
 IEnumerator SetReadyToHit () {
  anim.SetBool (attackBool, true);
        yield return new WaitForSeconds (0.5F);
             anim.SetBool (attackBool, false);
    yield return new WaitForSeconds (0.5F);
     readyToHit = true;
}
```

Also note the call to StopSearching when inflicting the damage. If the player should "bump into" the NPC from behind, it is of course more than logical that the bat notices the hero and attacks him at the latest. We now come to the actual logic in the update method for executing and switching between the different statuses is responsible. As long as the player is not discovered, we execute the WaypointWalk method . If this is not the case, the question now arises whether she is attacking the player or is in the exceptional state where she can no longer move. For this we now check whether

- the bat is still alive
- the player is still alive

- an obstacle (portcullis) has been registered
- the bat is in shock
- the bat is ready to strike again If everything is okay, the speed parameter of the NavMeshAgent is set to the attack speed, the acceleration is increased to two and the position of the hero is assigned as the target.

But should z. For example, if the bat is in shock or the player is dead, the NavMeshAgent's Stop method is called and the NPC stops on the spot

```
void Update () {
    if (! enemySonar.playerDetected) {
        WaypointWalk ();
    }
        else if (
            enemyHealth.health> 0 && playerHealth.health> 0 &&!
enemySonar.obstacleDetected &&! enemyHealth.isShocked && readyToHit)
 {
   agent.speed = attackingSpeed; agent.acceleration = 2;
agent.SetDestination (player.transform.position);
}
else
{
  agent.Stop (true);
  }
}
```

If you have now programmed the script and added it to the "Bat" prefab, we are finally finished with our opponent

Opening scene

For a real game, of course, we not only need the actual game and its level, but also an opening scene with a start menu, but such a start scene does not only serve to actually start the game. Often old saved games are loaded here or variables are assigned default start values. We also use this in our opening scene to assign the available life points of our player.

Create a starting scene

For such an opening scene, of course, we also need our own scene, which we can now create quickly and easily.

1. Open the dungeon scene and save it with File / Save Scene as under the new name “Startmenu” in the “Scenes” folder.

2. Drag the “Main Camera” out of the “Player” and delete the rest of the hero, ie the entire “Player” object.

3. Position the "Main Camera" GameObject in a suitable place, e.g. B. facing a wall with torches.

4. Remove all script components from the "Game Controller".

5. Add “introMusic” to the audio source of the “Game Controller” again and activate Loop and Play On Awake.

6. Remove all child objects from the "GUI" container except for "MessageText".

7. Delete all other elements that do not belong to the actual dungeon, ie all "Bat" instances, all "Stone" instances, all "BucketEmpty" instances, all "Gate" instances, the "IronKey", the "QuestTrigger", the "Waterdrop" particle system, the "Inventory" object and the "TooltipText" object. In the end, you should now get a scene that you can start without doing anything great.

8. If this now works properly, add both scenes to the Build Settings. To do this, start each scene and then execute Add Current in the Build Settings .

9. Finally, move the new scene “Startmenu” up in the Build Settings so that it has the index 0 and the scene “Dungeon” has the index 1

Create start menu

Now we want to create a small menu script that we can use to start and end the game. We also want to use a third button to display a small descriptive text that provides information about the game. For this we use the "MessageText" object again, which we have already used several times in the dungeon scene. For this, create a script called MainMenu, which you then add to the "Game Controller"

Start menu with OnGUI

To define the GUI objects, we need a variable of the Rect type in the OnGUI variant. In the start method, we then initialize the variables or build the references to them. If you would like to have the buttons in a different size, you simply need to adjust the width and / or height values accordingly.

```
private GUIText messageText; private Rect rect;
 void start () {
      messageText = GameObject.FindGameObjectWithTag ("Message"). guiText;
        messageText.text = "";
         rect = new rect (50,50,200,30);
   }
```

First, we define the three buttons in the OnGUI method and call up a separate method for each of them, in which we then write the associated code to be executed. For the positioning of the controls, we have the fixed

Y position 50 for the first button and increase it by 50 with each additional button so that they are displayed one below the other at the end.

