A Practical Introduction to 3D Game Development

Yasser Jaffal

bookboon
The eBook company

Yasser Jaffal

A Practical Introduction to 3D Game Development

A Practical Introduction to 3D Game Development
1st edition

ISBN 978-87-403-0786-3

Contents

About this Book

What this book is all about?

This book is a practical introduction to programming 2D and 3D games, techniques used in programming these games, and how to benefit from these techniques. It illustrates a large number of mechanics used in video games and shows by example how to program these mechanics and combine them to achieve the desired behavior. It illustrates also how to put the player into control and deal with interactions between player and various game elements.

The book focuses on programming as one of the important fields regarding video game development. However, game development is a huge world with tons of arts and skills to learn. The book has also exercises that allow you to evaluate your understanding of the covered topics. Each one of these exercises has a new idea that is not previously discussed, and challenges you to program your own variations of the examples.

Who can benefit from this book?

Everyone. Regardless of the purpose for which you want to learn game programming, and, most importantly, regardless of your current knowledge and experience in programming, you can benefit from the topics covered in this book. It gives you the basic knowledge you need to start quickly and effectively in the world of game development, with a focus on game logic and mechanics programming.

Does the book require specific game engine / programming language?

From a technical point of view, this book and its examples deal only with Unity3D game engine. Additionally, all scripts in the book are written using C# programming language. It has, however, a vision of being engine/programming language independent. Therefore, I have tried to avoid using any templates that are specific to Unity3D game engine and build everything from scratch depending on basic functions that are most likely to be found in all game engines. I am really interested in seeing someone applying the examples of this book using other engines and programming languages.

How to read this book?

This book offers you a non-linear approach to reach the knowledge you seek. If you already know what do you want to learn and what type of games you wish to create, you can jump from chapter to chapter reading only the sections you need. Say, for example, that you want to create car racing game. In this case, you have to read the first chapter which covers common basics then jump to section 6 of chapter 2 to see how to implement the input system you need. In this case, you are not interested in applying mouse look for first person control, which is covered in another section of chapter 2.

1 Basics of Scene Construction

Introduction

In this chapter, we are going to learn about the basic objects that construct a 3D scene, most important properties of these objects, and relations between these objects. If you are already familiar with 3D engines or 3D design software, you might find the subjects of this chapter familiar; since there are similarities between Unity's way of constructing 3D scenes and other 3D software/engines. In this case, you can safely skip this chapter without worrying about missing important topics. In this chapter, we are not going to cover advanced subjects regarding scene construction, but rather stick to basics that allow us to carry on in our journey towards the development of a 3D computer game.

After completing this chapter, you are expected to:

- Be able to construct a scene using basic 3D shapes.
- Understand the properties of objects in 3D scene (position, rotation, scale).
- Use relations between different objects in 3D space and their effects on the objects to accelerate and facilitate scene construction.
- Understand and use rendering properties (textures, materials, shaders) to enrich your 3D objects in the scene.
- Use different types of lights and understand their properties.
- Use camera to render the scene for the player.
- Write simple scripts that modify the properties of the objects at run time to achieve desired effects.

1.1 Basic shapes and their properties

In addition to its capability of importing 3D models from most known 3D design software, Unity provides us with a collection of game objects that represent basic 3D shapes. These shapes include cube, sphere, plane, cylinder, and many others. These objects make it possible to construct a basic scene and interact with it. Scene construction is done by simply adding a number of these shapes and modifying their properties; such as position, rotation, and scale.

> To add a new game object to the scene, go to *Game Object* menu, then select *Create Other*. You'll find the basic shapes in the third section of the menu starting from *Cube* and ending with *Quad*

Once you add a new shape game object to the scene, it appears in the scene window and the hierarchy. If you can't see the object in the scene, you can simply select it from the hierarchy and then press F key on the keyboard while the mouse pointer is inside the scene window.

Initially, the hierarchy contains only one object, which is the main camera. This camera is responsible for rendering the scene for the player. In other words, it is the player eye on the game world. Let's now start with a small scene that consists of a number basic shapes. Try to construct a scene similar to the one in Illustration 1 by yourself. If you find this difficult, you can follow the steps after the Illustration.

To modify the position, rotation, or scale of a game object; use the buttons at the top left of Unity's main window. Alternatively, you can use the properties of *Transform* component in the inspector window (see Illustration 2)

Illustration 1: Simple scene constructed using the basic 3D shapes provided by Unity

To construct the scene we see in Illustration 1, we need first to know the type and the properties of each shape we are going to add. The values of *position*, *rotation*, and *scale* are determined by the three axes of the 3D space, which are x (+right, -left), y (+up, -down), and z (+inside screen, -outside screen). The 3D Coordinate system used in Unity follows the left-hand rule. To remember this rule, hold your left hand with your index pointing forward, your thumb pointing up, and your middle pointing right. Your three fingers represent the positive directions of the axes in the 3D space, where the middle finger represents the x-axis, the thumb represents the y-axis, and the index represents the z-axis.

Follow these steps to construct a scene similar to Illustration 1:

1. Create a plane and position it at the center of the 3D space (0, 0, 0). The plane is a 2D shape that covers a 10*10 area on the xz plane, supposing that we use the default scale (1, 1, 1). This plane represents the ground in our scene.
2. Create 4 cylinders and position them at 1 on the y axis, so they sit on the top of the ground plane. Now we need to distribute them uniformly around the origin. For example, we can use the values of (2, 3.5), (-2, 3.5), (-2, -3.5), and (2, -3.5) as the values of (x, z) position for the each one of these cylinders. Finally, scale the cylinders to 0.5 on x and z axes to make them thinner. By completing this we have successfully added the pillars of the arcs to the scene.
3. Now we need to add the two bars at the top of the arcs. These bars are two cubes positioned at (0, 2, 3.5) and (0, 2, -3.5). To extend these cubes, scale them to 6 on x axis and to 0.5 on both y and z axes.

4. Finally, add 3 cubes and position them near the center of the scene by using the 3D gizmo, and rotate them with different angles. Set the y position for two cubes to 0.5, and the third to 1.5, which will make it sit on top of the other two. By adding these boxes we have completed the construction of our simple scene.

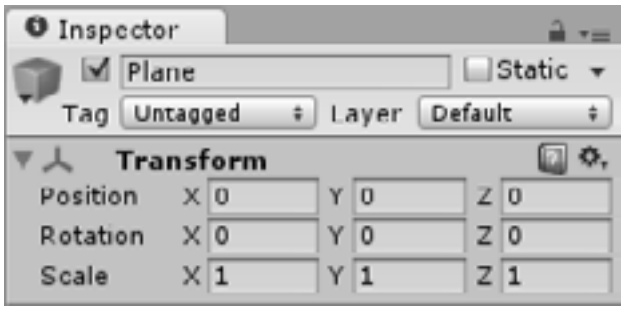


Illustration 2: Transform component

You can see the final result in *scene1* in the accompanying project. You might have noticed that we did not need more than the alternation of the positions, scales, and rotations of the basic shapes to construct our scene. You might have also noticed the existence of several components in the inspector window that are added to the game object. One of these components is *Transform* which we've just used. Each one of the components has its unique function and plays specific role in the look or the behavior of the game object. For instance, *Mesh Renderer* component is responsible for rendering the object. Try to disable this component and see what happens.

To disable a specific component, simply uncheck the check box at the top of the component in the inspector window

If you still unsure about the difference between scene, game object, and component; refer to Illustration 3, which summarizes scene construction in Unity.

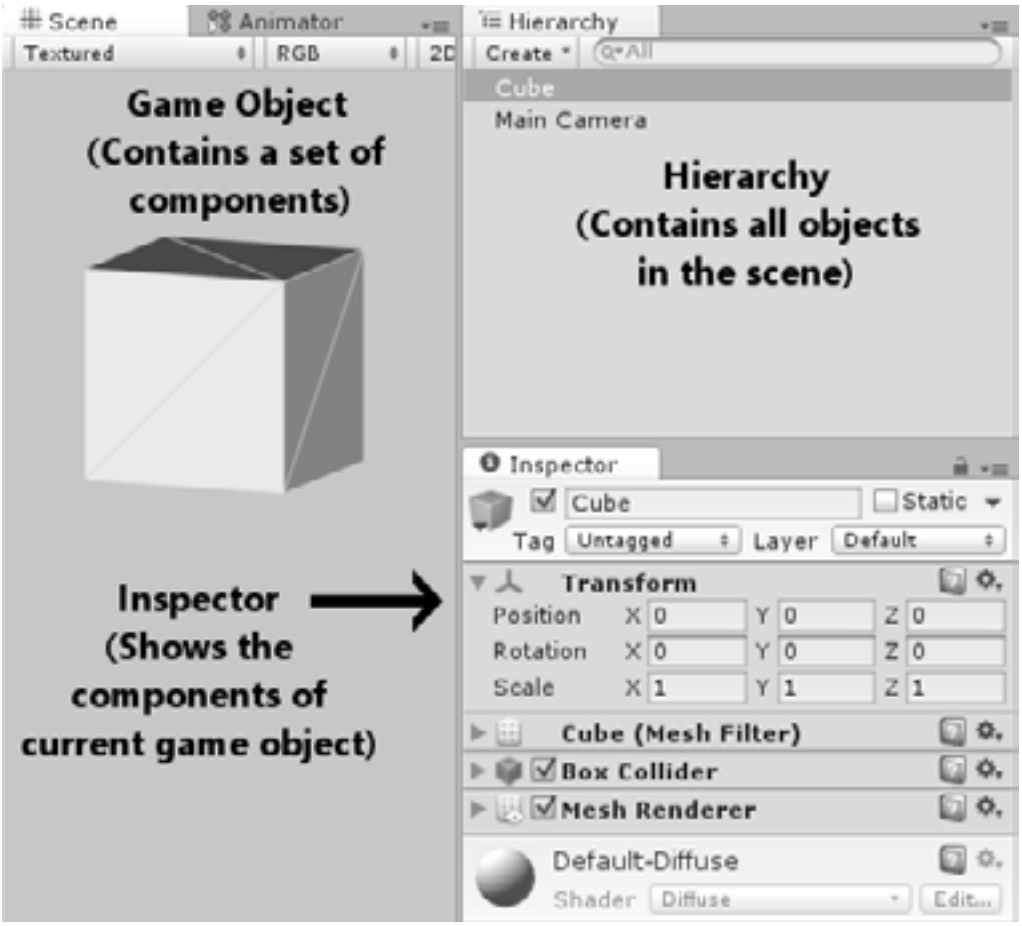


Illustration 3: The relation between the objects and the components in Unity

1.2 Relations between game objects

You might have noticed during your work in the last section that each game object can be handled independently without affecting the other objects. You might need, however, to move these objects as whole to another place, or make several copies of them. In either case, you need first to select all objects together.

A better approach will be to handle all related scene elements as one unit by adding logical relations between them. In most 3D programs and game engines, objects can be connected together using child/parent relations. In such relations, changes applied to the parent object affect its children but not vice-versa. However, children can override the values resulted from changes to parent.

In Illustration 1, we can identify different building blocks that construct the scene: the ground, the two arcs, and the three boxes. Supposing that the ground is the root of the scene (so that all objects move together if we move the ground), it is reasonable to make it the parent of all other objects in the scene, excluding the main camera.

To build a parent-child relation between two objects in Unity, simply drag the child into the parent inside the hierarchy. To unparent, drag the child out to an empty position in the hierarchy.

It is also reasonable to have each one of the two arcs as a whole unit, even it is not easy to determine which part of the arc must be the parent. In such case, we can simply add a logical parent, which is an empty object that is used as the root of other arc objects.

To add an empty game object in Unity, go to *Game Object* menu and select *Create Empty*

Let's add two empty objects and name them *arc 1* and *arc 2*. It is always a good idea to give meaningful names to game objects in the scene, in order to make dealing with scene elements easier. This is especially useful in large scenes that have many objects.

To change the name of a game object, you can either select it in the hierarchy and press F2, or change the name directly from the name field at the top of the inspector

Position these two empty objects at the center of the scene and add them as children for the ground. Now add the objects of each arc as children to one of these empty parents. Finally, add the three boxes as children for the ground. The final hierarchy should look like Illustration 4.

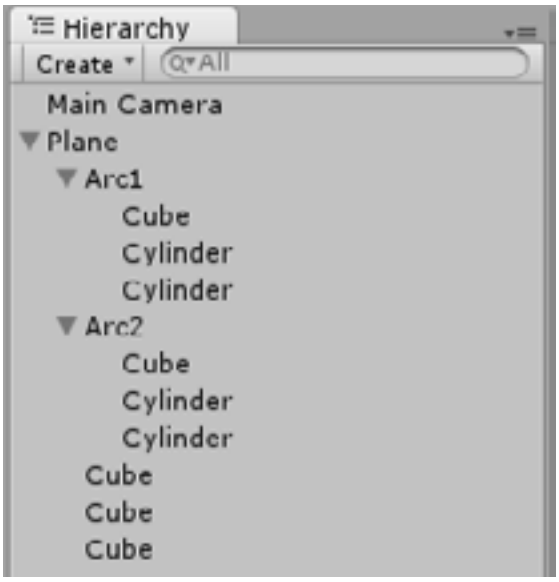


Illustration 4: The hierarchy of the scene with the relations between the objects

If you move the ground object to another position in the scene, you'll notice that its children move with it. However, examining position values in the inspector reveals that only the values of the ground object position change, while the position values of the children remain the same. Although this is unexpected, it can be justified by recognizing that the position values of the children are not absolute, but rather relative to the position of their parent. In other words, what you see in the inspector is the position of the child inside the *local space* of its parent, rather than the *world space*. The same applies to scale and rotation of the child. We are going to discuss these concepts later on, so don't worry about them at the moment.

1.3 Rendering properties

In this section, I am going to introduce to you the basic properties of object rendering in Unity. As I have mentioned earlier, *Mesh Renderer* component is responsible for rendering the object. Therefore, it can be only found in objects that are visible in the game. Therefore, it is missing in *arc 1* and *arc 2* as well as the main camera. It is not, however, the only component related to rendering; and we are going to deal with other components later on.

First element to discuss in this section is *texture*. A textures a 2 dimensional image painted over the surface(s) of a 3D shape to give it unique look. For example, we can assign a sand texture to the ground, a brick texture to the arcs, and a wood texture to the boxes. The textures we are going to use are shows in Illustration 5. The files containing these textures must be added to the project before they can be assigned to game objects.

To add texture files to the project, drag them from their current position inside Unity's project explorer. The best practice is to create a new folder for texture files and place the files inside it. To create a new folder, right click the root folder *Assets,* then select *Create* > *Folder.* Finally, to assign a specific texture to an object, drag the texture from the project explorer to the target object inside the scene window

Once you assign the texture to an object, Unity automatically creates a new material for that texture and adds the material to the renderer of the target object. Materials are automatically added to *Materials* folder, which Unity also automatically creates in the same location of *Textures* folder. Practically, the texture cannot be directly assigned to a game object. Alternatively, the material that is added to the renderer of the object has a shader, and the texture is set as a property of that shader. Shaders specify the final form of the material after applying all effects. Unity uses *Diffuse* shader as default. To fully understand the relation between textures, materials, shaders, and the renderer; see Illustration 6.

Illustration 5: Adding texture files to a Unity project

Illustration 6: Renderer component and its relation with materials, shaders, and textures

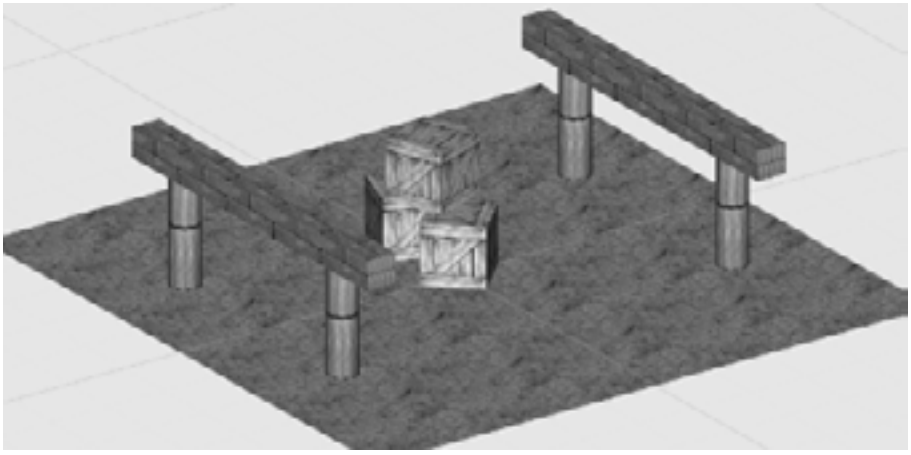


Illustration 7: The textured version of the scene

One of the interesting properties of shaders is *Tiling*. Tiling specifies how many times the texture should be repeated on each surface of the object on both x and y coordinates of the image of the texture. *Offset*, on the other hand, specifies whether the texture should be shifted vertically or horizontally. We can make use of tiling to enhance the view of the objects. Let's set tiling values for *Sand* to 5 on both x and y axes, and for *Brick* to 5 on x axis only. As for *Wood*, each box surface should show the texture only once; therefore we keep the tiling values at 1. You can see the final result in Illustration 7, the result can also be found in the accompanying project in *scene1 textured*.

1.4 Light types and properties

Lighting is an essential element that contributes in scene building. By using lighting, illuminated areas can be distinguished from dark ones, and shadows can be generated. Additionally, light can be used to focus player attention to a specific part of the scene, or even to design puzzles.

In this section, our aim is to introduce types of lights that Unity provides, and how they can be used in scene construction. First type to discuss is *ambient light*, which represents default lighting color on scene-wide level, without adding any other light source. Therefore, it can be used to simulate day/night, sunrise/sunset times, and so on. These effects can be achieved by changing the color of the ambient light.

Because ambient light is a scene-wide property, it is not bound to a particular game object in the scene, but can rather be adjusted from *Render Settings* window, which contains all scene-wide properties for the current scene.

To change the ambient light color, open *Edit* menu and select *Render Settings*. You can then see these settings in the inspector and adjust the *Ambient Light* property. To make the changes in lighting visible in Unity editor, you need first to switch lighting on by using  button.

From an artistic point of view, the importance of ambient light lies in the general feeling it gives to the player. For example, lava environments use warm light colors such as orange and red, while cold environments use blue. Green ambient color is used to give the feeling of a humid environment. Illustration 8 shows how changing the color of ambient light affects the scene.

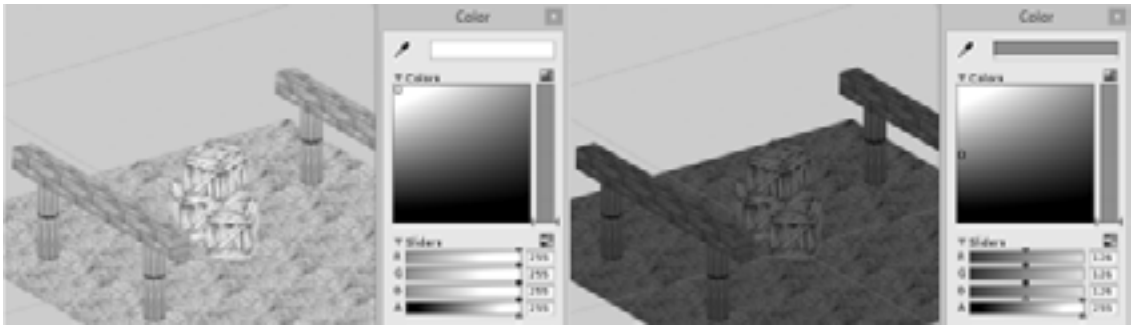


Illustration 8: Effect of ambient light on the scene

The second type of lights is *Directional Light*, which can exist once per scene. Directional light represents the major light source that has the largest effect outdoors. Therefore, it can represent the sun in day scenes, or the full moon in night scenes. Directional light, as the name suggest, has a specific emitting direction, unlike the ambient light. It has also other properties such as *Intensity*. Let's now add a directional light to the scene and adjust its properties.

To add a directional light to the scene, go to *Game Object* > *Create Other* > *Directional Light* then select it from the hierarchy to adjust its properties

The directional light affects all areas of the scene equally. Therefore, it does not matter where it is positioned in the space, or what its scale is. However, altering the rotation of the directional light will change the its emitting angle. For example, setting the rotation to (90, 0, 0) will make the light emit vertically from above; just like the noon sun. To make the effect of the directional light easily visible in our scene, it is better to set the rotation of the directional light object to another value, such as (50, -45, 0). Unity draws short beams in the emitting direction of the directional light to make it easier for us to recognize its rotation, like the ones in Illustration 9.

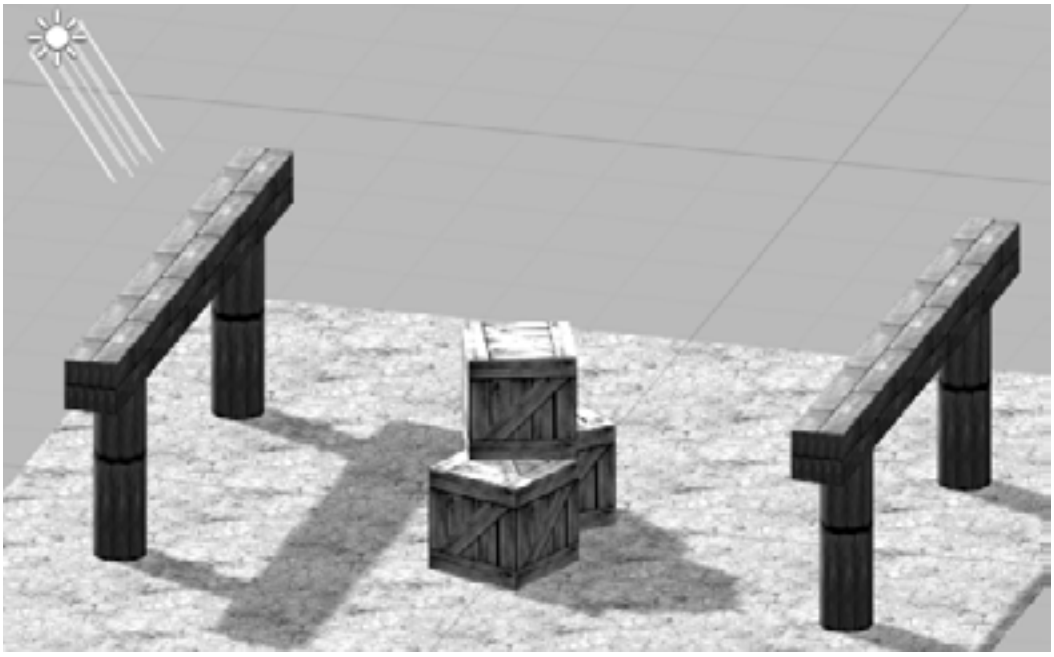


Illustration 9: The scene after adding the directional light

Another important property of the directional light is *Intensity*, which controls how strong is the effect of the light on the objects in the scene. A very high intensity will minimize the effect of textures and make the surfaces of the objects look like flat surfaces that have the color of the directional light.

The other two types of light available in Unity are *Point Light* and *Spot Light*. Even we are not going to use them in the current scene, it is a good idea to know how they work. Point light emits light equally in all directions, just like an ordinary electric light bulb. Spot light, on the other hand, emits beams in one direction, forming a spot of light on the surface of the target. You can think of search lights and car lights as examples of spot light. Illustration 10 shows examples of point and spot light. Each one of these lights has its unique properties that you may discover yourself.

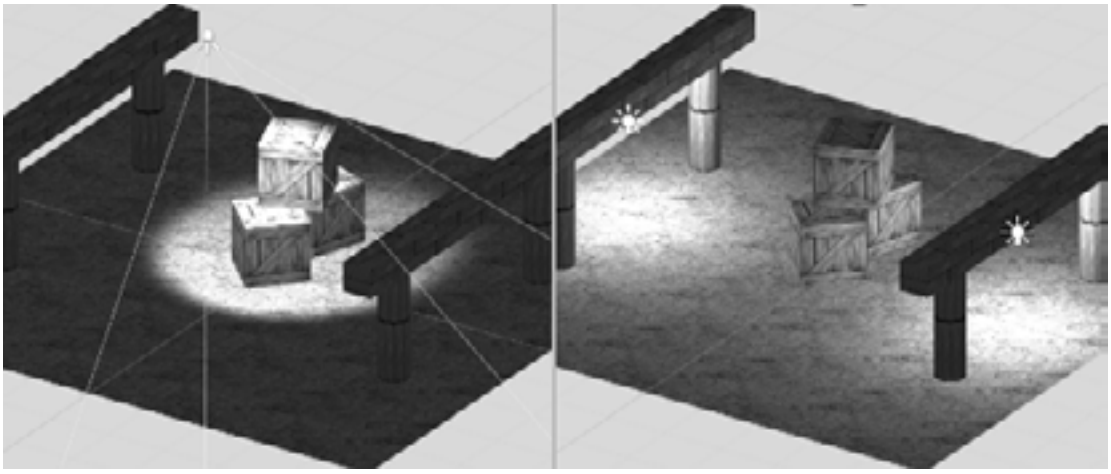
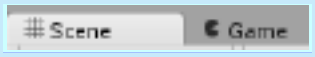


Illustration 10: Spot light (left) and point light (right) and their effect on the scene

1.5 Camera

When we are done constructing the scene, it is time to know how it going to look for the player when the game starts. Until now, we have been dealing with the scene from the *Scene* window. The other window (*Game*) displays the scene as it appears to player when the game starts. While Unity switches between these two windows automatically when you start or stop the game, you may switch between them at any time to see the game from the player perspective.

To switch between *Scene* view and *Game* view, use 

tabs at the top of the scene view

The main difference between *Scene* and *Game* views is that the latter does not allow you to surf freely in the scene and limits you to the view of the main camera, from which the player observes the game world. If you select the *Main Camera* game object from the hierarchy, you will see a number of properties including:

1. *Background*: the color that fills the horizon of the scene. Empty areas that are not covered by any visible game objects are going to appear in this color.
2. *Projection*: determines whether the distance between an object and the camera affects the size of that object when rendering it. In the default *Perspective* projection, far objects are rendered smaller than near ones. This behavior is similar to human vision system. In the *Orthographic* projection, all objects are rendered with their original size, regardless of how near or far they are from the camera. Orthographic projection is useful in some cases, like 2D games.
3. *Field of View*
4. *Clipping Planes (Far and Near)*

Field of view option is only available in the perspective projection. In this projection, the field of view takes a shape of a frustum. If you complete this frustum to a pyramid, the head of the pyramid is at the position of the camera, and the base of the pyramid is the far clipping plane. The near clipping plane is what turns the pyramid into a frustum, while the field of view is the angle between left and right sides of the frustum. Illustration 11 shows how does the camera use the view frustum to determine the visible objects in the scene. Illustration 12 shows the same scene from the perspective of the camera, and the player as well.

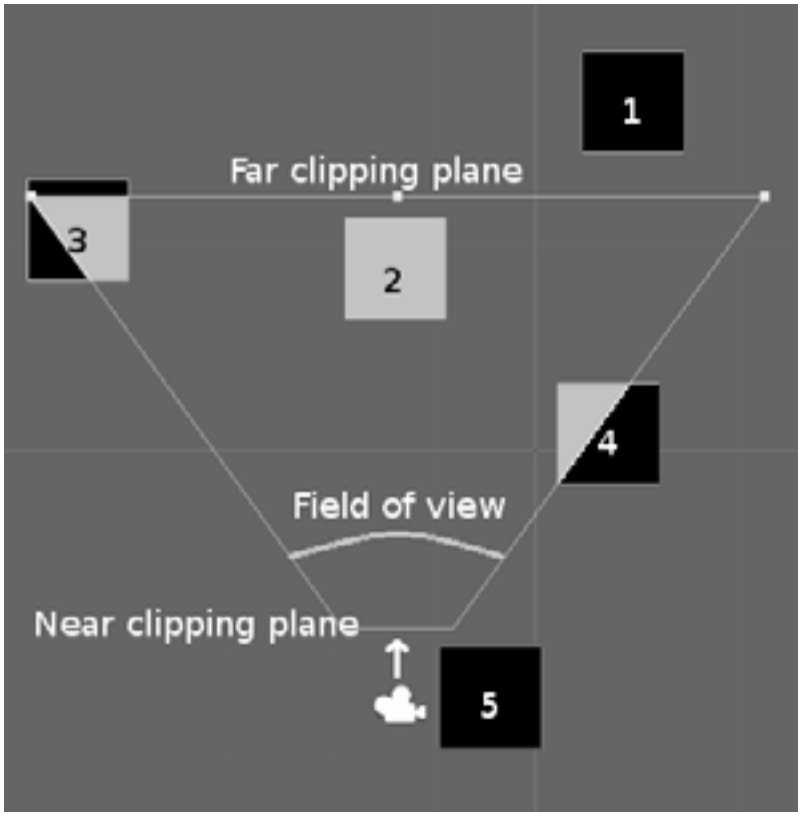


Illustration 11: The view frustum of the camera. The black shaded areas mark the invisible parts of the objects

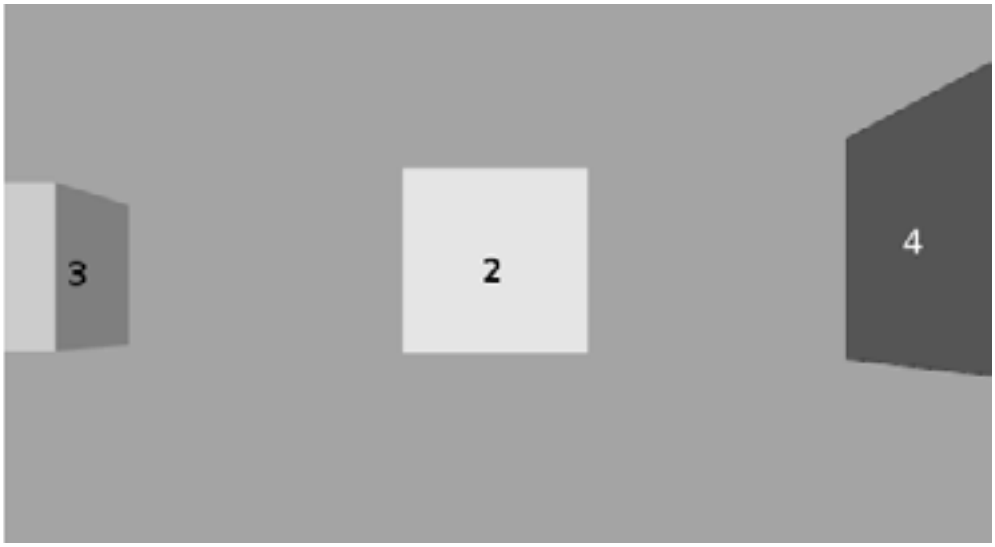


Illustration 12: The scene of Illustration 11 as seen by the camera

1.6 Controlling objects properties

After we have constructed the scene and had an idea about how it is going to look like for the player, it is time to do some programming. Programming scrips is a core element in game development, since scripts define the behavior of game objects during play time. Let's begin with script that has a simple function: displacement of the objects. Let's also take the camera as the first object to add scripts to.

It is recommended that you create a new folder called *scripts* inside the root folder of the project *Assets* to save our scripts in. After creating the folder, we are going to create our first script *CameraMover* into it.

To create a new script, right click *scripts* folder and select Create > C# Script then name the file *CameraMover.cs*

Unity supports three different programming languages, but I will stick to *C#* in this book. However, if you are familiar with *Javascript*, you may use it instead of *C#* by changing the syntax of the scripts listed in the book. It is also advisable to use *MonoDevelop* development environment included with Unity instead of *Microsoft Visual Studio*. Listing 1 shows the default template for all *C#* scripts created in Unity.

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

Listing 1: Default script template in Unity

Unity helps us by adding the most common functions, *Start()* and *Update()*. The first one is important to initialize the script (i.e. to set the default values of the variables), and is called once at the beginning of the script life cycle. The second function *Update()* is called once per frame, in order to perform the required changes on the properties of the object over time.

Unity handles scripts like other components we've seen so far, such as *Renderer* and *Transform*. Therefore, the scripts are not active unless they are added to game objects. Scripts must inherit from *MonoBehavior* class in order to be recognized by Unity as components. If you don't have much experience in object-oriented programming, you might not be sure what does inheritance mean in this context. But that's fine, since all you need is to keep the structure of the default template. Next step is to add our newly created script to the camera. Once we do this, all behavior we code in the script applies to the camera game object.

To add a script to a game object, select the target game object from the hierarchy, and then drag the script inside the inspector. Alternatively, you may click *Add Component* button at the bottom of the inspector and type the name of the script in the search box then hit Enter.

We are now ready to code the behavior in our script. In the cut-scenes of some video games, the camera moves around slowly and shows the player the scene from different angles. Why don't we try something like this? Assuming that we want the camera to keep moving upwards slowly, we need to move it by constant speed in the positive direction of the y axis. Additionally, we can add a variable that controls movement speed. Listing 2 illustrates necessary code for camera movement. Double clicking a script file loads the default script editor (*MonoDevelop* in our case) and opens the script for you to edit the code.

```
1. using UnityEngine;
2. using System.Collections;
3.
4. public class CameraMover : MonoBehaviour {
5.
6.       public float speed = 1;
7.
8.       // Use this for initialization
9.       void Start () {
10.
11.      }
12.
13.      // Update is called once per frame
14.      void Update () {
15.           transform.Translate(0, speed * Time.deltaTime, 0);
16.      }
17. }
```

Listing 2: Camera moving script

It is a good idea now to describe the mechanism that Unity uses to run the scripts. When the game starts, Unity calls *Start()* function from all active scripts in the scene. By doing this, Unity makes sure that all scripts are initialized and ready to enter the game update loop. In this loop, frames are constructed and rendered through various steps. These steps include reading user input, moving and animating objects, running the physics simulation and Artificial Intelligence (AI) algorithms, executing the game logic, and rendering the frame. In order to have an acceptable play experience for the player, at least 25 to 30 frames must be rendered every second. In each iteration of this loop, Unity passes through all active scripts in the scene and calls *Update()* function from them. This procedure continues as long as the game is running.

In line 6 of Listing 2, we declare a floating point number with the value of 1. This will be the speed of camera movement. The value of *speed* is multiplied by the *delta time* in line 15 to perform a translation on the y axis using *Translate()* function. *Translate()* takes the displacement values on x, y, and z axes. Notice that we provided a non-zero value for the y axis only, since we don't want to move the camera on neither x nor z axes.

In line 15, we need to move the object upwards with specific distance every frame, and we need to compute this distance. As we know from physics, the speed of an object equals the distance the object moves in the time unit. Therefore, we need to multiply the speed by the time unit to compute the distance we need to move the object with. But how to get this time? Since translation is performed once every frame, the time we need to know is the time passed since the rendering of the previous frame (i.e. since the last time the object moved). This value is given to us by Unity in the variable *Time.deltaTime*. So, when we multiply this value by the movement speed, we get the distance we need to move the object with. All you have to do now is to save the script and start the game to see the result.

To start / stop the game, click on  button

One interesting feature of Unity is the ability to modify the default values of the public variables directly from the inspector, so we do not have to change the code and recompile it after every modification. Illustration 13 shows how do public variables appear in the inspector. You may try to change the speed to a negative value and see the result you expect.

Illustration 13: The script component as it appears in the inspector

After running the game for a while, you are going to see that the scene you have constructed is no more visible to the camera as it moves higher and higher. Let's try to fix this little problem by forcing the camera to always look at the center of the scene, regardless of its current position. This is achieved by altering the rotation of the camera downwards as it goes higher. Fortunately, we don't need to bother ourselves with the dirty details of this rotation, since Unity provides this functionality directly through *transform.LookAt()* function. Simply add the following line after line 15 in Listing 2.

```
transform.LookAt(Vector3.zero);
```

What we expect to see now is vertical movement of the camera and rotation towards the origin of the scene, which keeps the scene visible. If you start the game now, you are going to get a change in the z position of the camera with the time. As the camera gets higher on the y axis, it gets also closer to the center of the xz plane. This unexpected movement along z axis can be justified by understanding the difference between translation in the *local space* and translation in the *world space*.

By default, *Translate()* function is applied in the local space of the object, which is affected by its position and rotation. World space, on the other hand, has fixed x, y, and z axes that are constant among all game objects all the time. To understand the concept of local space, consider an airplane object like the one in Illustration 14. This plane has a local space that is different from the global world space. The right wing of the plane, which points to the positive direction of the x axis of the local space. The plane front points towards the positive direction of the z axis of the local space, and the positive y axis of the local space is perpendicular to the upper surface of the plane and points upwards. (As a sort of motivation, we are going to build this plane model and fly with it in the next chapter).

Back to our camera, when it rotates to look at the center of the scene, its local y axis becomes no more parallel to the y axis of the world space as it originally was. Therefore, when you move the camera along its local y axis, it going actually to move along the z axis of the world space as well. Illustration 15 shows the local space axes of the camera after the rotation.

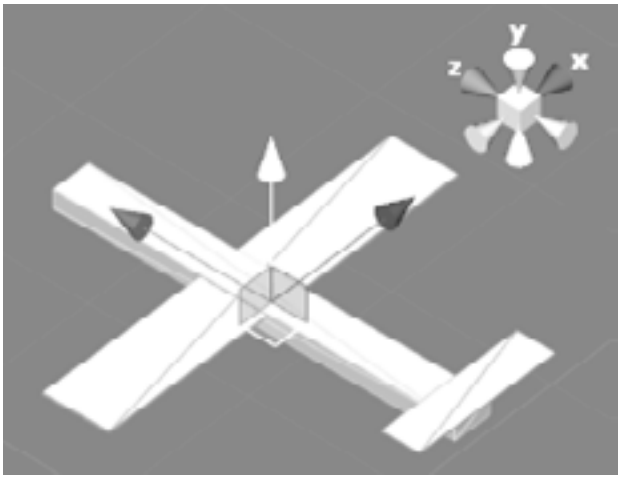


Illustration 14: Local space axes of a plane model

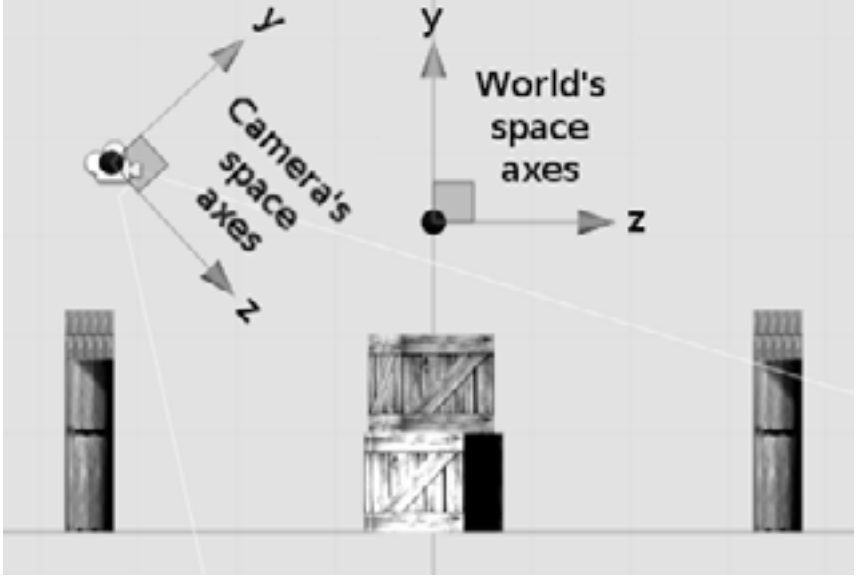


Illustration 15: The difference between world axes and camera's local space axes

To fix this unwanted behavior, we need to tell Unity to move the camera on the axes of the world space rather than the axes of its local space, which changes when camera rotates. So our new line 15 in Listing 2 is

```
transform.Translate(0, speed * Time.deltaTime, 0, Space.World);
```

Start the game now to see the difference. Let's now do something that is more exciting. What if we change the rotation of our directional light over the time? How will this rotation affect the shadows of scene elements? To do this we need to create a new script called *LightRotator* and add it to the directional light. This script is shown in Listing 3.

```
using UnityEngine;
public class LightRotator : MonoBehaviour {

    public float speed = 10;//10 degrees per second

    void Update () {
         transform.Rotate(speed * Time.deltaTime, 0, 0);
    }
}
```

Listing 3: Light rotator script

This script rotates the directional light around its local x axis. Since the directional light emits in the positive direction of its local z axis, the rotation changes the angle between the emitted light beam and the horizon. This gives an effect similar to sunrise and sunset (for better understanding of this rotation, use the left-hand rule, rotate your hand around the middle finger, and see how the direction of your index changes). The speed used here is a little bit higher than the speed of camera movement, because we want to see the effect light rotation before the camera goes far away from scene elements.

In this chapter we have learned how to construct a simple scene using basic shapes with different positions, rotations and scales. We have also learned how to use 2D images as textures to give details to the shapes. We have introduced different types of lighting sources and discussed their interesting properties. We have also written some scripts that change the properties of the objects during game execution.

Notice that there are more advanced topics regarding scene construction that were not covered in this chapter. The aim of this chapter was to provide a quick introduction to scene construction, so that we understand the structure of the scene and be able to interact with it. This was important because interaction is a core element in game development. There will be more on textures, materials, lighting, and shaders in the coming chapters.

Exercises

1. We've discussed the use of basic shapes to construct a scene. Use them to construct a more complex scene. For example, draw a house with a garden surrounded by a wall. You can look up in the internet for free textures to use.
2. Add point lights to the scene you have constructed in exercise 1. Select appropriate positions for these lights, and adjust their properties to make a night scene with electric lights. Remember to choose a dark color for the ambient light.
3. Modify *CameraMover* script to make the camera rotate around the scene horizontally, while keeps looking at the center of the scene. Axes of local space can help you to achieve this behavior. Add the modified version to the camera in the scene you have constructed in exercise 1.
4. Try to make use of relations between objects to add a spot light to the camera. This spot light must move along with the camera and focus on the position where the camera looks.

2 Handling User Input

Introduction

In this chapter we are going to learn about the first practical step towards building a complete computer game, which is reading user input. Computer games, regardless of their genre and mechanics, must have a form of user input; because interacting with the player is crucial in any digital game. We are going to learn about different techniques of reading and handling user input, and our focus will be on the scripts that react appropriately to this input in the scene.

After completing this chapter, you are expected to:

- Read keyboard input
- Implement platformer games input system
- Read mouse input
- Implement mouse look and develop first person input system
- Implement third person input system
- Implement controls of car racing games
- Implement controls of flight simulation games

2.1 Reading keyboard input

The keyboard is probably the most important input device in PC systems. Almost all PC games depend on number of keyboard keys that perform basic functions. Most commercial games give the player the ability to change key mapping to customize the controls for his needs.

Unity allows us to read keyboard input by using two methods. The first method is to read the key code directly, by telling you that the player is currently holding, for example, A or Z key. The other method is to use Unity's input manager to bind the keys with commands you define, so that you can later change the mapping between keys and commands. I am going to cover the first method only in this book, since the vision of the book is to be as general as possible and avoid Unity-specific functions as possible. This should make the book more useful for developers who use other engines.

To learn how to to read input, let's create a new scene in our project or create a whole new project. What we need now is a scene that contains the camera in its original position (0, 0, -10), in addition to a cube in the middle of the scene as in Illustration 16.

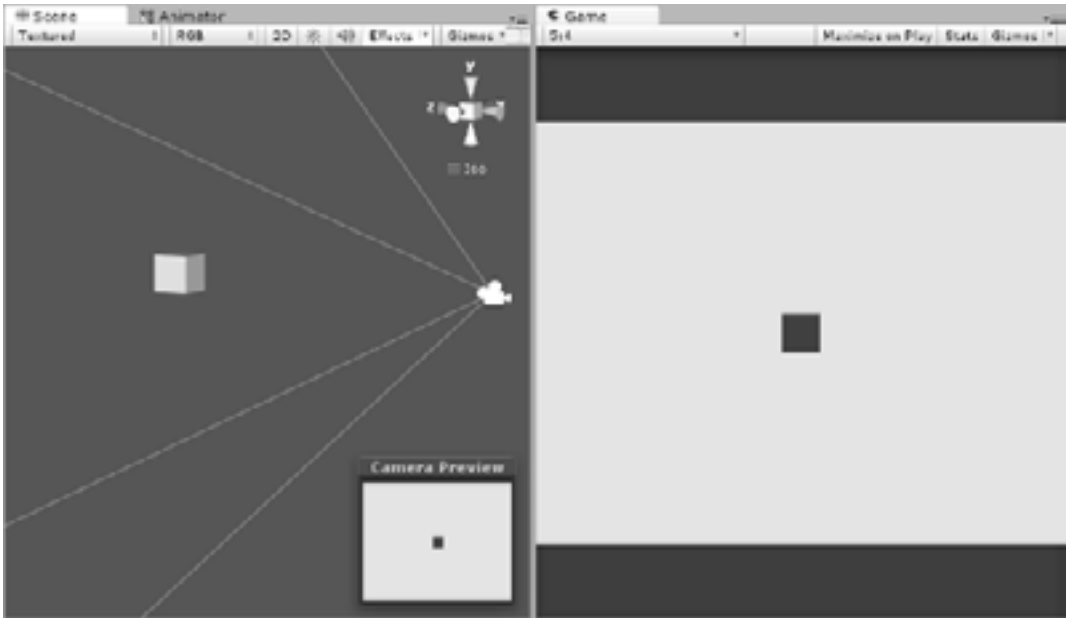


Illustration 16: A simple scene to demonstrate reading keyboard input

After creating the scene, create a new script in *scripts* sub folder that we dedicated for script files, and name it *KeyboardMovement*. Let's say that we want to move the cube in the four directions: up, down, left, and right depending on which keyboard arrow key the player is pressing. Listing 4 shows the required code to implement such functionality. All you have to do is to add the script to the cube and start the game to move it using the keyboard arrow keys.

```
1. using UnityEngine;
2. using System.Collections;
3.
4. public class KeyboardMovement : MonoBehaviour {
5.
6.       public float speed = 10;
7.
8.       // Use this for initialization
9.       void Start () {
10.
11.       }
12.
13.       // Update is called once per frame
14.       void Update () {
15.            //Check up
16.            if(Input.GetKey(KeyCode.UpArrow)){
17.                 transform.Translate(0, speed * Time.deltaTime, 0);
18.            }
19.            //Check down
20.            if(Input.GetKey(KeyCode.DownArrow)){
21.                 transform.Translate(0, -speed * Time.deltaTime, 0);
22.            }
23.            //Check right
24.            if(Input.GetKey(KeyCode.RightArrow)){
25.                 transform.Translate(speed * Time.deltaTime, 0, 0);
26.            }
27.            //Check left
28.            if(Input.GetKey (KeyCode.LeftArrow)){
29.                 transform.Translate(-speed * Time.deltaTime, 0, 0);
30.            }
31.       }
32. }
```

Listing 4: A simple script that interprets presses on the keyboard arrow keys into movement

As we see in Listing 4, player input reading is a continuous process as long as the game is running. Therefore, we need to handle the input inside *Update()* function. In lines 16, 20, 24, and 28; we call *Input.GetKey()* and pass to it a key code to check its state. The *KeyCode* enumerator includes codes for all keyboard keys, so we only have to choose the appropriate one. *Input.GetKey()* returns *true* if the given key is pressed during the current frame, and returns *false* otherwise. By using *if* keyword, we build conditional statements that bind movement to certain direction by a specific key on the keyboard. Here we use again *transform.Translate()* to move the object on x and y axes, and use positive and negative values of *speed* to specify the movement direction.

