6+

GENNADY GURYANOV

Making PC game together

Gennady Guryanov

Making PC game together

https://litres.ru/74152559
ISBN 9785007015523

Аннотация

The book tells about the creation of the software core of a single-player 3D computer game in the adventure genre, a first-person quest-puzzle in the Unity3d development environment for a personal computer.

The full cycle of creation from scratch to a working version of the game is presented, without graphic content. The book is addressed to schoolchildren, beginners and enthusiasts.

Содержание

Making PC game together

Gennady Guryanov

Cover designer Gennady Guryanov
Illustrator Gennady Guryanov

ISBN 978-5-0070-1552-3
Created with Ridero smart publishing system

Making a PC Game Together

About the book

The book tells about the creation of the software core of a single-player 3D computer game in the adventure genre, a first-person quest-puzzle in the Unity3d development environment for a personal computer.

The full cycle of creation from scratch to a working version of the game is presented, without graphic content. The book is addressed to schoolchildren, beginners and enthusiasts.

The author of the book has been a developer of computer games since 2006, he has a large number of released projects under his belt, including those made alone, for example, computer games in the adventure and quest genre AA-class (double A) «Secret Doctrine» and «Fish».

From the author

Dear reader!

Modern computer technologies have opened up previously unseen opportunities for the implementation of creative ideas with bright artistic and semantic possibilities. You now have the opportunity to create beauty, exciting kind stories, fantastic colorful antics and foggy thoughtfulness!

Being the creator of several games for a personal computer made alone — 4 small and 2 very large, I, the author of this book, will show you the way to the daily joy of creativity and your success. Create your worlds, give them to people, sow in people's souls the trembling of your heart and kind stories…

Friends! I warn you against making different games-surrogates such: «horror stories», «horrors», violence, gloom and stupidity.

Take responsibility boldly for those ideas that your heart tells you! Welcome to the world of creativity!

Let's learn how to create a computer game together!

1. What are we doing?

The structure of this book is made in such a way that it is suggested to read it without skipping chapters. The chapters in this book are made speculatively and to a certain extent conditional.

In the book we will make the basis of a computer game with puzzles, a walker in three-dimensional space, a game for a personal computer, for one player. That is, this game will not be with flat objects — «flats», but just like in the real world — 3d — with three-dimensional ones. In such a game, events develop by solving puzzles, riddles. The abbreviation 3d means «3 dimensional», i.e. 3-dimensional space. This is a space in which there are 3 dimensions: length, height and width, for example, the one in which you and I live in real life. This also means that in such a game you can walk, run, jump and even fly like in real life. All other games that are called 2d mean that their game space is flat or two-dimensional, it has only length and height or vice versa: length and width, or even height and width. For example, a flat 2d space is a piece of paper.

How does a computer game work? It can be conceptually divided into two parts: the visual part — everything that the player sees (pictures, artistic design, animations, objects, the player's location, etc.) and the program code part (what events are in the game and how they happen). These two parts can work

independently of each other. For example, if we have a certain number of 3D models, we can arrange them in a certain way — beautifully, and such a three-dimensional scene can already please the eye in itself. And vice versa, having a program code, you can, to put it cheerfully, hang almost any visual part on it, from a stylized cubes to detailed objects and the environment.

2. «What's the big deal?»

The development environment or, as people call it, the «engine» acts as a pot in which the soup is cooked during the game creation process. This «stock pot» has certain dimensions, the material it is made of, and it holds all the soup. Having different products for the soup (cabbage, carrots, potatoes, onions), you can put them in «stock pot» of different shapes and sizes, and even in very small ones, but the essence of the «soup» will not change. Today, the most popular «stock pots» are on everyone's lips: Unreal Engine and Unity3d Engine. This book tells about creating a game in the Unity3d environment. There is an opinion that the Unity3d development environment is simplified and created for developing primitive phone games. This is not true. If we know what we want to get, then in Unity3d we can achieve the highest graphic results. It is also widely believed that only the Unreal Engine development environment can produce «photorealistic» results for games and programs. This is a misconception based on competitive advertising, nothing more. As for the most important aspect — the convenience of creating a game, Unity3d is undoubtedly the best choice for a creative person. The deepest intuitiveness of Unity3d and its simple logic allows you to create what you want almost immediately, just on the fly, without even knowing the program. What does this intuitiveness mean? It means that