```
void OnGUI () {
        rect.y = 50;
   if (GUI.Button (rect, "Start")) {
     StartGame ();
}
        rect.y + = 50;
          if (GUI.Button (rect, "Control and Info")) {
              ShowInfo ();
}
     rect.y + = 50;
      if (GUI.Button (rect, "Quit")) {
     CloseGame ();
   }
}
```

In the StartGame method , we now start scene 1 with LoadLevel. We also use the PlayerPrefs class to write down the start value for the "LPs" parameter, which we use for the life points in the LifePointController of the dungeon scene. We also delete the content of messageText if any content is displayed there.

```
void StartGame () {
    messageText.text = "";
           PlayerPrefs.SetInt ("LPs", 3);
             // start value
         Application.LoadLevel (1);
}
```

In the ShowInfo method, which is called by the "Control and Info" button, we only assign an info text to the messageText variable.

```
void ShowInfo () {
    messageText.text = "You have been banished to a vaulted cellar. \ n" +
"Monsters and puzzles are waiting for you there. \ n \ n" + "To control: \ n" +
"A / D: Go left / right \ n "+" W / S: go forward / backward \ n "+" Q / E:
turn left / right \ n "+" left mouse button: pick up objects \ n "+" right mouse
button / space bar: attack ";
}
```

Finally , there is the CloseGame method, which calls the Quit method of the Application class and thus closes the game.

```
void CloseGame () {
  Application.Quit ();
}
```

Note that the Quit method cannot be tested in the editor. To test this functionality you should build or create the game as a stand-alone.

If you now start the game with the “Start menu” scene, you should see the menu from which you can also start the game. If the player now dies and has lost all lives, he returns to the menu after pressing the [Space] key (or [Escape] or [Return]).

MainMenu .cs

The complete MainMenu script that you should add to the "Game Controller" looks like the following listing.

MainMenu.cs

```
using UnityEngine;
using System.Collections;
 public class MainMenu: MonoBehaviour {
    private GUIText messageText;
      private rect rect;
        // Use this for initialization
  void start () {
```

```
    messageText = GameObject.FindGameObjectWithTag ("Message").
guiText; messageText.text = "";
      rect = new Rect (50,50,200,30);
     }
  void OnGUI () {
   rect.y = 50;
          if (GUI.Button (rect, "Start")) {
       StartGame ();
    }
    rect.y + = 50;
        if (GUI.Button (rect, "Control and Info")) {
         ShowInfo ();
       }
     rect.y + = 50;
          if (GUI.Button (rect, "Exit")) {
        CloseGame ();
    }
  }

void StartGame () {
      messageText.text = "";
              PlayerPrefs.SetInt ("LPs", 3);
              // start value
        Application.LoadLevel (1);
    }
    void ShowInfo () {
messageText.text = "You have been banished to a vaulted cellar. \ n" +
"Monsters and puzzles are waiting for you there. \ n \ n" + "To control: \ n" +
```

```
"A / D: Go left / right \ n "+" W / S: go forward / backward \ n "+" Q / E:
turn left / right \ n "+" left mouse button: pick up objects \ n "+" right mouse
button / space bar: attack ";
}
  void CloseGame () {
  messageText.text = "";
   if (Application.isWebPlayer) messageText.text = "This function is" + "" +
"not supported in the web player";
 else
    Application.Quit ();
 }
}
```

Web player adjustments

If you now want to publish the game, it is of course first of all important that you add all the scenes belonging to the game to the "Scenes in Build" list. In our case, however, there are two other special features, especially with the web player build, that you should pay attention to and adapt accordingly.

Change web player template

The first peculiarity concerns attacking. Since the player can throw the stones not only with the [Space] key, but also with the right mouse button, there is a conflict with a standard Unity function. With the right mouse button, the player normally calls up a context menu in the web player with which he can switch to full view, among other things, so that this context menu does not pop up every time the player wants to throw a stone, we can now suppress this context menu. To do this, go to the Player Settings of the

web player via Edit / Project Settings / Player and select “No Context Menu” in the Web Player Template area.

Changeover to uGUI

If you have Unity version 4.6 or higher, you will already be able to use the new uGUI system. Since this was still in the beta stage when this book was written, I initially assumed the conventional GUI options in the previous sections for reasons of compatibility, i.e. the GUI elements and OnGUI programming.

If you want to change the game to uGUI, you must of course first remove the GUIElements from the two scenes and replace them with uGUI controls. In addition, the access in the scripts must be adjusted accordingly. Let's go over these things to adjust. Download the sample game with a uGUI interface. On my website http://www.hummelwalker.de/buch-zusatzmaterial/ you can download the complete sample game, converted to uGUI. You can find the password in Chapter 1 of this book.

Script adjustments

First replace the following types in the scripts Inventory, InventoryItem, ItemProperties, MainMenu and PlayerHealth, HoverEffects, EPController, LifePointController, Shooting, WaterQuest, which of course also includes the transfer variables of the methods: ƒ GUIText by text ƒ GUITexture by image ƒ Texture2D by Sprite (except in the HoverEffects script) In order for the text and image types to be available in the scripts at all, you must first include the UnityEngine.UI namespace in these scripts with using. Also note the GetComponent assignments that you make with these data types do in the scripts. Of course, you have to change these as well.