Sometimes we need to read the key only once instead of the repetitive reading in every frame. For instance, the jump action in platformer games usually requires the player to release the jump button/key and press it again to make a new jump, instead of jumping continuously by simply keeping the jump key pressed. For similar scenarios, we use *Input.GetKeyDown()*, which gives us *true* only at the first time the player presses the key. After that, it keeps returning *false* until the key is released and pressed again. You can examine the difference between *Input.GetKey()* and *Input.GetKeyDown()* by replacing one by another in Listing 4.

If you are not familiar with programming, and hence do not really recognize the difference between using *if* alone and *else if*; I will explain this with the help of Listing 5. By using only *if*, we allow each key to be scanned independently, which in turn allows all keystrokes to affect the object simultaneously. So if the player holds both up arrow and right arrow, the object will move on both x and y axes resulting in diagonal displacement. However, if the player presses up and down arrows together, they will cancel each other effects and the object isn't going to move at all. Here where *else if* comes into play. If we want to prevent reading two opposite directions at the same time, we use *else if* and hence give the priority to the key that has the first condition check.

```
16. //Try to read the up arrow, if it is not pressed try with the down arrow
17. if(Input.GetKey(KeyCode.UpArrow)){
18.      transform.Translate(0, speed * Time.deltaTime, 0);
19. } else if(Input.GetKey(KeyCode.DownArrow)){
20.      transform.Translate(0, -speed * Time.deltaTime, 0);
21. }
22.
23. //Try to read the right arrow, if it is not pressed try with the left arrow
24. if(Input.GetKey(KeyCode.RightArrow)){
25.      transform.Translate(speed * Time.deltaTime, 0, 0);
26. } else if(Input.GetKey (KeyCode.LeftArrow)){
27.      transform.Translate(-speed * Time.deltaTime, 0, 0);
28. }
```

Listing 5: Using *else if* to restrict the reading on one key and give priority to specific keys over others

All of what we have seen with arrows applies to all other keys as well. All you have to do is to pick the key you need from *KeyCode* enumerator. You can see the result in *scene2* in the accompanying project.

2.2 Implementing platformer input system

Platformer games are probably the most famous 2D games available, and they also have some known titles among 3D games as well. Games such as *Super Mario* and *Castlevania* belong to this category of games, in addition to known modern game titles such as *Braid*, *FEZ*, and *Super Meat Boy*. These games depend primarily on jump mechanic to move between platforms, defeat enemies, and solve puzzles. They might also have other mechanics such as shooting.

Since we are now able to read user input from the keyboard, we can make a basic input system based on the arrow keys for movement and space bar for jumping. Since we have not yet learned how to detect collisions between objects, we are not able to find out whether the player character is currently standing on a platform or it should fall down. Therefore, we are going to consider that the character is standing on the ground as long as its position has specific y value.

In the previous example, we have seen how to implement movement to right and left. So, we are going now to implement the desired system to have gravity, and hence jumping and falling. We need also that the camera follows the player character as this character moves. Before starting to write the platformer input system scripts, we have to construct a basic scene to implement our work in it. Build a scene that is similar to Illustration 17 by making use of basic shapes, relations between objects, and appropriate textures. Notice that I have used *Quad* basic shape to make the 2D player character.

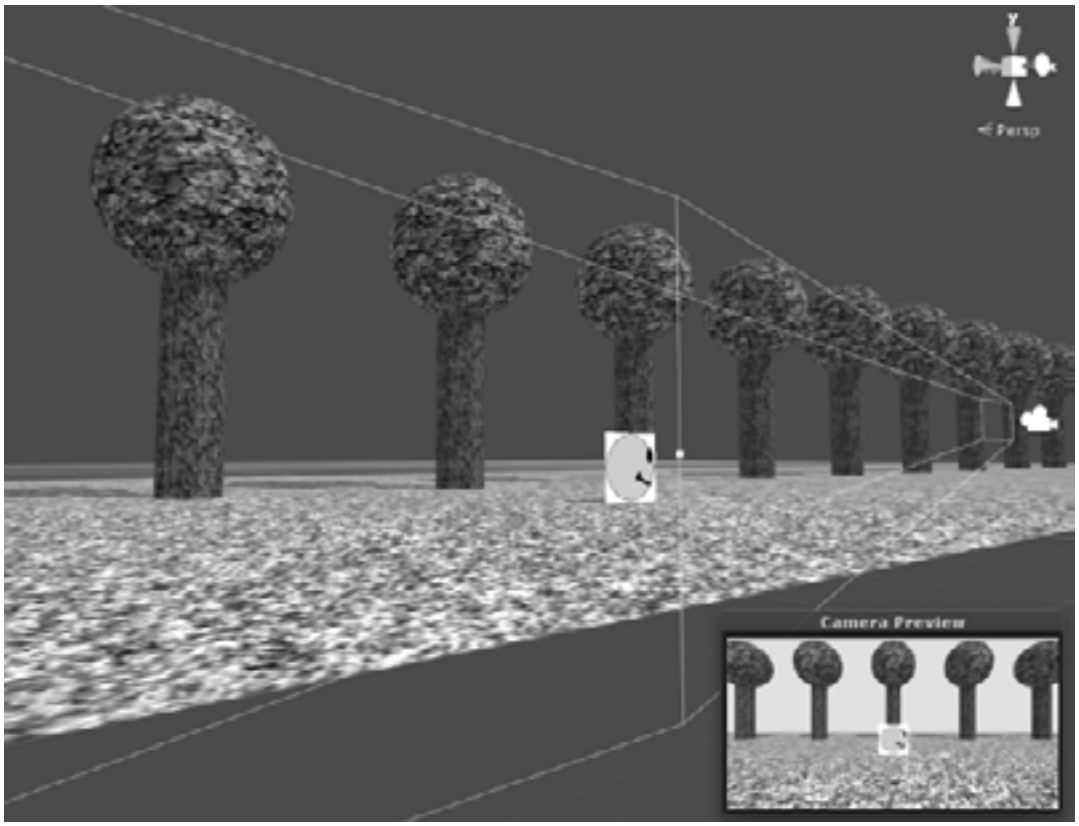


Illustration 17: The scene we are going to use to implement the platformer input system

The purpose of using the background you see in Illustration 17 is to be able to see the camera movement, specifically when we implement player character tracking function. Notice that the scene extends horizontally along the x axis, while its depth along the z axis is a bit smaller. The reason is obvious: platformer games depend mostly on horizontal movement. You can build something simpler if you find this one tedious to construct yourself, even I encourage you to try anyway; so you get more comfortable with scene construction details such as adjustment of texture tiling values to cope with object scaling.

Let's now write the script that controls the character. This script has several tasks to do: firstly, it ensures the application of gravity by dragging the object towards the ground if its y position is higher than the ground level. Secondly, it must not allow the character to sink below the ground level. Finally, and most importantly, it should allow the player to control the character by using keyboard keys. Listing 6 shows *PlatformerControl* script, which performs all of the above-mentioned tasks.

```
1. using UnityEngine;
2. using System.Collections;
3.
4. public class PlatformerControl : MonoBehaviour {
5.
6.       //Vertical speed at the beginning of jump
7.       public float jumpSpeed = 7;
8.
9.       //Falling speed
10.      public float gravity = 9.8f;
11.
12.      //Horizontal movement speed
13.      public float movementSpeed = 5;
14.
15.      //Storing player velocity for movement
16.      private Vector2 speed;
17.
18.      // Use this for initialization
19.      void Start () {
20.
21.      }
22.
23.      // Update is called once per frame
24.      void Update () {
25.           //Read direction input
```

```
26.            if(Input.GetKey(KeyCode.RightArrow)){
27.                 speed.x = movementSpeed;
28.            } else if(Input.GetKey(KeyCode.LeftArrow)){
29.                  speed.x = -movementSpeed;
30.            } else {
31.                 speed.x = 0;
32.            }
33.
34.            //Read jump input
35.            if(Input.GetKeyDown(KeyCode.Space)){
36.                 //Apply jump only if player is on ground
37.                 if(transform.position.y == 0.5f){
38.                        speed.y = jumpSpeed;
39.                 }
40.            }
41.
42.            //Move the character
43.            transform.Translate(speed.x * Time.deltaTime,
44.                                          speed.y * Time.deltaTime,
45.                                          0);
46.
47.            //Apply gravity to velocity
48.            if(transform.position.y > 0.5f){
49.                 speed.y = speed.y - gravity * Time.deltaTime;
50.            } else {
51.                 speed.y = 0;
52.                 Vector3 newPosition = transform.position;
53.                 newPosition.y = 0.5f;
54.                 transform.position = newPosition;
55.            }
56.      }
57. }
```

Listing 6: The Script that implements the platformer input system

The first note regarding this script is its relative length and complexity, compared with what we have been dealing with so far. This is reasonable, since this is our first script that does some real game stuff, and there is much more yet to come! You might now have had an idea about amount of work required to make a real game, regardless of how small and simple it might be.

Let's now dive into the details of the script and discuss them. We have three speed variables that control the speed of the character movement. The first speed is *jumpSpeed*, by jump speed we mean the vertical speed upwards at the moment the character leaves the ground. This speed starts to decrease with time until it reaches zero. After that, it starts to degrade below zero, so the object begins to fall down again because of the gravity. The second speed is *gravity*, which represents how fast the vertical speed of the character decreases when it is in the air. We are going to discuss the relation between these two speeds in a moment.

The third speed is the horizontal speed defined by *movementSpeed* variable, which is the speed of the character horizontal movement towards left and right. When we combine these three speeds together, they give us a resultant velocity that has two components on x and y axes. That's why we define *speed*, which is a variable of type *Vector2* (a two dimensional vector), in order to store the values of these two components and use them in each update iteration. We need to keep the value of the velocity stored between the frames, which is specially important for the vertical speed. The vertical speed changes during the time between the frames, and hence we keep its value to be able to update the next frame correctly.

The first step in *Update()* function, which lies between lines 26 and 32, is known to us. It simply reads keyboard input and interprets it as horizontal movement. As you can see, the right arrow gives us the positive value of *movementSpeed*, while left arrow gives us the negative value of the same variable. If the player is not pressing any of these keys, we set the value of *x* member of *speed* to zero. We use the member *x* of *speed* variable to keep horizontal speed value to use it later for final displacement of the object.

The second step in update (lines 35 through 40) is to read the space bar and implement the jumping if possible. Notice here that we use *GetKeyDown()* in order to prevent consecutive jumping by keeping the space bar down, and force the player to release space and press it again to re-jump. Additionally, we do not allow the character to jump unless it is standing on the ground at the moment the player presses space. In our scene, the original y position of the character is 0.5, so we take it as the ground reference to determine whether or not the character is grounded. Once we make sure that the character is grounded, we change *y* member of *speed* to the value of *jumpSpeed*.

The third step is in lines 43 through 45, in which we perform the actual displacement of the object based on the values of speed we have computed in the previous steps. We use *transform.Translate()* to perform displacement based on the values stored in *speed*. The displacement takes place on x and y axes, and we multiply by *Time.deltaTime* as usual, to work out the distance from the speed values we have.

After moving the object, we have one step remaining. This step is to compute the new vertical speed of the object. As described earlier, the y value of 0.5 in the character position means that the character is standing on the ground. If this is not the case (lines 48 through 50), it means that the character is currently in the air; which requires us to reduce its vertical speed using gravity. This reduction, as you can see, is equal to *gravity* multiplied by delta time from the previous frame. Since the vertical speed gets smaller as the time passes, the vertical displacement in the next frame will be less than what it is in the current frame. This holds until, at some point, the vertical speed reaches zero. This can be observed by seeing that character ascendance gets slower and slower until the object stops in the air, after that it starts to fall with a small speed and gets faster with the time.

The other part of the last step (lines 50 through 55) applies in cases other than jumping (being in air). Here we have two possibilities: the y position of the object is either 0.5, or less than that. Values less than 0.5 can be caused by gravity displacement in the previous step. Since displacement depends on delta time, which we cannot control, we cannot guarantee the absence of values less than 0.5 (remember that we still have no collision detection, so the ground will not prevent the character from sinking). To be in the safe side, we reset the vertical speed to zero and make sure that our object y position is 0.5.

It is important to mention that Unity does not allow us to modify position members directly, so you cannot simply use the statement *transform.position.y = 0.5;*. Alternatively you have to store the value of the position in a new variable of type *Vector3*, and then modify the members through that new variable. Finally, you set the value of *transform.position* to the value of the modified vector. This is exactly what we do in lines 52 through 54 in Listing 6

When you are done building the scene as in Illustration 17 (or any similar scene, as long as it has objects in the background to recognize movement), and add *PlatformerControl* script to the player character, you are ready to run the game and test movement and jumping. Illustration 18 shows a screen shot from the game during jump.

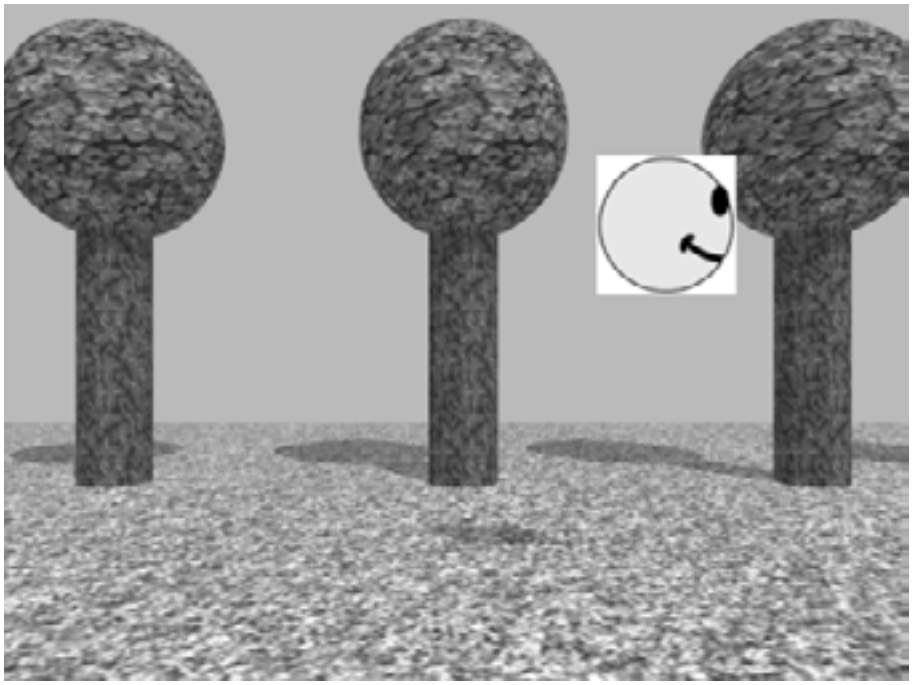


Illustration 18: Implementing the platformer input system and performing jump

What we have to do now is to make the camera follow the player character as it moves. Up to now, it is possible that the character leaves the field of view of the camera, which will make it invisible and make your game impossible to play. We can implement this function in two different ways: the first and the easiest method is to add the camera as a child to the character, so it follows the character as it moves horizontally and vertically. In this case, the relative position of the character inside game window will remain constant. The second and more advanced implementation (which we are going to use) is to write a script that allows the camera to follow the character, and gives the player a sort of margin in which he can move the character before the camera starts to follow it. This margin is going to be dynamically modifiable.

Let's discuss the implementation in detail. This method promises to have a control system that looks more professional and is more fun for the player to deal with. At the beginning, we have to have the character at the center of the camera view, then it starts to move, say, to the right. When the character is at a specific relative distance from the camera on the x axis, the camera begins to move right to follow it. Listing 7 shows the character following mechanism. So let's create a new script called *PlayerTracking* and attach it to our camera.

```
1. using UnityEngine;
2. using System.Collections;
3.
4. public class PlayerTracking : MonoBehaviour {
5.       //We need a reference to character transform
6.       public Transform playerCharacter;
7.       //Max movement distance to right before camera start following
8.       public float maxDistanceRight = 1.5f;
9.       //Max movement distance to left before camera start following
10.      public float maxDistanceLeft = 1.5f;
11.      //Max movement distance to up before camera start following
12.      public float maxDistanceUp = 1.0f;
13.      //Max movement distance to down before camera start following
14.      public float maxDistanceDown = 1.0f;
15.
16.      // Use this for initialization
17.      void Start () {
18.

19.      }
20.
21.      // Here we use LateUpdate instead of Update
22.      void LateUpdate () {
23.           //Current position of the camera
24.           Vector3 camPos = transform.position;
25.           //Current position of the character
26.           Vector3 playerPos = playerCharacter.position;
27.
28.           //Check if the camera is far behind the character
29.           if(playerPos.x - camPos.x > maxDistanceRight){
30.                camPos.x = playerPos.x - maxDistanceRight;
31.           }
32.           //Check if camera far front of player character
33.           else if(camPos.x - playerPos.x > maxDistanceLeft){
34.                camPos.x = playerPos.x + maxDistanceLeft;
35.           }
36.
37.           //Check if the camera is far below the character
38.           if(playerPos.y - camPos.y > maxDistanceUp){
39.                camPos.y = playerPos.y - maxDistanceUp;
40.           }
41.           //Check if the camera is far above the character
42.           else if(camPos.y - playerPos.y > maxDistanceDown){
43.                camPos.y = playerPos.y + maxDistanceDown;
44.            }
45.           //Set the position of the camera
46.           transform.position = camPos;
47.      }
48. }
```

Listing 7: The character tracking mechanism for the camera

There is a bunch of new things in this script that need to be discussed in detail. First of them is the variable called *playerCharacter* which has the type *Transform*. Until now we have been using variables that store numbers, and were able to use input fields in the inspector to set their values using the keyboard.

Due to the nature of this script, it is required to deal with more than one object. On one hand, we attach this script to the camera to control its movement. And, on the other hand, the script needs to deal with the character; in order to update the position of the camera according to the position of the character. That's why we have defined *playerCharacter* variable, which is going to be used to reference the object of the character, specifically the *Transform* component of that object. We need now is to bind this variable to the character object, so that the script knows the position of the character when the game runs. Illustration 19 shows how to bind a game object from the scene to a variable in a script.

To bind a game object from the scene to a variable in a script, drag the object from the hierarchy inside the filed of that variable in the inspector. So to bind the character object with *playerCharacter* variable in *PlayerTracking* script, first select the camera from the hierarchy, and attach the script to the camera if it has not yet been attached. Once the script is attached, you can see the field "*Player Character*" in the inspector. All you have to do now is to drag the character game object inside that field as in Illustration 19.

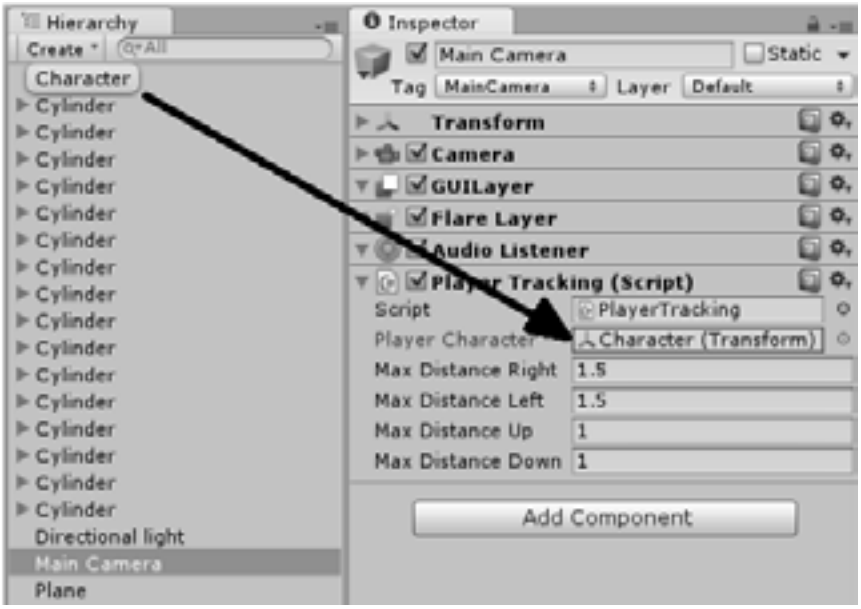


Illustration 19: Binding a game object from the scene with a variable in a script

In addition to *playerCharacter* variable, we have defined four variables to shape the frame in which the player character can move without moving the camera. These variables are named after positions relative to the camera. We have therefore *maxDistanceRight* and *maxDistanceLeft* to define maximum allowed distances on the x axis. So if the character is to the right of the camera, and the horizontal distance between the character and the camera on x axis is greater than *maxDistanceRight*, the camera will move right in order to prevent this distance from exceeding the defined limit. The same thing applies to *maxDistanceUp* and *maxDistanceDown* on the y axis. If we draw these limits as straight lines, they form a rectangle around the player character as shown in Illustration 20.

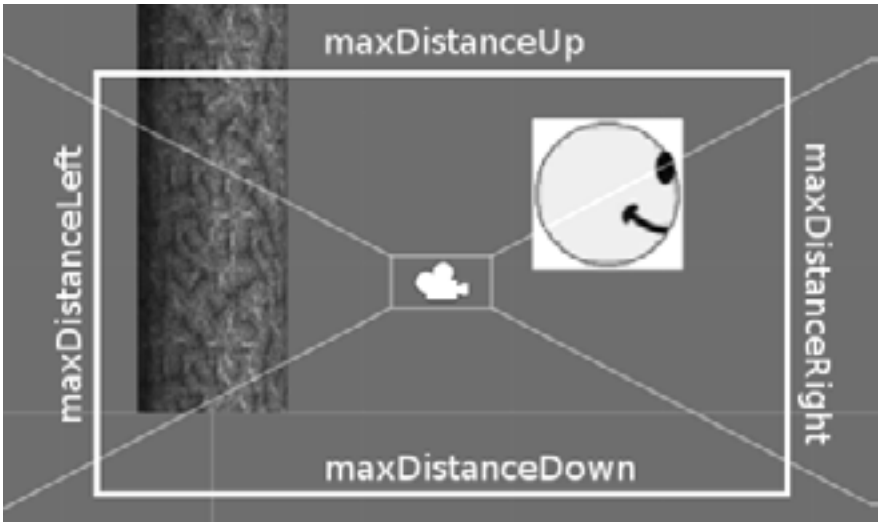


Illustration 20: The area in which the character can move without moving the camera

Once these distances have been defined, we need to track the position of the character in each update iteration to test if we should move the camera. First of all, notice that in line 22 we have called *LateUpdate()* function instead of *Update()* which we used to use previously. These functions are both called once every frame. It is guaranteed, however, that Unity will call *Update()* first from all scripts in the scene, then it will go through all scripts again and call *LateUpdate()*. In other words, it is guaranteed that the code in *LateUpdate()* is always executed after the code in *Update()* in any given update iteration.

Keep in mind that we have two scripts in the current scene: *PlatformerControl* which reads player input and performs character movement, and *PlayerTracking* which allows the camera to follow the character. Now we want to be sure that character movement completes before the camera moves. The easiest way to do that is to update camera movement from *LateUpdate()* function. This ensures that character movement is completed and the character is now in the new position, before the camera moves. Notice that if you use *Update()* for *PlayerTracking* instead of *LateUpdate()*, you might be lucky to have Unity call *Update()* from *PlatformerControl* first and then from *PlayerTracking*, hence you get a correct behavior. However, in programming we do not let luck control anything, and we always count for the worst case scenario.

As you see in lines 24 and 26, we start by storing the positions of both the camera and the character, in order to perform the required computations between them. After that we start to check for possible cases on the x axis. This checking takes place in lines 30 through 36. Keeping in mind that the positive direction of the x axis is to the right, we subtract x position of the camera from the x position of the character. If the result is greater than right limit defined by *maxDistanceRight*, we move the camera to follow the character.

One important issue here is: how to compute the new x position of the camera? The best way to answer the question is by an example: suppose that the maximum distance to the right is 1.5 as defined in line 8, and the character moves until it reaches the x position of 1.6, while the x position of the camera is still 0. Now if we compute the difference between the two positions, it is going to be 1.6 – 0 = 1.6, which is greater than the allowed value of 1.5. Therefore, we need to move the camera to the right. We need also to keep the character 1.5 units to the right of the camera, to allow the camera to move along and follow the character. In order to get the correct new position for the camera, we subtract the maximum allowed value from the current position of the character, which equals to 1.6 – 1.5 = 0.1. So 0.1 is the new x position we need to move the camera to. Camera movement is performed in line 30.

The same method applies to the case in which the character is to the left of the camera. The difference is that we add the value of *maxDistanceLeft* to the current position of the character to get the new position for the camera, like in line 34. The same applies also to the y movement of the character, along with the values of *maxDistanceUp* and *maxDistanceDown*.

The final step is to assign the new position we have computed in *camPos* variable to the position value of the camera transform, which we do in line 46. You can see the final result in *scene3* in the accompanying project.

2.3 Reading mouse input

After we have learned how to read and use keyboard input, let's now move to the other major input device in PC games: the mouse. If you are interested in computer games, you definitely know the importance of this device for these games. For instance, it is a major input device in shooting games, and it is also used to give tons of commands in real-time strategy games. Additionally, it is the major input device when it comes to game menus and the user interface in general.

What is interesting for us in this section is reading two-dimensional movement of the mouse, in addition to various mouse buttons and mouse wheel scrolling. Let's begin with a new simple scene that has one object: a sphere located at the origin. We will add a script to this sphere that reads mouse movement and interprets it to displacement of the sphere on x and y axes. The script will also read the clicks on mouse buttons and use them to change the color of the sphere. Finally, mouse wheel scrolling will be used to change the scale of the sphere. Before we begin, it is advised that you position the camera in a relatively far distance from the sphere (0, 0, -70 for example), which should prevent the sphere from leaving the field of view easily. Listing 8 shows the code required to read mouse input and interpret it to achieve the desired behavior.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class MouseMovement : MonoBehaviour {
5.
6.        //Speed of object movement
7.        public float movementSpeed = 5;
8.
9.        //Colors
10.       public Color left = Color.red;
11.       public Color right = Color.green;
12.       public Color middle = Color.blue;
13.
14.       //increment/decrement of scale at each mouse scroll
15.       public float scaleFactor = 1;
16.
17.       //Mouse position in the previous frame,
18.       //important to measure mouse displacement
19.       Vector3 lastMousePosition;
20.
21.       void Start () {
22.            //To make displacement = 0 at the beginning
23.            lastMousePosition = Input.mousePosition;
24.       }
25.
26.       void Update () {
```

```
27.
28.           if(Input.GetMouseButton(0)){
29.                //Left button
30.                renderer.material.color = left;
31.           } else if(Input.GetMouseButton(1)){
32.                //Right button
33.                renderer.material.color = right;
34.           } else if(Input.GetMouseButton(2)){
35.                //Middle button
36.                renderer.material.color = middle;
37.           }
38.
39.           //Calculate mouse displacement
40.           Vector3 mouseDelta = Input.mousePosition - lastMousePosition;
41.           transform.Translate(
42.                movementSpeed * Time.deltaTime * mouseDelta.x,
43.                movementSpeed * Time.deltaTime * mouseDelta.y, 0);
44.
45.           //Update the last position for the next frame
46.           lastMousePosition = Input.mousePosition;
47.           //Reading wheel scrolling
48.           float wheel = Input.GetAxis("Mouse ScrollWheel");
49.           if(wheel > 0){
50.                //Wheel has been rotated upwards
51.                transform.localScale += Vector3.one * scaleFactor;
52.           } else if(wheel < 0){
53.                //Wheel has been rotated downwards
54.                transform.localScale -= Vector3.one * scaleFactor;
55.           }
56.      }
57. }
```

Listing 8: Reading mouse input

Let's discuss the important parts of this script. First of all we have the movement speed in line 7, in addition to three variables of type *Color* in lines 10 through 12. These variables are able to store a specific color, which can be either picked from the color palette in the inspector, or set directly from code just like what we do here. In line 15, we define *lastMousePosition* variable, which is going to hold the last position of mouse pointer and allow us to compute mouse displacement every frame update. At the beginning of the execution (line 23), we set *lastMousePosition* to the current position of the mouse, which we get from *Input.mousePosition*. By doing this, we guarantee that the displacement is going to be zero when the first frame is rendered.

After that we go into the update loop and start to read inputs sequentially. We begin with lines 28 through 37, in which we check whether the player is pressing one of the three mouse buttons. *Input.GetMouseButton()* function does the job for us, and all we have to do is to call it and pass to it the number of the mouse button we want to check. In the default mouse setup these numbers are 0 for the left button, 1 for the right button, and 2 for the middle button. What we do in these lines is simply changing the material color of the sphere based on which button is pressed.

In lines 40 through 43 we compute displacement distance of the cursor by subtracting its previous position from its current position. We then move the object on x and y axes by the computed displacement multiplied by movement speed. Since the mouse pointer position is by nature two dimensional, we do not include the *z* member of mouse position in our computations. After that, in line 46, we update the last position of the mouse and make it equal to the mouse position in the current frame. By doing this, we become ready to compute mouse displacement in the coming frame.

The last step is to read the mouse scroll wheel, which is performed through *Input.GetAxis()*. We pass to this function the name of the axis we want to read. Axes names can be set up in a custom window that we might discuss later. However, we have already an axis named *Mouse ScrollWheel* defined for us by Unity, so all we have to do is to pass that name to *Input.GetAxis()*. If there is no wheel input, the returned value is zero. On the other hand, scrolling up will return 1 and scrolling down will return -1. Based on the return value, we add or subtract a vector of scale 1 from *transform.localScale* vector of the object. The complete scene is available in *scene4* in the accompanying project.

2.4 Implementing first person shooter input system

First person shooters are among the most popular 3D games. Many known titles belong to this category, such as *Call of Duty* series, *Doom* series, and *Half-Life*. These games place the camera at the position of the player face and observe the game world through his eyes. They mainly use mouse to control the looking direction, and WSAD keys to move in the four directions.

In this section we are going to implement this kind of input systems using what we have learned so far. First of all we need to know how to create a player character for such kind of games. Since the character isn't going to be visible, it is enough for us to use a cylinder with a length of two meters, or two units in Unity editor. In fact, the default length of the cylinder object in Unity is 2 units, so all we have to do is to add a cylinder with the default scale. Since the camera should be placed at the position of the player eyes, we have to add it as a child to this cylinder. This makes the camera move and rotate with cylinder. We have also to add a ground and maybe few objects that construct a scene in which we can navigate, as in Illustration 21.

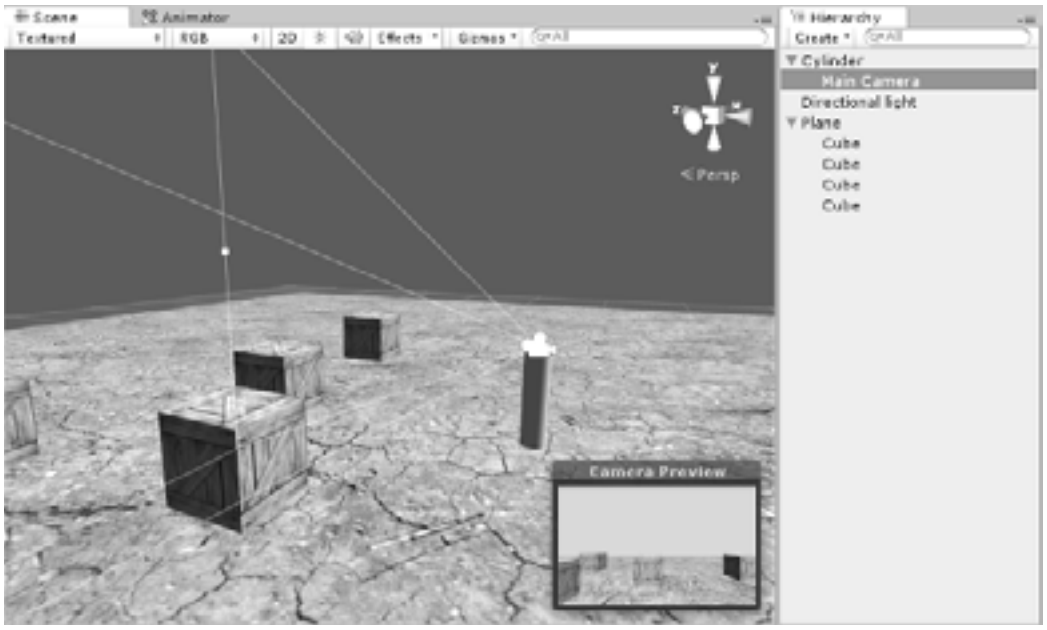


Illustration 21: The cylinder object used to create the first person input system

Notice that the camera is added as a child to the cylinder and it resides at the top of the cylinder upper surface. The mouse look mechanism is going to be as follows: when the player moves the mouse horizontally, we rotate the cylinder around the y axis leading to right or left rotation of the cylinder, depending on the direction of horizontal displacement of the mouse. On the other hand, when the player moves the mouse vertically, only the camera will be turned towards up or down. This means that we have two independent axes for horizontal and vertical rotations, which leads to a system similar to the tripod of the photography camera.

Regarding the movement, pressing W key moves the character forward in the direction the player currently faces (i.e. positive direction of the local z axis of the cylinder object). Similarly, pressing S key moves the character in the opposite of the direction it faces. This also applies to strafing right and left, where pressing D will move the character in the positive direction of its local x axis, and pressing A will move it in the opposite direction. Illustration explains cylinder reactions to user input.

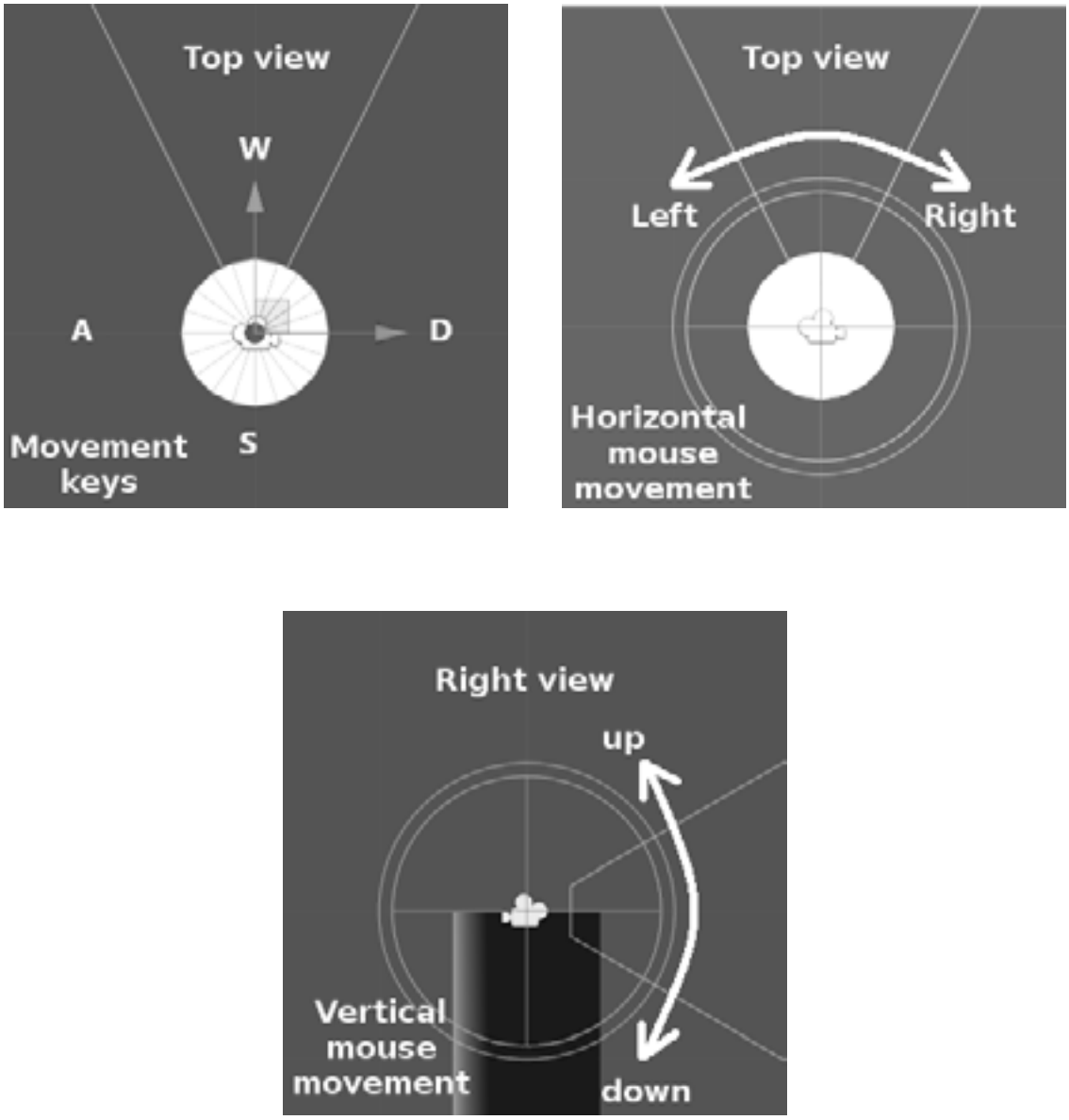


Illustration 22: Effect of the player input on the movement and the rotation of the character and the camera

If you compare this concept with the human body, you might say that the movement keys move the whole body in the four directions, while the horizontal mouse displacement rotates the body around itself on its vertical axis. On the other hand, the vertical mouse displacement moves the head only to look up and down.

One important note to mention before moving to the code. We must remember to somehow "lock" the vertical rotation of the camera towards up and down. This means that we set a maximum angle between the camera front vector and the horizon (60 degrees for example), in order to prevent the camera from rotating 180 degrees and hence becoming up side down. Back to our human body example, you see that the rotation of the human head is limited.

To implement the first person input system, we will use a script called *FirstPersonControl,* which is shown in Listing 9. This script must be attached to the cylinder, which in turn has the camera as a child as shown in Illustration 21.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class FirstPersonControl : MonoBehaviour {
5.
6.      //Vertical speed at the beginning of jump
7.      public float jumpSpeed = 0.25f;
8.
9.      //Falling speed
10.     public float gravity = 0.5f;
11.
12.     //Horizontal movement speed
13.     public float movementSpeed = 15;
```

```
14.
15.      //Mouse look speed on both axis
16.      public float horizontalMouseSpeed = 0.9f;
17.      public float verticalMouseSpeed = 0.5f;
18.
19.      //Max allowed cam vertical angle
20.      public float maxVerticalAngle = 60;
21.
22.      //Storing player velocity for movement
23.      private Vector3 speed;
24.
25.      //Mouse position in previous frame,
26.      //important to measure mouse displacement
27.      private Vector3 lastMousePosition;
28.
29.      //Store camera transform
30.      private Transform camera;
31.
32.      void Start () {
33.           lastMousePosition = Input.mousePosition;
34.           //Find camera object in children
35.           camera = transform.FindChild("Main Camera");
36.      }
37.
38.      void Update () {
39.           //Step 1: rotate cylinder around global Y
40.           //axis based on horizontal mouse displacement
41.           Vector3 mouseDelta = Input.mousePosition - lastMousePosition;
42.
43.           transform.RotateAround(
44.                       Vector3.up, //Rotation axis
45.                       mouseDelta.x *
46.                       horizontalMouseSpeed *
47.                       Time.deltaTime);//Angle
48.
49.           //Get current vertical camera rotation
50.           float currentRotation = camera.localRotation.eulerAngles.x;
51.
52.           //Convert vertical camera rotation from range [0, 360]
53.           //to range [-180, 180]
54.           if(currentRotation > 180){
55.                currentRotation = currentRotation - 360;
56.           }
57.
58.           //Calculate rotation amout for current frame
59.           float ang =
60.                -mouseDelta.y * verticalMouseSpeed * Time.deltaTime;
61.
62.           //Step 2: rotate camera around it's local X
63.           //axis based on vertical mouse displacement
64.           //First check allowed limits
65.           if((ang < 0 && ang + currentRotation > -maxVerticalAngle) ||
66.                (ang > 0 && ang + currentRotation < maxVerticalAngle)){
67.                camera.RotateAround(camera.right, ang);
68.           }
```

```
69.
70.           //Update last mouse position for next frame
71.           lastMousePosition = Input.mousePosition;
72.
73.           //Step 3: update movement
74.           if(Input.GetKey(KeyCode.A)){
75.                //Move left
76.                speed.x = -movementSpeed * Time.deltaTime;
77.           } else if(Input.GetKey(KeyCode.D)){
78.                //Move right
79.                speed.x = movementSpeed * Time.deltaTime;
80.           } else {
81.                speed.x = 0;
82.           }
83.
84.           if(Input.GetKey(KeyCode.W)){
85.                //Move forward
86.                speed.z = movementSpeed * Time.deltaTime;
87.           } else if(Input.GetKey(KeyCode.S)){
88.                //Move backwards
89.                speed.z = -movementSpeed * Time.deltaTime;
90.           } else {
91.                speed.z = 0;
92.           }
93.
94.           //Read jump input
95.           if(Input.GetKeyDown(KeyCode.Space)){
96.                //Apply jump only if player is on ground
97.                if(transform.position.y == 1.0f){
98.                       speed.y = jumpSpeed;
99.                }
100.          }
101.
102.          //Move the character
103.          transform.Translate(speed);
104.
105.          //Apply gravity to velocity
106.          if(transform.position.y > 1.0f){
107.               speed.y = speed.y - gravity * Time.deltaTime;
108.          } else {
109.               speed.y = 0;
110.               Vector3 newPosition = transform.position;
111.               newPosition.y = 1.0f;
112.               transform.position = newPosition;
113.          }
114.     }
115.  }
```

Listing 9: Implementing the first person input system

As you can see, some parts of the code are familiar since we have already dealt with them. You can refer to Listing 6 in page 26 to read the discussion over parts such as jumping.

Now let's get into the discussion of the first person input system. After declaring some variables we call the function *transform.FindChild()* in line 35. What does this function do is searching for the object with the provided name. This search is performed among the children of the current object only. In this case, we have passed the name *Main Camera*, which is the default name of the camera in Unity. Since we have already added the camera as a child to the cylinder, this function is going to find the camera and return it to be stored in *camera* variable. We are going to deal with this variable later on.

In the first step in lines 43 through 47, we call the function *transform.RotateAround(),* and its job is to rotate the object around a specific axis. Therefore, we provide a rotation axis and an angle. As for the axis it is *Vector3.up*, which is the positive direction of the global y axis that goes up. Since this is a vertical axis, the resulting rotation is going to be horizontal towards left or right. The rotation angle is a product of three values: first value is *mouseDelta.x*, which is the horizontal mouse displacement since the last frame. This value is positive when the mouse moves from left to right, which results in clockwise rotation as in part (b) in Illustration 22, and counter-clockwise rotation when the mouse moves in the opposite direction. The second value, *horizontalMouseSpeed*, represents the rotation speed. In most games, this value can be customized by the player in order to match the speed of mouse movement he is used to. The last value is *Time.deltaTime*, we have been dealing with this value for a while, since we usually need to compute the distance or angle from the speed.

In line 50, we compute current camera rotation around its local x axis. This value is always between 0 and 360, and it increases as the camera rotates clockwise. In other words, this value will increase when the camera looks down as in part (c) in Illustration 22. In lines 54 through 56, we convert this value to an angle between 180 and -180 by converting angles greater than 180 to negative angles (for example, 190 becomes -170 and so on). We store this value in *currentRotation* to benefit from it later on in computing the limits of camera rotation.

In lines 59 and 60, we compute the value of camera rotation for the current frame. This value consists of *-mouseDelta.y*, *verticalMouseSpeed*, and *Time.deltaTime,* and it is computed in a way similar to the one performed in a previous step to compute the cylinder rotation. The exception here is the use of the negative value of mouse vertical displacement *-mouseDelta.y*. The justification of that is: mouse movement upwards gives us a positive value for *mouseDelta.y*, and we need to convert it to a camera rotation upwards, which is in fact a counter-clockwise rotation around the local x axis of the camera. Therefore, the negative value results in the counter-clockwise rotation we need, and vice-versa for camera rotation downwards. After computing the angle we store it in the variable *ang*.

After computing rotation magnitude, what we need is to rotate the camera around its local x axis by this magnitude. Here we have three possibilities: first possibility is that the mouse did not move vertically, which results in zero value for *ang*. In that case we don't have to do anything. The second possibility is that the mouse moved upwards, which means that *ang* value is negative and will result in a camera rotation upwards. This is the case we check in line 65, to make sure that the resulting angle after rotation (*ang* + *currentRotation*) is greater than the minimum allowed value for camera rotation which is *-maxVerticalAngle*. The third and last possibility is that the mouse moved downwards, which gives us positive value for *ang.* In this case we have to make sure that *ang* + *currentRotation* is less than than *maxVerticalAngle*. This is what we do in line 66. If one of these conditions applies, we rotate the camera around its local x axis using *ang* value we have just computed. This rotation is applied using *transform. RotateAround()*, which we call in line 67.

In lines 74 through 92 we scan the inputs of the four direction arrows and add the appropriate direction to the final speed. This input might be forward, backwards, right, or left. The rest of lines have already been discussed in the platformer input system, so you can refer to the discussion in page 26. You can also see the final result in *scene5* in the accompanying project.