if an idea arises: «what if I try to do it this way, connect it, rotate it?», then you can immediately do it as logic, idea, thought suggests, and «this» will have the status — «you can». No other actions from the category «to rotate it, you must first add such an object and set the rotation point» are needed. For example, if you have a 3D model of an excavator with several parts, say, with a cabin, tracks, a bucket, then in Unity3d we can instantly, right in the scene, with one drag of the mouse, bind one part to another, in any order, for example, a cabin to a bucket, and a bucket to tracks, and this fantastic design is ready for use in this form from that moment, preserving the hierarchy of the changes made. The example, of course, is not very practical, but it shows that in Unity3d the game creator has almost complete creative freedom and intuitive understanding. Because design, assembly on the fly, an idea that has arisen, which can be immediately implemented, all this is the foundation of creativity and joy. On the contrary, in Unreal all of the above and much more is a continuous «headache» and restrictions: «first you need to do this, then like this, after that another trick…". And there is no need to talk about intuitive design creation and free assembly of elements directly in the scene in Unreal. Without going into additional details, we can say poetically: if working on Unity3d can be compared to the work of an artist's elegant brush, then working on Unreal can be compared to the work of a log in your hands, with which you are trying to draw something. The statement that the Unreal development

environment is suitable for projects created somewhere outside (in an external 3D modeling program, for example, in 3ds Max), given the capabilities of today's software, does not stand up to criticism. And it is even more impossible to talk about creativity and intuition in this case.

3. Deep Dive

To have a good understanding of how the game being created will work, it is necessary to understand that any computer program (a video game is primarily a computer program) on any computer on Earth is executed by the processor from top to bottom sequentially, command after command (we will not consider multithreading and several processor cores). A computer program is a set of notes in a notebook, in lines from top to bottom, just like a school notebook on the English language with an essay on the theme of Rabindranath Tagore's poems. Only instead of the artistic text of the essay, the programmer writes down commands for the processor in the notebook in the selected programming language. After launching such a program, the processor executes it, as I already said, from top to bottom, sequentially, command after command. In the last decades, concepts have appeared in programming: objects, instances, methods, etc., however, the essence of program execution has not changed — from top to bottom, sequentially.

The Unity3d development environment gives the creator the opportunity to make his game by placing objects and events as if separately: here you can put a character with a separate control program (in Unity3d this is called a script or scenario), here a grocery store with a separate program, and here a door leading to another level with its own program for transferring

to the level. It is necessary to remember that this is done only for the convenience of the game creator, and in fact all these «separate» programs are executed by the processor in order one after another, line by line.

How does this execution happen? The computer's microprocessor executes the program in cycles. Let's say there are 10 lines of code in the program. It is executed from top to bottom just like you read the text of this book. Moreover, until the depths of the processor and the computer's memory have finished calculating, for example, the third line of code, all the remaining lines from the fourth to the tenth will never be executed. All these seemingly simple nuances are extremely important as a basis. Knowing such nuances, you will be able to focus on creativity, on inventing «what can be implemented in the game», and not think «how can I do this, will there be an error and the game freeze?». So, when the execution ends after the tenth line, the cycle is repeated again from the first, and so on ad infinitum, until some other events occur, for example, the player simply exits the game.

Let's continue. The endless cycles just described are repeated every frame during the game. The more of these cycles the microprocessor manages to calculate in one second, the better and faster the game works. If the processor manages to calculate, for example, 60 such cycles in 1 second, then the game works at a speed of 60 FPS (from English Frames Per Seconds — frames per second) and the appearance of the game on the

player's monitor changes 60 times in 1 second, i.e. 60 frames per second. Such a game is perceived by the player's eye pleasantly, smoothly, without jerks and slowdowns. And vice versa, the fewer cycles per second the processor performs, the lower the FPS of the game and the quality of its work. It is also necessary to keep in mind that the processor is assisted in these calculations by the video card. We can say that this is the second processor of the computer, but designed only for graphics processing, in other words, only for drawing all the pictures and 3D objects of the game.

I want to note that there is no need to chase high FPS rates when developing a game, since today computer processors are able to cope with different loads equally successfully. Focus on creativity and on creating what you have in mind, and do not waste time on unnecessary searches for «salvation» of an additional 10 frames per second. In the end, computer technologies are developing so dynamically that in a period of 2—3 years, what works a little slowly today, tomorrow will work completely normally. Think about creativity, not about FPS.

4. C#

When creating the program code for our game, we will use the C# programming language (pronounced «see sharp»). This is a very convenient and flexible programming language. Its huge advantage is that it is strictly typed, i.e. in most cases it simply will not allow you to make a mistake when creating a program. For example, in this language you cannot simply say that a string of text is the same as a number. In other programming languages, this is possible, which in the process of creating a program can lead to the occurrence of implicit errors that lead to unpredictable consequences, and what is worse, the place where such an error occurred in the code is very difficult to find. To write programs in C#, we will need Microsoft Visual Studio. You can download it via Google, select the Community section, this option is free to download.