```
// messageText = GameObject.FindGameObjectWithTag ("Message").
// GetComponent <GUIText> ();
// GUIElementmessageText = GameObject.FindGameObjectWithTag ("Message").
GetComponent <Text> (); // uGUI
```

Inventory adjustments

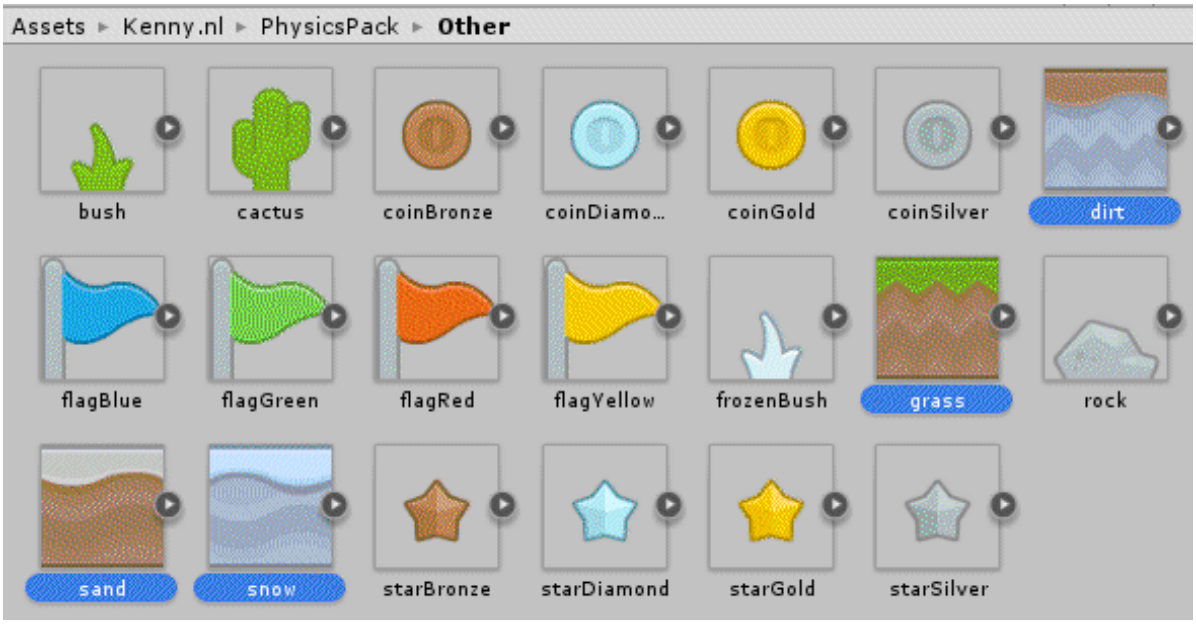
After these general changes, you must adapt the access to the variables to the new types in the inventory script. The properties of the image component in particular differ from the GUITex-ture component. Due to the slightly different reaction of this component, we will therefore first hide all graphic elements in the UpdateView method. Then we only activate those that should also have content (graphics) and assign them accordingly.