2.5 Implementing third person input system

Third person games include a collection of the most famous games, such as *Tomb Rider*, *Hitman*, *Splinter Cell*, *Max Payne*, and many more. They are based on placing the camera behind the player character at specific distance and height. Distance and height can change according to the play state, such as zooming in when the character aims with a weapon, and zooming out when the character is running fast.

You will learn in this section how to implement this type of input systems. We will benefit from a number of techniques we have learned so far, such as reading keyboard and mouse input, relations between objects, and rotation functions such as *transform.RotateArount()*. There are various methods that can relate the rotation of the camera to movement of the character. In *World of Warcraft*, for example, the mouse is used mainly to interact with the objects in the environment. Therefore, character rotation is performed using keyboard input, and camera rotation uses the right mouse button. In other games, such as *Hitman*, mouse pointer is used for aiming, so the character looks always at the position of the mouse pointer, and shooting is performed in that direction. In this latter case, the camera rotates automatically to follow the direction where the character looks.

In this section, we are going to deal with a simple system that rotates the character based on horizontal movement of the camera, in a manner similar to the one used in previous section for first person input system. We will also move the camera up and down based on vertical mouse movement. It is important to keep the camera always look at the character and follow it wherever it moves. We are going to implement this by using object relations between the character and the camera. Finally, we are going to allow the player to zoom the camera in and out using the mouse wheel. Let's begin by creating a simple character using basic shapes, it might look like the character in Illustration 23.

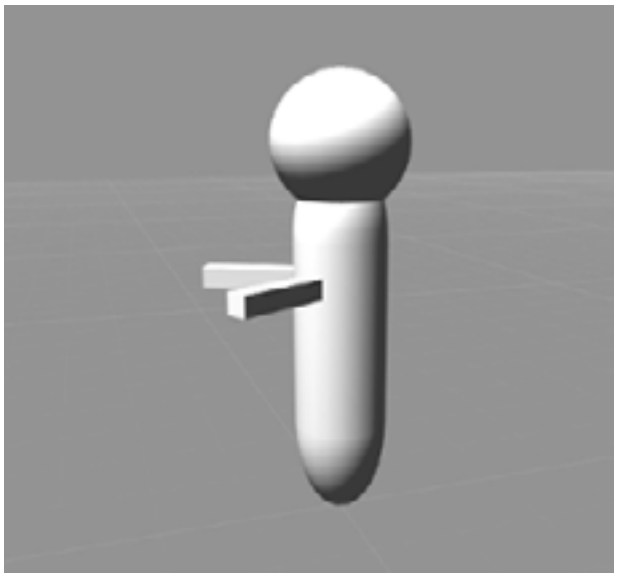


Illustration 23: A simple character we are going to use for the third person input system

After creating the character using the main body, which is a capsule object that has other objects (hands and head) as children, we need to add the camera as a child to this object. This is necessary to make the camera follow the character all the time. Next step is to add a script to the character object to control its movement. We are going also to add another script for the camera. Let's begin with the first script *ThirdPersonControl*, the script that we add to to the character to respond to keyboard input (movement and jumping). This script is shown in Listing 10.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ThirdPersonControl : MonoBehaviour {
5.
6.        //Vertical speed at the beginning of a jump
7.        public float jumpSpeed = 1;
8.
9.        //Falling speed
10.       public float gravity = 3;
11.
12.       //Horizontal movement speed
13.       public float movementSpeed = 5;
14.
15.       //Storing the player velocity for movement
16.       private Vector3 speed;
17.
18.       void Start () {
19.
20.       }
21.
22.       void Update () {
23.
24.             //Update movement
25.             if(Input.GetKey(KeyCode.A)){
26.                   //Move left
27.                   speed.x = -movementSpeed * Time.deltaTime;
```

```
28.            } else if(Input.GetKey(KeyCode.D)){
29.                 //Move right
30.                 speed.x = movementSpeed * Time.deltaTime;
31.            } else {
32.                 speed.x = 0;
33.            }
34.
35.            if(Input.GetKey(KeyCode.W)){
36.                 //Move forward
37.                 speed.z = movementSpeed * Time.deltaTime;
38.            } else if(Input.GetKey(KeyCode.S)){
39.                 //Move backwards
40.                 speed.z = -movementSpeed * Time.deltaTime;
41.            } else {
42.                 speed.z = 0;
43.            }
44.
45.            //Read jump input
46.            if(Input.GetKeyDown(KeyCode.Space)){
47.                 //Apply jump only if the player is on the ground
48.                 if(transform.position.y == 2.0f){
49.                        speed.y = jumpSpeed;
50.                 }
51.            }
52.
53.            //Move the character
54.            transform.Translate(speed);
55.
56.            //Apply gravity to velocity
57.            if(transform.position.y > 2.0f){
58.                 speed.y = speed.y - gravity * Time.deltaTime;
59.            } else {
60.                 speed.y = 0;
61.                 Vector3 newPosition = transform.position;
62.                 newPosition.y = 2.0f;
63.                 transform.position = newPosition;
64.            }
65.
66.      }
67. }
```

Listing 10: Reading movement input for third person input

We have discussed all these functions in the previous two sections, specifically in Listing 6 in page 26, and Listing 9 in page 40. Notice that this system is similar to the first person input system, except for reading mouse input and moving and rotating the camera. Camera rotation is the responsibility of *ThirdPersonCamera* script that we are going to attach to the camera. Listing 11 shows this script.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ThirdPersonCamera : MonoBehaviour {
5.
6.       //Character rotation speed
7.       public float horizontalSpeed = 0.4f;
8.
9.       //Camera vertical movement speed
10.      public float verticalSpeed = 5;
11.
12.      //Minimum and maximum allowed values
13.      //of the camera height
14.      public float minCameraHeight = 0.25f;
15.      public float maxCameraHeight = 15;
16.
17.      //Zoom control variables
18.      public float maxZoom = -10;
19.      public float minZoom = -30;
20.      public float zoomSpeed = 3;
21.
22.      //Should the camera move down when
23.      //the mouse moves up?
24.      public bool invertYMovement = true;
25.
26.      //Mouse position in the previous frame,
27.      //important to measure mouse displacement
28.      private Vector3 lastMousePosition;
29.
30.      //Reference to player game object
31.      Transform playerBody;
32.
33.      void Start () {
34.           lastMousePosition = Input.mousePosition;
35.           //Player must be the parent of the camera
36.           playerBody = transform.parent;
37.      }
38.
39.      void Update () {
40.           Vector3 mouseDelta = Input.mousePosition - lastMousePosition;
41.
42.           //Horizontal mouse displacement is interpreted
43.           //as rotation on the character
44.           playerBody.RotateAround(
45.                Vector3.up, //Rotation axis
46.                mouseDelta.x *
47.                horizontalSpeed *
48.                 Time.deltaTime);//Angle
49.
```

```
50.           //Vertical mouse displacement is interpreted
51.           //as vertical movement of the camera
52.           float yDelta = 0;
53.           if(invertYMovement){
54.                //Invert Y movement direction
55.                yDelta = -mouseDelta.y * verticalSpeed * Time.deltaTime;
56.           } else {
57.                //Use same Y movement direction
58.                yDelta = mouseDelta.y * verticalSpeed * Time.deltaTime;
59.           }
60.
61.           //Perform vertical movement of the camera
62.           transform.Translate(0, yDelta, 0, Space.World);
63.
64.           //Check if the Y position of the cam exceeds the allowed limits
65.           Vector3 newCameraPos = transform.localPosition;
66.           if(newCameraPos.y > maxCameraHeight){
67.                newCameraPos.y = maxCameraHeight;
68.           } else if(newCameraPos.y < minCameraHeight){
69.                newCameraPos.y = minCameraHeight;
70.           }
71.
72.           //Position the camera after fixing the Y position
73.           transform.localPosition = newCameraPos;
74.
75.           //Keep the camera looking at the character
76.           transform.LookAt(playerBody);
77.
78.           //Store the mouse position for the next frame
79.           lastMousePosition = Input.mousePosition;
80.
81.           //Apply zooming
82.           float wheel = Input.GetAxis("Mouse ScrollWheel");
83.           //Zoom in
84.           if(wheel > 0 && transform.localPosition.z < maxZoom){
85.                //Move the camera forward on its local Z axis
86.                 //using zoom speed
87.                transform.Translate(0, 0, zoomSpeed);
88.           } else //Zoom out
89.           if(wheel < 0 && transform.localPosition.z > minZoom){
90.                //Move the camera backwards on its local Z axis
91.                //using the negative value of zoom speed
92.                transform.Translate(0, 0, -zoomSpeed);
93.           }
94.      }
95. }
```

Listing 11: Camera script for the third person input system

Even this script is attached to the camera, it affects the character as well. At the beginning we declare a few variables that will help us to control the camera (lines 7 through 24). These variables control the speed of the horizontal rotation of the character and the vertical movement of the camera, in addition to the limits of vertical camera rotation and the limits of zooming in and out. Notice in lines 18 and 19 that the maximum and the minimum values of zooming are both negative values, since the camera must always be behind the character. According to the left-hand rule we use, positive direction of the z axis goes inside the screen.

In *Start()* function, specifically in line 36, we define the variable *playerBody* of type *Transform*. We are going to use this variable to reference the character. The transform of the character can be accessed through *transform.parent*, which returns the parent of the current game object. Remember that we attach this script to the camera, which is a child of the character.

In lines 44 and 45 we convert the horizontal displacement of the mouse to a rotation of *playerBody* around the y axis, in a way similar to what we have done in the first person system. In lines 52 through 62 we declare the variable *yDelta* to compute the vertical mouse displacement based on the value of *invertYMovement*. The value of *invertYMovement* decides whether the displacement of the camera will match the direction of the vertical mouse displacement or it will be in the opposite direction. After that we perform the vertical movement of the camera in the world space, which takes place in line 62.

After moving the camera vertically, we check in lines 65 through 70 whether the new vertical position of the camera is between the maximum and the minimum allowed values. These values are defined by *minCameraHeight* and *maxCameraHeight*. Notice that we check the position using *transform.localPosition* instead of *transform.position*. We do this in order to make the position test relative to the vertical position of the player character rather than the ground. This will make our computations applicable in all cases, including the cases where the vertical position of the character changes, such as jumping case. We store the position of the camera in *newCameraPosition*, then we check whether the member *y* of the new position is within the allowed limits. If a modification is necessary we perform it, and we finally store *newCameraPos* back in *transform.localPosition* in line 73.

Once we are done moving the camera, we need to make sure that the camera looks always at the character. So, in line 76, we call *transform.LookAt()* and pass to it *playerBody*, which refers to the character. In lines 81 through 93, we read the mouse wheel and interpret scroll up as zoom in and scroll down as zoom out. Camera movement along its local z axis is controlled by *zoomSpeed* and the position of the camera. This position after zooming must be between *maxZoom* and *minZoom*. Once again we use *transform.localPosition*, since camera zooming is performed against the character, rather than the world center. You can construct a simple scene to test this input system, and you can also see the final result in *scene6* in the accompanying project.

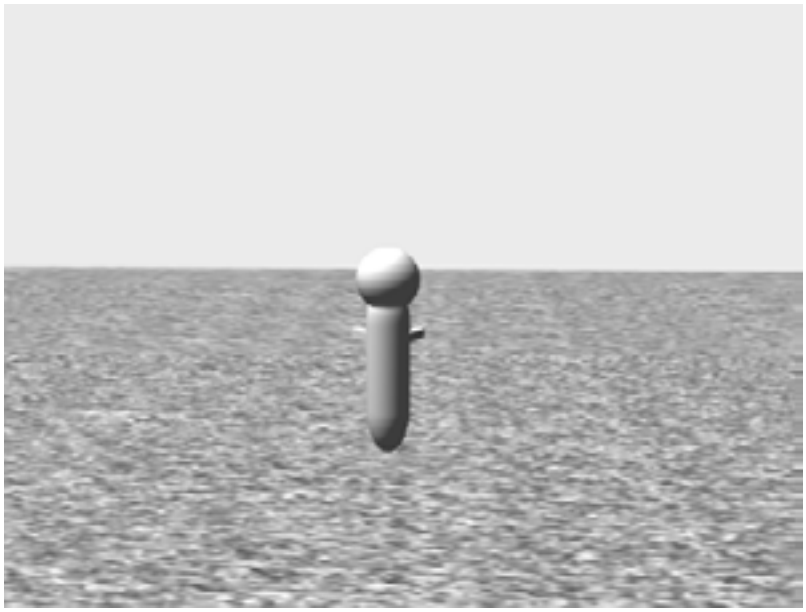


Illustration 24: A simple scene to demonstrate the third person input system

2.6 Implementing car racing games input system

Car racing input systems are similar to some extent to third person input systems, especially in terms of the position of the camera. However, there are still few differences as we are going to see. Let's begin with an object that represents the car, and here I am going to use a cube with dimensions of (2, 1, 3.5). Let's also add a ground, which is a plane with a scale of (100, 1, 100). Remember that the default side length of the plane is 10 units, so the total side length after scaling is going to be 100 * 10 = 1000 units. This is equal to one square kilometer, since each unit equals one meter. I am going also to use an asphalt texture for the ground, in addition to a directional light. Finally, we add two empty game objects as children to the car object, and these are going to be used as axes for car rotation. First object is *RotationAxisR*, which is located to the right of the car and has the local position (1, 0, 0), and the other is *RotationAxisL*, which has the local position (-1, 0, 0). The scene we are going to use is shown in Illustration 25.

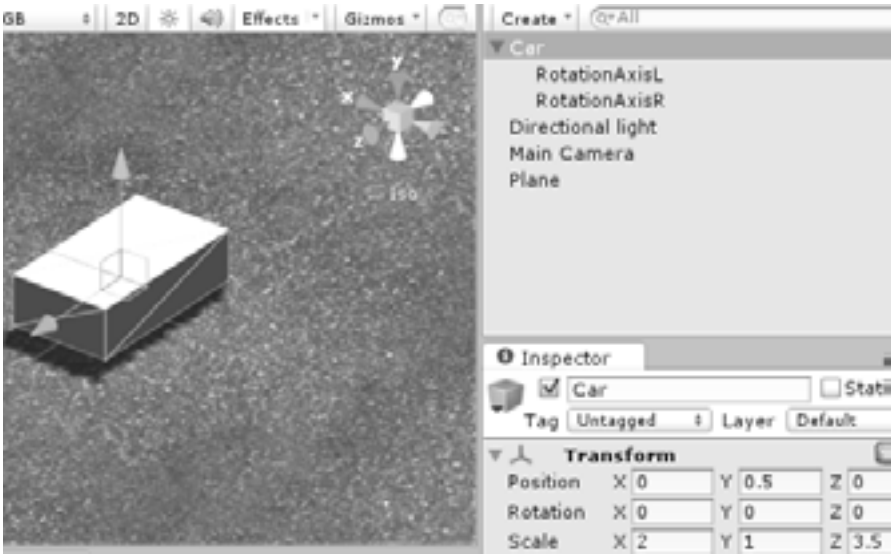


Illustration 25: A scene to demonstrate the car racing input system

The difference is that we are not going to add the camera as a child to the car like we have done with the player character in the third person input system, and this is going to serve a nice effect as we are going to see. Another difference is the way the car moves, which is not similar to the persons' movement we have seen so far. Car speed is not constant, but rather increases with time as long as the acceleration pedal is pressed, and decreases when the pedal is released. Additionally, using the brakes makes the car lose speed in a shorter time. Car rotation is also different from persons' rotation. Persons simply revolve around their selves, but cars need some area to turn within. Therefore, we can imagine two virtual axes to the right and the left of the car at some distance away from the center of the car body. This distance decreases as we steer more, so we can control the turning angle of the car. For the sake of simplicity, and because we deal only with digital input (keyboard) rather than analog input, I am going to use a fixed distance for the rotation axes.

Let's now write the script that will turn this static cube into a drivable car (behaviorally, not visually for sure!). So we can increase or decrease the speed, as well as turning right or left. Listing 12 shows *CarController* script we are going to use.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class CarController : MonoBehaviour {
5.
6.        //Max car speed in Km/h
7.        public float maxSpeed = 200;
8.
9.        //Increment in speed in Km/h
10.       public float acceleration = 20;
11.
12.       //Decrement of speed in Km/h
13.       public float deceleration = 16;
14.
15.       //Decrement of speed in Km/h when braking
16.       public float braking = 60;
17.
18.       //Decrement of speed in Km/h when turning right or left
19.       public float turnDeceleration = 30;
20.
21.       //Rotation speed when steering in degree/sec.
22.       public float steeringSpeed = 70;
```

```
23.
24.      //Rotation axis on right and left
25.      Transform rightAxis, leftAxis;
26.
27.      //Current car speed in m/s
28.      float currentSpeed = 0;
29.
30.      //Multiply by this value to convert from
31.      // Km/h to m/s
32.      const float CONVERSION_FACTOR = 0.2778f;
33.
34.      void Start () {
35.           //Find right and left rotation axes in children
36.           rightAxis = transform.FindChild("RotationAxisR");
37.           leftAxis = transform.FindChild("RotationAxisL");
38.      }
39.
40.      void Update () {
41.
42.           if(Input.GetKey(KeyCode.UpArrow)){
43.                //Acceleration pressed
44.                //Increase the speed by acceleration amount
45.                AdjustSpeed(
46.                       acceleration * CONVERSION_FACTOR * Time.deltaTime);
47.           } else {
48.                //Acceleration released
49.                //Decrease the speed by deceleration
50.                AdjustSpeed(
51.                       -deceleration * CONVERSION_FACTOR * Time.deltaTime);
52.           }
53.
54.           if(Input.GetKey(KeyCode.DownArrow)){
55.                //Braking pressed
56.                //Decrease the speed by braking amount
57.                AdjustSpeed(
58.                       -braking * CONVERSION_FACTOR * Time.deltaTime);
59.           }
60.
61.           //Turning right: no rotation if the current speed is < 5 Km/h
62.           if(Input.GetKey(KeyCode.RightArrow) &&
63.                currentSpeed > 5 * CONVERSION_FACTOR){
64.                AdjustSteering(steeringSpeed, rightAxis.position);
65.           }
66.
67.           //Turning left: no rotation if the current speed is < 5 Km/h
68.           if(Input.GetKey(KeyCode.LeftArrow) &&
69.                currentSpeed > 5 * CONVERSION_FACTOR){
70.                AdjustSteering(-steeringSpeed, leftAxis.position);
71.           }
72.
73.           //Perform movement on the local Z axis using current speed
74.           transform.Translate(0, 0, currentSpeed * Time.deltaTime);
75.
76.      }
```

```
77.
78.      //Adds a new value to the current speed
79.      //Checks max and min limits
80.      //The new value must be in m/s
81.      void AdjustSpeed(float newValue){
82.           currentSpeed += newValue;
83.           if(currentSpeed > maxSpeed * CONVERSION_FACTOR){
84.                currentSpeed = maxSpeed * CONVERSION_FACTOR;
85.           }
86.
87.           if(currentSpeed < 0){
88.                currentSpeed = 0;
89.           }
90.      }
91.
92.      //Rotates the car horizontally around the provided point
93.      //using the provided rotation speed in degree / second
94.      void AdjustSteering(float speed, Vector3 rotationAxis){
95.           //Rotate using the provided axis and steering speed
96.           transform.RotateAround(
97.                rotationAxis, Vector3.up, speed * Time.deltaTime);
98.           //If the current speed is > 30 Km/h,
99.           //reduce it by the amount of turn deceleration
100.          if(currentSpeed > 30 * CONVERSION_FACTOR){
101.               AdjustSpeed(
102.                      -turnDeceleration * CONVERSION_FACTOR *
103.                      Time.deltaTime);
104.          }
105.     }
106.  }
```

Listing 12: Car control script

In lines 6 through 19 we declare few variables related to the car speed, including max speed of the car *maxSpeed*, increment of the speed with the time *acceleration*, decrement of the speed with the time *deceleration*, decrement of the speed with the brakes *braking*, and decrement of the speed when the car turns right or left *turnDeceleration*. All these values are in km/h, which makes it easy for us to deal with them.

In line 22 we declare *steeringSpeed*, which is expressed in degrees per second. The variables *leftAxis* and *rightAxis* declared in line 25 are going to be used as references to the objects *RotationAxisR* and *RotationAxisL* which we have added as children to the car. We are going to use these objects as rotation axes when the car turns, since the car does not simply revolve around itself, but it needs some space to turn in. This space is determined by using these two axes: the farther a rotation axis is, the larger the resulting turning area is going to be.

The variable declared in line 28 stores the current speed of the car after applying acceleration, braking, and steering values that come from the player input. Notice that this speed variable, unlike the previous ones, is expressed in m/s rather than km/h. This is because the time unit used in Unity is the second, and the distance unit is the meter, which makes dealing with these units easier than the kilometer and the hour. To convert from km/h to m/s, we use the constant *CONVERSION_FACTOR* declared in line 32, which has a constant value of 0.2778. So all we have to do is to multiply any km/h value by *CONVERSION_FACTOR* to convert it to m/s. After that, in *Start()* function, we find the objects *RotationAxisR* and *RotationAxisL* and reference them using *rightAxis* and *leftAxis* consequtively, in order to use them when implementing car turning.

Before getting into the details of *Update()* function, I want to jump to lines 81 through 90, in which we declare a custom function called *AdjustSpeed()*. If you are unfamiliar with programming, we can simply say that declaring functions is useful when you need to repeat the same task several times in different places, and this task takes a number of lines to program, which makes writing it over an over tedious and more error prone. And this is the case when adjusting the speed of the car: we need first to add the new value to the current speed, then check whether the result exceeds the maximum speed, in that case we set the current speed to the value of maximum speed. We need also to check whether the result is less than zero, and set the current speed to zero in that case. So what we need to do is to call this function whenever we want to change the speed of our car, and provide it with *newValue* that we want add or subtract from the current speed. Before adding it, *newValue* need to be converted to m/s, which is the unit used by *AdjustSpeed()*.

We have another custom function in the lines 94 through 105, which is *AdjustSteering()*. We are going to use this function to implement turning right and left. When we call this function, we must provide it with a turn speed and a turn axis. *AdjustSteering()* rotates the car around *rotationAxis* using the value of *speed* variable. After that it checks the current speed of the car. If the current speed is greater than 30 km/h, it is reduced by *turnDeceleration*. Notice that *AdjustSteering()* calls *AdjustSpeed()* in order to apply deceleration. This is perfectly legal in programming languages; to have a function call another function.

Back to *Update()* function, and specifically to lines 42 through 52, where we check the state of the up arrow, and increment the current car speed by the value of *acceleration* if the player is holding this arrow. Notice that we change the speed by calling *AdjustSpeed()* function and providing it with *acceleration* multiplied by *CONVERSION_FACTOR* to convert km/h to m/s. Calling *AdjustSpeed()* without performing this conversion of units will result in a wrong value. If the player is not pressing up arrow key, we decrease the speed with the negative value of *deceleration.* We use the negative value here in order to have *AdjustSpeed()* subtract *deceleration* value from the current speed. In lines 54 through 59 we check the state of the down arrow, and activate the brakes if the player is pressing it. Activating the breaks means decreasing the speed with the value of *braking*. This means that the car will need less time to stop completely when the brakes are active.

Turning is performed in lines 62 through 71, were we check the state of the right arrow as well as the current car speed. We do not turn the car if its current speed is less than 5 km/h, since steering must not be able to move the car if it is completely stopped. Notice in line 64 that we use *rightAxis* as the rotation axis, along with the positive value of *steeringSpeed*. This is because turning right needs a positive (clockwise) rotation around the vertical axis (line 97). On the other hand, in line 70, we use the negative value of *steeringSpeed*, to achieve a counter-clockwise rotation around *leftAxis*.

After computing the speed of the car and performing the necessary rotation for turning, we need to move the car forward along its positive z axis. We use *currentSpeed* multiplied by the time. Remember that *currentSpeed* is expressed in m/s, so it is safe to multiply it directly by *Time.deltaTime* as in line 74. You can now test your car control script since it is ready, then you can move to the next step, which is writing the camera script.

The camera in car racing games follows the car at a specific distance behind it, and it has also a specific height above the vertical position of the car. Additionally the rotation of the camera after the car turns is not immediate, but rather has some latency, and has a speed less than the turn speed of the car. So if the car turns for long time, a part of its side will become visible to the player as in Illustration 26. This script we are going to add to the camera is *CarCamera* in Listing 13.

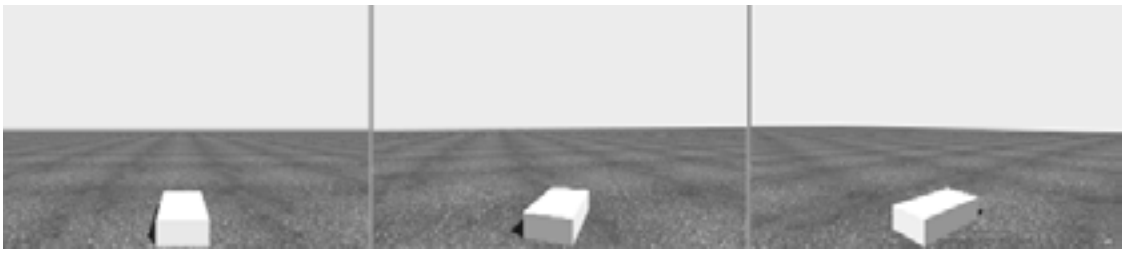


Illustration 26: Car rotation in front of the camera during turning, and the apparition of car side

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class CarCamera : MonoBehaviour {
5.
6.       //Reference to the car object
7.       public Transform car;
8.
9.       //Camera height above the Y position of the car
10.      public float height = 4;
11.
12.      //distance between the car and the camera
13.      //regardless of the height of the camera
14.      public float zDistance = 10;
15.
16.      //Seconds to wait before turning
17.      //the camera after the car turns
18.      public float turnTimeout = 0.25f;
19.
20.      //Speed of camera rotation
21.      //in degrees / sec.
22.      public float turnSpeed = 50;
23.
24.      //Time passed since the angle between the camera
25.      //and the car changed to a large value
26.      float angleChangeTime = -1;
27.
28.      void Start () {
29.           //Set position and rotation of the camera
30.           //to the same values of the car
31.           transform.position = car.position;
32.           transform.rotation = car.rotation;
33.      }
34.
35.      //We use late update to make sure that the car
36.      //moves before the camera
37.      void LateUpdate () {
38.            //We start by positioning the camera
39.           //at the same position of the car
40.           transform.position = car.position;
41.
42.           //Now move the camera backwards on its local
43.           //Z axis by the defined distance
44.           //and up by the defined height
45.           transform.Translate(0, height, -zDistance);
```

```
46.
47.           //Measure the angle between the camera and the car front
48.           float angle = Vector3.Angle(car.forward, transform.forward);
49.
50.           //Check if the angle is greater than the dead zone
51.           //The value of angle is always positive, regardless
52.           //of turn direction of the car
53.           if(angle > 1){
54.                //The difference is large
55.                //Check if it is time to start rotating the camera
56.                if(angleChangeTime == -1){
57.                       angleChangeTime = 0;
58.                }
59.
60.                //Add the delta time to
61.                //the total angle change time
62.                angleChangeTime += Time.deltaTime;
63.
64.                if(angleChangeTime > turnTimeout){
65.                       //It is time to start rotating the camera
66.                       //Perform a vector cross multiplication between
67.                       //the forward vectors of the camera and the car
68.                       float resultDirection =
69.                              Vector3.Cross(car.forward,
70.                              transform.forward).y;
71.
72.                       //The sign of the y value in the resulting vector
73.                       //determines the sign of rotation direction
74.                       float rotationDirection;
75.                       if(resultDirection > 0){
76.                       rotationDirection = -1;
77.                } else {
78.                       rotationDirection = 1;
79.                }
80.
81.                //now rotate the camera around the car in
82.                //the rotation direction using turn speed
83.                transform.RotateAround(car.position,
84.                                                  Vector3.up,
85.                       rotationDirection * turnSpeed * Time.deltaTime);
86.                }
87.           } else {
88.                //The difference is small,
89.                //reset the angle change time
90.                angleChangeTime = -1;
91.           }
92.      }
93. }
```

Listing 13: Camera script for the car racing input system

First of all remember that we are dealing with a camera that is independent of the car object, unlike third person camera, which was a child of the player character. Therefore we declare the variable *car*, in order to reference the car object, just like we have done in platformer input system (see Illustration 19 in page 31). Additionally, we have *height*, which is the vertical distance between the y position of the car and the y position of the camera. Finally, we have *zDistance*, which tells us how many meters there are between the car and the camera.

To control camera rotation we declare *turnTimeout*, which is the time the camera waits after the car turns, and before the camera starts to rotate behind it. As for *turnSpeed*, we are going to use it to determine the speed of camera rotation. Finally, we have *angleChangeTime*, which stores the time which the angle between the front vector of the car and the look direction of the camera becomes more than one degree. We need to store this time in order to compute the time passed since the angle change, and hence start to rotate the camera when this time exceeds *turnTimeout*. This is going to be discussed in detail shortly.

The first step in *Start()* function is to make sure that the camera is at the same position of the car, and is looking to the same direction of the front vector of the car. Therefore we copy the values from the position and the rotation of the car to the position and the rotation of the camera. It is a good time now to briefly discuss the programmatic concept of copying the value and copying the reference. In case of copying the value, like our case in *Start()* function, the variables involved remain independent of each other after copying is completed. This means that any future change on *car.position* or *car.rotation* is not going to affect the camera, which is not the case when copying by reference. However, I am not going to discuss reference copy unless it is the appropriate time to do so.

Notice that in line 37 we use *LateUpdate()* instead of *Update()*, in order to make sure that Unity executes the logic of *CarController* first and updates the position of the car before executing *CarCamera,* which makes the camera follow it (you can go back to page 30 for more on *LateUpdate()*). In lines 40 and 45, we position the camera in its correct place relative to the car. We perform this through two steps: firstly we position the camera at the same position of the car (line 40), then we use *transform.Translate()* to move it backwards and upwards using *zDistance* and *height*. We use the negative value of *zDistance* to have the camera move backwards and be behind the car, based on the left hand rule as always.

After updating the position of the camera, it is time to update the rotation. The first step is to get the angle between the car front direction *car.forward*, and the direction at which the camera is looking *transform.forward*. The forward vector of any object points to the positive direction of the local z axis of that object. We perform angle measurement in line 48, and store the angle value in *angle* variable. It is important to mention here that *Vector3.Angle()* measures the angle between the two given vectors, and returns the minimum possible angle between them. The returned angle is always positive, regardless of the order in which vectors are passed to the function.

All following steps in lines 53 through 85 are related to camera rotation after the car turns. These steps depend on having an angle between the car front and the direction of the camera, which is greater than one degree. This condition is checked in line 53 before moving to the next steps. These steps begin by testing the value of *angleChangeTime*, and reset it to zero if its is equal to -1 (lines 56 through 58). After that, in line 62, we accumulate the time passed since last frame to the value of *angleChangeTime*. In line 64 we check whether the time passed since the change in the angle exceeds waiting time specified by *turnTimeout*. If this is true, we begin to rotate the camera.

As we have learned before, we need three pieces of information to rotate the camera: a rotation axis, a rotation direction (clockwise or counter-clockwise), and a rotation speed. The rotation axis is the vertical axis located at the same position of the car, since we are going to rotate the camera around the car object. The speed of the rotation is already defined by *turnSpeed*, so what is left now is determining the rotation direction. This direction depends on whether the car has turned left or right. When the car turns right, we rotate the camera clockwise, and when the car turns left, we rotate the camera counter-clockwise.

To find the direction of the rotation, we use the vector cross product in the lines 68 through 70. The benefit we get from using cross product is the direction of the resulting vector, which is going to be important to us. This vector is perpendicular to the two vectors involved in the cross product operation. Since the camera direction and the car front are both horizontal, the resulting vector is vertical, and it points either up or down. The direction is determined by the smallest angle between the two vectors involved in the cross product. To illustrate, let's take the example in Illustration 27, which shows the case in which the car turns right.

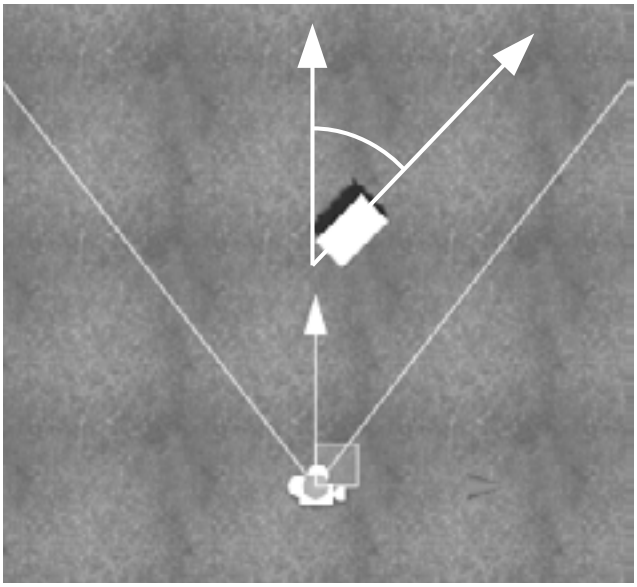


Illustration 27: Angle difference between the car front (a) and the camera direction (b)

In Illustration 27, the camera looks forward while the car front is a bit rotated to the right. To get the rotation direction, we need to apply left-hand rule on the cross product between these two vectors. This rule is different from the right-hand rule of cross product you might have learned in mathematics or physics class. So, according to this rule, the fastest way to let the first vector in the operation (car front) point to the direction of the second vector (camera front) is to rotate it counter-clockwise. It is clear in Illustration 27 that we need to rotate vector *a* counter clockwise to match the direction of vector *b*. This counter-clockwise rotation results in a vector that points downwards, hence has a negative value of *y*. This value is stored in *resultDirection* variable to be used later for determining the direction of camera rotation.

In lines 74 through 79, we determine the direction of camera rotation based on the value of *resultDirection*. If *resultDirection* is negative, left-hand rule tells us that rotation direction must be counter-clockwise, hence negative. Remember, however, that the front vector of the car is what we must rotate counter-clockwise, but we have no control over the car in this script. Therefore, a counter-clockwise rotation of the car can be substituted by a clockwise rotation of the camera. For that reason, you see that *rotationDirection* always has the opposite sign of *resultDirection*. Finally, we perform the rotation in lines 83 through 85 based on our computations. The final result can be seen in *scene7* in the accompanying project.

2.7 Implementing flight simulation input system

Last input system I am going to discuss in this chapter is the flight simulation input system. Surely, we are not going to implement a simulation of an airplane cockpit like what you might have seen in *Microsoft Flight Simulator*, but it is rather a simple system like the one you might have seen in some *GTA* series games. We are going to implement a flight jet with wings, not a helicopter.

This type of input systems is relatively simple to program, since it depends completely on rotations. Nevertheless, it might be hard to control for the players who are not used to this type of games. In this section I am going to cover a single state, which is the continuous flying of the plane. This means that taking off and landing are not going to be covered. Additionally, we are going to assume a constant flying speed that cannot be changed by the player. So let's at the beginning make a simple plane model like the one in Illustration 28 using cubes with varied sizes. These cubes are added as children to the plane body, so the plane can move as one unit.

Illustration 28: A simple plane model built using cubes

Let's now describe how the plane can be controlled. We let the plane fly forward automatically, and the player must not be able to stop it. This makes sense because the pilot can never stop the plane in the air. What the player can do is to rotate the plane around its local z axis using right and left arrows, in a movement known as *roll*. Another rotation the player can do is around the local x axis of the plane using up and down arrows, and this movement is known as *pitch*.

When the player presses down arrow, the front side of the plane raises, so the altitude of the plane increases as it moves. The opposite happens when the player presses up arrow, where the front side of the plane gets lower and the altitude decreases with the time. Right and left arrows do not affect the altitude or the direction of the plane, but they make it roll to the right or the left. So when the plane roll to the right and then raise its front, it is going eventually turn to right. This type of control needs some time to get used to if the player has no experience with it.

As for camera, we simply add it as a child to the plane and make it a little bit higher than it. Obviously, the camera is going to be behind the plane, so it gives a view similar to what you see in Illustration 29. It is a good idea to extend the far clipping plane of the camera to 5000 instead of 1000, because flight games depend on long view distances. This is a result of the nature of the planes, as they fly at high speed on high altitudes. The final step is to add *FlyController* script to the plane in order to control it. This script is shown in Listing 14.

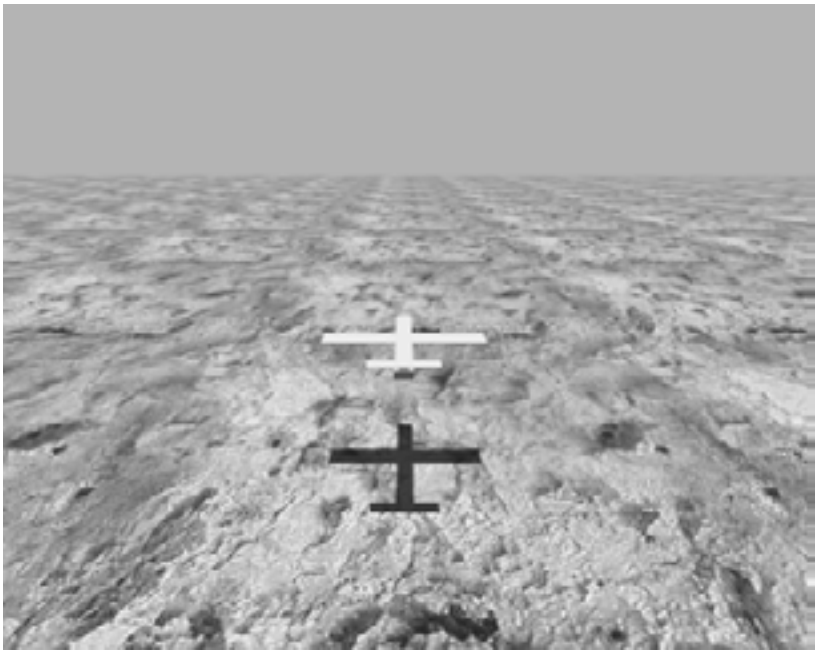


Illustration 29: The plane as seen by the camera during play

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class FlyController : MonoBehaviour {
5.
6.       //forward flying speed in m / s
7.       public float flySpeed = 166;
8.
9.       //Speed of the rotation around local z axis
10.      public float rollSpeed = 35;
11.
12.      //Speed of the rotation around local x axis
13.      public float pitchSpeed = 35;
14.
15.      void Start () {
16.
17.      }
18.
19.      void Update () {
20.           //rotation values for this frame
21.           float roll, pitch;
22.
23.           //Compute the rotation around z axis based on
24.           //right and left arrow keys
25.           if(Input.GetKey(KeyCode.RightArrow)){
26.                roll = -rollSpeed * Time.deltaTime;
27.           } else if(Input.GetKey(KeyCode.LeftArrow)){
28.                roll = rollSpeed * Time.deltaTime;
29.           } else {
30.                roll = 0;
```

```
31.           }
32.
33.           //Compute the rotation around x axis based on
34.           //up and down arrow keys
35.           if(Input.GetKey(KeyCode.DownArrow)){
36.                pitch = -pitchSpeed * Time.deltaTime;
37.           } else if(Input.GetKey(KeyCode.UpArrow)){
38.                pitch = pitchSpeed * Time.deltaTime;
39.           } else {
40.                pitch = 0;
41.           }
42.
43.           //Perform rotations around local
44.           // z and local x axes
45.           transform.Rotate(0, 0, roll);
46.           transform.Rotate(pitch, 0, 0);
47.
48.           //Move the plane forward based on flying speed
49.           transform.Translate(0, 0, flySpeed * Time.deltaTime);
50.
51.           //Do not allow the plane to sink below 5 meters
52.           Vector3 pos = transform.position;
53.           if(pos.y < 5){
54.                pos.y = 5;
55.           }
56.           transform.position = pos;
57.      }
58. }
```

Listing 14: The plane control system

As you can see the code is fairly simple and involves techniques we have already discussed. You can see the final result in *scene8* in the accompanying project.

This concludes this chapter about reading user input. We have learned how to read keyboard and mouse input and convert this input to meaningful actions that allow the player to interact with the game environment. Unity allows us to read input from wide range of input devices, such as game pads, joysticks, touch screens, steering wheels and others. Even it is not possible to cover all of them in this chapter, the basic idea is similar: you read the state of keys and buttons as well as the displacement of the fingers on a touch screen and interpret them to some actions in the game.

Exercises

1. In Listing 6 in page 26, we implemented a platformer input system. Some games of this type allow the player to increase the speed of character using a special key. Declare a new variable and call it *runSpeed*, and give it a value greater than normal speed if the character. After that add code that scans left shift key (use *KeyCode.LeftShift*). If the player is holding left shift, then use *runSpeed* for movement to make character move faster, or use default speed if the player is not pressing shift. You may apply running on other input systems as well if you wish.
2. When we implement person movement systems such as platformer, first person, and third person systems, we did not take into account acceleration and deceleration of the movement. Try to make use of acceleration/deceleration mechanism we implemented in Listing 12 in page 50 to enhance movement in these systems. For example, you can prevent sudden stop of the character during jumping, since it must be driven towards jump direction until it land on the ground again. You can implement this mechanism in any way you see appropriate.
3. Try to use car camera script in Listing 13 in page 54 with third person input system instead of adding the camera as child to the player character. What changes you need to do to let the player control the character easily using new camera system?
4. Modify plane control script in Listing 14 in page 59 so that plane return to its original rotation when the player releases control keys. You have to do computations similar to those in car camera, since you need to find the amount and direction of the rotation and when to stop rotating. Apply this to rotations on both x and z axes.

3 Basic Game Logic

Introduction

Computer game is in fact a piece of software, which has to abide to the logic of computer in dealing with facts, making decisions, and performing tasks. In this chapter we are going to learn a number of common game mechanics and how to program them. We have already dealt with a couple of mechanics which are moving and jumping.

After completing this chapter, you are expected to:

- Program simple shooting.
- Program collectables such as coins and other items
- Program objects holding and releasing.
- Program triggers and usable objects.

3.1 Shooting

Shooting mechanic is very common in 2D and 3D games. In this section we are going to discuss a simple projectile that moves forward with constant speed. This means that we are not going to discuss any external effects on the projectile such as gravity. Additionally, high speed projectiles such as rifle and shotgun bullets are not going to be covered, since they need a different technique that we are going to discuss later on.

Let's begin with a simple game that is similar to the classic game *Space Invaders*. We are going to build a simple space shuttle like in Illustration 30, and then we should add a script to control this shuttle. We are going to be able to move the shuttle in the four directions and shoot two different types of projectiles: bullets and rockets, which have similarities and dissimilarities. Since we are dealing with a top-down view, movement of the shuttle is going to be on the x and z axes. This time we should change the perspective of the camera to orthogonal, so the whole scene gets rendered in two dimensions, so the cubes look like rectangles.

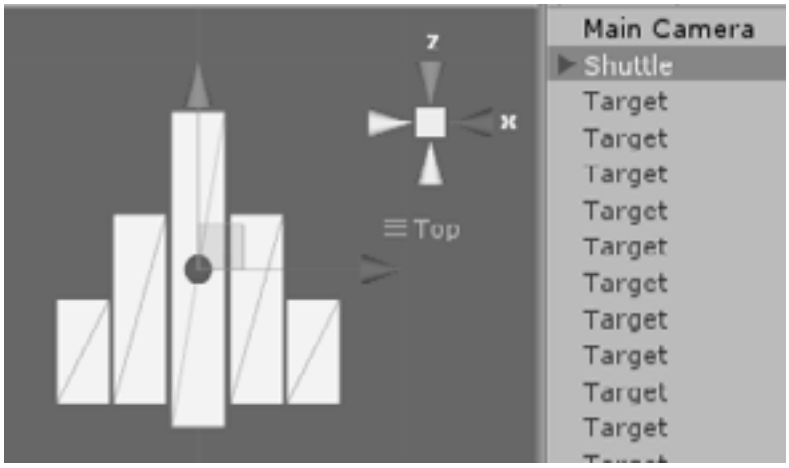


Illustration 30: The space shuttle to be used for shooting projectiles

In this section we are going to learn a new concept in Unity, which is the *prefab*. The idea of the prefab is based on creating a game object and add all of the necessary components, scripts, textures, and so on to that object. After that, we store this object as a prefab. This prefab can be used to generate unlimited number of copies of the original object, and we are able to modify a large number of objects from one place by modifying the prefab used to create them. Prefabs are going to be useful for making bullets and rockets, since shooting requires generating unspecified number of bullets and rockets during game execution.

There is going to be a number of scripts in this scene, therefore I am going to reveal them in a specific order that delays the scripts which have dependencies. So I am going to begin with independent scripts that do not need to reference other scripts. Therefore I am going to leave the shuttle for a while to discuss the targets that our shuttle will be shooting. We are going to use prefabs to create targets, since we are going to have a relatively large number of targets in the scene, and it would be great to be able to handle these objects from single place.

To create the target prefab, add a cube to the scene. We are going to add the script *Target* to this cube, which has only one variable called *hit*. *hit* is a boolean value that determines whether the target has already been hit or not. The script *Target* is shown in Listing 15.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Target : MonoBehaviour {
5.
6.       //The bullet sets this value to true when it hits the target
7.       public bool hit = false;
8.
9.       //Set this flag to true after calling destroy
10.      bool destroyed = false;
11.
12.      void Start () {
13.
14.      }
15.
16.      void Update () {
17.           if(hit){
18.                //The bullet has hit the target. Play destruction
19.                //animation, which is rotation and size reduction
20.                transform.Rotate(0, 720 * Time.deltaTime, 0);
21.                transform.localScale -= Vector3.one * Time.deltaTime;
22.
23.                //If we have not called Destroy() yet, call it now
24.                if(!destroyed){
25.                       //Delay destruction for one second so the player
26.                       //can see the animation
27.                       Destroy(gameObject, 1);
28.                       //Set the destruction flag to prevent multiple
29.                       //Destroy() calls
30.                       destroyed = true;
31.                }
32.           }
33.      }
34. }
```

Listing 15: Script for targets to be shot

This script checks the value of *hit*, and eventually destroys the object if the value is *true*. Since there are no statement in this code that changes the value of *hit*, the object maintains its state until another script modifies this value. We are going to see shortly how does the bullet check for a collision between itself and the object, and change the value of *hit* if such collision exists. The script also adds a sort of animation to the object, by rotating it and reducing its size with time once the object is hit. This animation makes the target looks like if it is falling down.