Very often, when writing game code, it is necessary to repeat the same program instructions many times. For this, there are methods or another name — functions. A method is a set of commands no different from other lines of code, which can be launched as a separate subroutine. In Unity3d, there are several fundamental methods that are responsible for almost all of the game's work. The first of them is the Update () method. This method is launched infinitely every frame of the game. The more often the processor can execute this method, the higher the FPS

of the game will be. Almost all of the code of our future game will be executed through this method. This code is usually called the game logic. The next fundamental method is called Start (), which is executed only once when the game starts. It is needed to prepare the necessary parameters for work — to initialize them. Moreover, in our game scene there can be many Start () and Update () methods at once: on the door of the transition to the level, on the character, on the grocery store, etc. How do they interact with each other? It is important to understand and remember that when the game is launched, all game objects that have a Start () method are executed first, and after that, all objects with an Update () method are executed. For example, if there are 100 objects in a running game scene, and each has an Update () method, this means that in 1 frame (mentioned above), the computer processor will execute all 100 Update () methods. For additional understanding, you can imagine that one Update () method is divided into 100 methods that must be executed during one frame of the game. In general, strictly speaking, from a performance point of view, such a large number of Update () methods in one scene is not very good. But such a division of one method into different objects is one of the great features of Unity3d. Thanks to this division, the process of creating a game becomes very convenient and understandable: here is a floor fan, it appears at the beginning of the game and while it is on, its Update () method is launched to rotate the blades of its propeller.

5. «to Rotate slowly!»

The introduction is over, now we can start to make. Let's start with preparing the project. A project is the entire conglomerate of code, objects, textures, templates, etc., located as files in a single folder that Unity3d uses. To create a clean new project, you need to run the Unity Hub program, it is used to organize projects, create and install different versions of Unity3d. Create a new project (New Project button) using any version of Unity3d starting from 6000. To download Hub, enter the words Unity hub in Google search. The first entry in the search is exactly what you need. When you launch Unity Hub, activate the license — Personal, it does not require payment for using the Unity3d editor.

To create our game, we will use a rendering pipeline called HDRP. Unity3d has 3 rendering pipelines: built-in, called BIP, universal URP and HDRP. A rendering pipeline is a structure that describes the necessary procedures to converting your 3D scene into an image on the computer screen. We can say briefly and clearly that the differences between them are in their graphical capabilities. BIP and URP pipelines provide mediocre graphics capabilities and are focused on creating games for mobile platforms (for phones) and virtual reality (VR). The coolest of all three is HDRP, it allows you to achieve the highest graphics quality, i.e. get the most beautiful picture of the game,

the most realistic 3D scenes. This pipeline is designed to create games that run on a personal computer (PC) and on consoles (like «Sony PlayStation»). We will use it.

After opening the created empty project, click Window> Package Manager. The Package Manager window will open. In this window, you need to switch the button at the top to see Unity Registry is displayed. This means that all packages available for download from the manufacturer Unity3d will be displayed. You need to find High Definition RP in this list, click on this inscription and in the right part of the window, click Install.

Dear reader, you can use the HDRP template from Unity Hub or the Render Pipeline Wizard at your discretion, but I intentionally describe how to create an HDRP project from scratch, so to speak, at the lowest level. The main reason is that from version to version in Unity3d, styles, settings, windows, etc. are changed, constantly being reworked by the Unity3d development team. All this tinsel can confuse a novice game creator. In addition, when using templates and «setup wizards», an empty project is very often created with «manufacturer's errors» (red messages in the console), which is additionally confusing, annoying and frustrating. The «low-level» method I describe allows you to install everything correctly on any version of the program without errors and — for sure.

We have just installed the rendering pipeline itself in our project, i.e. poured water into the stock pot for our «soup». Now we need to turn on the pipeline so that it can start working

— «light the gas under the pot». To do this, in the Project window, right-click — Create> Rendering> HDRP Asset. We have created a global object that controls the visualization of our new project. Now we need to «turn it on», to do this, click Edit> Project Settings> Graphics> Default Render Pipeline — drag the newly created HDRP asset into this empty slot. That's it, now the project is in the HDRP state and ready to work. The last step that needs to be done is to check that Visual Studio, which we downloaded and installed earlier, is used by default in our project. To do this, click Edit> Preferences> External Tools> External Script Editor and select Visual Studio from the drop-down list.

6. Here we go!

So, now let's get down filigree to the programs of our future game. Since there will be a lot of these programs or as they are also called «scripts», then in order not to get confused, you need to create a separate folder for them and call it Scripts. To do this, in the Project window, right-click on the Assets folder, Create> Folder, and call it Scripts. I suggest you take on board the idea that you need to delete and rename folders and project files in the Unity3d editor itself, but not through «Windows Explorer', since all Unity3d project files are linked by metadata, which in some cases is very capricious with all the ensuing negative consequences for the joy of creativity.

Try to do everything exactly as described below in the text so that there is no confusion and misunderstanding.

Let's create a player. The player is a game object with a camera, which is the main character of the entire game. The camera is an object that shows everything the player sees on the screen. In general, everything that is in a Unity3d scene or level is a game object — «GameObject', and all of them are located in the virtual space of our scene, so any object in Unity3d always exists with a «Transform' component, which sets the position and size of the object in space. Logic suggests that without this component, the object cannot exist, so the «Transform' component cannot be removed from the object.