```
void UpdateView () {
  int index = 0;
       i nt guiCount = guiItemTextures.Length;
            for (int i = 0; i <guiCount; i ++) {
          //guiItemTextures[i[.texture = null; // GUIElement guiItemTextures [i] .enabled = false; // uGUI guiItemQuantity [i] .text = ""; } foreach (KeyValuePair <string, ItemProperties> current in items) { //guiItemTextures[index[.texture = current.Value.texture; // GUIElement guiItemTextures [index] .sprite = current.Value.texture; // uGUI guiItemTextures [index]. enabled = true; guiItemQuantity [index] .text = current.Value.quantity.ToString (); index ++;
}
}
```

CHAPTER 4

Create a simple 2d 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.

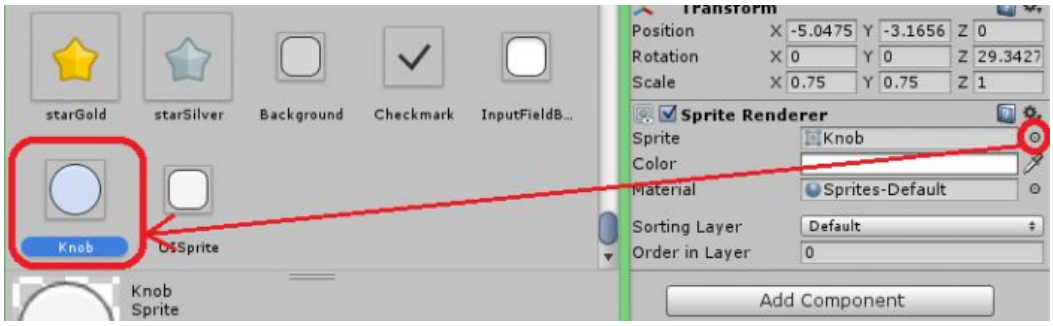
With this we have identified the basic programmable ballistics, and we have some small auxiliary programs for secondary functions.

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);
        }
```