Rotation is performed using the function *transform.Rotate()*, which we have dealt with in many cases before. On the other hand, reducing the object size is achieved by subtracting a small amount from the object scale. This amount is equal to a vector that has components with values equal to *Time.deltaTime*, which makes size reduction in a speed of one meter per second; so the object disappears after one second because of zero scale. Therefore, we call *Destroy()* in line 27 and pass to it the object we want to destroy (in this case it is the same target object, which can be achieved through the variable *gameObject*). In addition to the object, we also pass to *Destroy()* the time it should wait before performing the destruction. In this case the time is one second.

What we mean by destroying the object in this context is removing the object completely from the scene, and this can be observed by disappearance of the object from the hierarchy. To avoid calling *Destroy()* more than one time, we used the variable *destroyed* as a flag for calling that function. We need *destroyed* in this case because we have delayed the destruction to show the animation. This delay will cause *Update()* to be called several times during next second before the object is actually destroyed. During this second, we want to repeatedly call falling animation, but we want to call *Destroy()* only once.

We well now add another script to the target game object. The job of this script is to move the target so it does not stay still. This movement depends on time only and not on the player input like we have done several times before. The reason is obvious: the player do not control the targets, they move by themselves. The script is going to move the target in a specific direction with a constant speed. Listing 16 shows *AutoMover* script, which moves the object over the time in the direction specified by the *speed* vector. When the object leaves the field of view of the camera from one side, it should be repositioned in the opposite side. This movement is known as *wrapping*, and is common in many games, including the famous classic game *Pac-Man*. Listing 17 shows *Wrapper* script, which rotates the object around the scene based on its position on x and z axes.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class AutoMover : MonoBehaviour {
5.
6.       //Movement speed
7.       public Vector3 speed = new Vector3(0, 0, 0);
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.           //Move the object with the specified speed
15.           transform.Translate(speed * Time.deltaTime);
16.      }
17. }
```

Listing 16: A script that moves the object in a specific direction with a constant speed

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Wrapper : MonoBehaviour {
5.
6.       //When the target moves out of these bounds,
7.       //it should be wrapped around the scene
8.       public Vector3 limits = new Vector3(10, 0, 10);
9.
10.      void Start () {
11.
12.      }
13.
14.      void Update () {
15.           //Get the current position
16.           Vector3 newPos = transform.position;
17.
18.           if(transform.position.x > limits.x){
19.                //Object left from the right, return it from the left
20.                newPos.x = -limits.x;
21.           }
22.
23.           if(transform.position.x < -limits.x){
24.                //Object left from the left, return it from the right
25.                newPos.x = limits.x;
26.           }
27.
28.           if(transform.position.z > limits.z){
29.                ////Object left from the front, return it from the back
30.                newPos.z = -limits.z;
31.           }
32.
33.           //Set the new position after the modifications
34.           transform.position = newPos;
35.      }
36. }
```

Listing 17: The script that wraps the object around the scene if it leaves the view of the camera

Nothing new in these scripts except the wrapping step, which is as simple as modifying the values of x or z in the position if they exceed the preset limits. After adding *Target*, *AutoMover*, and *Wrapper* to the cube that we want use as a target, we are now ready to create a prefab for targets, which should allow us to add a number of targets to the scene.

To create a prefab, select the desired object from the hierarchy, and then drag it to any folder in the project explorer, preferably to a special folder named *prefabs* as in Illustration. After that you can add as many copies as you want to the scene by dragging the prefab to the scene view or the hierarchy. The objects that are connected to the prefab appear in the hierarchy in blue color. Any modification applied to the prefab such as adding a script or modifying a component affects all objects connected to this prefab.

Illustration 31: Creating a new prefab from an object

I believe it is a good time to dive again in programming details, and this time in some details of object-oriented programming. One of the important features of object orientation is the ability to reuse the code, and this is usually achieved by using inheritance. In Unity, however, there is a different technique that can be used, namely *composition over inheritance*. In this technique, we separate the different behaviors objects can expose into separate scripts. After that we can arbitrarily attach these scripts in any combination to the objects. For example, we can have a static target that does not move by attaching *Target* script only to it. If we want this target to move as well, all we have to do is to attach *AutoMover* script to it. Similarly, an object that has *AutoMover* script but does not have *Target*, is actually a moving object that cannot be shot and hit.

Now we need to create our bullet. The object we are going to use is a sphere with a scale of (0.25, 025, 0.25). The bullet moves forward with a constant speed when shot, and it also has a distance range. When the bullet moves beyond its distance range, it is automatically destroyed and removed from the scene. This first behavior of the bullet is provided by *Projectile* script shown in Listing 18. Additionally, the bullet must be able to detect any collision between itself and any one of the targets in the scene. If the bullet hits a target, this target must be registered as hit and the bullet must be destroyed immediately. This collision detection is the job of *TargetHitDestroyer* script shown in Listing 19.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Projectile : MonoBehaviour {
5.
6.       //Movement speed of the projectile in m/s
7.       public float speed = 15;
8.
9.       //How many meters can the projectile travel?
10.      public float range = 20;
11.
12.      //Compute distance moved so far. When total distance reaches
13.      //the range, the projectile must be destroyed
14.      float totalDistance = 0;
15.
16.      void Start () {
17.
18.      }
19.
20.      void Update () {
21.           //Compute the distance to move in current frame
22.           float distance = speed * Time.deltaTime;
23.           //Move the projectile forward on its local z axis
24.           transform.Translate(0, 0, distance);
25.
26.           //Add current frame distance to total distance
27.           totalDistance += distance;
28.
29.           //When the projectile reaches its range distance
30.           //we have to destroy it
31.           if(totalDistance > range){
32.                Destroy(gameObject);
33.           }
34.      }
35. }
```

Listing 18: A script for moving the bullet and setting a distance range for it

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class TargetHitDestroyer : MonoBehaviour {
5.
6.       void Start () {
7.
8.       }
9.
10.      void Update () {
11.           //Find all targets in the scene
12.           Target[] allTargets = FindObjectsOfType<Target>();
13.
14.           //Check each target if it is hit
15.           foreach(Target t in allTargets){
16.                //We care about targets that havn't already been hit
17.                if(!t.hit){
18.                       //Distance between the projectile the target
19.                       float distance = Vector3.Distance(
20.                                                transform.position,
21.                                                t.transform.position);
22.
23.                       if(distance <
24.                              t.transform.localScale.magnitude * 0.5f){
25.                              //The projectile touches the target
26.                              //Set hit flag to true.
27.                              t.hit = true;
28.
29.                              //Now destroy the projectile
30.                              Destroy(gameObject);
31.
32.                       }
33.                }
34.           }
35.      }
36. }
```

Listing 19: A script that detects collisions between an object and the targets in the scene

TargetHitDestroyer script introduces to us the concept of arrays. An array is a collection of objects that have the same type, and can be accessed through a single variable. In line 12 of Listing 19 we declare the array *allTargets*, in which we are going to store all the targets in the scene. You can easily recognize arrays through the square brackets [] in their declaration. In the same line we call the function *FindObjectsOfType()* and give it the type *Target*, which is the script added to the target prefab, and hence exists in all target objects. This function will search the scene for any object that has *Target* attached to it. *FindObjectOfType<Target>()* returns to us an array that contains all targets, and we store this array in *allTargets*.

Now we have to go through all targets stored in the array and test them one by one for possible collisions with our bullet. We use *foreach* loop to go through the elements of the array. The value of *t* changes at the beginning of each iteration of the loop, and takes the value of the next element in the array until it goes through all elements. The first thing to do in each iteration is to check whether the target has already been hit, and ignore it in if this is true (line 17). After that, we find the distance between the bullet and the target using *Vector3.Distance()*. If the distance is less than the "virtual" radius of the target, we count this as a hit (lines 23 and 24), and hence set the *hit* flag in the target to *true* and destroy the bullet (lines 27 and 30). I have mentioned that the radius of the target is virtual, since the target is a cube and therefore doesn't actually have a radius. However, we try to estimate a distance that can approximately simulate a border for the target object. Since the length of each cube edge is one, we multiply it by 0.5 to get the minimal possible distance between the surface of the cube and its center.

Once we have added these two scripts to the sphere that represent the bullet, we can create the prefab of the bullet in a similar way to what we have done for the target. Since we do not need a bullet in the scene at the beginning, you must delete the bullet object from the scene after creating the prefab.

To delete an object from the scene, simple select it from the hierarchy and hit *Delete* on the keyboard.

Now we need to create a prefab for the rocket, in a process that reflects the power of core reusability. First we need a shape to represent the rocking, and this shape is going to be a cube with a scale of (0.1, 0.1, 0.75). The rocket has a behavior similar to the bullet: it moves forward by a constant speed, hits the targets and destroys them. We can give these abilities to the rocket by adding *Projectile* and *TargetHitDestroyer* scripts to it. One additional feature the rocket has is the ability to lock on a target and follow it. To implement this feature, we add a third script to the rocket object. This third script is *TargetFollower*, shown in Listing 20.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class TargetFollower : MonoBehaviour {
5.
6.       //Target we are following
7.       Target currentTarget;
8.
9.       void Start () {
10.           //At the beginning we find the nearest target
11.           Target[] allTargets = FindObjectsOfType<Target>();
12.
13.           //Make sure there are targets in the scene
14.           if(allTargets.Length > 0){
15.                //Assume first target is the nearest
16.                Target nearest = allTargets[0];
17.
18.                //Find the nearest target
19.                foreach(Target t in allTargets){
20.                       //We don't care about targets that have been
21.                       //already hit
22.                       if(!t.hit){
23.                              //Distance between projectile
24.                              //and current target
25.                              float distance =
26.                                     Vector3.Distance(
27.                                           transform.position,
28.                                           t.transform.position);
29.
30.                              //Distance between projectile
31.                              //and nearest target
32.                              float minDistance =
33.                                     Vector3.Distance(
34.                                           transform.position,
35.                                           nearest.transform.position);
36.
37.                              //Update the nearest target if necessary
38.                              if(distance < minDistance){
39.                                     nearest = t;
40.                              }
41.                       }
42.                }
43.
```

```
44.                 //Set current target to nearest target
45.                 currentTarget = nearest;
46.       }
47. }
48.
49.       void Update () {
50.             //Make sure that current target
51.             //has not been already destroyed or hit
52.             if(currentTarget != null && !currentTarget.hit){
53.                   //Have the projectile to look at the target
54.                   transform.LookAt(currentTarget.transform.position);
55.             }
56.       }
57. }
```

Listing 20: Target following script of the rocket

The longest step in *TargetFollower* is the search algorithm we perform in *Start()*. This algorithm gets all targets in the scene, stores them in the array *allTargets*, and searches for the nearest target to the position where the rocket is created. Before searching for the nearest target, it is important to make sure that there are targets in the scene, which we do in line 14 by checking whether *allTargets.Length* is greater than zero. If there are targets in the scene, we take the first one and store it in *nearest*. It is important to realize that arrays have zero-based positions, which means that the first object in the array is located at the position zero. We use *allTargets[0]* o access the first object in the array. Storing the first element in *nearest* means that we consider it the nearest one, then we start to search for a possible closer object.

FindObjectsOfType<Target>() returns all objects in the scene, including hit objects which are currently playing falling animation. However, it does not return the targets which have already been destroyed and removed from the scene. In line 22, we make sure that the object has not yet been hit before considering it a candidate target. If the target has not yet been hit, we find the distance between its position and the position of the rocket and store it in *distance* (lines 24 through 28). Additionally, we find the distance between the rocket and the nearest target and store it in *minDistance* (lines 32 through 35). If *distance* is less than *minDistance*, this means that the current target is closer to the rocket than the nearest object we have so far. Therefore, in this case we set the value of the nearest target to current target (lines 38 through 40). The last step in *Start()* is to store the nearest target that we have found in *currentTarget* (line 45).

In *Update()* function of *TargetFollower*, we make sure that the current target is not null, which means it has a value stored in it. The current can be null when there are not targets in the scene, and therefore no nearest target or current target. After that, we check the current target to test if it has already been hit by another rocket or a bullet. A quick hint regarding && operator in line 52: this operator is called "And", and it requires both operands to be true in order to return a true. However, if the first operand is false, the second is not going to be evaluated. This behavior is necessary in this case, because we cannot check *nearestTarget.hit* if *nearestTarget* it self is null, otherwise we get an error.

By adding *TargetFollower* to the rocket, our prefab becomes ready to be created. So we create the rocket prefab and then delete the rocket object from the scene. Now we have our three prefabs: target, bullet, and rocket, we are ready to write the necessary scripts for the shuttle. Before moving on to the shuttle, we have to complete our scene by adding few targets. For example, you can add two rows of targets in front of the shuttle, by dragging target prefab into the scene several times in the desired positions. You can then set movement speed of first row targets to, say, (3, 0, 0) and the speed of second row targets to (-3, 0, 0). Now you have two rows of targets that move from right to left and from left to right. Now let's add the script that allows us to control shuttle movement, which is *ShuttleControl*, shown in Listing 21.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ShuttleControl : MonoBehaviour {
5.
6.       //Shuttle movement speed
7.       public float speed = 7;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.           //Reads keyboard input and moves
15.           //the shuttle on local x and z axis
16.           if(Input.GetKey(KeyCode.UpArrow)){
17.                transform.Translate(0, 0, speed * Time.deltaTime);
18.           } else if(Input.GetKey(KeyCode.DownArrow)){
19.                transform.Translate(0, 0, -speed * Time.deltaTime);
20.           }
21.
22.           if(Input.GetKey(KeyCode.RightArrow)){
23.                transform.Translate(speed * Time.deltaTime, 0, 0);
24.           } else if(Input.GetKey(KeyCode.LeftArrow)){
25.                transform.Translate(-speed * Time.deltaTime, 0, 0);
26.           }
27.      }
28. }
```

Listing 21: A script for controlling the shuttle

In addition to *ShuttleControl*, we need to add *Wrapper* and *TargetHitDestroyer* to the shuttle. This makes the shuttle wrap from right to left and vice-versa, and if the shuttle hits a target, it is going to be destroyed as well as the target. Two additional scripts are needed: one for shooting bullets, and another one for launching rockets. These scrips are *BulletShooter* shown in Listing 22, and *RocketLauncher* shown in Listing.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class BulletShooter : MonoBehaviour {
5.
6.       //Prefab of the bullet the shuttle shoots
7.       public GameObject bullet;
8.
9.       //How many seconds must pass between two consecutive bullets?
10.      public float timeBetweenBullets = 0.2f;
11.
12.      //When did the shuttle shoot the last bullet?
13.      float lastBulletTime = 0;
14.
15.      void Start () {
16.
17.      }
18.
19.      void Update () {
20.           //We use left control for bullet shooting
21.           if(Input.GetKey(KeyCode.LeftControl)){
22.                //Check if the time between bullets as already passed
23.                if(Time.time - lastBulletTime > timeBetweenBullets){
24.                       //Create new bullet using the prefab.
25.                       //The bullet is at the same position of the
26.                       //shuttle and looks at the same direction of it
27.                       Instantiate(bullet, //object to create
28.                             transform.position, //position of the object
29.                             transform.rotation);//rotation of the object
30.
31.                       //Register the time in which we shot the bullet
32.                       lastBulletTime = Time.time;
33.                }
34.           }
35.      }
36. }
```

Listing 22: Bullet shooting script

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class RocketLauncher : MonoBehaviour {
5.
6.       //Prefab of the rockets the shuttle launches
7.       public GameObject rocket;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.           //We use spacebar (discrete presses) for rocket launching
15.           if(Input.GetKeyDown(KeyCode.Space)){
16.
17.                //How many rockets there are in the scene?
18.                TargetFollower[] rockets =
19.                       FindObjectsOfType<TargetFollower>();
20.
21.                //Do not allow more than one rocket at the same time
22.                if(rockets.Length == 0){
23.                       //Create new rocket at the position of the
24.                       //shuttle and at the same rotation of it
25.                       Instantiate(rocket,
26.                              transform.position, transform.rotation);
27.                }
28.           }
29.      }
30. }
```

Listing 23: Rocket launching script

After attaching *BulletShooter* script to the shuttle, the first thing we have to do is to set the value of *bullet*. This variable is going to be used as a reference to the prefab needed to create new bullets. Recall that we have already prepared this prefab and saved it in our project. So we need now to tell the script which prefab should be the source when making bullet copies. Binding a prefab to a variable in a script is accomplished by dragging the prefab to the field of the variable in the inspector as in Illustration 32.

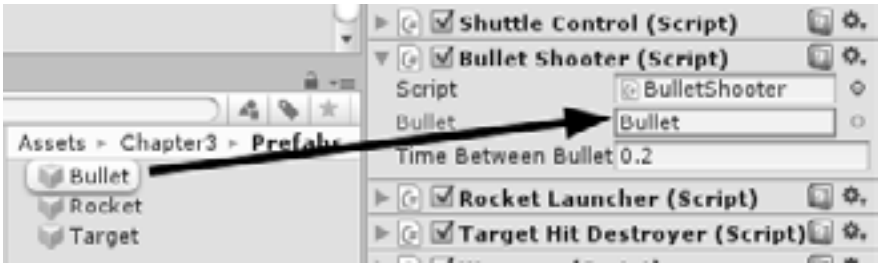


Illustration 32: Binding a prefab to a variable in a script

This script keeps shooting bullets as long as the player is pressing left control key. However, there is a preset time gap between two consecutive bullets. The variable *timeBetweenBullets* determines how many seconds must pass before a new bullet can be shot, and the variable *lastBulletTime* stores the last time of the last shooting. In line 23, we subtract *lastBulletTime* from the current time to get how many seconds passed so far since last shooting. If this time is greater than *timeBetweenBullets*, then we instantiate a new bullet from the prefab. The function *Instantiate()* takes a prefab to make a copy of, a position and a rotation for the new object. We call this function in line 27 and pass to it the bullet prefab through *bullet* variable. The position and rotation we provide to *Instantiate()* are the position and rotation of the shuttle, which makes the bullet get out from the shuttle.

We use a similar technique in *RocketLauncher* script. We have, however, two differences. The first difference is the use of discrete key presses on the space bar as a trigger for rockets, unlike the continuous reading of left control key in *BulletShooter*. The second difference is the limitation on the number of rockets that may exist in the scene simultaneously. In lines 18 and 19 we get an array of all rockets in the scene by finding all objects of type *TargetFollower*. Since this script exists only in rocket objects, the number of elements in the array is exactly the same number of the rockets that are in the scene. If the length of this array is zero, then there are no rockets in the scene and we may instantiate a new one (lines 22 through 27). Illustration 33 shows a screen shot of the demo. The result can also be seen in *scene9* in the accompanying project.

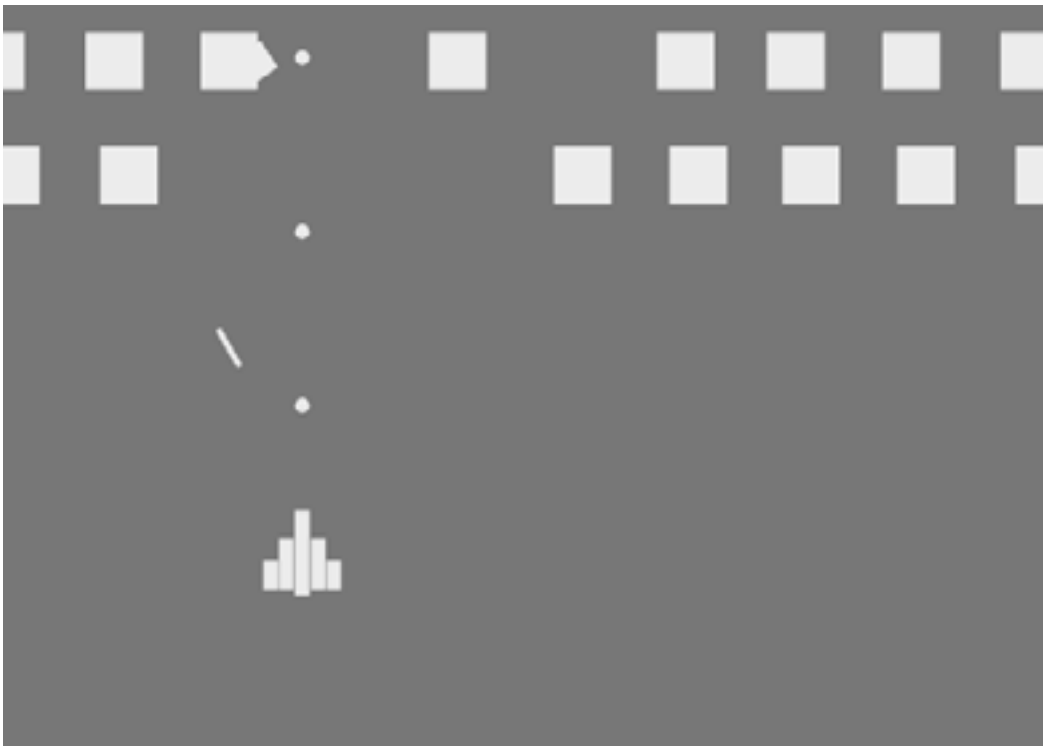


Illustration 33: A screen shot of space shuttle demo

3.2 Collectables

Many computer games depend on spreading collectable items around game world to motivate the player to explore the world or involve in some challenges to get rewards. For example, player can collect coins that can later be used to buy virtual tools or learn new abilities like in many RPG games. In this section we learn how to make such collectables.

Collectables share a common behavior, which is obviously the ability to be collected when someone touches them. This *someone* is not necessarily the character of the player, since it can also be a non-player character (NPC). Therefore, we are going to make a script that marks *collectables*, and another script that marks *collectors*. When any collector hits any collectable, a collection attempt is triggered. This attempt can either succeed or fail depending on many factors as we are going to see. In this section I am going to create a ball that represents the player, and this ball moves and rolls along x and z axes. The camera looks at the ball from above, and follows ball movement on x and z axes. The ball is able to collect coins, and has an inventory box to store the collected coins. Additionally, there are two types of *food* this ball can collect, and each type increases the size of the ball by a specific amount for limited time. To begin, create a ball with scale (1, 1, 1) to represent the player, and a ground plane with scale (10, 1, 10). I assign a glass texture to the ball and a wood ground texture to the ground like in Illustration 34. Let's also position the camera over the ball with a height of 15, and rotate it to look downwards to be able to see the ball.

Illustration 34: Glass ball that serves as player character

To control the ball, we create *BallRoller* script. We also add *ObjectTracker* script to the main camera to make it follow the ball. Listing 24 shows *BallRoller* control script, and Listing 25 shows *ObjectTracker* script for the camera. Remember to assign the ball game object to *objectToTrack* variable in *ObjectTracker* to tell the camera which object it must track.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class BallRoller : MonoBehaviour {
5.
6.       //Movement speed on x and z axes
7.       public float moveSpeed = 5;
8.
9.       //Rolling speed of the ball
10.      public float rollSpeed = 360;
11.
12.      void Start () {
13.
14.      }
15.
16.      void Update () {
17.           //Move along global x axis and roll around global z axis
18.           if(Input.GetKey(KeyCode.UpArrow)){
19.                transform.Translate(0, 0,
20.                       moveSpeed * Time.deltaTime, Space.World);
21.
22.                transform.Rotate(rollSpeed * Time.deltaTime,
23.                                      0, 0, Space.World);
24.           } else if(Input.GetKey(KeyCode.DownArrow)){
25.                transform.Translate(0, 0,
26.                       -moveSpeed * Time.deltaTime, Space.World);
27.
28.                transform.Rotate(-rollSpeed * Time.deltaTime,
29.                                      0, 0, Space.World);
30.           }
31.
32.           //Move along global z axis and roll around global x axis
33.           if(Input.GetKey(KeyCode.RightArrow)){
34.                transform.Translate(moveSpeed * Time.deltaTime,
35.                                      0, 0, Space.World);
36.
37.                transform.Rotate(0, 0,
38.                                 -rollSpeed * Time.deltaTime, Space.World);
39.           } else if(Input.GetKey(KeyCode.LeftArrow)){
40.                transform.Translate(-moveSpeed * Time.deltaTime,
41.                                      0, 0, Space.World);
42.                transform.Rotate(0, 0,
43.                                 rollSpeed * Time.deltaTime, Space.World);
44.           }
45.      }
46. }
```

Listing 24: The script that controls ball movement using arrow keys

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ObjectTracker : MonoBehaviour {
5.
6.       //Object to track
7.       public Transform objectToTrack;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.           //Set (x, z) position to (x, z)
15.           //position of the tracked object
16.           Vector3 newPos = transform.position;
17.           newPos.x = objectToTrack.position.x;
18.           newPos.z = objectToTrack.position.z;
19.           transform.position = newPos;
20.      }
21. }
```

Listing 25: The script that lets the camera track the player

The ball is going to be a collector, which collects coins or other things (collectables) by touching them. Therefore, we need now two additional scripts: *Collectable*, which marks an object as collectable, and *Collector*, which checks for collision with the collectables that exist in the scene. *Collectable* script is shown in Listing 26.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Collectable : MonoBehaviour {
5.
6.       //Distance between the center and
7.       //the external collision surface
8.       public float radius = 0.5f;
9.
10.      void Start () {
11.
12.      }
13.
14.      void Update () {
15.
16.      }
17.
18.           //This function is going to be invoked by the collector
19.           //When it touches this collectable
20.           public void Collect(Collector owner){
21.                //Tell all other scripts in the collectable object
22.                //to run collection logic if they have
23.                SendMessage("Collected",
24.                       owner, SendMessageOptions.RequireReceiver);
25.      }
26. }
```

Listing 26: Script for collectable objects

As you can see, both *Start()* and *Update()* functions are empty. This means that *Collectable* script is passive, and all what it does is to wait for the collector to call its *Collect()* function. The collector is also going to use *radius* value to test collision with the collectable. When *Collect()* function is called, the script sends a message called *Collected* and attaches an object with this message *owner*. In this context, the value of *owner* refers to the collector that called *Collect()* (i.e. the collector that has just collided the collectable). The attachment is important because it tells who should receive this collectable, in case there are multiple collectors in the game.

The question now is: who is going to receive the message *Collected*, which has been sent by *Collectable* script? The answer is: all scripts attached to the same game object of *Collectable*. We are going to see how to receive this message and how to write an appropriate logic to handle it. Therefore, the only job for *Collectable* is to tell other scripts that the object has collided with a collector. Notice that we use *SendMessageOptions.RequireReceiver*, which requires that at least one script receive this message, otherwise an error is raised. We require a receiver for this message since a collectable that does not include any other logic does not make sense.

Lets move now to the other side of the collection process, and have a look at *Collector* script. This script is shown in Listing 27.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Collector : MonoBehaviour {
5.
6.       //Radius of collision with collectables
7.       public float radius = 0.5f;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.           //Get all collectables in the scene
15.           Collectable[] allCollectables =
16.                                FindObjectsOfType<Collectable>();
17.
18.           //Check for collision with collectables
19.           //Use the radii to determine collision distance
20.           foreach(Collectable col in allCollectables){
21.                float distance =
22.                       Vector3.Distance(transform.position,
23.                                                col.transform.position);
24.
25.                //If the distance is less than the sum of the radii
26.                //Then we can try to collect this collectable
27.                if(distance < col.radius + radius){
28.                       //Tell the collectable that this collector
29.                       //is trying to collect it
30.                       col.Collect (this);
31.                 }
32.           }
33.      }
34. }
```

Listing 27: Script for collecting collectable objects

Notice that the function of *Collector* script is very abstract: it checks only whether there are collisions with collectables, and calls *Collect()* function from the colliding collectables. Collision check is performed by comparing the distance between the collector and the collectable with the sum of their virtual radii (line 27). Radii are virtual because the collector and the collectable are not necessarily spherical shapes, but this method is enough to serve the purpose in our case. If a collision is detected, the collector calls *Collect()* function of the collectable, and gives itself as a value for *owner* parameter. As you see, a script can get the value of itself by using the word *this* (line 30).

Now we have the mechanism that can detect a collision between a collector and a collectable. Next step is to determine what should be done after this collision. Theoretically, there is an infinite number of collectables, and each one of these need to be handled differently. For example, coins increase the amount of money the player has, while health portions restore player's health. In our example game, we have two main types of collectables: coins and food. Coins are going to increase the amount of money in the category box, while food increase the size of the ball (player character) with a specific factor for limited time. By increasing the size of the ball, food helps the player to collect coins faster. There are two types of food: green and red, and each one of them has its own factor if size increment as well as time limit. Before going into the details of these collectables, let's have a quick look at Listing 28, which shows *YRotator*, a simple script that rotates an object around the global y axis with specific speed.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class YRotator : MonoBehaviour {
5.
6.       //Rotation speed in degree/second
7.       public float speed = 180;
8.
9.       //Should the angle be randomized at the beginning?
10.      public bool randomStartAngle = true;
11.
12.      void Start () {
13.           if(randomStartAngle){
14.                //Rotate with a random angle between 0 and 180
15.                transform.Rotate(
16.                       0, Random.Range(0, 180), 0, Space.World);
17.           }
18.      }
19.
20.      void Update () {
21.           //Simply rotate around global y
22.           transform.Rotate(
23.                0, speed * Time.deltaTime, 0, Space.World);
24.      }
25. }
```

Listing 28: A script to rotate objects around the global y axis

It is possible to specify *randomStartAngle* to avoid having a lot of object that rotate similarly and hence have a nice randomness in the scene. Notice the use of *Random.Range()* function, which can be used along with lower and upper limits to generate random numbers in between. For example, we use it here to get a random angle at the beginning of the rotation, which lies between 0 and 180 degrees.

All collectables in our scene are going to have *Collectable* and *YRotator* script. For instance, you can say that we are going to use rotation as a sign to tell the player that this object can be collected. Lets begin with the coin, which can be made of a cylinder with a scale of (1, 0.02, 1), and we can put a golden texture on it to give the feeling of a real coin. It is also a good idea to add a point light as a child to the coin. We are going to give this light a yellow color, and position it in the center of the coin. It is strongly recommended that you create a prefab for the coin, since we are going to need a large number of them in the scene. Illustration 35 shows how our coin is going to look like.

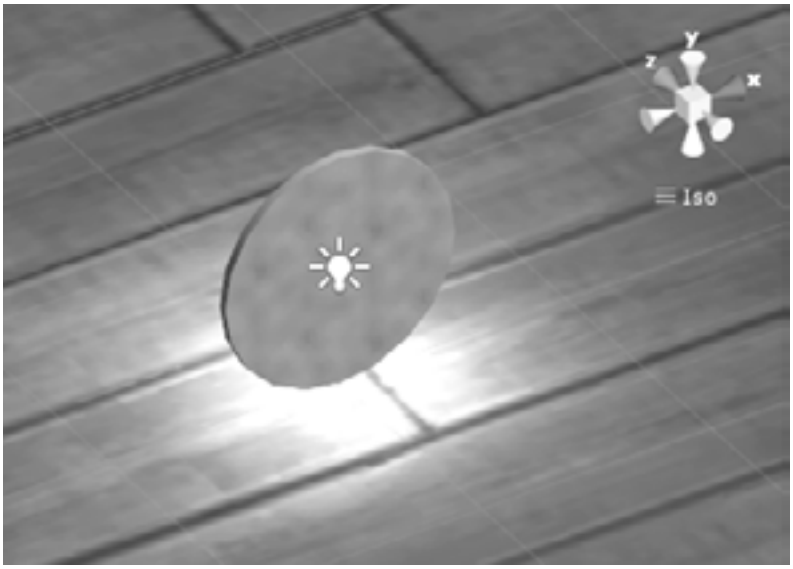


Illustration 35: The coin we are going to use

The coin is now collectable, which means that collisions with collector can be detected. Additionally, it has the y-rotation feature, so it is going to rotate in its position around the global y. We need now to specify happens when a collector tries to collect this coin. Obviously, it is going to increase the amount of money the collector has in his inventory box. Therefore, I am going to begin with *InventoryBox* script shown in Listing 29.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class InventoryBox : MonoBehaviour {
5.
6.        //How much money does the player have?
7.        public int money = 0;
8.
9.        void Start () {
10.
11.       }
12.
13.       void Update () {
14.
15.       }
16. }
```

Listing 29: The inventory box for the collector

This is a simple script we have to attach to the ball, in order to make it able to collect money (coins). This script can be extended to include whatever inventory you may think of. However, for our case we need only one variable, which is *money*. The importance of this script is the ability to gives to the collector. If there is a collector who does not have an inventory box, its collision with coins is going to be ignored, since coins require inventory box to be collected into. This can be useful, for example, if you want to have NPCs that can collect many things (weapons, power-ups), but not coins. Now we move back to our coin and add to it the script *Coin*, which specifies the behavior of a coin. This script is shown in Listing 30.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Coin : MonoBehaviour {
5.
6.       //This value is going to be added to the
7.       //money of the inventory box upon collection
8.       public int coinValue = 1;
9.
10.      void Start () {
11.
12.      }
13.
14.      void Update () {
15.
16.      }
17.
18.      //We declare this function to receive Collectable's
19.      //command to run collection logic
20.      public void Collected(Collector owner){
21.           //Check if the collector has an inventory box
22.           InventoryBox box = owner.GetComponent<InventoryBox>();
23.           if(box != null){
24.                //Inventory box exists,
25.                //so increase money by coin value
26.                box.money += coinValue;
27.
28.                //Done, destroy coin object
29.                Destroy(gameObject);
30.           }
31.      }
32. }
```

Listing 30: The script of a collectable coin

We can have coins with different amounts, by setting the value of *coinValue*. Just like *Collectable*, *Start()* and *Update()* functions are empty. The new function we add to this script is *Collected*, which takes a value of type *Collector*. This owner is in fact the collector who is trying to collect this coin. Back to *Collectable* script (Listing 26 in page 76), recall that *Collectable* sends a message to other scripts to inform them about a collision with a collector. That message is called *Collected* and includes an attachment, which is the collector. By declaring the method called *Collected()* and giving it the parameter *owner*, we specify *Coin* script as a receiver of this message. Consequently, when *Collected* message is received, *Collected()* function is executed. What does this function do is to get the inventory box of the collector (*owner*). By calling *owner.GetComponent<InventoryBox>()*, we try to get a reference to the inventory box, and store this reference in *box*. If no inventory box found, the value of *box* becomes *null*, and hence nothing is done. However, if the inventory box exists, the amount of money inside that box is increased by *coinValue*. Finally, the coin game object is destroyed.

Illustration 36 illustrates the interactions between *Collector*, *Collectable*, *Coin*, and *CategotyBox*, along with all interactions among these scripts.

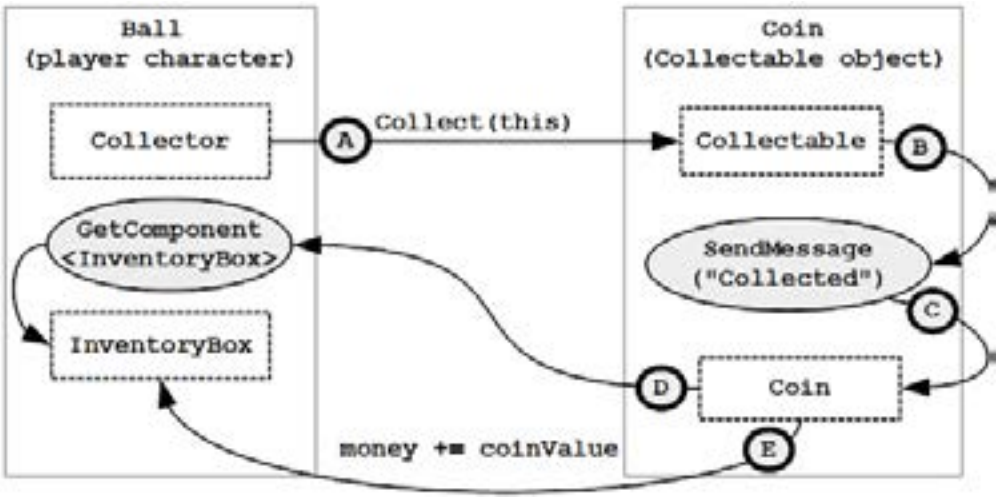


Illustration 36: Coin collection mechanism with the interactions among all involved scripts

Diagram in Illustration 36 summarizes the programmatic steps of coin collection. In step A the collector calls *Collect()* function of *Collectable* script attached to the coin, giving itself as the owner of what is to going to be collected. In step B, the collectable sends the message *Collected* to all scripts attached to the coin. In step C, *Coin* script receives the message by executing its own function *Collected()*, which eventually performs steps D and E. *Coin* script tries in step D to find *InventoryBox* script inside the collector. Recall that the variable *owner* inside *Collected()* function refers to the collector. Step E (money amount increment) is executed in case the inventory box is found, otherwise this step is not executed, and hence the coin is neither collected nor destroyed.

A similar process takes place in case of food collection. However, collection logic as well as the effect on the collector are different. We are going to have two types of food, and both of them are going to have the same structure. Additionally, we are going to create a separate prefab for each type of them. Taking food shall increase the size of the ball for a limited time. The two types of food we are going to create vary in these two values. Listing 31 shows *SizeChanger* script, which handles food collectables. At the other end, Listing 32 shows *Food* script, which we are going to add to our food objects. Food collectables are also going to have *Collectable* and *YRotator* scripts as well.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class SizeChanger : MonoBehaviour {
5.
6.       //current size of the object
7.       float currentSize = 0;
8.
9.       //Reference to Collector script
10.      Collector col;
11.
12.      void Start () {
13.      col = GetComponent<Collector>();
14.      }
15.
16.      void Update () {
17.
18.      }
19.
20.      //Called by Food script to try to increase the size
21.      public bool IncreaseSize(float amount, float duration){
22.           //Size can be increased only if it is zero
23.           if(currentSize == 0){
24.                //Set increment amount and increase the scale
25.                currentSize = amount;
26.                transform.localScale = Vector3.one * currentSize;
27.                //Call DecreaseSpeed() function after duration
28.                Invoke("DecreaseSize", duration);
29.
30.                //If there is a collector, increase its radius
31.                if(col != null){
32.                col.radius = col.radius * currentSize;
33.                }
34.                //Return true = food has been taken
35.                return true;
36.           }
37.           //Return false = food has NOT been taken
38.           return false;
39.      }
40.
41.      //Resets size to one
42.      public void DecreaseSize(){
43.           transform.localScale = Vector3.one;
44.           //If there is a collector, restore its original radius
45.           if(col != null){
46.                col.radius = col.radius / currentSize;
47.           }
48.           //Set current size back to zero
49.           currentSize = 0;
50.      }
51. }
```

Listing 31: A script that reacts to food collection by changing ball size

```
52. using UnityEngine;
53. using System.Collections;
54.
55. public class Food : MonoBehaviour {
56.
57.      //Size increment for the food taker
58.      public float sizeIncrementAmount = 2;
59.
60.      //How long can this food increase size (seconds)?
61.      public float incrementDuration = 5;
62.
63.      void Start () {
64.
65.      }
66.
67.      void Update () {
68.
69.      }
70.
71.      //We declare this function to receive Collectable's
72.      //command to run collection logic
73.      public void Collected(Collector owner){
74.           //Collector must have a size changer to take food
75.           SizeChanger changer = owner.GetComponent<SizeChanger>();
76.           if(changer != null){
77.                //Size changer found, try to take food
78.                bool canTake =
79.                       changer.IncreaseSize(
80.                             sizeIncrementAmount, incrementDuration);
81.                //Has the food been taken?
82.                if(canTake){
83.                       //canTake = true, so the food has been taken
84.                       Destroy(gameObject);
85.                }
86.           }
87.      }
88. }
```

Listing 32: A script for collectable food

SizeChanger in Listing 31 script begins with looking for a *Collector* attached to the same game object. If the collector exists, it is stored in *col* variable. There are two major functions in *SizeChanger*: *IncreaseSize()* and *DecreaseSize()*. *IncreaseSize()* is called by *Food* script, when the collector hits a collectable that has the script *Food* attached to it. *IncreaseSize()* begins with checking the value of *currentSize* variable; if the value is not zero, the function returns *false*, which means no food can be taken right now. However, if *currentSize* is zero, the value of *amount* parameter is stored in *currentSize*, and the scale of the ball (collector) is increased with the same value. In line 28, we use *Invoke()* function to setup the execution of another function later on. In this case, we specify *DecreaseSize()* function by writing its name, and we set the delay to the value of *duration* parameter. Finally, if *col* variable is not null (i.e. *Collector* script is attached to the game object), we multiply its radius by the value of *currentSize*, so the radius becomes suitable for the new size of the ball.

When the specified delay time for calling *DecreaseSize()* is over, Unity calls the function. The job of this function is to reset all variables to their default values; so the scale is set back to *Vector3.one*, the radius of the collector is divided by *currentSize*, and finally *currentSize* is set back to zero. It is important to notice that as long as *cuurentSize* is not zero, this means that the effect of an already taken food is still active, and this case no additional food collectable can be taken. This rule is enforced by *Food* script.

Food script in Listing 32 handles *Collected* message sent by *Collectable* like what we have seen in *Coin* script (Listing 30 in page 80). As we have already seen, *Coin* script depends on *InventoryBox*, so if the latter is missing, hitting a coin does not have any effect. Similarly, *Food* script depends on *SizeChanger*, since its job is to increase the size of the collector. Therefore, the first step in *Collect()* function is to make sure that the collector trying to collect this food has a *SizeChanger*, otherwise nothing happens. If *SizeChanger* exists, we declare the variable *canTake*, to test whether *SizeChanger* is in a state that allows it to take the food. Recalling *IncreaseSize()* function in *SizeChanger*, it returns *false* if the size is already increased. This returned value is stored in *canTake* to be checked in the next step (line 31). A *false* value of *canTake* means that this food has not been taken by *SizeChanger*, so we just ignore the hit and the food game object is not destroyed. However, if *canTake* is *true*, this means that the food has been taken by *SizeChanegr* and the size of the collector has been incareased. In this case, the food object is destroyed.

When applying the effect of *Food* on the collector, we use the values of *sizeIncrementAmount* and *incrementDuration*. Since these variables are public, their values can be set from the inspector. This makes it possible to make the two types of food we want to have. Now we can create two game objects, say, cubes with different sizes and textures, and attach to each one *YRotator*, *Collectable*, and *Food* scripts. The only difference regarding the scripts is going to be in the values of *sizeIncrementAmount* and *incrementDuration* variables. So let's make the "green food" with size increment of 2 and a duration of 7.5 seconds, and the "red food" with size increment of 3.5 and a duration of 5 seconds. These types of food are shown in Illustration 37. The final result can be seen in *scene10* in the accompanying project.

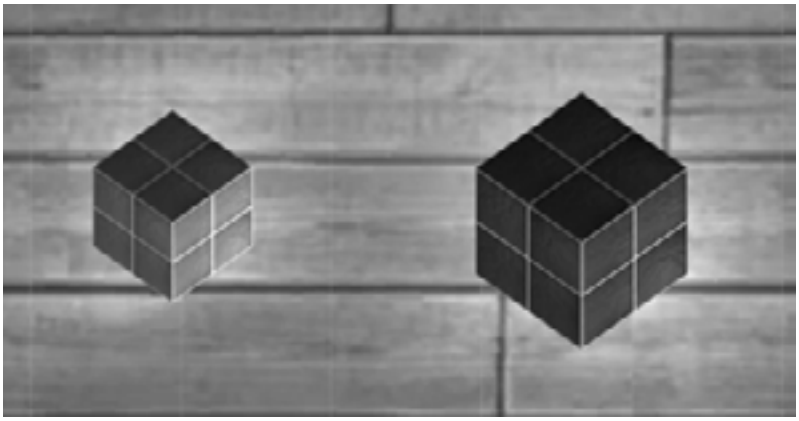


Illustration 37: Red food object (right) and green food object

3.3 Holding and releasing objects

It is necessary sometimes to give the player the ability to move objects around the scene, so he can stack some boxes to access a high place, or remove some obstacles off the way, and so on. Moving objects can be accomplished in different ways. For example, the player can push objects by moving towards them, or he can use some super powers or devices to hold objects (recall, for example, the gravity gun in Half-life 2). In this section, we are going to make use of relations between objects to implement a mechanism that allows the player to hold some objects, move while holding them, and release/discard these objects at any position.

In this section, I am going to reuse first person input system we have developed in section 2.4. What we are going to do is to make the player able to hold the boxes and release them by pressing E key. The scene we are going to use can be found in *scene9* in the accompanying project. We need two scripts to apply holding/releasing mechanism. The first script is *Holdable*, which we are going to add to all objects that can be hold by the player. This script is shown in Listing 33, and it is an empty script that has only a radius for checking distance with the holder. In our scene, we have to add this script to the boxes. It is a better idea always to make a prefab of a holdable box and add copies of it to the scene.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Holdable : MonoBehaviour {
5.
6.       //Radius of the holdable object
7.       public float radius = 1.5f;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.
15.      }
16. }
```

Listing 33: Holdable script

Most of the job is going to be in *Holder* script, which is shown in Listing 34. This script need to be attached to the cylinder that represents the player.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Holder : MonoBehaviour {
5.
6.       //Radius of the holder
7.       public float radius = 0.5f;
8.
9.       Holdable objectInHand;
10.
11.      void Start () {
12.
13.      }
14.
15.      void Update () {
16.           if(Input.GetKeyDown(KeyCode.E)){
17.                //If there is no object in hand,
18.                //look for one and try to hold it
19.                if(objectInHand == null){
20.                     //Get all holdables in the scene
21.                     Holdable[] allHoldables =
22.                          FindObjectsOfType<Holdable>();
23.                     foreach(Holdable holdable in allHoldables){
24.                          //Find distance between
25.                          //holder and holdable
26.                          float distance =
27.                               Vector3.Distance(
28.                                         transform.position,
29.                                         holdable.transform.position);
```

```
30.
31.                                 //Holdable must be close enough
32.                                 bool close =
33.                                       distance < radius + holdable.radius;
34.
35.                                 //Player must be facing the holdable
36.                                 Vector3 dVector;
37.                                 dVector = holdable.transform.position
38.                                                   - transform.position;
39.
40.                                 float ang =
41.                                       Vector3.Angle(dVector,
42.                                             transform.forward);
43.
44.                                 if(close && ang < 90){
45.                                       //Now we can hold it
46.                                       //1. Set it as object in hands
47.                                       objectInHand = holdable;
48.
49.                                       //2. Add it as a child to move with
50.                                       //the holdable
51.                                       holdable.transform.parent = transform;
52.
53.                                       return;
54.                                 }
55.
56.                           }
57.                     } else {
58.                           //There is an object already in hands
59.                           //Now we have to release it
60.                           objectInHand.transform.parent = null;
61.                           objectInHand = null;
62.                     }
63.               }
64.         }
65. }
```

Listing 34: Holder script

The idea of the script is simple: when the player presses E key, we make sure that there is no object in hand. If this is true, we try to find a suitable object to hold. If such object is found, the holder holds it. On the other hand, if there is already an object in hand, this object is released. The variable *objectInHand* stores the object that is currently hold. If the value of this variable is *null*, it means that no object is in hand (line 19).