So, the player. Let's create a program for the player, for this we right-click in the «Project' window in the «Scripts' folder that we created — Create> Scripting> Empty C# Script, we'll call it «FPCharacter'. Double-click on this file to open the «Visual Studio' editor. We'll see the following lines of code:

```
using UnityEngine;
public class FPCharacter
{
}
```

Let's take a quick look at them. The line 'using UnityEngine' means that this program or script will use (English: using) the base classes of the Unity3d Engine development environment.

The line 'public class FPCharacter' means that this scenario is a class (a class is the basic data type of a programming language) called FPCharacter, and this class is publicly accessible thanks to the 'public' keyword, i.e. it can be accessed by any code in the project.

It is also necessary to keep in mind that all game logic scenarios in Unity3d are, so to speak, derived from the base class «MonoBehaviour'. Therefore, in order for our game scenario to work, it is always necessary to add a colon and MonoBehaviour to the class name. Now this same code should look like this:

```
using UnityEngine;
public class FPCharacter: MonoBehaviour
{
}
```

Now we have created the 'skeleton' of our first program. Now we need to fill it with data and operations. All the content of the class or its «body» should be written between curly brackets.

Let's specify or declare variables. A variable is a piece of your computer's memory that has a type, i.e. who it is, a name — what its name is, and stores some values. Let's write:

public float speed = 2.0f;
public float speedfast = 50.0f;
public float gravity = -9.8f;

The first line means that a variable called 'speed' is declared, has a value equal to 2.0, is public and has a type — 'float', i.e. a numeric type with a floating point. For example, in C# there is an 'int' type, i.e. an integer type, into which you can enter the value 1, 2, 3, etc., but you cannot write 1.4 or 2.3. It is into the 'float' type that you can enter the values 2.3 or 1.4. Also, according to the rules of the C# language, when declaring a variable of the 'float' type, you need to put the letter 'f' in the value indication to separate the variable from another type called 'double'. Also, at the end of an instruction or variable declaration, you must always put a semicolon. When transferring our code to machine instructions for the processor (run our program), a semicolon indicates that this instruction or variable declaration is finished and processor can move on to the next steps.

Now the three variables we have created will help us control the program, i.e. the player we are making. The variables are 'speed', 'speedfast' and 'gravity'. The program now looks like this:

```
using UnityEngine;
public class FPCharacter: MonoBehaviour
{
public float speed = 2.0f;
public float speedfast = 50.0f;
public float gravity = -9.8f;
}
```

Press Ctrl + S to save our script. To move consistently and not get confused, return to the Unity3d editor in our project and create the player game object itself. Right-click in the «Hierarchy' window> Create Empty. Let's immediately name this game object «Player'. And now just drag and drop our created «FPCharacter' script on it (You can drag and drop the script onto the object in the «Hierarchy' window or in the «Inspector' window). The «Inspector' window is the most frequently used window in Unity3d, it displays almost all the properties and parameters of the objects of the future game.

Now we see our «FPCharacter' script attached on the «Player' object as its component, just like the «Transform' described above. And we see all three variables of our script with their values set. Note that if you remove the 'public' keywords, the variables will disappear from the «Inspector' and their setting in the Unity3d editor will become impossible.

7. Character Controller

Let's continue by writing this line of code:

CharacterController CharControl;

This is a variable of the «CharacterController' type, which is called «CharControl'. The declaration of such a variable occurs in the same way as we did above. To better understand the analogy and remember the principle, imagine that we had a variable of the 'float' type called 'speed'. And now we have a variable of the «CharacterController' type called «CharControl'. Fix in your mind that all declarations in the future will occur in the same way, then the picture of what is happening will be immediately clear.

«CharacterController' is a specialized Unity3d class created for universal and fast creation of movements and collisions of characters in the game, in our case the player himself.

Before continuing, it is important to understand that in C#, variable types can have a specific value or only references to other variables. For example, the 'float' variables described above are value types, i.e. they can be used to specify that speed = 2.0f; Reference types, such as «CharacterController', are only a pointer (reference) to the corresponding class. Logic suggests that if this is a just reference, then you need to get something that can be referenced, i.e. get a specific class or object. To do this, we use one of the main methods that I described earlier, it

is called Start (). Let's write a line of the Start () method in the «body» of our FPCharacter script:

```
using UnityEngine;
public class FPCharacter: MonoBehaviour
{
public float speed = 2.0f;
public float speedfast = 50.0f;
public float gravity = -9.8f;

CharacterController CharControl;

void Start ()
{
}
}
```

Let's write a line of code to this method to obtain a specific «CharacterController' class:

```
void Start ()
{
CharControl = GetComponent <CharacterController>
();
}
```

You may notice that the class and method declarations look similar — between the curly braces is the main code, the so-called «body» of the structure.