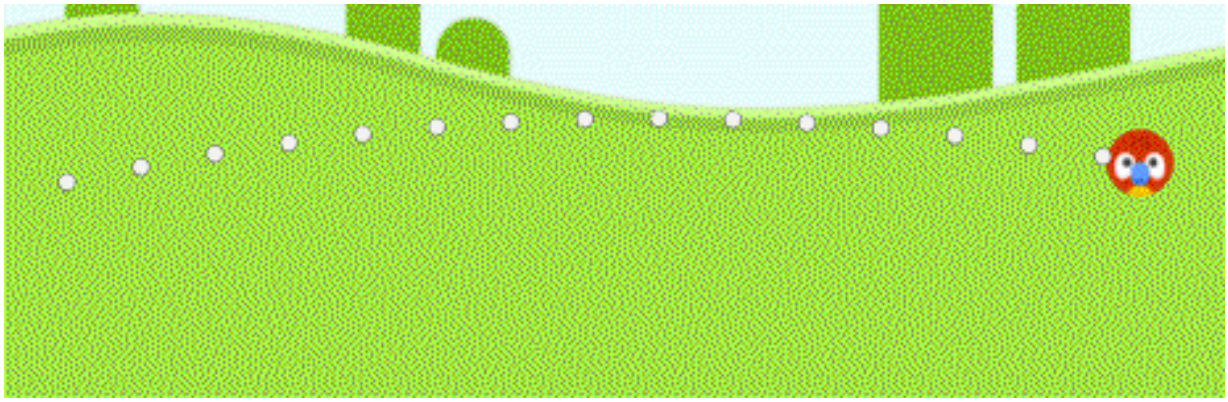
```
        AddPathPoint ();

        // 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:

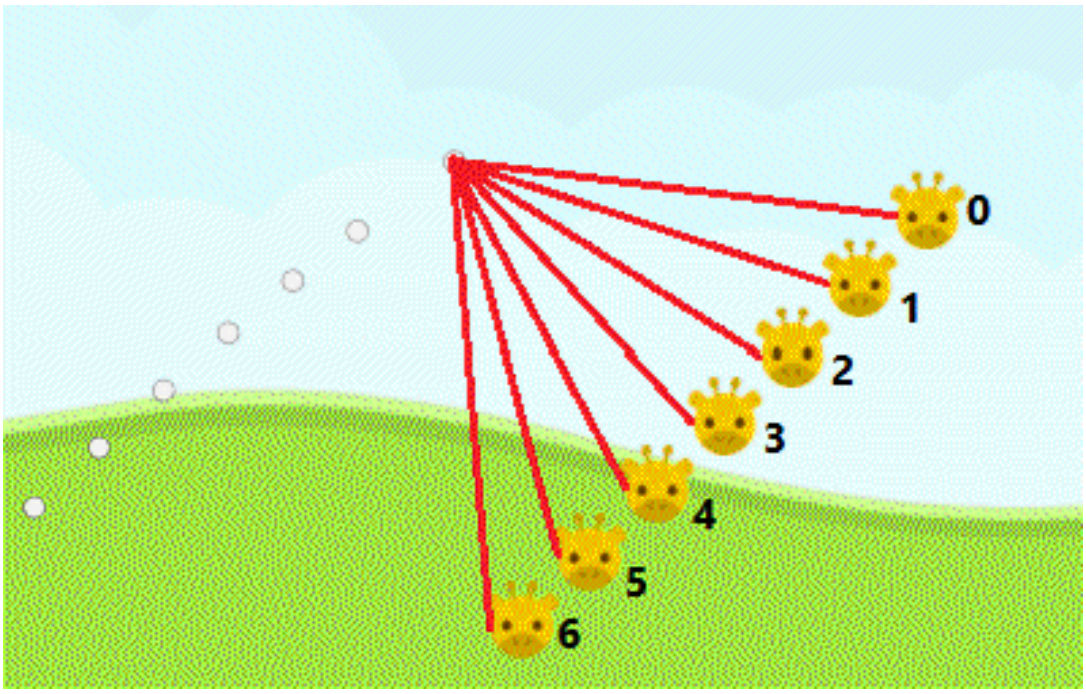
```
using UnityEngine;
using System.Collections;
```