To hold an object, we perform a number of steps. We begin by getting all holdable objects in the scene and iterate over them (lines 21 through 23). For each object, we compare the distance between it and the holdable, and if the distance is less than the sum of radii, the value of *close* variable becomes *true* (lines 26 through 33). The *true* value of *close* means that the object is close enough to be hold. However, there is another condition to check. It is necessary for the holder to face the holdable object before being able to hold it. This condition can be checked by measuring the angle between holder's looking direction (*transform.forward*) and the straight line between the position of the holder and the position of the holdable (lines 36 through 42). Vector math tells us that we have to subtract the position of the holder from the position of the holdable, in order to get a vector that represents the line between these two objects. If the angle between these two vectors is less than 90, we consider this as facing (line 44).

After checking all relevant condition, it is time to do the actual holding of the holdable object. This step is fairly simple. Firstly, we have to store the holdable we found in *objectInHands* variable. Since the value of this variable is not *null* anymore, no other object can be hold, and pressing E again is going to release it. Secondly, we set the parent of the holdable transform to be the holder itself. By doing this, we ensure that the holdable moves with the holder wherever it goes, and rotates with it as well (lines 44 through 51). In line 53, we use *return* to stop the execution of *Update()*. This step enhances the performance by avoiding unnecessary check of the rest of holdable objects, since we have already found what we need. Lines 57 through 62 apply when the player hits E key while holding an object. In this case, *objectInHand* is released by setting its parent to *null*, so it is not a child of the holder anymore. Finally, it is necessary to set the value of *objectInHand* to *null*, in order to free the space for holding another object in the future. The final result can be seen in *scene11* in the accompanying project.

3.4 Triggers and usable objects

In addition to object holding, players can perform another actions that manipulate the scene. For example, the player can activate or deactivate some devices, such as electrical lights. This activation or deactivation is called triggering, because a player performs an action that triggers another action. When the player switches light on or off, he deals in fact with the power switch, and the switch makes the effect. The switch in this case is called the trigger, since it is responsible for performing the action. In the same example, the player is the activator of the trigger, and the one who decided to perform the action.

In this section we are going to look at a general solution for usable triggers. Therefore, the code might seem complex at the beginning, but in the long run it provides a portable pattern that can be used in almost every situation. The scene we are going to use in this section is a bit more complex than previous scenes, and it going to have a number of objects that are necessary to illustrate the complete picture. Let's begin with Illustration 38, which shows how does our scene look like.

Illustration 38: The scene we are going to use to illustrate triggers and usable objects

In the scene you see in Illustration 38, we have a first person character that is going to represent the player. Additionally, we have (from left to right): a point light with an object above it to represent an electrical light, a standing control panel to control the fan on the wall, and an electrical switch on the wall to turn the light on or off. The usable objects in this scene are the fan and the light, and we are going to be able to use them through the triggers (the switch and the control panel).

So let's begin with the trigger that can be used by the player to use objects. Listing 35 shows *SwitchableTrigger* script. I used the term s*witchable*, since there are other types of non-switchable triggers, such as time triggers and hit triggers that activate automatically when the player touches them.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class SwitchableTrigger : MonoBehaviour {
5.
6.       //The trigger switches between these states
7.       public TriggerState[] states;
8.
9.       //Index of the current state
10.      public int currentState = 0;
11.
12.      //Minimum distance to intarct with this trigger
13.       public float activationDistance = 3;
14.
15.      //Last time the current state changed
16.      float lastSwitchTime = 0;
17.
18.      void Start () {
19.
20.      }
21.
22.      void Update () {
23.
24.      }
25.
26.       //Tries to switch the state of the trigger
27.      //Returns true if switching was successful
28.      public bool SwitchState(){
29.           //If states array is empty we do nothing
30.           if(states.Length == 0){
31.                       return false;
32.           }
33.
34.           //Get the current state
35.           TriggerState current = states[currentState];
36.
37.           //Check if the rest time of current state is over
38.           if(Time.time - lastSwitchTime > current.restTime){
39.                //It is over, we can switch to next state
40.                currentState += 1;
41.
42.                //If we are in the last state, return to the first
43.                 if(currentState == states.Length){
44.                       currentState = 0;
45.                 }
46.
47.                //Get the new state
48.                TriggerState newState = states[currentState];
49.
50.                //Send all messages of the new state
```

```
51.                 foreach(TriggerMessage message
52.                                                 in newState.messagesToSend){
53.                         //Get the receiver of the message
54.                         GameObject sendTo = message.messageReceiver;
55.                         //Get the name of the message
56.                         string messageName = message.messageName;
57.                         //Send the message
58.                         sendTo.SendMessage(messageName);
59.                 }
60.
61.                 //Finally, record switching time
62.                 lastSwitchTime = Time.time;
63.                 return true;
64.             } else {
65.                 return false;
66.             }
67.       }
68. }
69.
70. //Small structure to represent state
71. [System.Serializable]
72. public class TriggerState{
73.       public float restTime;
74.       public TriggerMessage[] messagesToSend;
75. }
76.
77. //A structure to represent messages sent by trigger
78. [System.Serializable]
79. public class TriggerMessage{
80.       public GameObject messageReceiver;
81.       public string messageName;
82.  }
```

Listing 35: Switchable trigger script

Before going into the details of *SwitchableTrigger* itself, let's jump to lines 72 through 76 and 79 through 83. In these lines we have two small classes that are a bit different than scripts we are used to. Firstly, notice that they do not extend *MonoBhaviour*, and, secondly, they have the [*System.Serializable*] before class declaration. These classes are going to be used as boxes that combine a number of variables. For example, if I declare a variable of type *TriggerState*, this variable includes two variable inside it: *restTime* and the array *messagesToSend*. The importance of [*System.Serializable*] is it makes variables of this class visible in the inspector, just like all variable types we have been using up to now.

TriggerState is going to be used to specify how many states a switch can have. The number of states is most of time 2 (on/off), but sometimes we need more than two states. Each state has a *restTime*, expressed in the number of seconds. During rest time, the trigger is locked and its state cannot be switched until rest time is over. This is useful for tasks that take time, such as opening an electrical door. Each state has an array of *TriggerMessage*, the class which is going to be used to specify what happens at every state change. Each one of these messages has a name and a receiver, to which the message is going to be sent. The receiver can be any game object in the scene, and it must have a script that receives the message. In other words, at least one script attached to the receiver must have a function that has the same name of the message.

Back to *SwitchableTrigget*, this script has a number of interesting variables. First variable is an array of states (called *states*). Each time the user activates the trigger, it tries to switch to the next state in the array, and if is already in the last state, it goes back to the first state. Second variable is a public index to specify the currently active state. This index refers to one of the states in *states* array, and is increased at each switching. Finally, we have the variable *activationDistance*, to specify the minimum distance between the trigger and the player that allows the player to use it. In addition to these public variables, *lastSwitchTime* stores the last time this trigger has been used, to be able to lock the trigger for the rest time of the current state.

SwitchState() function is called whenever the player tries to use the trigger. It returns *true* if the state has been successfully switched, or *false* otherwise. One reason that leads to unsuccessful switching attempt is that the rest time of the current state has not yet passed. If it is possible to switch the state, the value of *currentState* is incremented by 1, or set back to zero if the current state is the last one in *states* array. After setting the new state, the trigger iterates over all *TriggerMessage* values stored in the list of messages of the new state, and sends each message once to the specified receiver (lines 51 through 59). The last step before returning *true* is to record the current time as *lastSwitchTime*, to be able to compute the rest time of the current state. Illustration 39 shows the mechanism of manipulating multiple objects through multiple states of a single trigger.

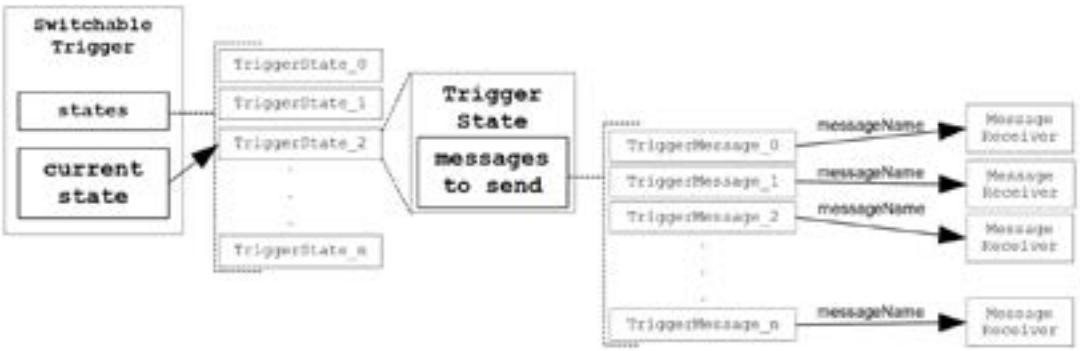


Illustration 39: Triggering mechanism: when the state changes, the new state sends all messages stored in messagesToSend array

The idea will be more clear when we discuss the examples. The first example is a switch that controls an electrical light. The switch as well as the light has two states: on and off. When the player switches the trigger, two things happen: the light is changed from on to off or vice-versa, and the switch button is moved upwards or downwards. This means that we have two states for the switch trigger, and each state has two messages to send: one message to the light, and another message to the switch object. Next step is to write two scrips that are capable of receiving the messages and performing actions based on them. The first script is *LightControl* shown in Listing 36. This script can receive two messages: *SwitchOn* and *SwitchOff*.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class LightControl : MonoBehaviour {
5.
6.       //The light we are going to control
7.       Light toControl;
8.
9.       void Start () {
10.           toControl = GetComponent<Light>();
11.      }
12.
13.      void Update () {
14.
15.      }
16.
17.      //Receives "SwitchOn" message
18.      public void SwitchOn(){
19.           toControl.enabled = true;
20.      }
21.
22.      //Receives "SwitchOff" message
23.      public void SwitchOff(){
24.           toControl.enabled = false;
25.      }
26. }
```

Listing 36: The script that controls the light based on received messages

This script must be attached to a light object, and it starts by finding the light component and storing it in *toControl*. When it receives *SwitchOn* message, it executes *SwitchOn()* function and enables the attached light component. The opposite happens when it receives *SwitchOff* message, by disabling the controlled light component. The other script that receives messages is *ZFlipper*. This script rotates the object to which it is attached 180 degrees around object's local z axis. This rotation is performed when the message *Flip* is received. But why we are going to use this script? Illustration 40 shows the object we are going to use as switch. It is clear that when this object is rotated 180 degrees around its local z axis, the texture flips upside down, resulting in an effect similar to switch movement.

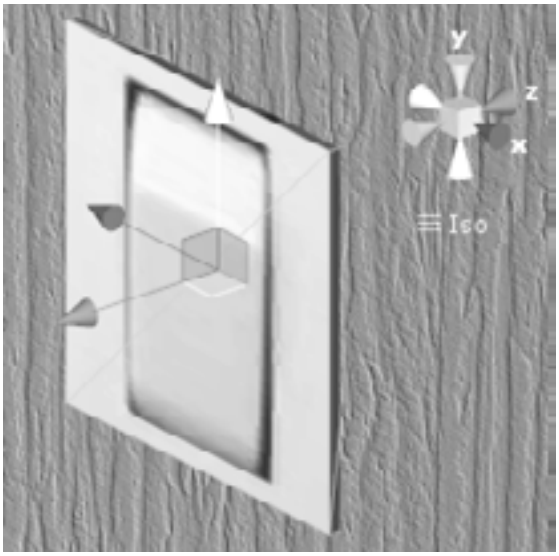


Illustration 40: Power switch object

ZFlipper script is shown in Listing 37.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ZFlipper : MonoBehaviour {
5.
6.       void Start () {
7.
8.       }
9.
10.      void Update () {
11.
12.      }
13.      //Rotates 180 degree around local z axis
14.      public void Flip(){
15.           transform.Rotate(0, 0, 180);
16.      }
17. }
```

Listing 37: A simple script to flip switch object

The function *Flip()* is executed when the message *Flip* is received. Now we have the trigger that is capable of sending messages and usable objects that can receive messages. We need to attach *SwitcahbleTrigger* script to the switch object and set the appropriate number of states and messages per state. Illustration 41 shows this script in the inspector after preparing it for use.

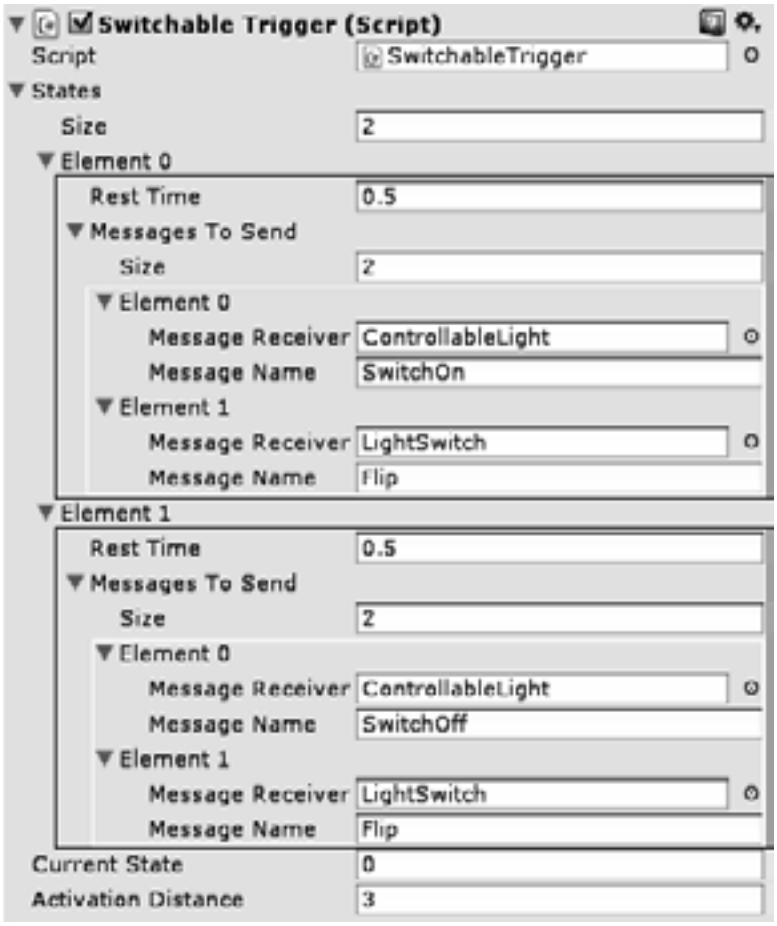


Illustration 41: Light switch trigger completely configured

Illustration 41 shows that the trigger has two states. The first state sends two messages when activated: *SwitchOn* message to *ControllableLight*, and *Flip* message to *LightSwitch*. *ControllableLight* is the light game object that has the script *LightControl*, and *LightControl* is the switch object itself. This means that the trigger sends *Flip* message to itself. The second state also sends two messages when activated. The difference is the message it sends to the light, which is *SwitchOff* this time. At the beginning, the light is switched on, therefore we set *currentState* to 0, which is the first state. To complete the functionality, we add *TriggerSwitcher* script to the cylinder that represents the player. This script reads E key from the keyboard and activates any switchable trigger nearby. This script is shown in Listing 38.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class TriggerSwitcher : MonoBehaviour {
5.
6.       void Start () {
7.
8.       }
9.
10.      void Update () {
11.           if(Input.GetKeyDown(KeyCode.E)){
12.                //Find all switchable triggers in the scene
13.                SwitchableTrigger[] allST =
14.                       FindObjectsOfType<SwitchableTrigger>();
15.
16.                //search for suitable trigger
17.                //Suitable = near + facing
18.                foreach(SwitchableTrigger st in allST){
19.                       float dist =
20.                            Vector3.Distance(transform.position,
21.                                                 st.transform.position);
22.
23.                       //If distance less than activationDistance,
24.                       //of the trigger, then it is close.
25.                       if(dist < st.activationDistance){
26.                            Vector3 distVector =
27.                                 st.transform.position
28.                                       - transform.position;
29.
30.                            float angle =
31.                                 Vector3.Angle(distVector,
32.                                                 transform.forward);
33.                            //If angle < 90, it is facing
34.                            if(angle < 90){
35.                                 //facing trigger = we can use it
36.                                 st.SwitchState();
37.                            }
38.                       }
39.                }
40.           }
41.      }
42. }
```

Listing 38: The script that allows the player to use switchable triggers

When the player presses E key, the script searches for all switchable triggers in the scene, and finds the distance between the player and each one. If the distance is less than activation distance set in the trigger, and the player is facing the trigger with an angle less than 90, *SwitchState()* function of the trigger is called. After adding *TriggerSwitcher* script to the cylinder, light switching functionality becomes ready, so you can test it. You can also see the result int *scene12* in the accompanying project.

To solidify the idea, I am going to illustrate a second example with multiple states. Recall the scene in Illustration 38, there is a fan on the wall and a switch standing in the middle of the room. This switch is going to be used to turn the fan on and change its speed. Let's begin with the script of the fan which is *ControllableFan* shown in Listing. This script sets 3 different speeds for the fan, as well as off state.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ControllableFan : MonoBehaviour {
5.
6.       public float speed1 = 20;
7.       public float speed2 = 40;
8.       public float speed3 = 60;
9.
10.      float currentSpeed = 0;
11.
12.      void Start () {
13.
14.      }
15.
16.      void Update () {
17.           transform.Rotate(0, currentSpeed * Time.deltaTime, 0);
18.      }
19.
20.      public void SetSpeed1(){
21.           currentSpeed = speed1;
22.      }
23.
24.      public void SetSpeed2(){
25.           currentSpeed = speed2;
26.      }
27.
28.      public void SetSpeed3(){
29.           currentSpeed = speed3;
30.      }
31.
32.      public void SwitchOff(){
33.           currentSpeed = 0;
34.      }
35. }
```

Listing 39: Fan script

The script has three functions that change the value of *currentSpeed*, which is the variable that directly affects the rotation speed of the fan. Additionally, the script has *SwitchOff()* function which sets the speed to zero, hence stops the rotation. To control the fan, we attach *SwitchableTrigger* script to the switch in the middle of the room. We need to setup the script as in Illustration 42.

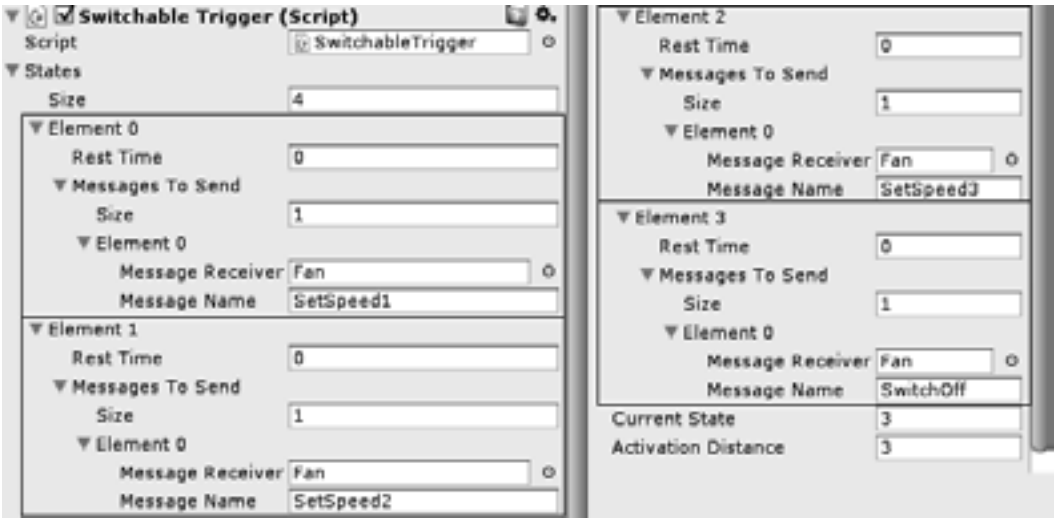


Illustration 42: Configured switchable trigger for the fan switch

This time we have four different states for the switch. The first three states send messages that change fan speed, while the fourth one switches the fan off. Since the fan is off at the beginning of the game, the current state is set to 3, which is the index of the off state in the switch. If the player switches this trigger, the state switches to the first, and sends *SetSpeed1()* message to the fan object. You can see the final result in *scene12* in the accompanying project.

Exercises

1. Change the hit animation in *Target* script (Listing 15 in page 63) so that the target the bullet hits moves fast to the front instead of rotation.
2. Change *TargetFollower* script (Listing 20 in page 69) so that it locks the rocket on the farthest target from the shuttle instead of the nearest.
3. Add a new feature to *ShuttleControl* script (Listing 21 in page 70) to allow the player to rotate the shuttle using horizontal mouse movement. You must read the displacement of the mouse pointer and rotate the shuttle around the y axis.
4. Attach the script *BulletShooter* (Listing 22 in page 71) to the target prefab to make the targets able to shoot bullets on the shuttle. You have to add a new script that checks for collision between the bullet and the shuttle. Therefore you are going to need a new bullet prefab. Finally, you have to make sure that all targets look backwards (i.e. positive direction of their z axes point towards the shuttle).
5. Add a third type of collectables to our ball example in section 3.2, which has a limited time effect on the collector. This effect is doubling the value of the collected coins during the effect period, so if the collector collects this *Doubler*, and then collects a coin with value 1, then the variable *money* in *InventoryBox* must be increased by 2. You can select the ball object from the hierarchy during play, in order to be able to observe the value of *money* all the time.
6. Make a switchable trigger that cycles the color of a light between three values: red, yellow, and green. You can refer to triggers in *scene12* in the accompanying project to understand the idea.

4 Physics Simulation

Introduction

Up to now, we have been dealing with semi-static objects that do not move or rotate unless controlled by some script. In this chapter, we introduce physics simulation, an important function of any game engine. Physics simulation gives the objects realistic behavior and hence helps us making better, more fun games.

After completing this chapter you should be able to:

- Use basic physics functions such as gravity and collision detection
- Make physics-enabled vehicles (cars)
- Create physical player character
- Use ray casting to simulate shooting
- Make physics projectiles
- Simulate explosions and destruction
- Create breakable objects

4.1 Gravity and Collision Detection

In previous chapters, we were able to simulate gravity by setting a ground reference and moving the towards it as the time goes. In this chapter we make advantage of built-in physics simulator in Unity. To apply physical characteristics to an object, we need to add two components: *Collider* and *Rigid Body*. All basic shapes in Unity have colliders by default. For example; if you add a sphere object, you can notice that it has a *Sphere Collider* component attached to it as in Illustration 43.

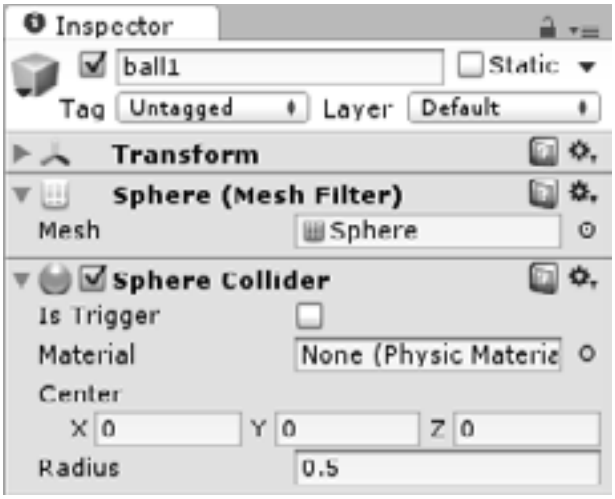


Illustration 43: The Collider component

The shape of the collider is independent from the shape of the object itself. Consequently, it is possible to have an object that looks as cube but behaves physically as sphere. It is also possible to position the center of the collider away from the center of the shape, or scale its size to make it larger or smaller than the visible object. These modifications can be made by setting the values of *Center* and *Radius* variables of the collider. Once an object has a collider, it becomes a solid entity that collides with other objects in the scene. However, collisions between objects can be detected and resolved only when the objects are under the control of the physics simulator. This fact justifies why colliders have had no effect in our previous examples, even they existed in all objects we have made.

The second important component for physics simulation is the rigid body, which is shown in Illustration 44. When this component is added to the object, it makes it become physically active. This component has a collection of interesting properties, which we are going to discuss soon.

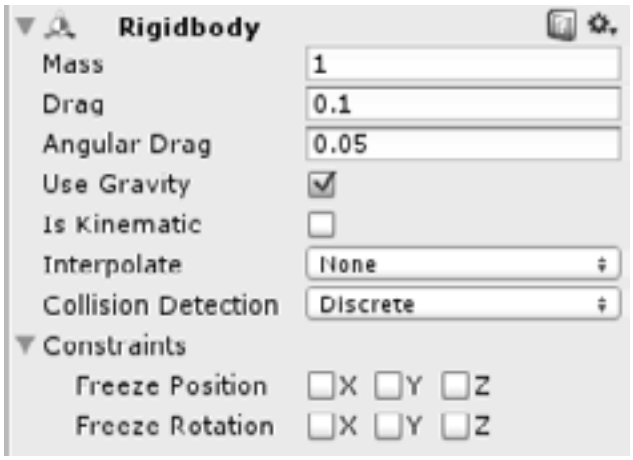


Illustration 44: The Rigid Body component

To illustrate the role each property plays in the rigid body, make a simple scene like the one in Illustration 45. All objects in the scene have colliders by default, and we are going to add rigid body components to the four balls; so that they become affected by gravity and other forces.

To add a rigid body to an object; first select the object from the hierarchy, then go to Component > Physics > Rigid Body.

Illustration 45: A simple scene to demonstrate the properties of the rigid body

The first important property of the rigid body is *Drag*, which is the amount of air resistance applied to the object while moving. Larger air resistance leads to faster lose of speed for the object. To test the effect of the drag, set the *drag* value for the balls to 0.1, 1.5, 0.2 and 2.5 starting from the top most ball. If you run the game now, all balls fall down, and each one of them moves along its track. You can notice that the balls with smaller drag values move faster and fall from the edge of the track, while the balls with larger drag values stop moving before reaching the end of the track. This result is shown in Illustration 46, and it can also be seen in *scene13* in the accompanying project.

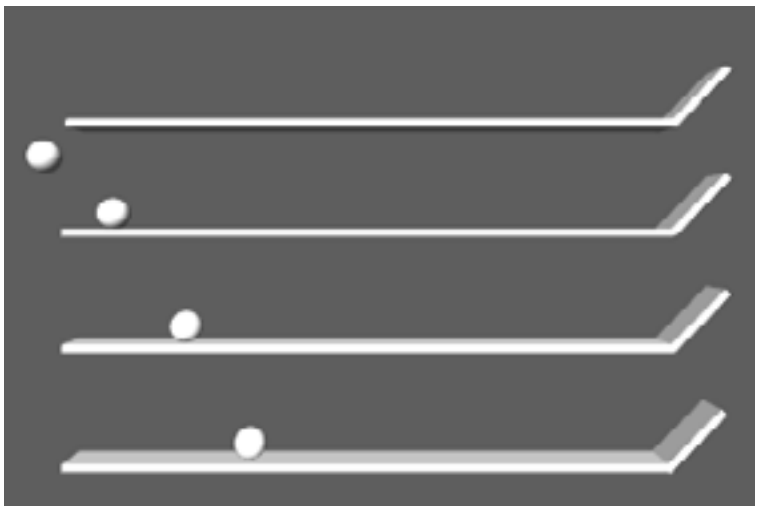


Illustration 46: The effect of the drag on the movement of the objects: upper balls have lower drag values

The second important property of the rigid body is its mass. The mass of the rigid body determines how *strong* is the gravity force applied to it. However, it does not affect the *velocity* in which the object moves downwards. The next example is show in Listing 47. The scene consists of four cubes with a mass of 0.25 (250 gram) for each one, and four balls with masses of 10, 7.5, 5 and 1, starting from the top most ball.

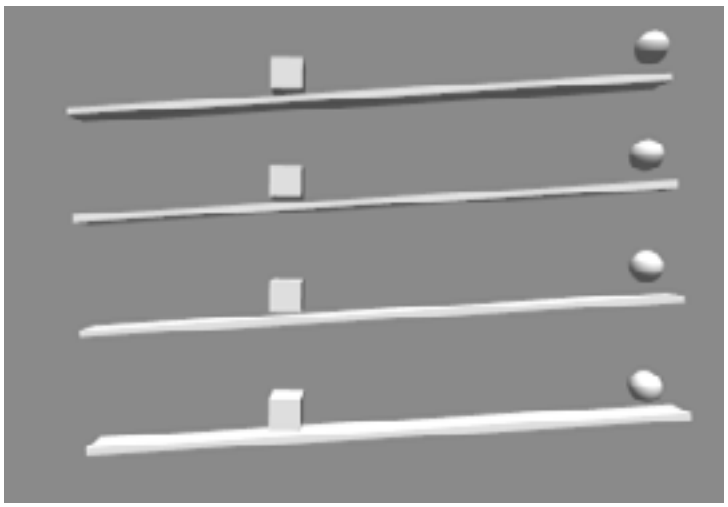


Illustration 47: A scene to demonstrate the effect of the mass of the rigid body

When the game runs, we expect each one of the balls to move along the track, hit the cube, and push it to some distance before both of them stop (or fall off the edge of the track). However, we want restrict the movement of the balls as well as the cubes to x and y axes only, to prevent the ball from falling from the side of the track. This is possible by setting the constraints of the rigid body components attached to the balls as well as the cubes. What we need to do is to freeze the movement on the z axis, and freeze the rotation around the y axis. The freeze of the rotation is important for the cubes, in order to keep their orientation when they are hit by the balls. The constraints of all rigid bodies of balls and cubes should look like Illustration 48.

Illustration 48: Movement and rotation constraints of the rigid body

The four balls have different masses, but they have the same drag. Therefore, when the game runs all balls start to move with equal velocity along the track. The effect of the mass appears when a ball collides with the cube. The ball with greater mass is going to push the cube for longer distance before both of them stop due to friction force. The result you are going to see is similar to Illustration 49. This demo can be viewed in *scene14* in the accompanying project.

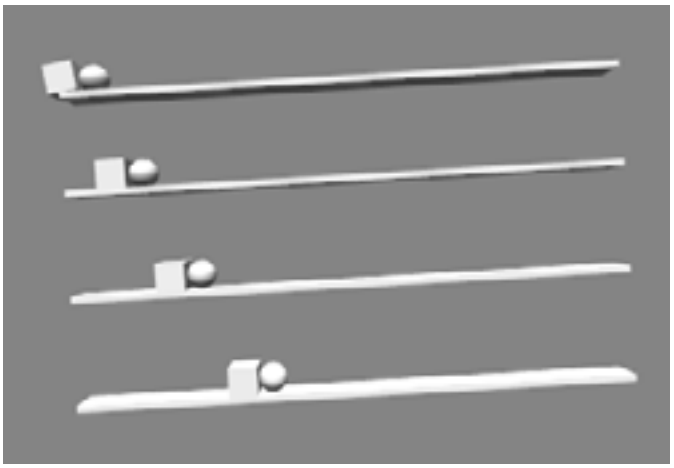


Illustration 49: The effect of the mass on the rigid body: upper balls have greater masses, and the cubes have equal masses

The physics simulator detects collisions between objects and resolves them physically by moving and rotating objects in a manner that would be exposed by similar objects in the real world. In our part, we might need to know when these objects collide with each other, in order to perform some actions in code based on the collisions. For example, when a rocket hits a target, it must be destroyed. Our next example consists of four balls and two cubes shaped as planes as in Illustration 50. The balls have rigid bodies attached, while the planes do not.

Illustration 50: A scene to demonstrate collision detection and handling in code

Before jumping to code, we have to set *Is Trigger* property in the collider of the upper plane to *true*. Trigger colliders are different in terms of collision resolution. The physics simulator tells us when an object collides with a trigger, but it does not handle this collision physically. In other words, when a ball hits the upper plane it is going to simply keep falling; as the trigger does not block the movement of other objects. Now we want to write a script in which we are going to handle the collisions between a ball and the other objects. The script is shown in Listing 40, and we are going to attach it to all balls in the scene.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ColorBall : MonoBehaviour {
5.
6.       //Color of the ball
7.       public Color color;
8.
9.       void Start () {
10.           renderer.material.color = color;
11.      }
12.
13.      void Update () {
14.
15.      }
16.
17.      //To handle collision with solid colliders
18.      void OnCollisionEnter(Collision col){
19.           //access the colliding object and change its color
20.           col.collider.renderer.material.color = color;
21.      }
22.
23.      //TO handle collision with triggers
24.      void OnTriggerEnter(Collider col){
25.           col.renderer.material.color = color;
26.      }
27. }
```

Listing 40: A script for handling collisions

This script allows us to select a color from the inspector, and sets this color to the material of the object when the game starts. For example, we can set the colors for the four balls to red, yellow, green, and blue (from left to right). We have two functions to handle two different types of collisions. The first one is *OnCollisionEnter()*, which is called whenever the ball hits a solid collider. This function is called only once upon the first contact between the two colliding objects. When *OnCollisionEnter()* is called, it is provided with a reference to collision data through *col* variable. This variable allows us to access the other object involved in the collision by calling *col.collider*. In this case, we simple access the renderer of the other object and change its color to match the color of the ball. Similarly, *OnTriggerEnter()* function is called when the ball hits a trigger collider (in our case the upper non-blocking plane). One difference regarding *OnTriggerEnter()* is the parameter provided to it. Since there are no detailed collision data, the variable *col* refers to the collider of the other object directly. Therefore, we are able to directly access the renderer and change its color.

Before starting the game, we need to vary the falling speed of the four balls. One possible method is to set a different drag to each one of them. When the game starts the balls start to fall down because of gravity. Whenever a ball hits a plane, it changes its color to match the color specified in *ColorBall* script. Illustration 51 shows a screen shot during game run. The final result can be seen in *scene15* in the accompanying project.

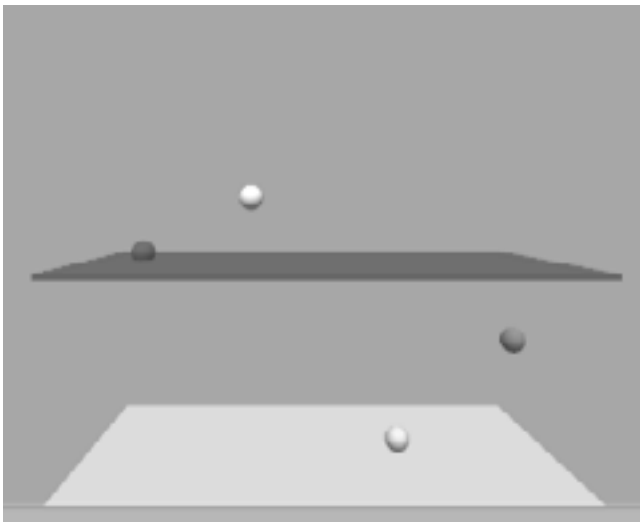


Illustration 51: Handling collisions and changing the colors of planes accordingly

4.2 Physical Vehicles

In section 2.6, we implemented a simple car input system. In that system, we have simulated everything manually: acceleration, braking, and steering. In this section we are going to use physics simulator to construct a more realistic vehicle. This vehicle is going to have four wheels with suspension springs. First step is to construct the vehicle like in Illustration 52.

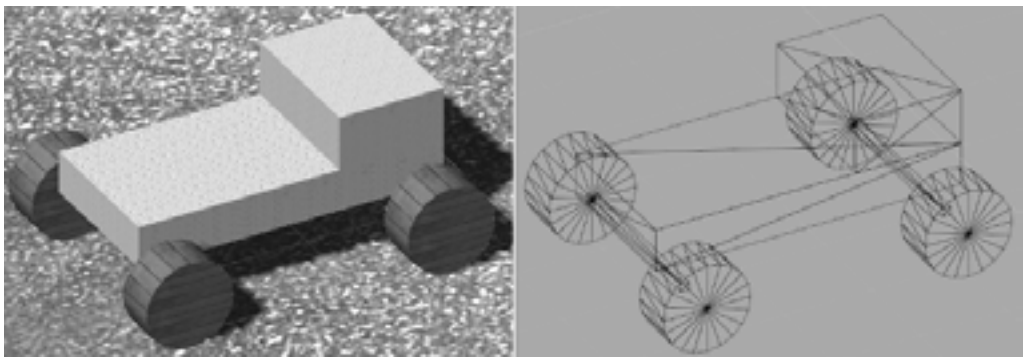


Illustration 52: A simple vehicle constructed using basic shapes

Front and back axes can be made using cubes, as well as the cabin and the main body. On the other hand, cylinders can be used to create the four wheels. After constructing the vehicle, we have to remove all colliders from all parts, except the front and the back axes. This last step is necessary to create a custom collision shape that helps our vehicle to behave better, as we are going to see soon. Finally, create an empty object and add all these parts as its children, so that it looks like Illustration 53.

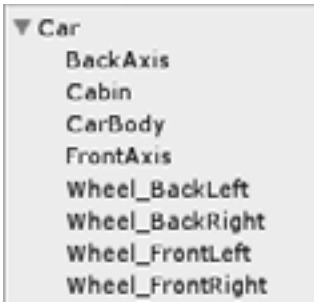


Illustration 53: Vehicle parts added as children to an empty game object

Next step is to add a collider to the root object of the car. I am going to use a capsule collider, because it prevents the car from flipping on its back and force it to roll until it gets back on its wheels. The capsule must extend from the back to the front of the car. Vertically, the capsule must be raised so its bottom side touches the bottom side of the car body. Another important modification we have to do is to increase the size of the colliders of front and back axes. These colliders must be extended along the x axis until the ends of the colliders reach the outer side of the wheels. The purpose of this collider extension is to prevent the wheel collider from sinking into the ground accidentally (specially when the vehicle jumps and lands on the wheels of one side), which may make the vehicle stuck with a wheel under the ground. Illustration 54 shows the correct size and position of the collider.

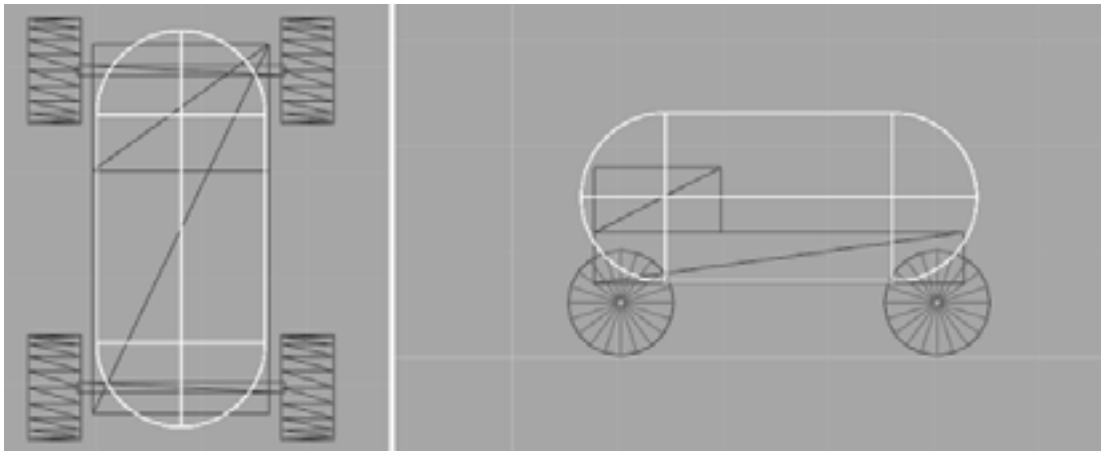


Illustration 54: Capsule collider added to the vehicle

The second component we have to add to the root object of our vehicle is a rigid body. For out vehicle we need a reasonable mass such as 1500 kilograms. Additionally, we need to set the drag to a relatively high value, in order to give the feel of a heavy object that needs great force to move and stop after short time when there is no force to move it. Therefore, we need a value such as 0.25 for the drag, and a value of 0.75 for the angular drag. By default, Unity sets the center of mass of an object at the origin of the local space of the object. However, sometimes we need to have a different center of mass. In case of our vehicle, we need to set the center of mass a bit lower, in order to prevent the vehicle from flipping easily when it turns right or left. To perform that, add an empty game object as a child to the game object of the vehicle, name it *CenterOfMass*, and position it at (0, -0.5, 0.2). We are going to specify this position as the center of the mass of our vehicle later on using a script.

Now we need to have realistic wheels for our vehicle. These wheels are going to be the core element in the simulation, since they are responsible for applying motor torque, braking, steering, and spring suspension. A unique property of Unity, which does not necessarily apply to other game engines, is the separation between the physical *wheel collider* component and the visual wheel object. Therefore, we are going to add an empty object for each wheel collider, instead of adding these collider directly to the wheels we have made.

The best practice when using wheel colliders is to add an empty game object as a child to the vehicle, and add all empty objects of the wheel colliders as children to it. Wheel colliders in Unity have zero width, therefore, we need two colliders for each one of the relatively wide wheels of our vehicle. One of these colliders must be positioned near the outer side of the wheel, and the other near the inner side. To summarize, we need an empty game object, and let's call it *WheelColliders*, and additional eight empty game objects added to it, as in Listing 55.

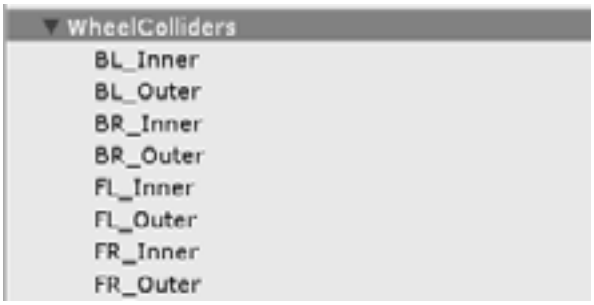


Illustration 55: Empty objects to hold wheel colliders. The names of the objects describe the position of the collider

After adding a *wheel collider* component to each one of the empty objects, we need to set their properties. Refer to Listing 56 for the appropriate values for wheel colliders.

You can select multiple objects and add the same component to each one of them at once. You may also set the properties of the component while multiple objects are selected, so that your changes are applied to all selected objects.

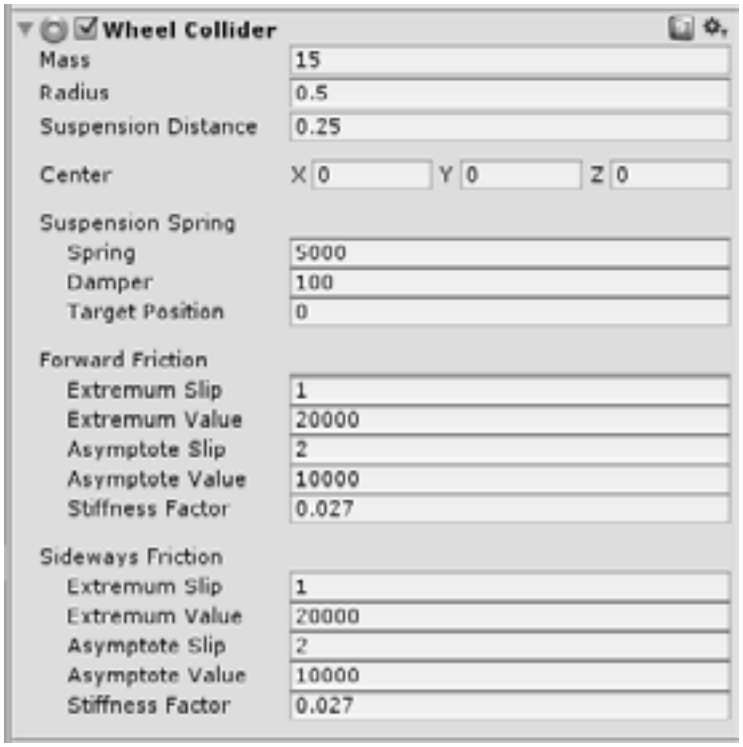


Illustration 56: Setting the properties of the wheel colliders

There is a bunch of interesting properties to deal with: the *mass* of the wheel is set to 15, assuming that each wheel wights 30 kilograms (remember that we have inner and outer collider for each wheel), the *radius* can be adjusted visually by calibrating its value until the collider has the same size of the visual wheel, and the *suspension distance* is the length of the suspension spring when it is fully extended and in our case it is 25 centimeters. *Suspension spring*, *forward friction*, and *sideways friction* categories contain a number of values that have to do with wheel damping and friction. Sometimes it takes a long time to calibrate these values, so I am not going to discuss their details. However, it is useful to read about them in Unity documentation or other sources on the internet; in order to learn how to configure them accurately to achieve the desired result. Finally, we need to position the colliders correctly, like in Illustration 57.