Now the variable «CharControl' we created before, using the

function «GetComponent <> () ' (get component), becomes a reference to the component «CharacterController'. But where is this component that we got? If the function of getting starts as we have it immediately with the words «GetComponent' after the equal sign, then getting something happens right «here», in this game object on which the code script is located.

The general view of the program is now as follows:

```
using UnityEngine;
public class FPCharacter: MonoBehaviour
{
public float speed = 2.0f;
public float speedfast = 50.0f;
public float gravity = -9.8f;

CharacterController CharControl;

void Start ()
{
CharControl = GetComponent <CharacterController>
();
}
}
```

A new word 'void' appeared before the word «Start', it means «empty», this is the most frequently used type when specifying a method. The fact is that a method in the C# programming language can produce some result of its calculations. In this case,

instead of 'void', the type that should be the result is put. For example, if the result of the calculations is a floating number (number with comma), then instead of 'void', 'float' is put.

As I said earlier, the Start () method is executed once. Now, when the program is launched, the reference to the «CharacterController' component will be stored all the time while the program is running.

Switch back to the Unity3d editor, select our «Player' object in the «Hierarchy' window and click the «Add Component' button in the «Inspector' window. In the menu that opens, enter the first few letters in the search bar, for example: 'char' and we can immediately see that the search results already show the required «CharacterController' component — add it. Now we see 3 components on our «Player' object: «Transform', «FPCharacter' and «CharacterController'. Set the object position (in the «Transform' component) equal to 0,0,0, so the object will move to the zero position of the scene space, coordinate axes. Make it a rule when creating and setting up a new object to always set it to position 0,0,0 and rotation 0,0,0. This will save you from a lot of confusion associated with placing objects «where you need them». To get closer to our object in the «Scene' window, you can always press the «F» key on the keyboard, it will focus the scene view on the selected object.

We see our object as a green wire frame, these are the boundaries of the «CharacterController' component, i.e., the actual size and height of the player. We need to change several

parameters, set the values MinMoveDistance = 0, Radius = 0.35, Height = 1.8. So we get a game character 1.8 meters tall. A unit of measurement equal to 1 in Unity3d is equal to 1 meter. Maintaining approximately the exact dimensions as in reality is very important when creating a game, since too high dimensions, for example, if the player is not 1.8, but 18 Unity3d units, can negatively affect many different aspects. The same applies to excessive reduction. Therefore, if, for example, you are going to create some 3D models in the 3D modeling program 3ds Max, then you need to set the System Unit Scale: 1 Unit = Meters, Display Unit Scale: Metric, Millimeters, rotate the rotation point (green marker) up by 90 degrees and export the 3D model in the. FBX format with Units = Meters forcedly set. With this method of sending models to the Unity3d development environment, your models will always correspond to real sizes and will always be rotated correctly. Of course, it will be necessary to maintain the correct sizes when modeling in the 3ds Max program according to its rules.

8. If () condition operator

Back to the code. It's time for operators. An operator is a symbol or a text word that denotes some actions with data. Let's clarify one of the most important operators of the entire C# programming language, it is called 'if' (if) — condition. It can be characterized as one of the most fundamental along with the cyclic repetition operator 'for' (for) — cycle. It can be conceptually said that all programs on planet Earth are variations of codes based on the 'if' condition operator and cyclic repetition operators for, regardless of the specific programming language. Let's write the following lines of code in the «body» of our FPCharacter script, right under the Start () method:

```
void Update ()
{
if (Input.GetKey (KeyCode. LeftShift))
{
_speedfast = speedfast;
}
}
```

First, now we have the previously described Update () method, which is launched every frame during the game program's operation. Second, we can now see two nested «bodies» of structures: one of them is the contents of the Update

() method, i.e. everything that is between its curly brackets, and the second «body» belongs to the if () operator — is, so to speak, within its scope of influence. There is an interesting feature here that must be mentioned: if you declare any variable «in the body» of the if () operator, it will be available only «in the body» of the if () operator. On the contrary, if you declare a variable «in the body» of the Update () method, it will be available both in Update () and in if ().

Now the if () operator. This is an operator that is needed to simply describe the condition: «if — then». In our case, now, this is a description of the action «if such and such a key is pressed, then do such and such an action». The Input.GetKey (KeyCode. LeftShift) code uses a class and a method to determine whether the user has pressed a certain key on the keyboard, in this case, «Left Shift'. This entire expression is the condition of the if () operator. If the key is pressed, then the code located «in the body» of the if () operator is executed.

What's interesting is that the «key pressed» check will happen, for example, 60 times per second if the game runs at 60 FPS — this is exactly what was described at the beginning of the book.

Let's continue. «In the body» of the if () operator, one new variable '_speedfast' has appeared. First, we declare it, «in the body» of our «FPCharacter' class, immediately under the previously declared speed variables, we write the following line of code:

float _speedfast;

Now we have declared a 'float' type variable without specifying the 'public' keyword. This is done so that it does not appear in the «Inspector' window of the editor, since it does not require manual installation and is not used anywhere except this script (FPCharacter), i.e. it is internal or local. Also, we have not set its value, so in this case this variable is equal to 0.