```
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);
    }
}
```

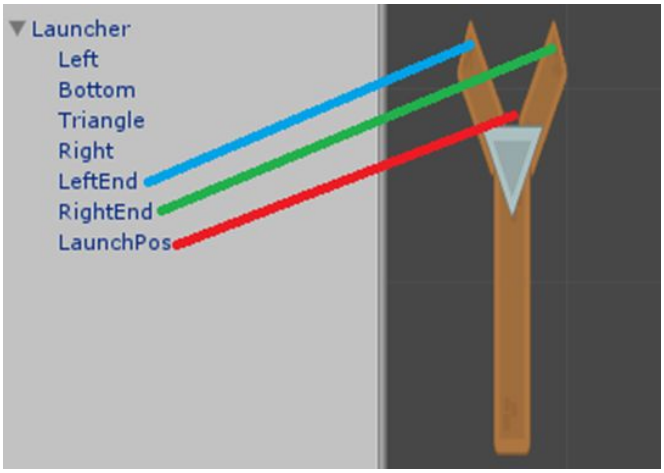
The idea of doing this attack is to receive the DoSpecialAttack message and then create the specified number of fragments using the specified template. In order to prevent collisions between fragments and some of them and also between fragments and the original projectile - where a collision can occur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

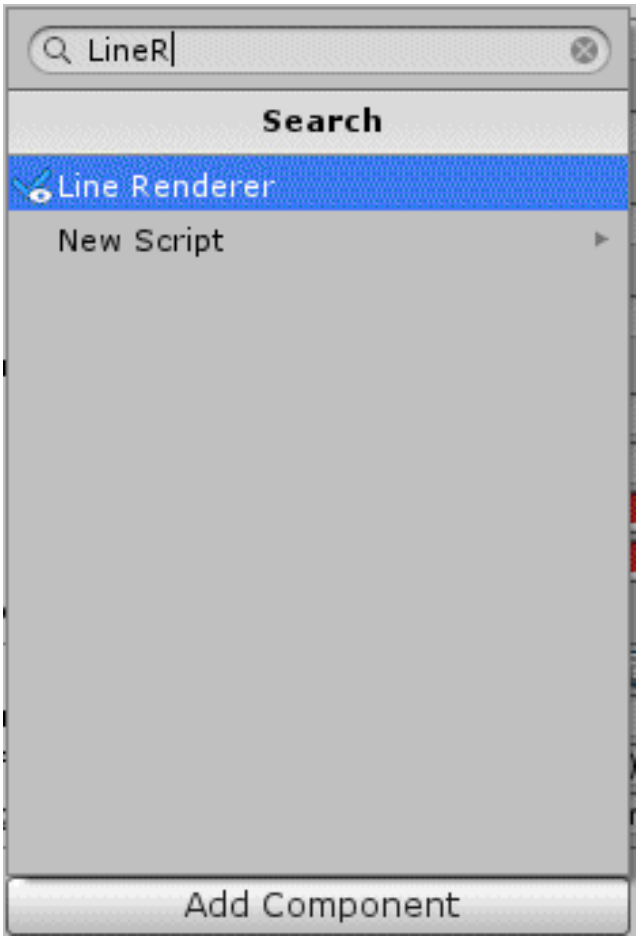
Here I will talk a little bit more about the BringCurrentProjectile () function to explain the mechanism you use to achieve the smooth movement of the projectile from its current position towards the launch site. The smooth movement in game engines depends on the elapsed time factor between each two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

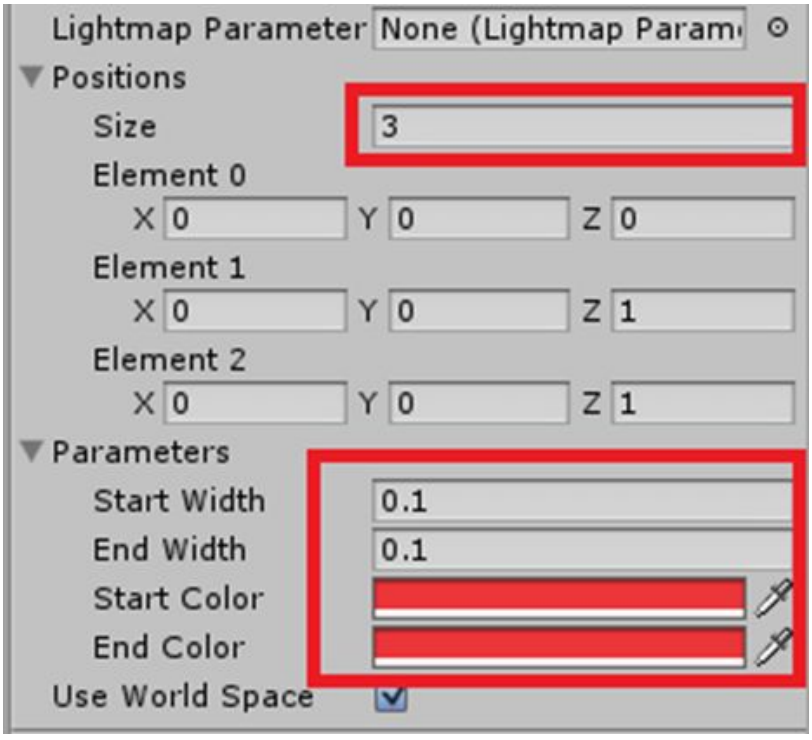
The three functions HoldProjectile (), DragProjectile (), and ReleaseProjectile () call the Hold (), Drag (), and Launch () functions of the current projectile currentProjectile. The function that needs some explanation here is DragProjectile () because it contains an additional step that is not present 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

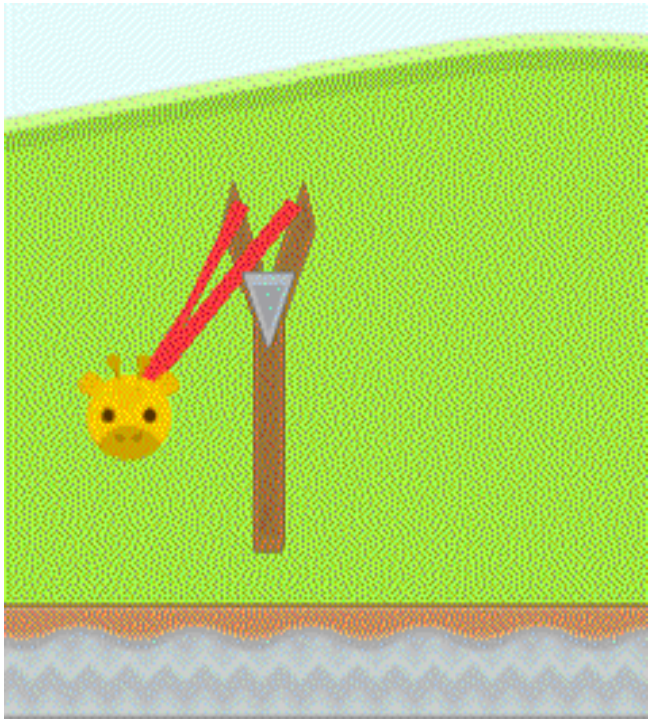
described in the following narrative:

```
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:

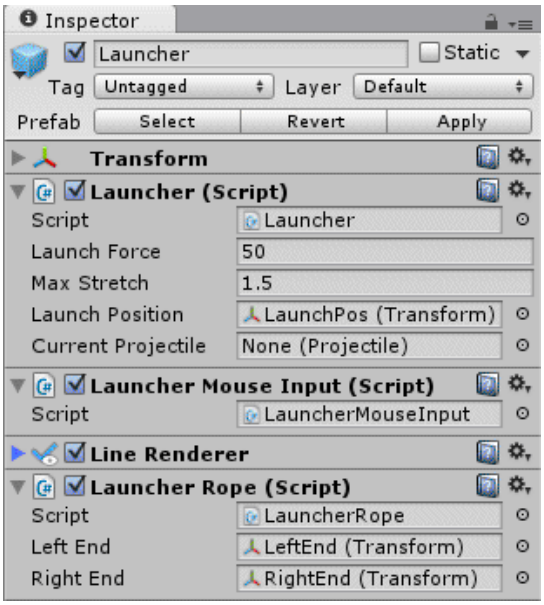
```
using UnityEngine;
using System.Collections;

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?
```

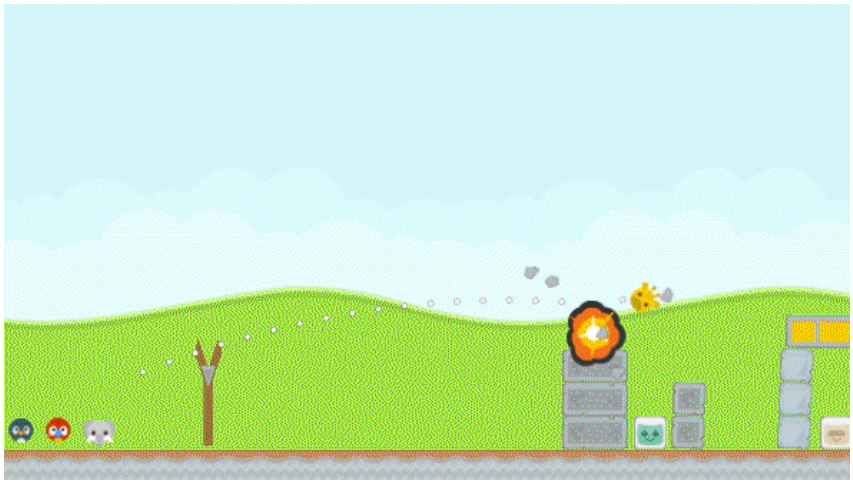
```
            if (projectileCol.bounds.Contains (mouseWorldPos))
            {
               // Yes, that is, the mouse button was pressed over the projectile
               // Hold the projectile
               launcher.HoldProjectile ();
            }
         }
      }
   }

   // 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!



*