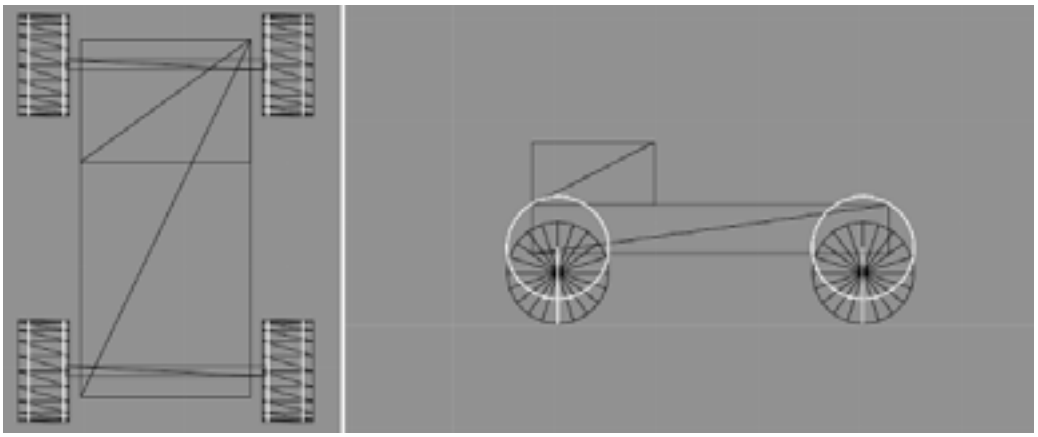


Illustration 57: The correct positioning of the wheel colliders in the vehicle. The colliders are shown in white color

The vertical line of the collider in Illustration 57 represents the spring suspension distance for each wheel collider. On the other hand, the circle shows the position of the collider when the spring is completely pressed. Therefore, we must position the colliders so that the lower end of the line is at the same level of the lowest part of the visual wheel. Before moving on, make sure that the complete hierarchy of your vehicle matches the hierarchy in Illustration 58.

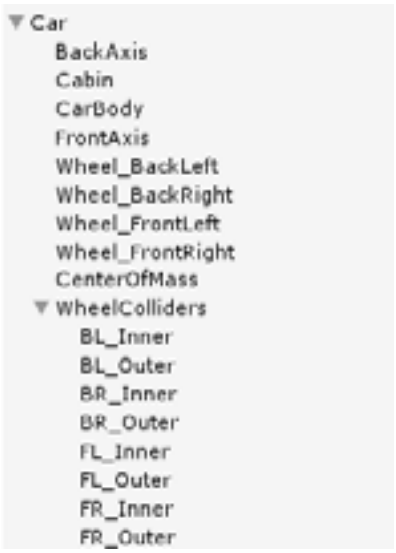


Illustration 58: The complete hierarchy of the vehicle

Our vehicle is now ready to be controlled, and therefore we need a number of scripts that work together to give us an acceptable look and feel of a real car. First and major script is *PhysicsCarDriver*, shown in Listings 41 through 43, which is going to provide basic functions of a controllable car. These functions are independent from user input, which makes the script general to a degree that allows the car to be controlled through AI driver. I have separated this script over multiple listings since it is relatively long. Listing 41 shows the variables we need to control the vehicle.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PhysicsCarDriver : MonoBehaviour {
5.
6.       //All colliders of front wheels
7.       public WheelCollider[] frontWheels;
8.
9.       //All colliders of back wheels
10.      public WheelCollider[] backWheels;
11.
12.      //Car's center of mass
13.      public Transform centerOfMass;
14.
15.      //Max torque of the motor
16.      public float maxMotorTorque = 9500;
17.
18.      //Braking power
19.      public float brakesTorque = 7500;
20.
21.      //Angle to rotate wheels when steering
22.      public float maxSteeringAngle = 20;
23.
24.      //steering rotation speed in degrees per second
25.      public float steeringSpeed = 30;
26.
27.      //maximum car speed in km/h
28.      public float maxSpeed = 250;
29.
30.      //maximum speed moving reverse
31.      public float maxReverseSpeed = 20;
32.
33.      //current steering position
34.      float currensSteering = 0;
35.
36.      //maximum car speed in rpm
37.      float maxRPM, maxReverseRPM;
38.
39.      //Input flags
40.      bool accelerateForward,
41.           accelerateBackwards,
42.           brake, steerRight, steerLeft;
43.
```

Listing 41: Variables declarations for *PhysicsCarDriver* script

First of all we have two arrays of type *WheelCollider*, in order to reference the colliders of the front and the back wheels of our vehicle. We need references to all colliders, since they are the place where we control vehicle movement. Separating front and back wheels is necessary as the steering is going to be applied to front wheels only. The next variable is *centerOfMass*, which is going to store a reference to *CenterOfMass* empty object we have created earlier. The variables *maxMotorTorque* and *brakesTorque* represent the magnitude of the force used to accelerate and decelerate the wheels. To control steering, we use *maxSteeringAngle* and *steeringSpeed*. These two variables are going to be applied to front wheels only, since we do not usually steer the back wheels. The last two public variables of the script are *maxSpeed* and *maxReverseSpeed*, which set the speed limits of our vehicle when driving forward or backwards.

In addition to the public variables, we have *currentSteering*, which is used to store the current steering angle of the front wheels. We have also *maxRPM* and *maxReverseRPM*, and we are going to use these two variables to represent *maxSpeed* and *maxReverseSpeed* in terms of rotations per minute. Having the speed in such unit is necessary, since wheel collider uses this unit to express the speed. Finally, we have a set of flags that store the current control state of the vehicle. If there is a command from the controller (player or AI) to accelerate forward, then *accelerateForward* variable is set to *true*, otherwise it is going to be *false*. Similarly, the other four flags represent the states of their relative commands. The next part of the script is shown in Listing 42, which contains *Start()* and *FixedUpdate()* functions.

```
44.      void Start () {
45.           //Convert max speed to rpm
46.           maxRPM =
47.                KmphToRPM(frontWheels[0], maxSpeed);
48.           maxReverseRPM =
49.                KmphToRPM(frontWheels[0], maxReverseSpeed);
50.
51.           //Set the center of mass for better car turning
52.           rigidbody.centerOfMass =
53.                centerOfMass.localPosition;
54.      }
55.
56.      //We use fixed update for physics
57.      void FixedUpdate () {
58.           //Update acceleration
59.           if(accelerateForward){
60.                foreach(WheelCollider wheel in frontWheels){
61.                     UpdateWheelTorque(wheel, maxMotorTorque);
62.                }
63.
64.                foreach(WheelCollider wheel in backWheels){
65.                     UpdateWheelTorque(wheel, maxMotorTorque);
66.                }
67.                accelerateForward = false;
68.           } else if(accelerateBackwards){
69.
70.                foreach(WheelCollider wheel in frontWheels){
71.                     UpdateWheelTorque(wheel, -maxMotorTorque);
72.                }
73.
74.                foreach(WheelCollider wheel in backWheels){
75.                     UpdateWheelTorque(wheel, -maxMotorTorque);
76.                }
77.                accelerateBackwards = false;
78.           } else {
79.                foreach(WheelCollider wheel in frontWheels){
80.                     UpdateWheelTorque(wheel, 0);
81.                }
82.
83.                foreach(WheelCollider wheel in backWheels){
84.                     UpdateWheelTorque(wheel, 0);
85.                }
86.           }
87.
88.           //Update steering
89.           if(steerRight){
90.                UpdateSteering(steeringSpeed * Time.deltaTime);
91.                steerRight = false;
92.           } else if(steerLeft){
93.                UpdateSteering(-steeringSpeed * Time.deltaTime);
94.                steerLeft = false;
95.           } else {
```

```
96.                 UpdateSteering(0);
97.           }
98.
99.           //Update brakes
100.          if(brake){
101.                foreach(WheelCollider wheel in frontWheels){
102.                      wheel.brakeTorque = brakesTorque;
103.                }
104.
105.                foreach(WheelCollider wheel in backWheels){
106.                      wheel.brakeTorque = brakesTorque;
107.                }
108.                brake = false;
109.          } else {
110.                foreach(WheelCollider wheel in frontWheels){
111.                      wheel.brakeTorque = 0;
112.                }
113.
114.                foreach(WheelCollider wheel in backWheels){
115.                      wheel.brakeTorque = 0;
116.                }
117.          }
118.     }
119.
```

Listing 42: *Start()* and *FixedUpdate()* functions of *PhysicsCarDriver* script

In *Start()* function, we first convert *maxSpeed* and *maxReverseSpeed* from km/h to RPM. The conversion is performed by *KmphToRPM()* function, which we are going to discuss in details shortly. The converted values are stored in *maxRPM* and *maxReverseRPM* variables. The second important thing to do in *Start()* is to change the center of mass of the rigid body, so it takes the new position from the local position of *centerOfMass*.

After the initialization we move to the update function. This time we deal with a new variation of update, which is *FixedUpdate()*. This function is guaranteed by Unity to give the same *deltaTime* at each iteration; so it is used for tasks that depend on accurate timing such as physics simulation and collision detection. Therefore, we perform all tasks related to car driving inside *FixedUpdate()*. In lines 59 through 86, we check *accelerateForward* and *accelerateBackwards* flags. If the value of either flag is *true*, we apply the max motor torque to front and back wheels. Notice that we apply the torque through *UpdateWheelTorque()* function, which is responsible for checking RPM limits before applying the torque, as we are going to see soon. After applying the torque we reset the corresponding flag to *false*. If both *accelerateForward* and *accelerateBackwards* are *false*, this means that there is no command to move the car. Consequently, we apply a torque of zero to all wheels.

The same mechanism is used with steering in lines 89 through 97: if *steerRight* or *steerLeft* flag is set to *true*, we apply the corresponding steering by calling *UpdateSteering()* function. However, if both flags are *false*, we apply a zero steering to the front wheels. The details of *UpdateSteering()* functions are going to be covered shortly. The last section of *FixedUpdate()* between lines 101 and 117 handles braking input. If braking flag is *true*, *brakesTorque* is applied to all wheels, otherwise a brake torque of zero is applied. A torque can be applied to a wheel collider through *motorTorque* variable. The negative value means that we want the wheel to spin counter clockwise, hence moving the vehicle backwards. Unlike the case of applying motor torque, brake torque does not need to check any conditions before being applied to the wheels. This sounds logical if you recognize the fact that nothing bad happens when pressing the brakes pedal of a stopped car. The last part of *PhysicsCarDriver* is shown in Listing 43, and it covers all other functions of the script.

```
120.     //Drive car forward
121.     public void AccelerateForward(){
122.          accelerateForward = true;
123.          accelerateBackwards = false;
124.     }
125.
126.     //Drive backwards
127.     public void AccelerateBackwards(){
128.          accelerateBackwards = true;
129.          accelerateForward = false;
130.     }
131.
132.     //Turn steering wheel to right
133.     public void SteerRight(){
134.          steerRight = true;
135.          steerLeft = false;
136.     }
137.
138.     //Turn steering wheel to left
139.     public void SteerLeft(){
140.          steerLeft = true;
141.          steerRight = false;
142.     }
143.
144.     //Apply braking to all wheels
145.     public void Brake(){
146.          brake = true;
147.     }
148.
149.     //Applies torque to the wheel and checks RPM limits
150.     void UpdateWheelTorque(WheelCollider wheel, float torque){
151.          wheel.motorTorque = torque;
152.          if(wheel.rpm > maxRPM || wheel.rpm < -maxReverseRPM){
153.               wheel.motorTorque = 0;
154.          }
155.     }
156.
```

```
157.     //Updates steering angle
158.     void UpdateSteering(float amount){
159.          if(amount != 0){
160.               currensSteering += amount;
161.          } else {
162.               //Steering is released,
163.               //return steering to straight
164.               //steering dead zone is
165.               //between -3 and 3 degrees
166.               if(currensSteering > 3){
167.                      currensSteering -=
168.                                steeringSpeed * Time.deltaTime;
169.               } else if(currensSteering < -3){
170.                      currensSteering +=
171.                                steeringSpeed * Time.deltaTime;
172.               } else {
173.                      currensSteering = 0;
174.               }
175.          }
176.          //Apply max and min steering angles
177.          if(currensSteering > maxSteeringAngle){
178.               currensSteering = maxSteeringAngle;
179.          }
180.
181.          if(currensSteering < -maxSteeringAngle){
182.          currensSteering = -maxSteeringAngle;
183.          }
184.          //Apply steering angle to front wheels only
185.          foreach(WheelCollider wheel in frontWheels){
186.               wheel.steerAngle = currensSteering;
187.          }
188.     }
189.
190.     //Converts Km/h tp RPM based on
191.     //the radius of provided wheel
192.     float KmphToRPM(WheelCollider wheel, float speed){
193.          //Meters per hour
194.          float mph = speed * 1000;
195.          //Meters per minute
196.          float mpm = mph / 60;
197.          return mpm / (wheel.radius * 2 * Mathf.PI);
198.     }
199.   }
```

Listing 43: Control and other functions of *PhysicsCarDriver* script

AccelerateForward() is a public function that can be used by other scripts to set *accelerateForward* flag. This function protects the script from a contradictory input by setting *accelerateBackwards* to *false*. The same mechanism is used by *AccelerateBackwards()*, *SteerRight()*, *SteerLeft()* and *Brake()* functions. *UpdateWheelTorque()* function takes a wheel collider and a torque amount to apply to it. After applying the toque, it checks the new speed of the wheel in RPM. If the speed is greater than *maxRPM* or less than *-maxReverseRPM*, a zero torque is applied to prevent the vehicle from exceeding its speed limits.

UpdateSteering() function takes a float value in degrees, and adds it to *currentSteering*. If the passed value is zero, the function returns the steering to the straight position by using *steeringSpeed*. The steering dead zone is set between -3 and 3 degrees, so if the steering value is within these limits, it is going to be set instantly to zero (straight). After setting the new value of *currentSteering*, the function checks the limits by comparing *currentSteering* with *maxSteeringAngle* and *-maxSteeringAngle*. Finally, the value of *cuurentSteering* is stored in *steerAngle* member of all wheel colliders in *frontWheels*.

The last function we are going to discuss in this script is *KmphToRPM()*. This function takes two parameters: a wheel collider and a speed value expressed in km/h. The first step is to convert *speed* from km/h to meter/minute and store the result in *mpm* variable. The second step is to convert the speed from meter/minute to RPM. However, this conversion depends on the circumference of the wheel. The circumference tells us how many meters the wheel moves during one complete rotation. Therefore, we divide *mpm* by the circumference to get the speed in RPM and return it. After adding this script to the root object of the vehicle, we are ready to move to our next script.

Now we need another script that enables the player to provide his input to control the vehicle. This script is going to read input from the keyboard and invoke the corresponding functions from *PhysicsCarDriver*. Listing 44 shows *KeyboardCarController* script, which handles player input.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class KeyboardCarController : MonoBehaviour {
5.
6.       //Reference to car we are going to drive
7.       PhysicsCarDriver driver;
8.
9.       void Start () {
10.           //Get the attached car driver
11.           driver = GetComponent<PhysicsCarDriver>();
12.      }
13.
14.      void Update () {
15.           //Use up and down arrows for acceleration
16.           if(Input.GetKey(KeyCode.UpArrow)){
17.                driver.AccelerateForward();
18.           } else if(Input.GetKey(KeyCode.DownArrow)){
19.                driver.AccelerateBackwards();
20.           }
21.
22.           //Use right and left arrows for steering
23.           if(Input.GetKey(KeyCode.RightArrow)){
24.                driver.SteerRight();
25.           } else if(Input.GetKey(KeyCode.LeftArrow)){
26.                driver.SteerLeft();
27.           }
28.
29.           //Use space for braking
30.           if(Input.GetKey(KeyCode.Space)){
31.                driver.Brake();
32.           }
33.      }
34. }
```

Listing 44: A script to handle user input that controls the vehicle

This script is fairly simple; all it has to do is to read the state of keyboard keys and call the matching function from *PhysicsCarDriver* attached to the same object (root of the vehicle). The last script we are going to add to the vehicle is *CarSpeedMeasure*, which measures the current speed of the car in km/h and prints it out for us in Unity's console. This script is shown in Listing 45.

The output of the console can be see by clicking the lower left corner of Unity's main window. Even if the console window is closed, the last output line is always shown at the lower left corner.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class CarSpeedMeasure : MonoBehaviour {
5.
6.       public WheelCollider wheel;
7.
8.       void Start () {
9.
10.      }
11.
12.      void Update () {
13.           //Print the speed in the console
14.           print (GetCarSpeed());
15.      }
16.
17.      //Converts RPM to Km/h based on
18.      //the radius of the wheel
19.      float GetCarSpeed(){
20.           //Meters per minute
21.           float mpm = wheel.rpm * wheel.radius * 2 * Mathf.PI;
22.           //Meters per hour
23.           float mph = mpm * 60;
24.           //Kilometers per hour
25.           float kmph = mph / 1000;
26.           return kmph;
27.      }
28. }
```

Listing 45: A script to measure current vehicle speed in km/h

Notice that the script needs a reference to one of the wheel colliders, since the measured speed is based on the current RPM of the wheels. Once again we use the circumference of the wheel, in order to calculate the distance traveled every complete rotation. Recall that we already have a camera script for car racing games, which is *CarCamera* (Listing 13). You can attach this script to the main camera, and set the root of the vehicle as the car to be followed by the camera. Now you have a vehicle that can be controlled using physics simulator, and you are ready to take a ride. Do not forget to add a ground before running the game, in addition to some obstacles like humps; in order to test how they effect the vehicle when driving over them.

Now we are going to perform some cosmetic enhancements to our vehicle. The vehicle is already functional and behaves as it should, but it does not reflect its state visually. First of all, we need to be able to see the wheels spinning as the vehicle moves. Additionally, we need to visualize the steering angle by turning the front wheels left or right. Finally, we have to visualize the effect of suspension springs by moving the wheels up and down relative to the car body. The common thing between these three visual enhancements is the fact that they are all applied to the visual wheels of the car. Therefore, we are going to write a single script, *CarWheelAnimator*, and let it do the job for us. This script is shown in Listing 46.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class CarWheelAnimator : MonoBehaviour {
5.
6.        //The actual wheel to read data from
7.        public WheelCollider wheel;
8.
9.        //Axis for vertical rotaion
10.       public Transform steeringAxis;
11.
12.       //Stores steering angle from last frame,
13.       //in order to be able to reset Y rotation
14.       //of the wheel
15.       float lastSteerAngle = 0;
16.
17.       //To save the original position at the position
18.       Vector3 originalPos;
19.
20.       void Start () {
21.             //Register the original position of the wheel
22.             originalPos = transform.localPosition;
23.       }
24.
25.       void LateUpdate () {
26.             //Convert wheel speed in rpm to degrees per second
27.             float rotationsPerSecond = wheel.rpm / 60;
```

```
28.           float degreesPerSecond = rotationsPerSecond * 360;
29.
30.           //Rotate around local y axis
31.           transform.Rotate(0, degreesPerSecond * Time.deltaTime, 0);
32.
33.           //Steering axis exists in front wheels
34.           if(steeringAxis != null){
35.                //Reset the steering to zero by sbtracting
36.                //the steering value of last frame
37.                transform.RotateAround(
38.                                  steeringAxis.position,
39.                                  steeringAxis.up,
40.                                  -lastSteerAngle);
41.                //Apply new steering value
42.                transform.RotateAround(
43.                                  steeringAxis.position,
44.                                  steeringAxis.up,
45.                                  wheel.steerAngle);
46.                //Update last steering angle value for the next frame
47.                lastSteerAngle = wheel.steerAngle;
48.           }
49.
50.           //Check if the wheel hits the ground
51.           WheelHit hit;
52.
53.           if(wheel.GetGroundHit(out hit)){
54.                //Wheel hits the ground.
55.                //Move the wheel up by spring pressing distance
56.                //Use world space
57.                float colliderCenter = hit.point.y + wheel.radius;
58.                Vector3 wheelPosition = transform.position;
59.                wheelPosition.y = colliderCenter;
60.                transform.position = wheelPosition;
61.           } else {
62.                //No hit, smoothly return wheel to its original position
63.                Vector3 pos = transform.localPosition;
64.                pos = Vector3.Lerp(transform.localPosition,
65.                                                originalPos, Time.deltaTime);
66.                transform.localPosition = pos;
67.           }
68.      }
69. }
```

Listing 46: A script to animate the visual wheels based on the properties of the wheel colliders

We need to attach this script to each one of the four visual wheels of our vehicle. We have two variables that need to be set from the inspector: *wheel*, which references the input wheel collider; from which the data is going to be read (RPM, steer angle, and suspension), and *steeringAxis*; which is the axis of vertical rotation of the visual wheel. This vertical rotation reflects the steer angle of the wheel, and hence is needed only for the front wheels. Before discussing the details of the script, we have to add two new empty objects to the hierarchy of our vehicles. These objects are *SteeringAxis_L* and *SteeringAxis_R*. As the names suggest, these objects are going to be the axes for steering rotation, so they must be positioned at the left and right ends of *FrontAxis*. Now we have to specify the appropriate source of data for each one of the four wheels, as well as the appropriate steering axes for the front wheels.

Back to the script, we have two additional variables: *lastSteerAngle*, which stores the steering angle from the previous frame, and *originalPos*, which stores the original position of the wheel when *Start()* function is called. Since this script performs animation tasks, the best practice is to update it using *LateUpdate()*; in order to make sure that all objects have updated state before animating them. The first task is straightforward, which is the spinning of the wheels. All we have to do is to read RPM values from the wheel collider, convert its value from minutes to seconds, and then use the converted value to rotate the wheel around its local y axis (remember that the wheel is a cylinder laying on its side, so the its local y axis goes from left to right). This rotation is performed in line 31.

If *steeringAxis* variable is not *null*, we update the steer angle of the wheel based on the steer angle of the collider. This task is performed through two steps: reset and set. The first step is to reset the rotation of the wheel back to straight position, which can be done by rotating the wheel around the local y axis of *steeringAxis* by the amount of *-lastSeerAngle*. Now we have to set the new steer angle by rotating the wheel around the same axis, but this time by amount equal to *wheel.steerAngle*, which is the current steer angle of the collider. Finally, we store the value of current steer angle in *lastSteerAngle*, to be able to reset the value in the next frame. Keep in mid that *steeringAxis* for the back wheels is *null*, so this step is not applicable to these wheels.

Finally, we have to update the y position of the wheel based on the state of the suspension spring. If you have already tested the vehicle, you might have noticed that some the wheels sink into the ground. This is a result of applying a pressure on the suspension springs, which moves the wheel collider upwards for a short time. However, we want to see something different: the wheel must remain on the ground, and the car body alone must be lowered.

To animate correctly, we have to know first whether the wheel is grounded. Therefore, we define in line 51 a variable of type *WheelHit*, which gives us details about the state of the wheel collider. The function *GetGroundHit()* returns *true* if the wheel collider is currently grounded, and stores the details in *hit* by using the keyword *out*. The variable *hit.point* stores the contact point between the wheel collider and the ground. Therefore, if we add the y member of this point to the radius of the collider; we get the correct y value for the position of the wheel. To keep thing simple, I am going to move the wheel in world coordinates only. All we have to do is to update the y position of the wheel so it matches the current center of the collider (lines 57 through 60).

In some cases, such as car jumping, the wheel collider does not touch the ground; which requires us to return the wheel to its original position before applying the effect of the suspension spring. If *GetGroundHit()* function returns *false*, then we know that the wheel is not on the ground. In this case we return it smoothly to its original local position, which we have already stored in *originalPos*. Smoothing transformations over several frames is essential for acceptable animation, so we are going to learn how to implement it. The core function when smoothing is the *Lerp()* function, which exists for a number of data types in Unity.

In this script we call *Vector3.Lerp()*, in order to smoothly return the wheel to its original position. *Lerp()* function takes three arguments: *Vector3 from*, *Vector3 to*, and *float t*. If the value of *t* is zero, *from* vector is returned, and if the value of *t* is 1, *to* vector is returned. If $t = 0.5$, the function returns the middle point between to vectors. We call this function in line 64 and provide it with a small value (*Time.deltaTime*), so we get a new point on the path between *transform.localPosition* and *originalPosition*. The returned point is closer to *transform.localPosition*, so we store it in *pos* and use it as the new position of the wheel. As a result, the wheel will keep moving smoothly towards *originalPos*, and eventually return to its original position. Illustration 59 shows the difference between pressed and rest states of the suspension springs. The final result can be found in *scene16* in the accompanying project.

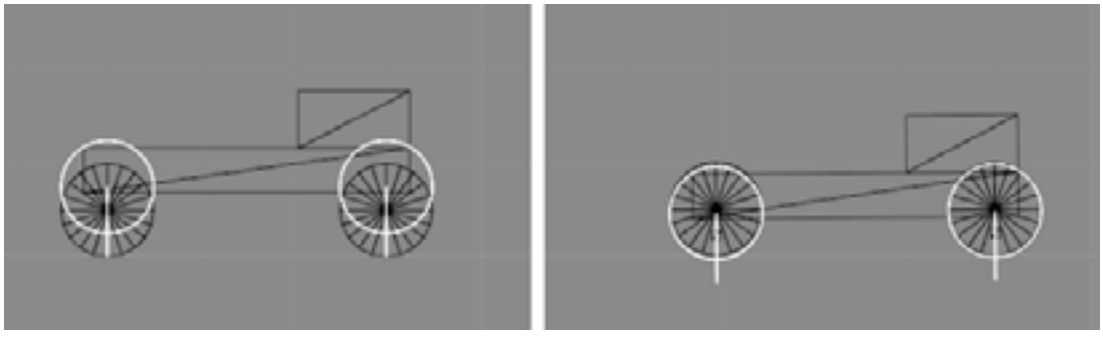


Illustration 59: Suspension springs at rest state (left) and while pressed (right). When springs are pressed wheel colliders sink into the ground and the car body is lowered, while visual wheels are kept on the ground.

4.3 Physical player character

In this section we are going to create a character controller that with physical behavior. This character can then be used for first person, third person, or even platformer input systems. The idea is to have a capsule collider with a rigid body attached to it. This capsule is going to be controlled by applying appropriate forces to it. To illustrate the physical character, I am going to make a first person input system. Therefore, we need to have a capsule with the main camera attached to it as child, and positioned at the top of the capsule. Let's consider that our character has a weight of 70 kilograms, which can be set from the rigid body component. Another important setting for the rigid body is to freeze the rotation on all axes, which means that physics simulator cannot rotate the character but only move it.

Following the same methodology we used in section 4.2 with car driver, we are going to make a character component that is totally isolated from user input. To control the character, we are going to write a second script that takes user input and calls appropriate functions from the character controller. Listing 47 shows *PhysicsCharacter* script, which we are going to attach to the capsule to turn it into a controllable character.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PhysicsCharacter : MonoBehaviour {
5.
6.       //maximum jump height in meters
7.       public float jumpHeight = 2;
8.
9.       //Horizontal movement speed
10.      public float movementSpeed = 8;
11.
12.      //Position of the character's feet
13.      public Transform feet;
14.
15.      //Input flags
16.      bool walkForward, walkBackwards,
17.           strafeRight, strafeLeft, jump;
18.
19.      void Start () {
20.
21.      }
22.
23.      public void WalkForward(){
24.           walkForward = true;
25.           walkBackwards = false;
26.      }
27.
28.      public void WalkBackwards(){
29.           walkBackwards = true;
30.           walkForward = false;
31.      }
32.
33.      public void StrafeRight(){
34.           strafeRight = true;
35.           strafeLeft = false;
36.      }
37.
38.      public void StrafeLeft(){
39.           strafeLeft = true;
40.           strafeRight = false;
41.      }
42.
43.      public void Jump(){
44.           jump = true;
45.      }
46.
47.      public void Turn(float amount){
48.           transform.RotateAround(Vector3.up, amount);
49.      }
50.
51.      //Fixed update is better with physics
52.      void FixedUpdate () {
53.           //Player can direct character only
54.           //when it is on the ground
55.           //or stuck somewhere with near-zero vertical velocity
```

```
56.           Vector3 velocity = rigidbody.velocity;
57.           if(OnGround() || (velocity.y >= 0 && velocity.y < 0.1f)){
58.                //Reset velocity on x and z to zero
59.                velocity.x = velocity.z = 0;
60.
61.                //update movement
62.                if(strafeLeft){
63.                      //Move left
64.                      velocity += -transform.right * movementSpeed;
65.                      strafeLeft = false;
66.                } else if(strafeRight){
67.                      //Move right
68.                      velocity += transform.right * movementSpeed;
69.                      strafeRight = false;
70.                }
71.
72.                if(walkForward){
73.                      //Move forward
74.                      velocity += transform.forward * movementSpeed;
75.                      walkForward = false;
76.                } else if(walkBackwards){
77.                      //Move backwards
78.                      velocity += -transform.forward * movementSpeed;
79.                      walkBackwards = false;
80.                }
81.           }
82.
83.           rigidbody.velocity = velocity;
84.
85.           //Read jump input
86.           if(jump && OnGround()){
87.                //v2^2 - v1^2 = 2as
88.                //v2 is zero (at max height)
89.                //v1^2 = -2as
90.                //v1 = sqrt(-2as)
91.                float v1 =
92.                      Mathf.Sqrt(-2 * Physics.gravity.y * jumpHeight);
93.
94.                //Use momentum formula p=mv with up as velocity direction
95.                rigidbody.AddForce(
96.                      Vector3.up * v1 * rigidbody.mass,
97.                      ForceMode.Impulse);
98.
99.                jump = false;
100.          }
101.
102.     }
103.
104.     //Checks whether the character is on the ground
105.     public bool OnGround(){
106.          //Cast a ray from feet position downwards.
107.          //The length of the ray is 10 cm.
```

```
108.          //If it hits the ground,
109.          //then the character is grounded
110.          if(Physics.Raycast(
111.               new Ray(feet.position, -Vector3.up), 0.1f)){
112.               return true;
113.          }
114.          return false;
115.     }
116.   }
```

Listing 47: Character controller based on physics simulation

We can set the values of *jumpHeight* and *movementSpeed* values of the character to match our needs. Additionally, we have the third variable *playerFeet*, which must reference an empty object that is a child of the capsule. This object must be positioned at the bottom of the capsule, where the feet are supposed to be. Let's begin from the last function in the script, *OnGround()*, at line 105. This function tests whether the character is currently standing on the ground. To perform this task, the function casts a ray using *Physics.RayCast()* function. This functions needs a ray and optionally a maximum distance for that ray, and tells whether this ray has hit something while traveling in its direction. In line 111 we create a new ray that starts from the position of the feet and goes downwards. We limit the travel distance of this ray to 10 cm only, so that it hits the ground only when the character is actually standing on it. If the ray hits the ground, we return *true* to indicate that the character is currently grounded.

In a similar implementation to *PhysicsCarDriver*, we declare a number of flag variables that describe the current control state of the character. Each one of these variables has a corresponding function that can be called to set its value. For example, calling *WalkForward()* sets the value of *walkForward* flag to *true*, as well as setting the value of *walkBackwards* flag to false. The same rule applies to all other flags. One interesting function is *Turn()*, which does not deal with any flags, but simple takes a value that represents a rotation degree, and rotates the capsule around the y axis by the provided degree.

The last (and most important) function we need is *FixedUpdate()*, which applies the state of control flags to the rigid body of the character. The first step is in line 56, in which we measure the current velocity of the character and store it in *velocity* variable. The next step is to update the movement of the object on x and z axes based on the state of walking and strafing flags. We assume that the character cannot direct itself while flying in the air (i.e. during jumping), so we make sure it is either grounded or stuck somewhere with a vertical velocity that is almost zero. This latter case can happen with non flat grounds and other cases. For example, consider the case where the character stands over a small gap between two boxes, like in Illustration 60. In this case *OnGround()* function will certainly return *false*, because the ray will be cast down through the gab, and hence will pass more than 10 cm before hitting the ground. However, we still want the character to be able to move, otherwise it is going to be stuck forever.

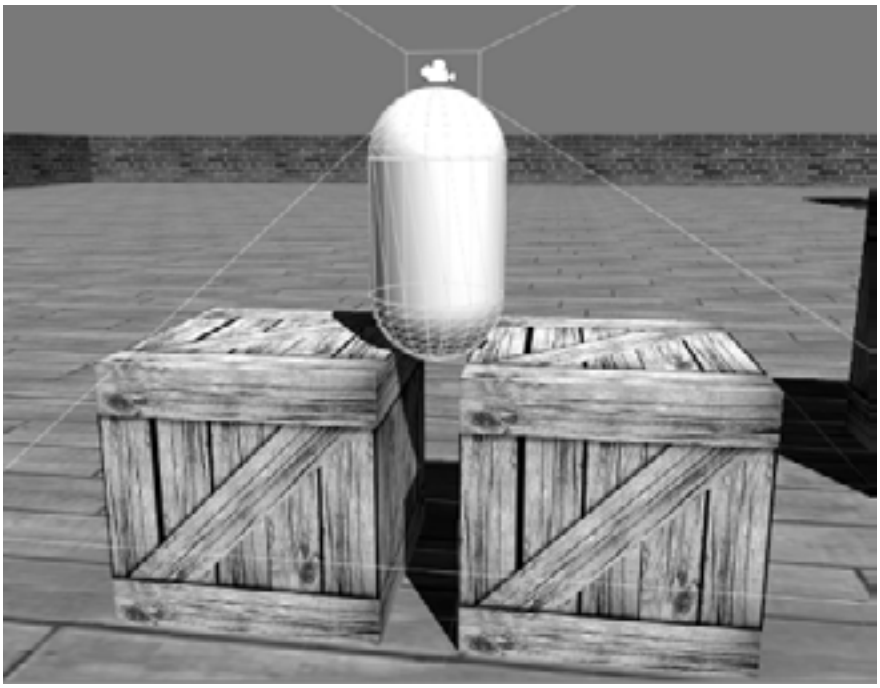


Illustration 60: The character in this case is not grounded. However, it must still be controllable

If controlling the character is allowed, we have to clear its current velocity on x and z axes, and then apply the new velocity. Steps in lines 62 through 80 are similar to their counterparts in the previously written *FirstPersonControl* script (Listing 9). The difference is in the implementation of the movement, since we implement it this time by setting the velocity of the rigid body on x and z axes in accordance to the control flags. In line 83, we store the newly computed velocity back into *rigidbody.velocity*, which makes the physics simulator move the object based on this new velocity. Keep in mind that all these steps did not touch the y value of the velocity, and hence have no effect on the jumping or falling state of the character.

The next step at line 86 is reading jump input and applying jump if the character is grounded. Jumping is implemented by applying a one-time force (pulse) to the character. This force must push the character upwards in the air, until it reaches the specified *jumpHeight* and then starts to fall down. From a physical point of view, our character is a projectile that is going to be thrown in the air with an initial speed. As the time goes, this speed is going to be reduced until it reaches zero at the maximum height. After that, the projectile begins to fall down again by the force of gravity. All we have to do is to compute the correct initial velocity for the projectile, which depends its mass. This computation requires us to dive into physics and remember some laws of projectiles. The following paragraph discusses in details how can we calculate the force magnitude in order to reach the desired jump height. If you do not like physics, you are free to skip it.

We use the projectile formula, , where v_2 is the velocity of the objects when it reaches its maximum height, is the initial velocity of the object when it leaves the ground, a is the acceleration, and s is the maximum height the object reaches. In our case, v_1 is the sole unknown we need to work out for. At the maximum height, the velocity of the object reaches zero before it starts to fall, hence . s in our case is the value of *jumpHeight*, which is also known to us. As the object goes up, it loses its velocity due to the acceleration of its weight, which is the acceleration of the gravity ( ). By working out for , we get , which is expressed in lines 91 and 92 in the script. Since jump force is a momentum that is applied once, we can use the momentum formula to compute the amount of the force we need to add to the rigid body of the object. This formula is expressed in line 96. We call *rigidbody.AddForce()* at line 95, which is used to add a force to the rigid body. After adding the appropriate jump force, we reset *jump* flag to *false*.

Now we need a script to read the input from the player and call the functions of the character to control it. This script is *FPSInput* shown in Listing 48.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class FPSInput : MonoBehaviour {
5.
6.       //Mouse look speed on both axis
7.       public float horizontalMouseSpeed = 0.9f;
8.       public float verticalMouseSpeed = 0.5f;
9.
10.      //Max allowed cam vertical angle
11.      public float maxVerticalAngle = 60;
12.
13.      //Mouse position in previous frame,
14.      //important to measure mouse displacement
15.      private Vector3 lastMousePosition;
16.
17.      //Store camera transform
18.      private Transform camera;
19.
20.      //The character to control
21.      PhysicsCharacter character;
22.
23.      void Start () {
24.           character = GetComponent<PhysicsCharacter>();
25.           lastMousePosition = Input.mousePosition;
26.           //Find camera object in children
27.           camera = transform.FindChild("Main Camera");
28.      }
29.
30.      void Update () {
31.           //Step 1: rotate cylinder around global Y
32.           //axis based on horizontal mouse displacement
33.           Vector3 mouseDelta = Input.mousePosition - lastMousePosition;
34.
35.           character.Turn(mouseDelta.x *
36.                horizontalMouseSpeed *
37.                Time.deltaTime);
38.
39.           //Get current vertical camera rotation
40.           float currentRotation = camera.localRotation.eulerAngles.x;
41.
42.           //Convert vertical camera rotation from range [0, 360]
43.           //to range [-180, 180]
44.           if(currentRotation > 180){
45.                currentRotation = currentRotation - 360;
46.           }
47.
48.           //Calculate rotation amout for current frame
49.           float ang =
50.                -mouseDelta.y * verticalMouseSpeed * Time.deltaTime;
51.
52.           //Step 2: rotate camera around it's local X
53.           //axis based on vertical mouse displacement
54.           //First check allowed limits
```

```
55.             if((ang < 0 && ang + currentRotation > -maxVerticalAngle) ||
56.                   (ang > 0 && ang + currentRotation < maxVerticalAngle)){
57.                   camera.RotateAround(camera.right, ang);
58.             }
59.
60.             //Update last mouse position for next frame
61.             lastMousePosition = Input.mousePosition;
62.
63.             if(Input.GetKey(KeyCode.A)){
64.                   character.StrafeLeft();
65.             } else if(Input.GetKey(KeyCode.D)){
66.                   character.StrafeRight();
67.             }
68.
69.             if(Input.GetKey(KeyCode.W)){
70.                   character.WalkForward();
71.             } else if(Input.GetKey(KeyCode.S)){
72.                   character.WalkBackwards();
73.             }
74.
75.             if(Input.GetKeyDown(KeyCode.Space)){
76.                   character.Jump();
77.             }
78.       }
79. }
```

Listing 48: A script to read user input and control the physics character

We have already discussed most of the functions in a similar script, which is *FirstPersonControl* shown in Listing 9. The two scripts handle the camera movement in the same way. However, *FPSInput* depends on *PhysicsCharacter*, and cannot control the character directly by displacing it. You can notice the difference between the two scripts in lines 61 through 77 of *FPSInput.* In these lines, we use player input to call functions from *PhysicsCharacter* script, which must be attached to the same object. Before testing your character, it is a good idea to disable the renderer of the capsule. A complete physics character can be found in *scene17* in the accompanying project. It is worth pointing out that if you add some basic shapes with rigid bodies and adequate masses, you would be able to push them and move them using the physics character.

4.4 Ray cast shooting

In this section we are going to extend the first person character we made in section 4.3, by giving it the ability to shoot bullets. This time we are going to learn a new technique to implement shooting, which is ray casting. We have already dealt with a simple usage of ray casting to test character grounding. However, in this section we need more detailed information about the result of ray casting. Let's begin by adding a simple gun model and a crosshair for the character. These two objects must be added as children for the camera, and must be adjusted to give the first person view. When the game runs, the gun and the crosshair should appear as in Illustration 61.

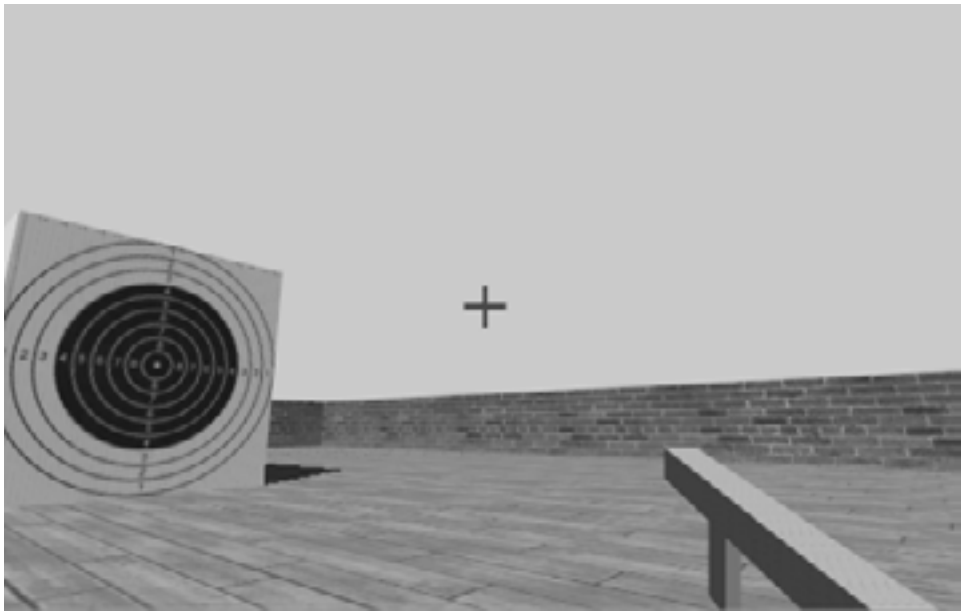


Illustration 61: A simple gun and a crosshair made of basic shapes and added to the first person camera

As you would expect, we are going to handle mouse left-click as fire command. When the player fires the weapon, he casts a ray towards the aim position of the crosshair. This ray is the bullet fired from the weapon, and we are going to be able to check if it hit something and handle the hit. The main script in shooting mechanism is *RaycastShooter*, shown in Listing 49. Before discussing the details of the script, we need to know a number of basic properties of the ray cast shooter, which are listed below:

1. The shooter casts one ray at a time.
2. The ray has a maximum range that can be set from the inspector.
3. There is a time gap between two consecutive ray casts (fire rate).
4. Ray shooter has vertical and horizontal thresholds of inaccuracy, which are expressed in terms of maximum angle between the ray that passes through the center of the crosshair and the actual ray cast by the shooter. Each time a ray is cast, it is going to be rotated around x and y axes by a random value between positive and negative values of the threshold.
5. The shooter has a configurable value for damage caused by its bullets. This value represents the maximum damage when the target is at zero distance from the shooter. The damage power decreases as the distance between the shooter and the target increases, until it reaches zero at the maximum range of the shooter.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class RaycastShooter : MonoBehaviour {
5.
6.       //How far can each bullet travel
7.       public float maxRange = 100;
8.
9.       //how many seconds to wait between two consecutive shoots
10.      public float shootRate = 0.1f;
11.
12.      //vertical inaccuracy in degrees
13.      public float verticalInaccuracy = 1;
14.
15.      //horizontal inaccuracy in degrees
16.      public float horizontalInaccuracy = 1;
17.
18.      //Bullet damage from zero-distance
19.      public float power = 100;
20.
21.      //Possition and direction of casted rays
22.      public Transform muzzle;
23.
24.      //Last time shooting is performed
25.      float lastShootTime = 0;
26.
27.      //Store last inaccuracy vector
28.      Vector2 inaccuracyVector;
29.
30.      void Start () {
31.
32.      }
33.
34.      void Update () {
35.
36.      }
37.
38.      //Shoot a bullet using ray cast
39.      //return true if a bullet has been shot
40.      public void Shoot(){
41.           if(Time.time - lastShootTime > shootRate){
42.                //Get some random values for inaccuracy
43.                inaccuracyVector.y = Random.Range(
44.                                                  -horizontalInaccuracy,
45.                                                  horizontalInaccuracy);
46.
47.                inaccuracyVector.x = Random.Range(
48.                                                  -verticalInaccuracy,
49.                                                  verticalInaccuracy);
50.
51.                //Rotate the muzzle to apply inaccuracy
52.                muzzle.Rotate(inaccuracyVector.x, 0, 0);
53.                muzzle.Rotate(0, inaccuracyVector.y, 0);
54.
55.                //A variable to store hit data
```

```
56.                 RaycastHit hit;
57.
58.                 //Perform the ray cast
59.                 if(Physics.Raycast(
60.                             new Ray(muzzle.position, muzzle.forward),
61.                             out hit, maxRange)){
62.
63.                       //Overwrite hit.distance
64.                       //with the value of scaled damage
65.
66.                       hit.distance =
67.                             power * (1 - (hit.distance / maxRange));
68.                       hit.transform.SendMessage(
69.                                   "OnRaycastHit", hit,
70.                                    SendMessageOptions.DontRequireReceiver);
71.
72.                 }
73.
74.                 //return muzzle to its original rotation
75.                 muzzle.Rotate(-inaccuracyVector.x, 0, 0);
76.                 muzzle.Rotate(0, -inaccuracyVector.y, 0);
77.
78.                 //Register last shooting time
79.                 lastShootTime = Time.time;
80.
81.                 //Inform other scripts that shooting happened
82.                 SendMessage("OnRaycastShoot",
83.                             SendMessageOptions.DontRequireReceiver);
84.           }
85.      }
86.
87.      //Get last inaccuracy
88.      public Vector2 GetLastInaccuracyVector(){
89.           return inaccuracyVector;
90.      }
91. }
```

Listing 49: Ray cast shooting scrip

First variables represent shooting properties mentioned earlier, which are *maxRange*, *shootRange*, *verticalInaccuracy*, *horizontalInaccuracy*, and *power*. The variable *muzzle* represent to the position and direction of the ray that is going to be cast. To apply the fire rate, we need to store the time of the last ray cast. The variable *lastShootTime* is where we are going to store this value. Each time a ray is cast, we generate two random values for the angles of inaccuracy, and store these values in *inaccuracyVector* for later use. Inaccuracy mechanism is going to be covered in details soon.

The main function of this script is *Shoot()*, in which the ray casting is performed. Before shooting, the function checks whether minimum time gap between two consecutive shoots has already passed. If this is true, it generates two random numbers for x and y inaccuracy angles. The horizontal inaccuracy is a random value between *-horizontalInaccuracy* and *+horizontalInaccuracy*, and, similarly, the vertical inaccuracy is generated in the range [*-verticalInaccuracy*, *+verticalInaccuracy*]. The generated random values are stored in *x* and *y* members of *inaccuracyVector*. Initially, the muzzle must be positioned so that its positive z axis points forward towards shooting direction. Before shooting, we rotate the muzzle around its local x and y axis by the values of the two generated inaccuracy angles.

After performing inaccuracy effect, we are now ready to perform the actual shooting. The variable *hit* declared in line 56 is the place were hit data are going to be stored (if there is a hit surely). In line 59 through 61 we call *Physics.RayCast()*, in order to perform shooting. The ray starts from the position of the muzzle and heads towards the direction of its positive z axis. The variable *hit* is provided as store place for hit data using the keyword *out*. This keyword marks an output parameter, unlike input parameters we are used to use when calling functions. In other words, the function *Physics.RayCast()* is going to set the value of *hit*, instead of reading its value. Finally, we tell the function that the length of the ray must not exceed the value of *maxRange*. Lines 66 through 70 are executed if the ray has hit something. The variable *hit.distance* stores the distance between ray generation position and the object that has been hit by the ray. We need this distance in order to compute the scaled damage we are going to apply to the hit object.

Remember that we store the damage of the bullet in the variable *power*, which is the damage applied to the object when shot from zero-distance. As the distance between the shooter and the target increases, the damage caused by the bullet decreases linearly, until it reaches zero at *maxRange*. When the bullet hits the target, the damage it causes interests us more than the distance from which it has been shot. Therefore, we compute the damage based on the distance, and them overwrite *distance* member of *hit* variable with the scaled damage (lines 66 and 67). Finally, we inform the target that it has been hit by sending *OnRaycaseHit* message to it and providing *hit*, which contains necessary data such as damage and direction.

When shooting is completed, we have to return the muzzle back to its original rotation by rotating it again with the negatives of random inaccuracy variables. After that, the current time is stored in *lastShootTime*, and the message *OnRaycastShoot* is sent. This latter message can be useful if we need to do some stuff in combination with shooting, such as playing a sound or an animation. We can also access the values of last inaccuracy vector, by calling *GetLastInaccuracyVector()*. This can help us in camera shaking, or gun animation as we do in *RaycastShooterAnimator* shown in Listing 50. The job of this script is to animate the gun during shooting, so that the player gets on-screen feedback that the shooting actually happened.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class RaycastShooterAnimator : MonoBehaviour {
5.
6.       //Distance to move backwards when animating
7.       public float zDistance = 0.15f;
8.
9.       //Reference to shooter
10.      RaycastShooter shooter;
11.
12.      //Original position
13.      Vector3 originalPosition;
14.
15.      //Original rotation
16.      Quaternion originalRotation;
17.
18.      void Start () {
19.           shooter = GetComponent<RaycastShooter>();
20.           originalPosition = transform.localPosition;
21.           originalRotation = transform.localRotation;
22.      }
23.
24.      void LateUpdate () {
25.           //Slowly return to original position and rotation
26.           transform.localPosition =
27.                Vector3.Lerp(transform.localPosition,
28.                      originalPosition, Time.deltaTime * 10);
29.
30.           transform.localRotation =
31.                Quaternion.Lerp(transform.localRotation,
32.                      originalRotation, Time.deltaTime * 10);
33.      }
34.
35.      void OnRaycastShoot(){
36.           //Shooting happend: animate based on in
37.           Vector2 rotation =
38.                shooter.GetLastInaccuracyVector();
39.           transform.Rotate(rotation.x, 0, 0);
40.           transform.Rotate(0, rotation.y, 0);
41.           transform.Translate(0, 0, -zDistance);
42.      }
43. }
```

Listing 50: A script to animate the gun during shooting

The script begins by storing the original local position and rotation of the gun. This step is necessary to insure that we return the gun to its original position after animating it. The script also needs a reference to *RaycastShooter*, in order to read inaccuracy values and use them in the animation. One additional interesting variable here is *zDistance*, which is the amount of movement on the local z axis of the gun. When the player shoots, the gun moves very fast (instantly, in fact) to this z position, in order to simulate shooting reaction. After that, the gun return slowly and smoothly to its original position. In addition to the movement along z axis, the gun rotates around its local x and y axes by the values of last inaccuracy vector accessed through *shooter.GetLastInaccuracyVector()*. In *LateUpdate()*, we make sure that the gun returns to its original position by calling the functions *Vector3.Lerp()* for the position and *Quaternion.Lerp()* for the rotation. However, we multiply *Time.deltaTime* by 10 in order to get a faster return, which is needed in case of high shoot rate.