The general form of our script code is now as follows:

```
using UnityEngine;
public class FPCharacter: MonoBehaviour
{
public float speed = 2.0f;
public float speedfast = 50.0f;
public float gravity = -9.8f;

float _speedfast;

CharacterController CharControl;

void Start ()
{
CharControl = GetComponent <CharacterController>
();
}
void Update ()
{
if (Input.GetKey (KeyCode. LeftShift))
```

```
{
_speedfast = speedfast;
}
}
}
```

Now, when you press the Left Shift key on the keyboard, the '_speedfast' variable will take the value of the 'speedfast' variable. But now such an assignment of values will make little sense since it has no reverse effect, so let's continue examining the if () operator. The 'speedfast' variable is needed to speed up the movement of our character while holding the Left Shift key, i.e. when the Left Shift key is not pressed, the character's speed should be normal. To implement this, we write the following lines of code, right after the curly bracket of the «body» of the if () operator:

```
else
{
_speedfast = 1;
}
```

We have a new part of the if () operator, which is called 'else' (else), this is branching, i.e. if the condition specified by the code in the if () operator is not met for some reason, then the execution of the condition can be branched — to execute a piece of code located «in the body» of the else operator. Finally, the if () operator in our case will look like this:

```
if (Input.GetKey (KeyCode. LeftShift))
```

{
_speedfast = speedfast;
}
else
{
_speedfast = 1;
}

Thus, each frame, many times in the Update () method, a check will be made for the Left Shift key being pressed: if it is pressed, then the internal (local) 'speed' variable will be equal to the manually set 'speedfast' variable, or if the Left Shift is not pressed, then the local variable will be equal to 1. It is necessary to remember that the 'else' operator cannot be used without if (). There are also other condition branches when additional data for checking could be added to the 'else' operator. This will be discussed later in the book. Let's continue our «Player' script. Let's write the following lines of code «in the body» of the Update () method:

float offsetX = Input.GetAxis («Horizontal») * speed * _speedfast;

float offsetZ = Input.GetAxis («Vertical») * speed * _speedfast;

We declare two local variables offsetX and offsetZ, which read the pressing of the orientation keys in space. Again we see the Unity3d class called Input and getting the coordinate axes «Horizontal' and «Vertical'. The two words «Horizontal' and

«Vertical' are, so to speak, collective for all the buttons of the keyboard, joystick, etc., divided into 2 categories: horizontal and vertical. Vertical in this case is understood as the opposite of the horizontal axes, because the character's movement is left-right and forward-backward, but not left-right, up-down. Receiving data through the Input class from the player's keyboard, we also multiply the data by the speed and by the fast speed (for example, running), which we have just analyzed. The asterisk sign means ordinary multiplication.

9. Movement

Let's add the following line of code, which forms a local variable of a vector in space, in three coordinates — X, Y, Z:

Vector3 movement = new Vector3 (offsetX, 0, offsetZ);

«Vector3» in Unity3d is actually just 3 float variables described earlier put together, but is used to describe position in space and direction of movement.

We see a new operator called 'new', this operator is needed to create a new instance of the data type. In this case, we create a new vector with two variables offsetX and offsetZ. The Y variable, which describes the up-down movement, is missing and is not needed now.

The correct directions along the axes in Unity3d can be easily remembered using associative memorization. For example, to remember that the Z coordinate is forward movement, you can imagine that Z is like a zigzag movement of a boat on water going into the distance, i.e. we are sailing forward (forward-backward). Y is like an hourglass that only flows down (up-down), and for X there is remains only left-right.

Now we have made a movement vector, which will be responsible for the movement of our character forward-backward and left-right, depending on which keys on the keyboard the player presses. But now, with our current code, the movement forward-backward and left-right will be slower than

the diagonal movements. This will happen because the diagonal movement will take into account the offset values of both offsetX and offsetZ together. To make the movement uniform in all directions, you need to add the following code as next line in your code:

movement = Vector3.ClampMagnitude (movement, speedfast);

«ClampMagnitude' is an internal Unity3d method that will limit the magnitude of our motion vector so that the speed of movement is uniform in all directions.

Let's add next line of code:

movement. y = gravity;

A little higher we did not use the Y variable, which is needed for moving up and down. Now we add it too. In our movement vector there are 3 variables and we can get or set each of them separately, now it looks like 'movement. y', i.e. we set this variable equal to gravity so that our future player can freely fall down if he has no support under his feet.

Next we must make sure that our future player can move the same way speed on computers of different power, at different game speeds, at different game FPS, etc. To do this, we will write the following line of code:

movement = movement * Time.deltaTime;

In this line of code we link our movement vector via the «Time' class (a multitasking class of the game time category) with the rendering time of the previous frame — 'deltaTime'.