Up to now, we have a physical first person character that can move, jump, push objects, and aim at them using a gun with a crosshair. The remaining step in regard to character is giving the player the ability to shoot using mouse button. This task is as simple as reading mouse input and calling *Shoot()* function from *RaycastShooter* script. Listing 51 shows *GunInput* script, which reads mouse input and triggers the shooter.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class GunInput : MonoBehaviour {
5.
6.       //If true, the player don't have to release
7.       //the mouse button between shoots
8.       public bool continuous = true;
9.
10.      void Start () {
11.
12.      }
13.
14.      void Update () {
15.           //Send Shoot message on mouse click
16.           if(continuous){
17.                if(Input.GetMouseButton(0)){
18.                     SendMessage("Shoot");
19.                }
20.           } else {
21.                if(Input.GetMouseButtonDown(0)){
22.                     SendMessage("Shoot");
23.                }
24.           }
25.      }
26. }
```

Listing 51: A script to read left mouse button and activate shooting

The variable *continuous* allows us to control the type of shooting, so we can for example force the player to release mouse button before shooting again. An interesting detail in this script is its independency from *RaycastShooter*, which makes it reusable with other types of shooters or weapons. The only requirement that other shooters must have is the ability to receive *Shoot* message sent by this script. The three scripts *RaycastShooter*, *RaycastShooterAnimator*, and *GunInput* must be added to the gun game object. After adding them, we have to set the muzzle for *RaycastShooter*, which is in this case the crosshair object. We can set a scene to test shooting, which consists of some static targets as well as dynamic boxes and balls with rigid bodies. The scene may look like Illustration 62.

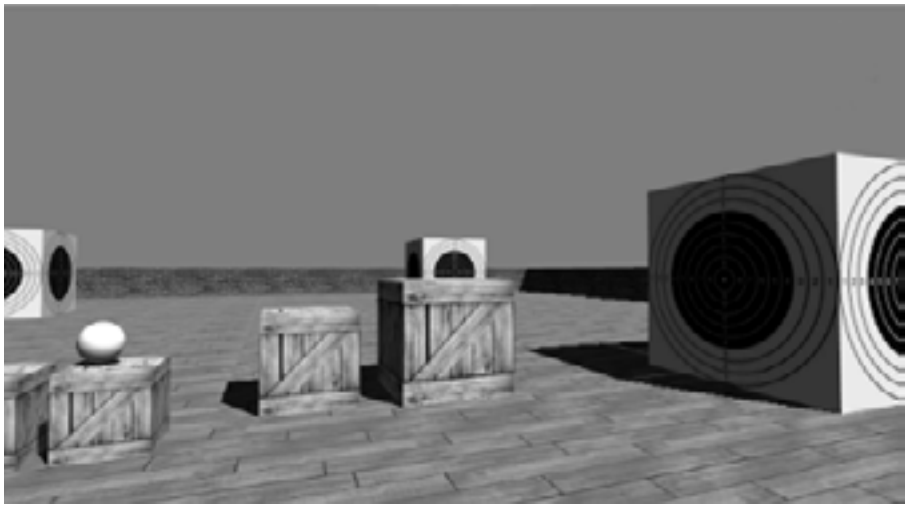


Illustration 62: A scene to test ray cast shooting

Now we have to specify what happens when an object is shot. Hit reaction can vary from an object to another, depending on how does each object respond to *OnRaycastHit* message sent by *RaycastShooter*. To illustrate the variance of hit reactions, we are going to write two scripts that handle *OnRaycastHit* differently. The first one is *BulletHoleMaker*, shown in Listing 52. As the name suggests, this script creates a bullet hole at the position of the hit. This hole is in fact an instance of a prefab, which is made of a quad with bullet hole texture on it.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class BulletHoleMaker : MonoBehaviour {
5.
6.       //Object to instantiate as hole
7.       public GameObject holePrefab;
8.
9.       //Seconds to wait before destroying hole object
10.      public float holeLife = 15;
11.
12.      void Start () {
13.
14.      }
15.
16.      void Update () {
17.
18.      }
19.
20.      //Receive OnRaycastHit message and generate hole at hit position
21.      void OnRaycastHit(RaycastHit hit){
22.           GameObject hole = (GameObject) Instantiate(holePrefab);
23.           hole.transform.position = hit.point;
24.           //If there is a rigid body, add the hole as a child to it
25.           //This makes the hole move along with the hit object
26.           if(hit.rigidbody != null){
27.                hole.transform.parent = hit.transform;
28.           }
29.           //Since we use a quad, we rotate it towards inside
30.           //This might be different if you use another shape
31.           hole.transform.LookAt(hit.point - hit.normal);
32.           //move it away from hit surface with a tiny amount
33.           //This ensures that the hole is always on top
34.           hole.transform.Translate(0, 0, -0.0125f);
35.
36.           //Invoke destruction after a while
37.           Destroy(hole, holeLife);
38.
39.           //Report some data
40.           print (name + " took damage of " + hit.distance);
41.      }
42. }
```

Listing 52: A script to generate a hole at the position of the bullet hit

If this script is attached to an object, it responds to *OnRaycastHit* by creating a hole at the position of the ray hit. First step is to instantiate the prefab provided through *holePrefab* and position it at *hit.point*, which is the position of the hit in world coordinates. If the hit object has a rigid body attached to it, then there is a possiblity that the object moves or rotates at any moment in the future. Therefore, it is necessary in this case to add the hole as a child of the hit object, so that the object and the hole displace and rotate as one unit. An important question to answer is: how must the hole be directed? The obvious answer is outwards. The variable *hit.normal* returns a vector perpendicular to the surface that has been hit, where the direction of this vector points outside from the object. By adding this vector to the position of the hit, we get the correct position and direction of the hole. However, the quad object in Unity has only one rendered face, which is the face that looks at the negative direction of the local z axis. Therefore, the positive z axis of the quad must look towards inside like in Illustration 63, in order to have the rendered face visible. Consequently, we set *hit.point – hit.normal* as look point for the quad.

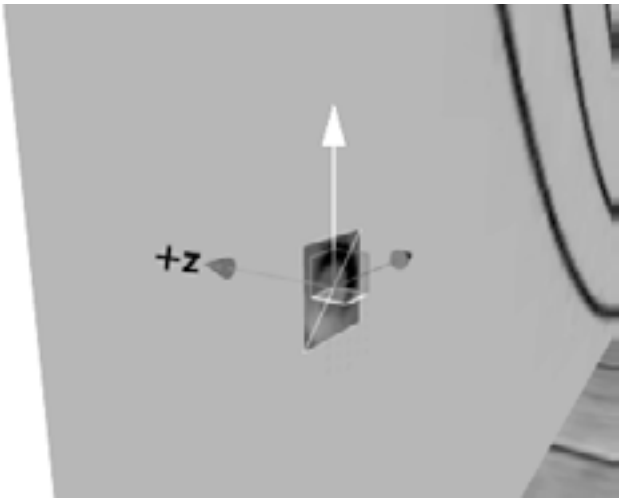


Illustration 63: Hole object made of quad, the positive z axis is looks inside so that the rendered face looks outside

After rotating the hole correctly, we need to make sure that it is on top of the surface. This guarantees that the hole is always in front of the object and hence visible. However, the distance must be very small, otherwise the space between the surface and the hole becomes visible. In the case of the quad we use, it is enough to move it along its negative z axis by 0.0125. Additionally, we need to keep the number of holes in the scene limit in order to avoid performance issues. Therefore, we specify a life time for the hole that can be set through *holeLife*. After creating, rotating, and positioning the hole, *Destroy()* is called for the newly created hole and is given *holeLife* value as wait time before destruction. Remember that we used the member *distance* of *RaycastHit* to store the scaled damage value computed using the power of the bullet. This value is printed along with the name of the hit object so that you get an idea about the scaled value of the damage. All we have to do now is to attach this script to any object we want, and provide it with the prefab of the hole it should create.

The second script we are going to write as a handler for *OnRaycastHit* is *BulletForceReceiver*. This script, shown in Listing 53, is specific for objects that have rigid bodies, and hence have dynamic physical behavior.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class BulletForceReceiver : MonoBehaviour {
5.
6.       void Start () {
7.
8.       }
9.
10.      void Update () {
11.
12.      }
13.
14.      //Receive OnRaycastHit message and apply an impulse force
15.      void OnRaycastHit(RaycastHit hit){
16.           rigidbody.AddForceAtPosition(
17.                -hit.normal * hit.distance, hit.point, ForceMode.Impulse);
18.      }
19. }
```

Listing 53: A script that receives ray cast hit and reacts by applying impulse force to the rigid body

When *OnRaycastHit* message is received, this script applies an impulse force of the rigid body once. The function used to apply the force is *AddForceAtPosition()*, which allows us to specify a point to apply the impulse on. This addition is important to give the realistic behavior we expect. For example, when the shot is near the upper side of the hit object, the upper side must absorb the greatest amount of the hit which might cause object to rotate. We take *hit.point* as the position where we want to apply the force. The magnitude of the force is the scaled damage stored in *hit.distance*, and the direction is inside the object *-hit.normal* (remember that *hit.normal* points outside from the object). You can experement the full functional physics character as well as ray cast shooting in *scene17* in the accompanying project.

4.5 Physics projectiles

In section 3.1 we implemented a simple projectile system, which consisted of objects that move with constant speed over the time. In this section we learn how to use rigid bodies with impulse forces to create more realistic projectiles. Turning any rigid body into a projectile is as simple as adding an impulse force with specific direction and magnitude to it. In this section, we are going to create a ball that has a rigid body. This ball is going to be thrown on a stack of boxes in a mechanic similar to Angry Birds. Illustration 64 shows the scene we need to setup for our projectile demo.

Illustration 64: A scene to demonstrate physics projectiles

To launch the ball (i.e. the projectile), the player has to hold the left mouse button on it and drag to the left. As indication, a line will be drawn between the ball and the mouse position. The longer this line is, the greater is the impulse force that launches the ball. This simple mechanism allows the player to control both the direction and the magnitude of the force that launches the projectile with ease. One more important note: even the scene is constructed in 3D, we are going to limit both movement and rotation in 2D. This means that we must freeze the movement on z axis for all objects, as well as freezing rotation around x and y axes. Not only that, we have also to make sure that all objects in the scene have the same z value for position, to make sure they collide with each other. Additionally, we need to add a new component called *Line Renderer* to the ball. This component draws a 3D line in the space that passes through a provided list of positions in the space. Illustration 65 shows the properties of the line renderer we need to add to the ball, which is going to be the indicator of launch direction and force.

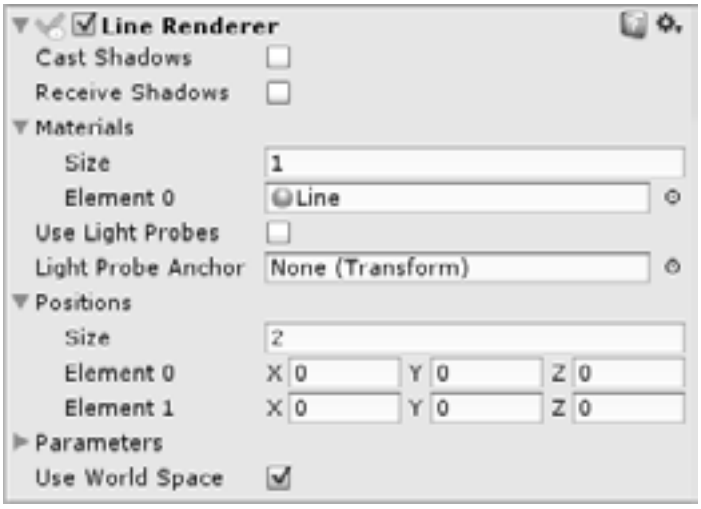


Illustration 65: Line renderer that draws launch direction indicator

You can use any material you want to render the line. Here I use a custom material called *Line*, which is simply a gradient of blue and orange. The number of positions we need is 2: a start point and an end point. The positions are by default zero, but we are going to change them according to user input. The script *PhysicsProjectile* shown in Listing 54 must be added to the ball object to turn it into a controllable projectile that can be launched using mouse drag.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PhysicsProjectile : MonoBehaviour {
5.
6.       //Launch force multiplier
7.       public float launchPower = 300;
8.
9.       //How long seconds to keep the projectile alive after launching?
10.      //-1 = keep for infinity
11.      public float lifeTime = 7;
12.
13.      //Has the player hold the mouse down on the object?
14.      bool mousePressed = false;
15.
16.      //Has the projectile been already launched?
17.      bool launched = false;
18.
19.      //Position to generate launch power from
20.      Vector3 launchPosition;
21.
22.      //Line to show launch direction
23.      LineRenderer line;
24.
25.      //Reference to main camera,
26.      //necessary for launch point specification with the mouse
27.      Camera cam;
28.
29.      void Start () {
30.           //Get the attached line and find the camera
31.           line = GetComponent<LineRenderer>();
32.           cam = Camera.main;
33.      }
34.
35.      void Update () {
36.           //We draw line after pressing the mouse on the projectile,
37.           //and before launching the projectile
38.           if(!launched && mousePressed){
39.                //Create a ray that goes from
40.                //camera position into the screen,
41.                //and passes throug mouse pointer position
42.                Ray cameraRay = cam.ScreenPointToRay(Input.mousePosition);
43.
44.                //Find the distance between the camera and the projectile
45.                float dist = Vector3.Distance(
46.                          cam.transform.position, transform.position);
```

```
47.
48.                //Set the launch position to the point on the ray that
49.                //has the same distance from the camera as the projectile
50.                launchPosition = cameraRay.GetPoint (dist);
51.
52.                //Update line start and end positions
53.                //Line starts at the position of the projectile
54.                line.SetPosition(0, transform.position);
55.
56.                //Line ends at the launch position
57.                line.SetPosition(1, launchPosition);
58.
59.                //After drawing the line, make sure that projectile
60.                //and lanunch position has the same z position
61.                launchPosition.z = transform.position.z;
62.           }
63.      }
64.
65.      //Called when a mouse button is pressed on the object
66.      void OnMouseDown(){
67.           mousePressed = true;
68.      }
69.
70.      //Called when a mouse button is released
71.      //and the same button has been already pressed in the object
72.      void OnMouseUp(){
73.           //Projectile must not has been alread launched
74.           if(!launched){
75.                //Set launched to true and destroy the line component
76.                launched = true;
77.                Destroy(line);
78.
79.                //Destroy object after its lifetime
80.                Destroy(gameObject, lifeTime);
81.
82.                //Apply a force that is directly proportional with
83.                //the distance between launch position and
84.                //the position of the projectile
85.                Vector3 forceDirection =
86.                      transform.position - launchPosition;
87.                forceDirection = forceDirection * launchPower;
88.                rigidbody.AddForce(forceDirection, ForceMode.Impulse);
89.           }
90.      }
91. }
```

Listing 54: A script to control the rigid body with mouse and apply impulse force to it

The script has two public variables: *launchPower*, which is the magnitude of the launch force we are going to multiply with distance specified by the player, and *lifeTime* which is the amount of time to keep the projectile after launching it. In addition to public variables, we have two flags: *mousePressed* which becomes *true* when the player presses the left mouse button over the projectile, and *launched*, which stays *false* until the projectile has been launched. The second flag is necessary to ensure that the player is allowed to launch the projectile only once. Additionally, we have *launchPosition*, which is the other end of the line that specifies the direction of the force to be applied. Finally, we need references to both the main camera and the line renderer that is attached to the object. The variable *Camera.main* gives us the reference to the main camera.

Launching the projectile is handled through in *OnMouseDown()* and *OnMouseUp()* functions. After holding the mouse during aiming, *LateUpdate()* handles updating the line that indicates launching direction and magnitude. *OnMouseDown()* function is called when the player clicks the mouse over the object, so we use this function to set *mousePressed* flag to *true*. After pressing the mouse over the projectile, the player should move the mouse to set the launch direction. Therefore, we use *LateUpdate()* function to update the indication line, which applies only if the projectile has not yet been launched and the mouse is still pressed. The first step is convert the position of the mouse pointer from screen coordinates to 3D space coordinates, in order to get the position of the other end of the line. To help us in this task, the camera provides us with the function *ScreenPointToRay()*.

To understand how does *ScreenPointToRay()* work, we have to imagine the mouse pointer as an object in front of the camera. The resulting ray starts from the position of the camera, passes through the position of the mouse pointer, and goes into the screen. After getting the ray, we need a point on that ray to be the second position for the line we are going to draw. This point must have the same distance from the camera as the projectile, which requires us to compute the distance between the camera and the projectile first. The function *cameraRay.GetPoint()* takes a distance and returns a point on the ray that has the provided distance from the origin of the ray, and this position is stored in *launchPosition*. The final step is update the line by calling *line.SetPosition()*. Each time we call this function we give it the index of the position and the point to set for that position. In the case of our line, we have only two positions: one is the position of the projectile itself *transform.position*, and the other is for *launchPosition*, which is the point on the ray we have computed. As a result, the line appears between the projectile and the mouse pointer as in Illustration 66. Keep in mind, however, that we change the actual launch position so that is has the same z value as the projectile, which guarantees generating a force in the correct direction. Nevertheless, drawing the line between the projectile and the actual launch position will make it appear to the player away from the mouse pointer, which is frustrating.

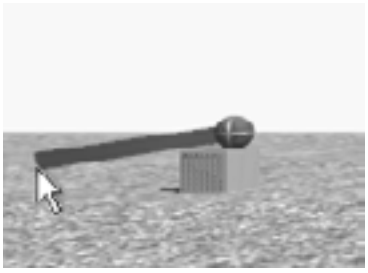


Illustration 66: Indication line drawn between the projectile and the mouse pointer

Now we have to handle releasing the mouse button and eventually launch the projectile. The event is handled through *OnMouseUp()* function, and it is handled only once. If the value of *launched* is *false*, this means that the projectile has not yet been launched. Therefore, the first thing we must do is to prevent future handling of mouse up event by setting *launched* to *true*. Before launching the projectile we have to destroy the line and invoke *Destroy()* for the projectile it self after the preset time. The next thing to do is to get the direction of the launching force. This direction is the vector between mouse position and projectile position, which we get by subtracting these positions and storing the result in *forceDirection*. We then multiply the vector by *launchPower* to magnify its effect, and finally add it as impulse force to the projectile. The last script to present is *ProjectileGenerator*, which is responsible for generating a new projectile after destroying the previous one. This simple script is shown in Listing 55. Illustration 67 shows the projectile hitting the boxes, which is the result of the functional demo that can be found in *scene18* in the accompanying project.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ProjectileGenerator : MonoBehaviour {
5.
6.       //Prefab to generate
7.       public GameObject projectile;
8.
9.       void Start () {
10.           //Try to generate a projectile once every second
11.           InvokeRepeating("Generate", 0, 1);
12.      }
13.
14.      void Update () {
15.
16.      }
17.
18.      void Generate(){
19.           //If there are no projectiles in the scene, generate one
20.           PhysicsProjectile[] prjectiles =
21.                FindObjectsOfType<PhysicsProjectile>();
22.           if(prjectiles.Length == 0){
23.                Instantiate(projectile,
24.                                  transform.position,
25.                                  transform.rotation);
26.           }
27.      }
28. }
```

Listing 55: A script to continuously generate projectiles

Illustration 67: The effect of throwing the projectile on stacked boxes

4.6 Explosions and destruction

This section has a special importance in developing action games, in which blowing up things and destroying some parts of the environment is a core mechanic. Having a physics simulator helps making realistic explosions, by giving us the ability to push object away with explosion force. The simulator also makes it possible to build destructible structures that are affected by explosions. In this section we are going to construct a simple building, which consists of building blocks. These blocks are affected by explosion force, which makes the building partially or completely destructible using explosions. In addition to being destructible, each block is going to have the ability to return to its original position and hence reconstruct the building. To begin, we have to create a prefab that represents these building blocks, and then make as many copies as we need. The building block we are going to use is a cube with brick texture like in Illustration 68.

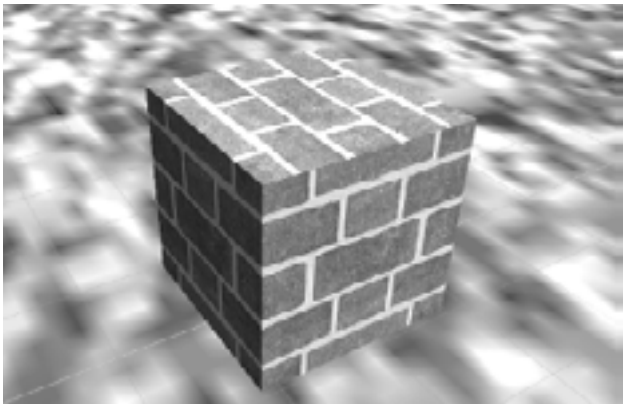


Illustration 68: Brick-textured cube to be used as building block for the destructible building

To build the prefab we need, first we have to add a rigid body component to it. However, this rigid body is going to be frozen by *Destructible* script, which is shown in Listing 56. This freezing must hold until an external force with enough magnitude moves the block. In our case, this external force will be an explosion.

```
1.  using UnityEngine;
2.  //We have to import this library
3.  using System.Collections.Generic;
4.
5.  public class Destructible : MonoBehaviour {
6.
7.       //We scan from the center of the object towards this direction
8.       //in order to find a dependency
9.       public Vector3 scanDirection;
10.
11.      //Desructibles that depend on this one
12.      List<Destructible> dependents = new List<Destructible>();
13.
14.      //To store original constraints before freezing
```

```
15.      RigidbodyConstraints original;
16.
17.      void Start () {
18.           //Try to find a destructible at the scan direction
19.           //If it is found, then add self as dependent to it
20.           RaycastHit hit;
21.           Ray scanRay = new Ray(transform.position, scanDirection);
22.
23.           if(Physics.Raycast(scanRay, out hit, scanDirection.magnitude)){
24.
25.                Destructible dependency =
26.                     hit.transform.GetComponent<Destructible>();
27.
28.                if(dependency != null){
29.                     dependency.dependents.Add(this);
30.                }
31.           }
32.
33.           //Store the original constraints then freeze the rigid body
34.           original = rigidbody.constraints;
35.           rigidbody.constraints = RigidbodyConstraints.FreezeAll;
36.      }
37.
38.      void Update () {
39.
40.      }
41.
42.      //Destruct this destructible by restoring its original constraints
43.      public void Destruct(){
44.           rigidbody.constraints = original;
45.           //Call Destruct() in dependents, and delay it a little bit
46.           foreach(Destructible dependent in dependents){
47.                if(dependent != null){
48.                     float time = Random.Range(0.0f, 0.01f);
49.                     dependent.Invoke("Destruct", time);
50.                }
51.           }
52.           //Inform other scripts that destruction heppened
53.           SendMessage("OnDestruction",
54.                            SendMessageOptions.DontRequireReceiver);
55.           }
56. }
```

Listing 56: A script for building blocks that construct a destructible structure

Before diving into the logic of the code, there is a couple of new things to introduce. First of all, we use the keyword *using* in line 3 in order to import a new library into our code. This library is called *System.Collection.Generic*. The details of the library are not as much important as knowing how to use it. By bringing this library to our script, we are able to declare a *List*, which we do in line 12. The declaration *List<Destructible>* means that this list accepts only one type of elements, which is *Destructible*. Lists are, like arrays, collections of objects. However, they are dynamic and we can add as much elements to a list as we need without having to worry about its capacity.

The most two important variables in this class are *scanDirection* and *dependents*. To realize their importance we have first to understand how the destructible building is built. When we arrange the blocks in some order either vertically or horizontally, we must build relations between them. For instance, a pillar like in Illustration 69. This pillar consists of 8 building blocks arranged vertically.

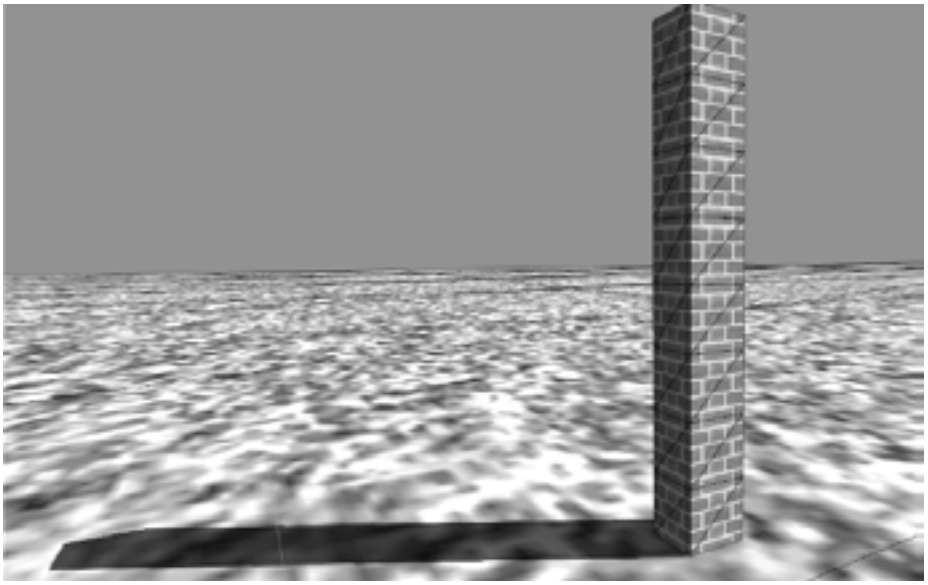


Illustration 69: A pillar made of 8 destructible building blocks

Logically, when the bottom most block is destructed, then the whole pillar must fall apart (unless you want Super-Mario-like blocks that float in the air). The question is how to tell the physics simulator that each block in the pillar depends on the one below it, and hence must be destructed when the dependency is destructed. This is the job of *scanDirection*, which is a vector that starts from the center of the block and goes in the specified direction. In case of the- pillar, this vector must point downwards with a magnitude greater than 0.5, which is the distance between the center of the block and its bottom face. Before discussing the mechanism of *scanDirection*, it is important to understand the dependency structure. For each destructible, there is a list of other destructible objects that depend on it. When the dependency is destructed, then all object in its *dependents* list must be destructed as well.

At the beginning of the game, and as we already know, *Start()* function is called. Upon start, each destructible casts a ray with the direction and the magnitude specified in *scanDirection*. What the destructible tries to do by casting this ray is to find a dependency. This dependency must be have *Destructible* script attached, which is checked in line 28. If the condition is satisfied, the destructible which has cast the ray adds itself to *dependents* list inside the destructible which the ray hit. In case of the pillar shown in Illustration 69, *scanDirection* should have a value such as (0, -0.6, 0), so that each scan ray is cast downwards. As a result, each block will add itself to *dependents* list of the block below it, while the bottom most block isn't going to be able to find any dependencies.

After building dependencies among building blocks, the next step is to freeze the block so it does not get affected by physics simulation. Therefore, the *rigidbody.constraints* of the block are saved in *original*, and then changed to *RigidbodyConstraints.FreezeAll*. This guarantees that both position and the rotation of the block are conserved up to the moment we decide to allow them to be changed. The block remains frozen until *Destruct()* function is called (or *Destruct* message is received). To destruct the block, the first thing to do is to free it from any constraints by resetting *rigidbody.constraints* to its original value stored in *original*. After that, the destruction must be propagated to all dependencies by calling *Destruct()* for all elements in *dependents* list. However, it is better to delay the destruction by a tiny amount of time, in order to give the sense of physical relation between the blocks. Finally, it won't hurt if we send an informative message to all other script connected to the block, telling them that destruction just happened. This makes it possible to do some relevant effects, such as playing sound or making some dust. Illustration 70 shows a simple building constructed with destructible building blocks.

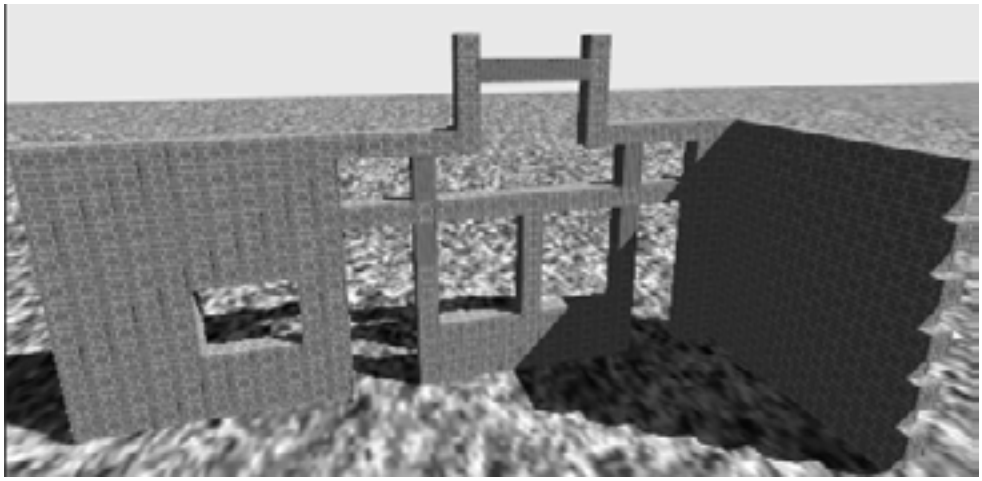


Illustration 70: A building constructed completely from destructible building blocks

The *scanDirection* for each block depends on the blocks around it. White arrows in Listing 71 indicate scan directions of building blocks in left and right walls: each arrow starts from a block and points to its dependency.

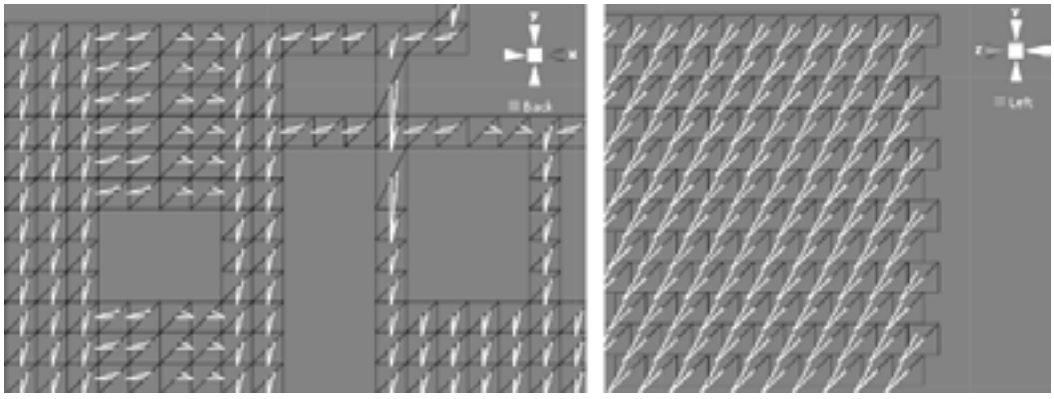


Illustration 71: Comparing block arrangement between left and right walls: the blocks of left wall scan down, left, or right, while the blocks of the right wall scan towards bottom left

As a result, a dependency tree will be built between blocks, which specify what depends on what. The tree that results from the above scans is shown in Illustration 72.

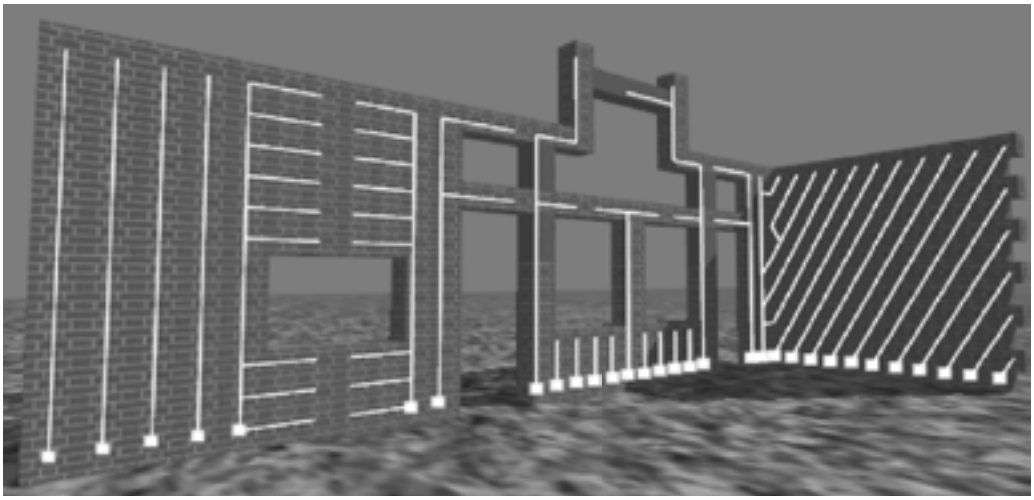


Illustration 72: Dependency trees between building blocks: when the root destructs, the whole tree must eventually be destructed. Roots are displayed as small squares

Now we want to be somehow able to perform the desired destruction on the building. To do this, we are going to use a script that generates an explosion when the left mouse button is clicked. *MouseExploder* script shown in Listing 57 is the second script we need to add to the prefab of our building block.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class MouseExploder : MonoBehaviour {
5.
6.       //How strong is the explosion?
7.       public float explosionForce = 40000;
8.
9.       //Range of explsion effect
10.      public float explosionRadius = 5;
11.
12.      //Position of the explosion
13.      //Relative to the position of the object
14.      //Expressed in world space
15.      public Vector3 explosionPosition = new Vector3(-1, 0, -1);
16.
17.      void Start () {
18.
19.      }
20.
21.      void Update () {
22.
23.      }
24.
25.      //Perform explosion on mouse click
26.      void OnMouseDown(){
27.           //Get all rigid bodies
28.           Rigidbody[] allBodies = FindObjectsOfType<Rigidbody>();
29.
30.           //Calculate the position of the explosion
31.           Vector3 explosionPos = transform.position;
32.           explosionPos += explosionPosition;
33.
34.           //Find bodies within radius, send destruction
35.           //message in case they are destructible,
36.           //and finally apply explosion force to them
37.           foreach(Rigidbody body in allBodies){
38.                float dist =
39.                     Vector3.Distance(
40.                          body.transform.position,
41.                          explosionPos);
42.
43.                if(dist < explosionRadius){
44.                     body.SendMessage("Destruct",
45.                          SendMessageOptions.DontRequireReceiver);
46.
47.                     body.AddExplosionForce(
48.                          explosionForce, //Explosion strength
49.                          explosionPos, //Explosion position
50.                          explosionRadius);//effective radius
51.                }
52.           }
53.      }
54. }
```

Listing 57: A script that generates explosion force based on mouse click

The first thing you can notice in the script is the relatively high magnitude of *explosionForce*, which is reasonable since we are talking about an explosion that can destroy a building. The radius in which the explosion takes effect is set by *explosionRadius*, which means that only objects within this distance are affected by explosion force. This script creates an explosion when a block is clicked. However, to make the explosion more obvious and effective, we move for a short distance from the block position and perform the explosion there. This distance is determined by *explosionPosition*, which is relative the position of the target block that was clicked.

OnMouseClick() handles mouse click on the destructible block by adding explosion force to all surrounding rigid body within the given radius. First step is to get all rigid bodies and store them in *allBodies* array. After that, *explosionPos* is computed by adding *explosionPosition* relative position to the position of the block. The *for* loop in line 37 iterates over all rigid bodies and finds the distance between each body and the computed *explosionPos*. If this distance is less than *explosionRadius*, we send *Destruct* message to the rigid body and then call *AddExplosionForce()*. *AddExplosionForce()* takes 3 arguments: the strength of the explosion which is *explosionForce*, the position of the explosion which is *explosionPos*, and the radius of explosion effect which is *explosionRadius*. The importance of sending *Destruct* message is to unfreeze the rigid body, in case it has *Destructible* script attached to it. Without this step, the explosion isn't going to have any effect on the rigid body. Illustration 73 shows a demo explosion. Notice that the effect of the explosion is purely physical, and has no visual effects such as fire or smoke.

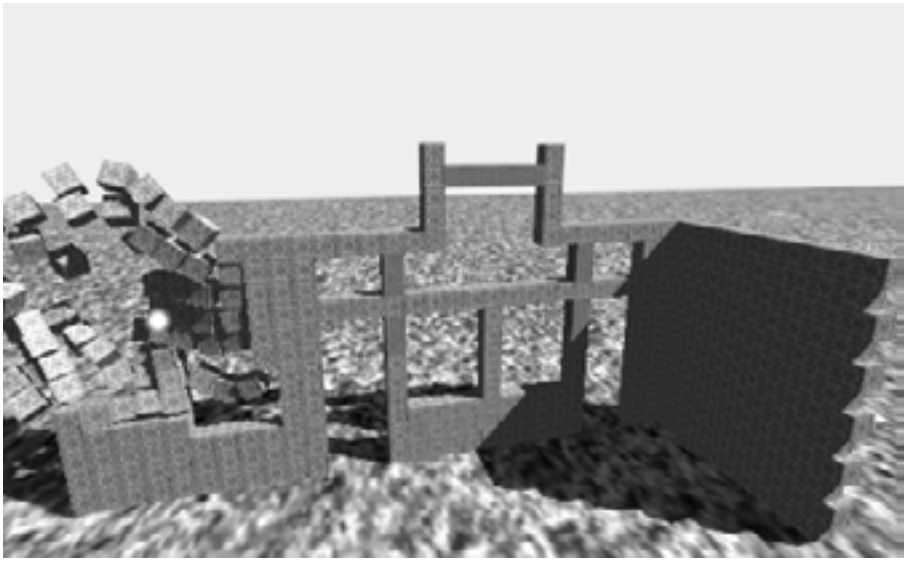


Illustration 73: Effect of the explosion force on destructible blocks: the blocks are pushed away from the position of the explosion, which is shown as bright point

To make our example more fun, I would like to add an interesting feature to our building blocks. What if we become able to reconstruct the whole building after destroying it? This idea can be simply implemented by storing the original position of each block at the beginning, and then returning it smoothly to its position when, say, space bar is pressed. The script *Returner* does this interesting job when attached to the prefab of the building block. This script is shown in Listing 58.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Returner : MonoBehaviour {
5.
6.       //Original position of the object
7.       Vector3 originalPos;
8.       //Original rotation of the object
9.       Quaternion originalRot;
10.      //Is the object currently moving to its original position?
11.      bool returning = false;
12.
13.      void Start () {
14.           //Record original position and rotation
15.           originalPos = transform.position;
16.           originalRot = transform.rotation;
17.      }
18.
19.      void Update () {
20.
21.           //When space is pressed, initialize returning
22.           if(Input.GetKeyDown(KeyCode.Space)){
23.           Return();
24.           }
25.
26.      if(returning){
27.           //During return freese all constraints to prevent
28.           //external forces from affecting the rigid body
29.           if(rigidbody.constraints !=
30.                RigidbodyConstraints.FreezeAll){
31.                //Clear any linear or angular
32.                // velocities to stop object
33.                rigidbody.velocity = Vector3.zero;
34.                rigidbody.angularVelocity = Vector3.zero;
35.                //Now freeze the rigid body
36.                rigidbody.constraints =
37.                     RigidbodyConstraints.FreezeAll;
38.           }
39.
40.           //Smoothly return position and rotation
41.           //to their original values
42.           transform.position =
43.                Vector3.Lerp(transform.position, //from
44.                                  originalPos, //to
45.                                  Time.deltaTime * 3);//amount
```

```
46.
47.           transform.rotation =
48.                Quaternion.Lerp(
49.                                transform.rotation, //from
50.                                originalRot, //to
51.                                Time.deltaTime * 3);//amount
52.
53.           //If the object is too near, manually set original values,
54.           //and set returning to false
55.           float remaining =
56.                           Vector3.Distance(transform.position, originalPos);
57.                if(remaining < 0.01f){
58.                     transform.position = originalPos;
59.                     transform.rotation = originalRot;
60.                     returning = false;
61.                }
62.           }
63.      }
64.
65.      //Return the object to its original position
66.      public void Return(){
67.           returning = true;
68.      }
69. }
```

Listing 58: A script to reset the position and rotation of building blocks when space is pressed

The first thing this script does is storing the original (initial) position and rotation of the block in *originalPos* and *originalRot* variables. Notice that rotation is stored in a variable of type *Quaternion*. During *Update()* we scan for space key press. If it is pressed, we call *Return()* function. This function simply sets *returning* flag to *true*, in order to let the object move towards its original position in each frame. If *returning* is *true*, the object must be moved towards *originalPos* and rotated towards *originalRot*. Before performing the translation and rotation, we have to remove all forces and prevent them from affecting the rigid body. Therefore, we ensure that the rigid body is frozen and clear all velocities affecting it (lines 29 through 38). After that, we use *Vector3.Lerp* and *Quaternion.Lerp* to translate and rotate the object smoothly. By multiplying *Time.deltaTime* by 3, we ensure a faster return. Finally, we find the remaining distance to reach the original position, if it is less than 0.01, we directly assign the original values of position and rotation to the transform. The final result can be experimented in *scene19* in the accompanying project.

4.7 Breakable objects

In the previous section, we learned how to construct a building using destructible building blocks. However, for smaller destroyable (or breakable) objects, we need a more appropriate approach. For objects like boxes that player break in order to get items, or glass windows that collapse when shot and turn into small pieces; we need a different method of destruction. This method can be used for one-way destruction, in which the destroyed object cannot be constructed again. The idea is to remove the original object from the scene, and instantly create a collection of pieces that are smaller in size and have the same texture of the original object. These pieces can be made using prefabs as we are going to see. Consider the building we constructed in the previous section, what about adding some glass windows? First of all, we need a prefab for the window, which must be breakable (and destructible as well). The glass block might look like Listing 47.

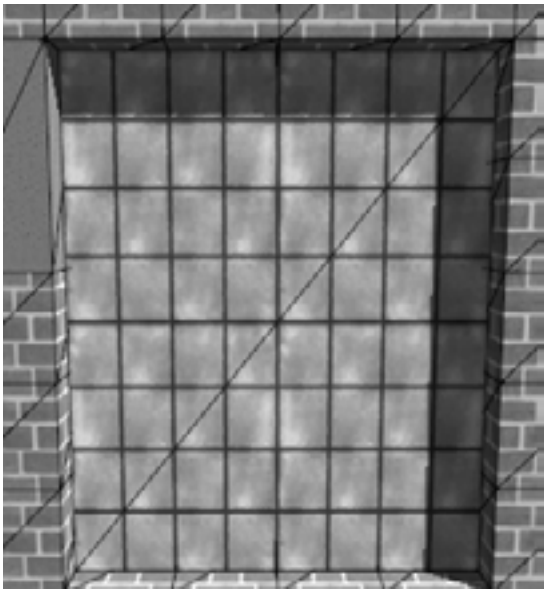


Illustration 74: Glass block to be used as breakable window

We need to add *Destructible* script to this window in order to have it behave like any other block in the building. Additionally, we need to write a new script that turns this window into a breakable object. Consequently, we are going to call this script *Breakable*, and it is shown in Listing 59.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class Breakable : MonoBehaviour {
5.
6.       //Objects to generate when broken (pieces)
7.       public GameObject[] pieces;
8.
9.       //How many times each piece should be generated?
10.      public int pieceCopies = 5;
11.
12.      //How strong this breakable explodes?
13.      public float explosionPower = 10;
14.
15.      //How long to keep pieces?
16.      public float pieceLifetime = 10;
17.
18.      void Start () {
19.
20.      }
21.
22.      void Update () {
23.
24.      }
25.
26.      //Break the breakable
27.      public void Break(){
28.           //Generate the pieces with spiecified count
29.           foreach(GameObject piecePrefab in pieces){
30.                for(int i = 0; i < pieceCopies; i++){
31.                     GameObject piece =
32.                          (GameObject)Instantiate(piecePrefab);
33.
34.                     //Get a random position within
35.                     //the borders of the object
36.                     Vector3 borders = transform.localScale;
37.                     Vector3 randPos;
38.
39.                     randPos.x =
40.                      Random.Range(-borders.x * 0.5f, borders.x * 0.5f);
41.
42.                     randPos.y =
43.                      Random.Range(-borders.y * 0.5f, borders.y * 0.5f);
44.
45.                     randPos.z =
46.                      Random.Range(-borders.z * 0.5f, borders.z * 0.5f);
47.
48.                     //Place the piece at the random position
49.                     piece.transform.position =
50.                          transform.position + randPos;
```

```
51.
52.                         Vector3 explosionPos = transform.position;
53.                         float explosionRad = transform.localScale.magnitude;
54.
55.                         piece.rigidbody.AddExplosionForce(explosionPower,
56.                                                           explosionPos,
57.                                                           explosionRad);
58.
59.                         //If life time is spiecified,
60.                         //destroy the piece after it
61.                         if(pieceLifetime > 0){
62.                               Destroy(piece, pieceLifetime);
63.                         }
64.                   }
65.             }
66.
67.             //Inform other scripts that the object has been broken
68.             SendMessage("OnBreak", SendMessageOptions.DontRequireReceiver);
69.
70.             //Finally, destroy the breakable
71.             Destroy(gameObject);
72.
73.       }
74. }
```

Listing 59: A script to make an object breakable

Since we are going to replace this object with a number of pieces when broken, we need to provide the prefabs of these pieces in *pieces* array. The variable *pieceCopies* specifies how many times each piece prefab must be generated. For example if we have two prefabs and set *pieceCopies* to 5, we end up with a total of 10 pieces when the object is broken. *explosionPower* determines how strong is the generated explosion when this object brakes. This explosion is going to be used only to spread the pieces after destruction, and have no effect on other objects near the breakable. Finally, we can set a time after which remove the pieces from the scene using *pieceLifeTime*. If the value of this variable is zero or less, the pieces are going to be permanent.