Thus, any movement in the game can be calculated regardless of the specific computer and its power. Always use multiplication by 'Time.deltaTime' in any code based on some changes in values over time.

Also, the C# language provides the ability to write the last line of code, so to speak, in a shortened form, which is what we will do:

movement *= Time.deltaTime;

In this form, the line of code completely corresponds to the previous one, so we will leave it. Now we need to «shoe another leg», namely, our character will soon be able to move, but as soon as he turns, his movement will be disrupted, i.e. the character will look somewhere to the side, but will continue to move strictly forward-backward, left-right. To prevent this, we need to transform the calculations of our movement vector from local coordinates to global ones. Let's write the following line:

movement = transform.TransformDirection (movement);

The word 'transform' is a reference to the corresponding component, which is located on our game object Player. Here is the same technology as with «GetComponent <> ()» described earlier. If after the equal sign immediately write 'transform' or «GetComponent <> ()», it means that the operations are performed with the current game object on which our code script is located. «TransformDirection ()» is a Unity3d method that transforms our vector into world coordinates (global). Let me

briefly explain that world coordinates are the basis of the entire 3D scene of the Unity3d development environment, and they are immutable. And local coordinates belong to some object of the scene and constantly change their directions, depending on the rotation of this object.

Now our character's movements forward-backward, left-right will occur in accordance with its rotation.

The character's movement program is ready and needs to be executed, or rather applied to our Player game object. To do this, we will communicate our movement vector to the «CharacterController' component using its internal «Move' method, and add a line of code:

CharControl.Move (movement);

Thus, our lines of code, running infinitely every frame in the Update () method, will read the keys pressed by the player, form a vector of the direction and speed of the player's movement and move him. Now the entire written code looks like this:

```
using UnityEngine;
public class FPCharacter: MonoBehaviour
{
public float speed = 2.0f;
public float speedfast = 50.0f;
public float gravity = -9.8f;

CharacterController CharControl;
float _speedfast;
```

```
   void Start ()
   {
   CharControl = GetComponent <CharacterController>
();
   }

   void Update ()
   {
   if (Input.GetKey (KeyCode. LeftShift))
   {
   _speedfast = speedfast;
   }
   else
   {
   _speedfast = 1;
   }

   float offsetX = Input.GetAxis («Horizontal») * speed *
_speedfast;
   float offsetZ = Input.GetAxis («Vertical») * speed *
_speedfast;
   Vector3 movement = new Vector3 (offsetX, 0, offsetZ);
   movement = Vector3.ClampMagnitude (movement,
speedfast);
   movement. y = gravity;
```

```
movement *= Time.deltaTime;
movement = transform.TransformDirection (movement);
CharControl.Move (movement);
}
}
```

Let's save our script and switch to the Unity3d editor.

10. Launching the Player

Let's test our player, see how he moves. To do this, first of all, you need to switch all the editor windows to a convenient layout so that you can see both the «Game' window and the «Scene' window at once. In the upper right corner of the editor, click the «Select editor layout' button and choose from the list — «2 by 3», this is what you need.

Now let's make something like the ground so that our player does not fall down into the abyss. Click the GameObject tab> 3D Object> Plane at the top. A Plane game object with a collider will appear in the scene. Due to the fact that the calculation of collisions and physical interactions is a heavy-duty task even for today's computers, the Unity3d development environment has «Colliders' components (from the English collide — to collide), which perform the functions of virtual surfaces used to calculate collisions and simulate the physical behavior of objects. For example, a complex (i.e. multi-polygonal) sphere object can be enclosed in a collider in the form of a simple cube. Then such a sphere will become an obstacle to passing through it and can already be used in physical calculations. The main advantage in this case will be that the computer will need to calculate only 6 surfaces of the virtual cube for collisions, and not hundreds or more surfaces of the sphere itself.

So, the plane we just created already has a collider, i.e. it

can be used immediately as ground or an obstacle for falling down. Place our Player in the «Scene' window (green wire frame) slightly above this plane so that when the program starts, it does not «fall through the ground» (if suddenly the movement markers or better to say arrows disappear, you can press the «W» key to activate them again. It is also necessary that the «Player' object is selected in the «Hierarchy' window).

Press the Play button at the top in the middle (don't forget to click once in the «Game' window so that the playback focus is the same as in the finished game, otherwise the game controls will not work). Now you can press the WASD and arrow keys, and our game character will start moving and may even fall if he goes beyond the plane boundaries. If you press the Left Shift while moving, the player's movement speed will increase. But as we understand, there is no rotation of the virtual head using a computer mouse, so we will add this feature. Press Stop (same place at the top in the middle) and return to our «FPCharacter' code.

Write the following line of code «in the body» of the Update () method:

transform.Rotate (0, Input.GetAxis («Mouse X») * SensHoriz, 0);

And «in the body» of the «FPCharacter' class we declare a variable:

public float SensHoriz = 2.0f;

The line of code we wrote will rotate our Player game object

using the Rotate () method along the Y axis by reading the mouse movements. The «SensHoriz' variable is needed to adjust the sensitivity or speed of the head rotation. But this rotation will only happen to the sides, so it's time to add the «head itself» and the remaining up-down rotation.

11. Head

Let's go back to the editor, select our Player object in the «Hierarchy' window, right-click on it and «Create Empty'. This way, our Player object will have a child object, let's call it «Head' (To rename any object in Unity3d, you can press the «F2» key). This «Head' object will be our head. We need to set the Y value to 0.7 for the «Position' parameter of its «Transform' component, since we understand that the head should be above the «stomach» level. Now let's add eyes to our player's head, to do this, on the newly created «Head' object in the «Inspector' window, click «Add Component' and enter «Camera' in the search, add it. If your scene contains (see the «Hierarchy' window) a camera called «Main Camera', then delete it, since it is no longer needed. Let's go back to the code.

In order for our program, our script, to be able to control the newly created head with eyes, we need to programmatically interact with it, i.e. we need to get a reference to it, a reference to the head object. In general, in Unity3d, everything is an objects and you can get a reference to all of them and, accordingly, control them. So remember that.

First, in the body of the «FPCharacter' class, we declare a reference variable to the «Transform' component and 4 number variables:

Transform HeadTransf;

float RotatX;
public float SensVert = 2.0f;
public float MinVert = -70f;
public float MaxVert = 70f;

Now let's write a line of code to get a reference to an object in the Start () method:

HeadTransf = transform.GetChild (0);

The GetChild () method is needed to create a reference to a child object. When we created the «Head' object, you saw that it appeared as if under the «Player' object, i.e. it is its child object, and for the time being it repeats all the movements and rotations of its «parent», all this is a simple hierarchy. Such a hierarchy can be multi-level — a child object, a child object of a child object, etc., and is a fundamental feature of Unity3d, extremely easy to create and edit on the fly, and very useful in almost all types of work in the Unity3d development environment, both programming and artistic.

GetChild (0) means that we get the child object with index 0, i.e. simply the first from top to bottom in the hierarchy. In programming, the counting starts from zero, i.e. if you have 10 objects, then you can get each of them in order starting from 0 and ending with 9. To get used to this way of counting and quickly navigate in it, because in nature this is impossible, in nature nothing can be called a zero number, because if you have 10 apples in your hands, then any first apple is number 1, and the tenth is 10, so, to quickly navigate in such a counting system

when you have a specified number of elements or objects, then just subtract one from their total number and there will be no difficulties. If you have 23 apples, then in programming you need to address them as 0—22.

Now, in the Update () method, at the bottom, we write the following lines:

RotatX = RotatX — Input.GetAxis («Mouse Y») * SensVert;

RotatX = Mathf.Clamp (RotatX, MinVert, MaxVert);

HeadTransf. localEulerAngles = new Vector3 (RotatX, HeadTransf. localEulerAngles. y, 0);

The first one reads the mouse movement along the Y axis; the second one limits vertical movements using two variables MinVert and MaxVert, simulating real up-down limitations of a human head; the third line forms the rotation vector of our virtual head, i.e. applies the formed rotation.

The general appearance of our program is now as follows, compare it with your version:

```
using UnityEngine;
public class FPCharacter: MonoBehaviour
{
public float speed = 2.0f;
public float speedfast = 50.0f;
public float gravity = -9.8f;
public float SensHoriz = 2.0f;
public float SensVert = 2.0f;
```

```
public float MinVert = -70f;
public float MaxVert = 70f;

CharacterController CharControl;
float _speedfast;

Transform HeadTransf;
float RotatX;

void Start ()
{
CharControl = GetComponent <CharacterController>
();
HeadTransf = transform.GetChild (0);
}

void Update ()
{
if (Input.GetKey (KeyCode. LeftShift))
{
_speedfast = speedfast;
}
else
{
_speedfast = 1;
}
```

```
   float offsetX = Input.GetAxis («Horizontal») * speed *
_speedfast;
   float offsetZ = Input.GetAxis («Vertical») * speed *
_speedfast;
   Vector3 movement = new Vector3 (offsetX, 0, offsetZ);
   movement = Vector3.ClampMagnitude (movement,
speedfast);
   movement. y = gravity;
   movement *= Time.deltaTime;
   movement = transform.TransformDirection
(movement);
   CharControl.Move (movement);

   transform.Rotate (0, Input.GetAxis («Mouse X») *
SensHoriz, 0);

   RotatX = RotatX — Input.GetAxis («Mouse Y») *
SensVert;
   RotatX = Mathf.Clamp (RotatX, MinVert, MaxVert);
   HeadTransf. localEulerAngles = new Vector3 (RotatX,
HeadTransf. localEulerAngles. y, 0);
```

Конец ознакомительного фрагмента.