We call *Break()* function once whenever we want to break the object. This functions does three jobs: generates the pieces, inform other scripts that breaking has happened by sending *OnBreak* message, and finally destroys the object. The interesting part are the two nested *foreach* and *for* loops. The outer loops iterates over all prefabs in *pieces* array, and the second one iterates *pieceCopies* times, in order to instantiate the required number of objects from each prefab. The position where each piece is instantiated is random, but it must fall within the borders of the original object. These borders are determined using the local scale of the object. After moving the piece to its randomly generated position, we apply the explosion force to it. Keep in mind that the explosion is positioned at the center of the original object. As a result, all pieces will be pushed away from the center by the explosion force. When the piece is ready, we check whether *pieceLifeTime* is greater than zero, and eventually destroy the piece after the specified time. Once all pieces are generated, we send *OnBreak* message to inform other scripts that the object has been broken, and then destroy the original object.

In Unity, calling *Destroy* does not immediately destroy the object. The destruction is rather delayed until the end of the frame. In our case, this gives other scripts attached to the breakable object the chance to respond to *OnBreak* message.

To activate these breakable windows in the building we made in *scene19*, we must add a script that works as a bridge between *Destructible* and *Breakable* scripts. This script must be added to breakable objects in order to send *Break* message when *OnDestruction* message is received. The script *BreakOnDestruct* is shown in Listing 60. The complete building with destructibles and breakables is in *scene19* in the accompanying project.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class BreakOnDestruct : MonoBehaviour {
5.
6.       void Start () {
7.
8.       }
9.
10.      void Update () {
11.
12.      }
13.
14.      //Simply receive destruction message
15.      //and send break message
16.      void OnDestruction(){
17.           SendMessage("Break", SendMessageOptions.DontRequireReceiver);
18.      }
19. }
```

Listing 60: A simple script that receives *OnDestruction* message and reacts by sending *Break* message

Exercises

1. The vehicle we made in section 4.2 is a four-wheel drive vehicle. Change it to two-wheel drive so that only the back wheels are connected to the motor.
2. Create two boxes with rigid bodies and make the vehicle we made in section 4.2 carry them as it moves. What are the necessary changes you have to make on the vehicle in order to be able to carry such objects?
3. Implement a transmission control for the vehicle in section 4.2, so that the player can change the speed manually. Remember that a higher shift increases the maximum speed and decreases motor torque. Use A key to shift up and Z key to shift down. Implement at least 4 different speeds.
4. Use a modified version of *RaycastShooter* to implement a shotgun. This shotgun must be usable with *GunInput* script, and must cast at least 5 rays in each shot. These rays must originate from the muzzle and diffuse randomly by some degrees on x and y axes. Finally, disable continuous shooting for the shotgun.
5. Write a script that destroys the object if it takes a shot with damage more than 50. Add your object to *scene17* and test it.
6. Construct a tower like in Illustration 75, and modify scan directions of the destructible blocks to build logically correct relations. For example, the whole tower must fall when the pillar is destructed.

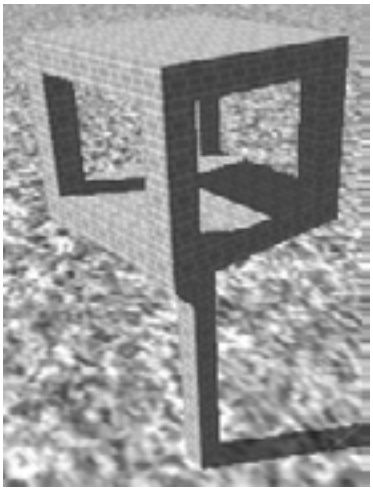


Illustration 75: The tower of Exercise 6

7. Create a breakable wood box that breaks when it receives *OnRaycastHit* message of *RaycastShooter* (Listing 49 in page 128). You need to use *Breakable* script from Listing 59 (page 150). Use pieces of appropriate numbers and sizes. As a plus, try to add a permanent piece that represents valuable object for the player.

5 Advanced Game Logic

Introduction

In chapter 3, we learned some basic mechanics that lots of games need. We continue in this chapter with even more mechanics and learn how to program them. The topics introduced in this chapter have been delayed until we discuss physics simulation and collision detection, since these topics depend on detecting collisions. For instance, it does not make sense to discuss doors and locks if these doors do not block player's movement.

After completing this chapter, you are expected to:

- Make doors, locks, and keys
- Program simple puzzles and unlock combinations
- Program player's health, lives, and score
- Program different types of weapons with ammo and reload mechanism

5.1 Doors, locks, and keys

In this section we are going to discuss two types of doors: rotating doors and sliding doors. Rotating doors are just like ordinary doors we usually see: they rotate around y axis and their rotation axis is on the left or right end of the door. On the other hand, sliding doors usually move in one dimension (right, left, or up) just like elevator doors.

Let's begin with rotating doors. To implement such doors with ease, we can use a new physics component called *Hinge Joint*. This component is specific for object that have limited freedom of movement, such as doors. To begin, we can create a simple room with floor and four walls; where one of these walls leads outside through a door opening. In this opening we locate our door like in Illustration 76. After that, we need to add a rigid body component to the door and configure it as in Illustration 77. Notice that we increase both drag and angular drag to make the door movement speed reasonable (otherwise it will feel too light).

Illustration 76: A room with a basic rotating door

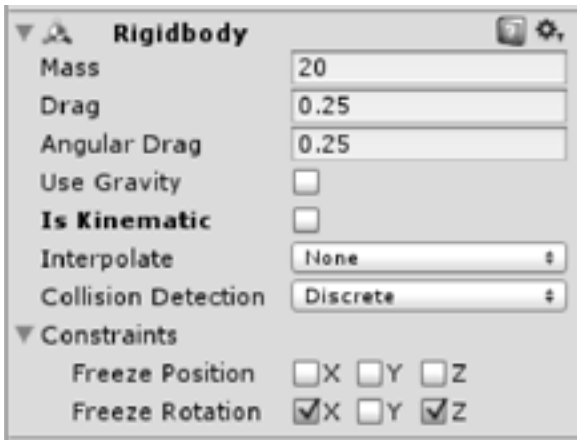


Illustration 77: Rigid body configuration of the rotating door

Finally, we need to add the hinge joint component to the door and configure it according to Illustration 78. Hinge joint is a physics component that is affected by external forces. Therefore, we will not need to press any keyboard key or mouse button to open the door. Alternatively, we have to exert a force with an appropriate magnitude.

The *Hinge Joint* component can be found in *Component* > *Physics* > *Hinge Joint* menu item.

Illustration 78: Configuring hinge joint component to create a rotating door

As you see, this is a relatively large component with lots of options to deal with. However, we are interested in some options that allow us to get the desired functionality. *Anchor* and *Axis* values specify the position and direction of the rotation axis. Since we are making a rotating door, the axis need to be on the side. If you consider z size of the door as its thickness, and x size of the door as its width, then the position of the rotation axis is (0.5, 0, 0), which is the right end of the door. Similarly, the direction of this axis needs to be (0, 1, 0), so that the axis goes along y axis. The second change we need is activating *Use Spring*, which generates a force that returns the door to its original position when there are no external forces affecting it. The related *Spring* and *Damper* values must be appropriately set, so they are neither too strong nor too weak. The values you see in Illustration 78 have been configured to be appropriate for the physics character we created in section 4.3. Finally, we have to activate *Use Limits*, in order to set maximum and minimum degrees of door rotation. In this case, we make a bi-directional rotating door that rotates 180 degrees. In other words, the door can be pushed from both sides and rotates up to 90 degrees. You may add a physics character with first person control to test the door.

Now we are going to lock this door and create a key. The player must posses this key in order to open the door and pass through. The key is going to be collectable and, when collected, is going to be added to the inventory box of the player. Therefore, we need a script similar to *Collectable* script we have created earlier (Listing 26 page 76), in addition to *InventoryBox* (Listing 29 page 79). For this latter script, we have to add a list to store the keys that the player has. These keys are simply strings. The modified version of *InventoryBox* is shown in Listing 61.

```
1.  using UnityEngine;
2.  using System.Collections.Generic;
3.
4.  public class InventoryBox : MonoBehaviour {
5.
6.       //How much money does the player have?
7.       public int money = 0;
8.
9.       //What keys does the player have?
10.      public List<string> keys;
11.
12.      void Start () {
13.
14.      }
15.
16.      void Update () {
17.
18.      }
19. }
```

Listing 61: The modified version of *InventoryBox* script

The mechanism we are going to use is the following: each key has a "secret text" that must be unique, and this text can be used to open all doors that are locked with the same secret text. The list *keys* in *InventoryBox* stores secret texts of the keys that player currently has. Now we have to 1) create a collectable key that gives the player the secret text, and 2) create a lock that prevents door from opening until the secret text is provided by the player.

Let's begin with the collectable key: we need to build a new collectable/collector mechanism, but this time we are going to make use of collision detection. Therefore, we do not need to iterate over all collectables in the scene and measure their distances like we did in section 3.2. Alternatively, we simply create a key script that responds to *Collect* message by giving the collector (owner) a new key in a form of secret text. Listing 62 shows *CollectableKey* script which implements the described function.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class CollectableKey : MonoBehaviour {
5.
6.       //Key to give to the player
7.       public string key;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.
15.      }
16.
17.      //Receive collect message
18.      public void Collect(GameObject owner){
19.           //Find the inventory box of the owner
20.           //Give the key to the owner by adding it
21.           //to the list of keys the owner has
22.           InventoryBox box = owner.GetComponent<InventoryBox>();
23.           if(box != null){
24.                box.keys.Add(key);
25.                //Finally, destroy the key object
26.                Destroy (gameObject);
27.           }
28.      }
29. }
```

Listing 62: A script for collectable key that gives the collector a secret text

What does the script simply do is to verify that *owner* has an inventory box. If this is true, it adds the secret key stored in *key* variable to the list of keys in the inventory box (*box.keys*). Finally, the collectable key is destroyed and removed from the scene, which is important to give the player the impression that he has already collected the key. The question now is: how the player is going to collect the key? The answer might vary depending on the situation: he might simple pick it from the floor, or it can be given to him by another character in the game, and so on. Generally, any event that ends by sending *Collect* message to key object and providing player's character as *owner* will eventually give the player the key. In our case, we simply collide with the key object and collect it. Consequently, we need a script that sends *Collect* message upon collision between the player and the key. This script is *CollisionCollector* shown in Listing 63.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class CollisionCollector : MonoBehaviour {
5.
6.       void Start () {
7.
8.       }
9.
10.      void Update () {
11.
12.      }
13.
14.      //Collect the collectable on collision
15.      void OnCollisionEnter(Collision col){
16.           SendCollectMessage(col.gameObject);
17.      }
18.
19.      //Collict on trigger hit
20.      void OnTriggerEnter(Collider col){
21.           SendCollectMessage(col.gameObject);
22.      }
23.
24.      void SendCollectMessage(GameObject target){
25.           //Send collect message to the colliding object.
26.           //Provide self as owner of what is to be collected
27.           target.gameObject.SendMessage("Collect",
28.                gameObject, //owner
29.                SendMessageOptions.DontRequireReceiver);
30.      }
31. }
```

Listing 63: Collector script based on collisions

This script handles the two types of possible collisions by handling *OnCollisionEnter* and *OnTriggerEnter* messages. Consequently, it sends *Collect* message to the colliding object and provides itself as the owner. This results in a generic collecting script that can collect any object as long as it handles *Collect* message, and not only keys. Before carrying on, it is a good idea to revise the list of scripts we need: we will use a *PhysicsCharacter* with *FPSInput*. These two scripts should be attached to capsule that represents the character and has the camera added as a child. In order to collect collectables, we need both *InventoryBox* and *CollisionCollector* scripts. Now we have to make an object that resembles the key we need to collect, and add the *CollectableKey* script to it. For example, you can make a simple key shape like in Illustration 79.

Illustration 79: A simple key shape to be used as collectable key object

After attaching *CollectableKey* script to the key object, we need to specify the secret text that uniquely identifies the key/lock pair and type the value in *key* field. For this example I use "door1". When the player character collides with this key, the value *door1* will be added to *keys* list inside *InventoryBox*. To complete the demo, we need to add a lock to our door. Remember that we have used *Hinge Joint* component to create the door, which makes door movement under the control of physics simulator. Therefore, to lock the door we need to freeze its position and rotation. This task is performed by *PhysicsDoorLock* script shown in Listing 64.

```
1.  using UnityEngine;
2.  using System.Collections.Generic;
3.
4.  public class PhysicsKeyLock : MonoBehaviour {
5.
6.       //Unique string to unlock this lock
7.       public string unlockKey;
8.
9.       void Start () {
10.      Lock ();
11.      }
12.
13.      void Update () {
14.
15.      }
16.
17.      //Lock the key by setting the rigid body to kinematic
18.      public void Lock(){
19.           rigidbody.isKinematic = true;
20.      }
21.
22.      //Try to unlock using the provided keys
23.      public void Unlock(ICollection<string> keys){
24.
25.           if(!rigidbody.isKinematic){
26.                return;
27.           }
28.
29.           //If one of the keys match, then unlock
30.           foreach(string key in keys){
31.                if(unlockKey.Equals(key)){
32.                      //Tell other scripts that unlocked succeded
33.                      SendMessage("OnUnlock",
34.                            SendMessageOptions.DontRequireReceiver);
35.
36.                       rigidbody.isKinematic = false;
37.                       return;
38.                }
39.           }
40.
41.           //Tell other scripts that unlocking has failed
42.           SendMessage("OnUnlockFail",
43.                SendMessageOptions.DontRequireReceiver);
44.      }
45. }
```

Listing 64: A script to lock physics door with provided string unlock key

By setting *isKinematic* property of *rigidbody* to *true*, the script tells the physics simulator that no external force can alter the position or rotation of the door. Nevertheless, the door must still be able to collide and block other objects. The script starts by calling *Lock()* function, which in turn sets *rigidbody.isKinematic* value to *true*. Anyone tries to unlock the door must provide a collection of strings (keys) that he has. If one of these keys matches *unlockKey*, the door is unlocked. Notice that we use *ICollection* generic list, which is the most generic type of collections available. As a result, the function can be called using *List<string>* or *string[]* without problems. The value of *rigidbody.isKinematic* determines whether the door is locked or not. If the door is already unlocked, the value is *false*. If the door has not yet been unlocked, every key in the provided *keys* collection is compared with *unlockKey*. If a match is found, the door is unlocked by resetting *rigidbody.isKinematic* back to *false*. Before that, the script informs other scripts about unlock by sending *OnUnlock* message. However, if none of the provided keys matches *unlockKey*, the function returns without unlocking the door and sends *OnUnlockFail*.

All we have to do now is to attach the script to the physics door we've made and set its *unlockKey* to "door1", so that it matches *key* value of the collectable key. The final step is to initialize unlock attempt. One of the options is to try to unlock the door when the player character touches it. To implement this option, we have to write a script that handles collision between player character and the door and eventually try to unlock the door. This script is *TouchUnlocker* shown in Listing 65.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class TouchUnlocker : MonoBehaviour {
5.
6.       void Start () {
7.
8.       }
9.
10.      void Update () {
11.
12.      }
13.
14.      //Send unlock message to colliding object
15.      void OnCollisionEnter(Collision col){
16.           //Get the inventory box
17.           InventoryBox box = GetComponent<InventoryBox>();
18.
19.           //Try all keys in the inventory box with the lock
20.           col.gameObject.SendMessage("Unlock",
21.                box.keys, //Colletion of keys to try
22.                SendMessageOptions.DontRequireReceiver);
23.
24.      }
25.
26. }
```

Listing 65: A script that tries to unlock the door when the player touches it

Whenever the player collides with an object, this script send *Unlock* message to that object and provides it with the list of keys stored in player's inventory box. If the colliding object is a locked door, it will try all provided keys to unlock itself. However, if the colliding object is not a door, the message is simply ignored and nothing evil happens. In *scene20* in the accompanying project, you can find two functional rotating doors: unlocked and locked with a collectable key.

The second type of doors we are going to implement is sliding door. This time we use a custom script instead of a physics component to implement the desired door movement. The generic functionality this script has to provide is moving the door along its local x axis. However, to make a real door, we need to do more than that. First of all, we need to provide the functionality of a generic door, such as opening, closing, locking and unlocking. In the case of rotating door, hinge joint properties did the job for us. We need, however, to do handle these situations by ourselves now. Therefore, we need *GeneralDoor* script that represents an abstract door, regardless of the actual way of opening and closing it. This script is shown in Listing 66.

```
1.  using UnityEngine;
2.  using System.Collections.Generic;
3.
4.  public class GeneralDoor : MonoBehaviour {
5.
6.       //Is the door initially open?
7.       public bool initiallyOpen = false;
8.
9.       //Key to unlock the door
10.      public string unlockKey;
11.
12.      //Internal state storage
13.      bool isOpen;
14.
15.      //Internal state of lock
16.      bool locked;
17.
18.      void Start () {
19.           //Lock the door if there is an unlock key provided
20.           locked = !string.IsNullOrEmpty(unlockKey);
21.           //Set the initial state of the door
22.           isOpen = initiallyOpen;
23.      }
24.
25.      void Update () {
26.
27.      }
28.
29.      //Open the door if not locked
30.      public void Open(){
31.           if(!locked){
32.                isOpen = true;
33.           }
34.      }
35.
36.      //Close the door if not locked
37.      public void Close(){
38.           if(!locked){
39.                isOpen = false;
40.           }
41.      }
42.
43.      //Lock the door
44.      public void Lock(){
45.           locked = true;
46.      }
47.
48.      //Try to unlock the door using provided keys
49.      public void Unlock(ICollection<string> keys){
50.           //Check if it already unlocked
51.           if(!IsLocked()){
52.                return;
53.           }
54.           //Try all keys to unlock the door
55.           foreach(string key in keys){
```

```
56.                 if(key.Equals(unlockKey)){
57.                       //Tell other scripts that the door has been unlocked
58.                       SendMessage("OnUnlock",
59.                             SendMessageOptions.DontRequireReceiver);
60.
61.                       locked = false;
62.                       return;
63.                 }
64.           }
65.           //Tell other scripts that unlocking failed
66.           SendMessage("OnUnlockFail",
67.                 SendMessageOptions.DontRequireReceiver);
68.     }
69.
70.     //Is the door currently locked?
71.     public bool IsLocked(){
72.           return locked;
73.     }
74.
75.     //Is the door currently open?
76.     public bool IsOpen(){
77.           return isOpen;
78.     }
79.
80.     //Switch the state of the door
81.     public void Switch(){
82.           if(IsOpen()){
83.                 Close();
84.           } else {
85.                 Open();
86.           }
87.     }
88. }
```

Listing 66: A script that handles basic functions of a door regardless of the actual implementation of these functions

You might have noticed that this script internally handles the state of the door, and provides public functions to check or modify this state. All functions may be called without any parameters, and their effect on the internal state is instant. This is true for *Open()*, *Close()*, and *Lock()* functions. The only exception is *Unlock()*; which requires the caller to provide a list of keys, and the state *locked* is not changed to *false* unless one of these keys matches *unlockKey*. The question now is how to make use of these functions to make an actual sliding door? The answer is simple: we make another script that continuously calls *IsOpen()* and *IsClose()* and consequently modifies the position of the door towards open or close positions. This script is *SlidingDoor* shown in Listing 67.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(GeneralDoor))]
5.  public class SlidingDoor : MonoBehaviour {
6.
7.        //Relative new position of door when opened
8.        public Vector3 slidingDirection = Vector3.up;
9.        //Speed of door movement
10.       public float speed = 2;
11.       //Store close and open positions
12.       Vector3 originalPosition, slidingPosition;
13.       //Reference to general door script
14.       GeneralDoor door;
15.       //Current state of the door
16.       SlidingDoorState state;
17.
18.       void Start () {
19.             //Initialize the variables
20.             door = GetComponent<GeneralDoor>();
21.             originalPosition = transform.position;
22.             slidingPosition = transform.position + slidingDirection;
23.             state = SlidingDoorState.close;
24.       }
25.
26.       void Update () {
27.             if(door.IsOpen()){
```

```
28.                 //The door must be open
29.                 if(state != SlidingDoorState.open){
30.                       //The door is not open, move it
31.                       //smoothly towards open position
32.                       transform.position =
33.                                   Vector3.Lerp(
34.                                   transform.position,
35.                                   slidingPosition,
36.                                   Time.deltaTime * speed);
37.
38.                       float remaining =
39.                             Vector3.Distance(
40.                                   transform.position, slidingPosition);
41.
42.                       //Check if door reached open position
43.                       if(remaining < 0.01f){
44.                             //Open position reached:
45.                             //change state of the door
46.                             state = SlidingDoorState.open;
47.                             transform.position = slidingPosition;
48.                             //Inform other scripts about open completion
49.                             SendMessage("OnOpenComplete",
50.                                   SendMessageOptions.DontRequireReceiver);
51.
52.                       } else if(state != SlidingDoorState.openning){
53.                             //Door just started to open,
54.                             //send a message to inform about that
55.                             SendMessage("OnOpenStart",
56.                                   SendMessageOptions.DontRequireReceiver);
57.
58.                             state = SlidingDoorState.openning;
59.                       }
60.                 }
61.           } else {
62.                 //The door must be close
63.                 if(state != SlidingDoorState.close){
64.                       //The door is not close, move it
65.                       //smoothly towards close position
66.                       transform.position =
67.                             Vector3.Lerp(
68.                                   transform.position,
69.                                   originalPosition,
70.                                   Time.deltaTime * speed);
71.                       float remaining =
72.                             Vector3.Distance(
73.                                   transform.position, slidingPosition);
74.
75.                       //Check if door reached close position
76.                       if(remaining < 0.01f){
77.                             //Close position reached:
78.                             //change state of the door
79.                             state = SlidingDoorState.close;
80.                             transform.position = originalPosition;
81.                             //Inform other scripts about close completion
82.                             SendMessage("OnCloseComplete",
```

```
83.                                  SendMessageOptions.DontRequireReceiver);
84.
85.                        } else if(state != SlidingDoorState.closing){
86.                              //Door just started to close,
87.                              //send a message to inform about that
88.                              SendMessage("OnCloseStart",
89.                                    SendMessageOptions.DontRequireReceiver);
90.
91.                              state = SlidingDoorState.closing;
92.                        }
93.                  }
94.            }
95.      }
96.
97.      void OnCollisionEnter(Collision col){
98.            if(state == SlidingDoorState.closing){
99.                  //Something interrupted the door while closing
100.                 //Inform about that
101.                 SendMessage("OnCloseInterruption",
102.                       col.gameObject,
103.                       SendMessageOptions.DontRequireReceiver);
104.           }
105.     }
106.
107.     //Enumeration of different door states
108.     enum SlidingDoorState{
109.           open, close, openning, closing
110.     }
111.  }
```

Listing 67: A script that implements sliding door functionality

Before discussing the details of this script, notice *RequireComponent* annotation we used before declaring the class. This annotation requires the game object to which *SlidingDoor* script is attached to have a *GeneralDoor* script as well. If you attach *SlidingDoor* script to an object, Unity automatically attaches *GeneralDoor* script if does not exist. Similarly, if you try to remove *GeneralDoor* script from an object while *SlidingDoor* is attached to it, the removal is refused by Unity. We benefit from this mechanism because *SlidingDoor* completely depends on *GeneralDoor* and cannot be used alone.

The *slidingDirection* vector determines the distance the door moves along its local axes when it is opened. For example, if the sliding direction is (0, 2, 0), the door is going to move two meters up when it is opened. The variable *speed* controls the speed of the door when it opens or closes. Opening and closing the door is in fact a process of smoothly moving it between *originalPosition* and *slidingPosition*. The initial position of the door when *Start()* is called is taken as *originalPosition* (close position). On the other hand, *slidingPosition* (open position) is computed by adding *slidingDirection* to *originalPosition*. In addition to *door* which references the attached *GeneralDoor*, we implement an internal state management by using the enumerator *SlidingDoorState* (line 107). The variable *state* of type *SlidingDoorState* tells us what the sliding door is doing at any given moment (opened, closed, opening, or closing). The initial state is set according to *door.initallyOpen*. Illustration 80 shows a double sliding door that has two parts with opposite sliding directions.

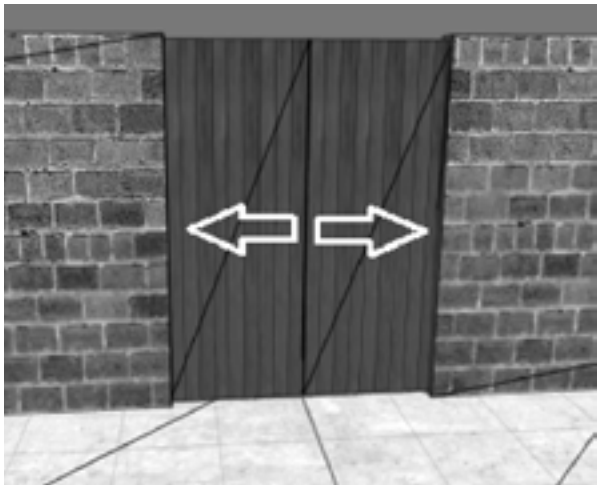


Illustration 80: A sliding door with two parts. Arrows indicate sliding direction of each part.

During *Update()*, the script checks the value of *door.IsOpen()*. If this function returns *true*, it means that the door must be open or, if it is not, must be opened. Therefore, we check *state*, and if its not *SlidingDoorState.open*, then we have three possibilities: the door is closed (*SlidingDoorState.closed*), being closed (*SlidingDoorState.closing*), or being opened (*SlidingDoorState.opening*). In the first two cases we have to change *state* to *SlidingDoorState.opening*, and simultaneously send *OnOpenStart* message to inform other scripts that the door has just started to open. On the other hand, if the state is already *SlidingDoorStart.openning*, then we smoothly move the door towards *slidingPosition*. The smooth movement is, as we have seen earlier, performed using *Vector3.Lerp()* function. Door movement has a dead zone of 0.01, after which the position of the door is set to *slidingPosition*. Same steps go in the other direction if *door.IsOpen()* returns *false*, since in that case the door must be closed. During closing, we keep an eye on *OnCollisionEnter* event. This allows us to detect anything that might block the door as it closes. A possible reaction is to reopen the door, or destroy the colliding object. The latter option allows the player to use the door as a weapon to eliminate enemies.

5.2 Puzzles and unlock combinations

This section is an extension of section 5.1, in which we will continue to work on the sliding door we have already made. This sliding door is going to be locked using an electrical central lock, and the player has to solve a simple puzzle to unlock the door and open it. What we need to do now is to add *SlidingDoor* script to the two parts of the door and configure their sliding directions to, say, (1.2, 0, 0) for the right part and (-1.2, 0, 0) for the left part. *GeneralDoor* script considers the door as unlocked if the value *unlockKey* is empty. Since we need locked doors, we need to put some value such as “door2” for this variable for both parts. Now we can create our central lock. This can be an empty game object that has the necessary scripts attached to it. The first script is the part of the lock that controls door parts. *CentralLock* script is shown in Listing 68.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class CentralLock : MonoBehaviour {
5.
6.       //Door object(s)
7.       public GeneralDoor[] targetDoors;
8.
9.       //Keys to unlock the doors
10.      public string[] keys;
11.
12.      //Should the doors be opened after unlocking?
13.      public bool autoOpen = true;
14.
15.      //Should the doors be closed before locking?
16.      public bool autoClose = true;
17.
18.      void Start () {
19.
20.      }
21.
22.      void Update () {
23.
24.      }
25.
26.      //Locks all target doors
27.      public void LockAll(){
28.           foreach(GeneralDoor door in targetDoors){
29.                if(autoClose){
30.                     door.Close();
31.                }
32.                door.Lock();
33.           }
34.      }
35.
36.      //Unlocks all target doors using available keys
37.      public void UnlockAll(){
38.           foreach(GeneralDoor door in targetDoors){
39.                door.Unlock(keys);
40.                if(autoOpen){
41.                     door.Open();
42.                }
43.           }
44.      }
45. }
```

Listing 68: A script that centrally controls locking/unlocking of multiple doors

This script references an array of doors *targetDoors* and has another array of keys to unlock them called *keys*. When *LockAll()* function is called, the script iterates over all referenced doors and calls *door.Lock()*. If *autoClose* option is selected, every door is closed before being locked. On the other hand, *UnlockAll()* function tries to unlock all doors by trying all keys on each one of them and, if *autoOpen* is selected, opens them. By having multiple keys, we can reference doors that does not necessarily have the same unlock secret text. This can be useful for a scenario in which you wish to create a central control room with secret entrance, and allow the player to unlock all the doors in the level from inside this room. Otherwise, the player has to find the key for each door to unlock it. An important detail to notice here that we reference doors through *GeneralDoor* script (the array *targetDoors* has the type *GeneralDoor[]*). This gives us the opportunity to reference multiple types of doors, not only sliding doors.

We can now attach this script to an empty game object, and then add both parts of the sliding door to its *targetDoors*. The second step would be adding "door2" unlock text to *keys* array. We keep both *autoOpen* and *autoClose* checked, so that all we have to do to open our sliding door is to call *UnlockAll()*. The question now is: who is going to call this function? And when it is going to be called? The answer is the puzzle system we are going to build shortly. Before introducing the programmatic details of the puzzle, let's briefly discuss its logic. The puzzle has four buttons, which can be switched between two color states: red and green. To unlock the door, the player must find the correct red/green combination between these four buttons (if you are curious about the total number of possible combinations, it is 4 to the power 2 = 16). These buttons can be arranged around the door like in Illustration 81.

Illustration 81: The four buttons of unlock puzzle arranged around the sliding door

Each one of these buttons must be switchable by the player. Therefore, we are going to reuse *SwitchableTrigger* script (Listing 35 page 90) and *TriggerSwitcher* script(Listing 37 page 92), which we created in section 3.4. Recall that adding *SwitchableTrigger* script has the function *SwitchState()*, which cycles between different states and can send different messages upon every switch. Additionally, *TriggerSwitcher* script gives the player the ability to activate these triggers by pressing E key. Whenever the player switches a puzzle button, we need to perform three tasks: first, we have to change the color of the switched button from red to green or vice-versa. Second, we have to change a global state that manages all switches and tests whether the unlock combination has been matched. Finally, we have to try to open the door, to see if the combination worked. This means that the door opens automatically once the player gives the correct combination, so he do not have to reach the door and try to open it every time.

So let's begin with the easiest part, which is changing the color of the switch. Listing 69 shows *ColorCycler* script, which simply cycles the main color of the material between the elements of a provided array of colors.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ColorCycler : MonoBehaviour {
5.
6.       //Color to cycle between
7.       public Color[] colors;
8.
9.       //index of the current color
10.      public int currentColor = 0;
11.
12.      void Start () {
13.           renderer.material.color = colors[currentColor];
14.      }
15.
16.      void Update () {
17.
18.      }
19.
```

```
20.      //Cycles to next color in the array
21.      public void CycleColor(){
22.           if(colors.Length > 0){
23.                currentColor++;
24.
25.                if(currentColor == colors.Length){
26.                     currentColor = 0;
27.                }
28.
29.                renderer.material.color = colors[currentColor];
30.           }
31.      }
32. }
```

Listing 69: A script that cycles the color of the object

Now all we have to do is to attach *SwitchableTrigger* script to each button (or easier: make a button prefab) and set the number of states to 2. When each state is activated, it sends *CycleColor* message to itself. Next we have to add *ColorCycler* script to the button and add red and green to *colors* using the inspector. Our button is now ready and cycles between red and green colors when switched. We have to have four copies of this button in the scene, and find a method to combine their states logically to form a puzzle. The puzzle itself can be any script that runs any logic we want, given that it sends *UnlockAll* to *CentralLock* script when certain condition is met. For our specific puzzle, we need a script that compares the colors of the four buttons with an internally stored unlock sequence of colors. If the colors match the sequence, *UnlockAll* message is sent to *CentralLock*, otherwise *LockAll* message is sent. This script is *ColorCodePuzzle* shown in Listing 70.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class ColorCodePuzzle : MonoBehaviour {
5.
6.       //Unlock sequence
7.       public Color[] unlockCode;
8.
9.       //Where to get input code
10.      public Renderer[] colorSources;
11.
12.      //Message to send upon code match
13.      public TriggerMessage[] matchMessages;
14.
15.      //Messages to send upon code mismatch
16.      public TriggerMessage[] mismatchMessages;
17.
18.      void Start () {
19.
20.      }
21.
22.      void Update () {
23.
```

```
24.      }
25.
26.      //Compare both codes
27.      public void CompareCodes(){
28.           //Assume combinations match
29.           bool match = true;
30.
31.           //Get combination from sources and compare it with unlock code
32.           for(int i = 0; i < colorSources.Length; i++){
33.                //One difference is enough to deny match
34.                if(!colorSources[i].material.color.Equals(unlockCode[i])){
35.                     match = false;
36.                }
37.           }
38.
39.           TriggerMessage[] toSend;
40.
41.           //If the code matches combination
42.           //Then send match messages
43.           if(match){
44.                toSend = matchMessages;
45.           } else {
46.                //else send mismatch messages
47.                toSend = mismatchMessages;
48.           }
49.
50.           //Send messages
51.           foreach(TriggerMessage msg in toSend){
52.                if(msg.messageReceiver != null){
53.                     msg.messageReceiver
54.                                .SendMessage(
55.                                    msg.messageName,
56.                                    SendMessageOptions.RequireReceiver);
57.
58.                }
59.           }
60.      }
61. }
```

Listing 70: The script of color combination unlock puzzle

For this script we have reused *TriggerMessage* small class we have created earlier in section 3.4 (Listing 35 page 90). This time we have two arrays of messages: *matchMessages*, which we send when colors match unlock code, and *mismatchMessages*, which we send otherwise. *unlockCode* is directly provided as an array of colors that can be set from the inspector, while other colors that user change come from different renderers. These renderers are referenced through *colorSources* array. To bind the four buttons with the puzzle, we have to reference their renderers as *colorSources*. Whenever a button is switched, it must first switch its color and then send *CompareCodes* message to the puzzle. For simplicity, we attach *ColorCodePuzzle* to the same object of *CentralLock*. The final configuration of each button, as well as the central lock and the puzzle is shown in Illustration 82. It is remember to notice that colors of the unlock code and the buttons must match perfectly in terms of color degree: not any green matches any green, but the numeric values of the colors must be the same. The final functional demo can be found in *scene20* in the accompanying project.

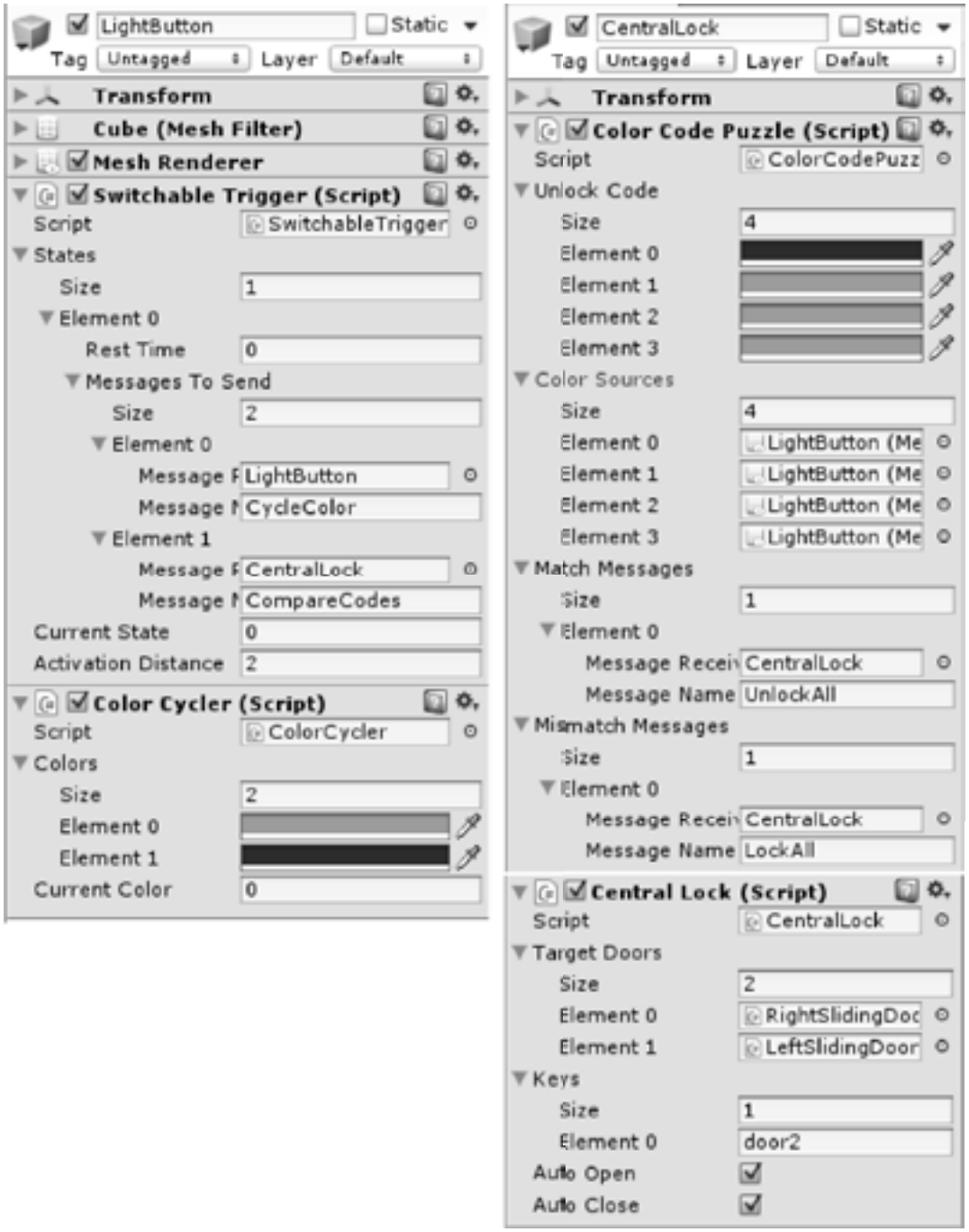


Illustration 82: Configuring buttons, puzzle, and central lock to implement color sequence unlock mechanism

5.3 Health, lives, and score

Player health is one of the vital things needed in many games. The player usually starts the game with full health represented most of times as numeric value of 100. As the player goes through the game he gets attacked by enemies, which causes his health to drop. Nevertheless, he can also pickup some objects that increases his health. If the health of the player reaches zero, the player dies. After his death, however, it is possible to give the player another chance by allowing him to have multiple lives. When the player loses all of his lives, the game is over. In addition to health and lives, it is possible to have a score system that makes the performance of the players comparable.

In this section, we are going to compile the three topics: health, lives, and score into a complete game. In this game, the player controls a cube inside a closed room, which is surrounded by cannons that shoot projectiles. The objective is to survive for the longest possible time by moving and avoiding these projectiles. The player has a health of 100 and three lives. Projectiles has two types: red projectiles that take 10 health points, and green projectiles that take 5 health points. The good player performance results in longer survival time, so it is a good idea to take the number of seconds the player survived as his score. Illustration 83 shows a top view of the room we are going to use. The barrels you see are the torrents (shooters), which are simply cylinders. It is important to rotate each shooter so that the positive direction of its local y axis points towards inside the room.

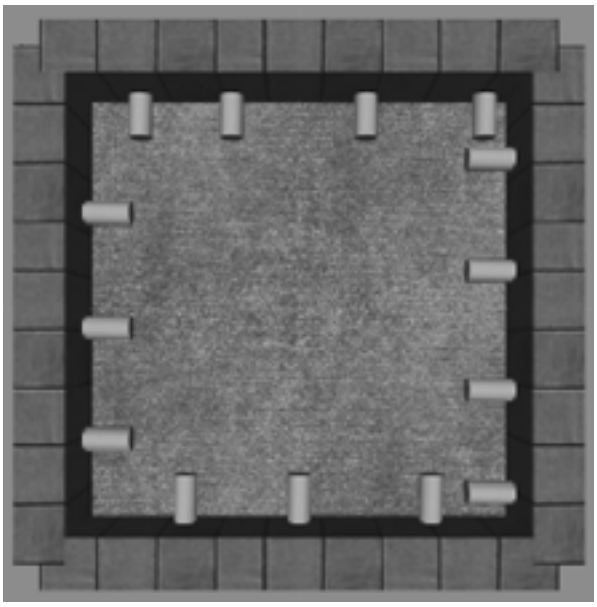


Illustration 83: Top view of the play room and the shooters surrounding it

To prevent the projectiles from colliding with the shooters, we have to remove colliders of all shooters. Each one of these shooter will be given a collection of projectile prefabs to randomly choose one from them and shoot it. Additionally, they will be given maximum and minimum time limits to randomly set pause time between shoots. These functions are encoded in *PhysicsShooter* script shown in Listing 71.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PhysicsShooter : MonoBehaviour {
5.
6.       //Prefabs of projectiles to shoot
7.       public GameObject[] projectils;
8.
9.       //min and max time between shots
10.      public float minTime = 1, maxTime = 6;
11.
12.      void Start () {
13.      ShootRandomly();
14.      }
15.
16.      void Update () {
17.
18.      }
19.
20.      void ShootRandomly(){
21.           //Shoot after random time
22.           float randomTime = Random.Range(minTime, maxTime);
23.           Invoke("Shoot", randomTime);
24.      }
25.
26.      void Shoot(){
27.           //Select random projectile
28.           int index = Random.Range(0, projectils.Length);
29.           GameObject prefab = projectils[index];
30.           GameObject projectile = (GameObject)Instantiate(prefab);
31.
32.           //Shoot the projectile
33.           projectile.transform.position = transform.position;
34.           projectile.rigidbody.AddForce
35.                       (transform.up * 6, ForceMode.Impulse);
36.
37.           //Reshoot after random time
38.           ShootRandomly();
39.      }
40. }
```

Listing 71: A script to randomize type of projectile and shoot timing

The script performs shooting by adding impulse force to the instantiated projectile, which must be chosen randomly from *projectiles* array. After each shooting, *ShootRandomly()* is called. This function generates a random value between *minTime* and *maxTime*, then uses this value as latency before invoking *Shoot()* again. Now we have to create the two projectiles that will be shot by these cannons. These projectiles must have the ability to decrease player's health when they hit him. Therefore we call their script *PainfulProjectile*, which is shown in Listing 72.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PainfulProjectile : MonoBehaviour {
5.
6.       //Damage caused by the projectile
7.       public int damage;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.
15.      }
16.
17.      void OnCollisionEnter(Collision col){
18.           //Tell other object about painful hit
19.           col.gameObject.SendMessage("OnPainfulHit",
20.                                      damage,
21.                                      SendMessageOptions.DontRequireReceiver);
22.
23.           //Destroy the projectile
24.           Destroy(gameObject);
25.      }
26. }
```

Listing 72: A script for projectile that decreases player's health by hitting him

The script is fairly simple: when it collides with another object, it informs the colliding object about the painful that occurred and passes the amount of damage that has to be taken. We can use this scripts to make prefabs for two types of projectiles with 5 and 10 damage power. For this example we can use shining green and red balls as projectiles. We can make them shining by adding a point line object with the same color of the texture as a child. Illustration 84 shows how these projectiles are going to look like. Since these projectiles need to be controlled by physics simulator, we need to attach rigid bodies to them. However, it is necessary to disable *Use Gravity* option for the rigid bodies to prevent them from falling on the ground and hence keep moving in straight line when launched.

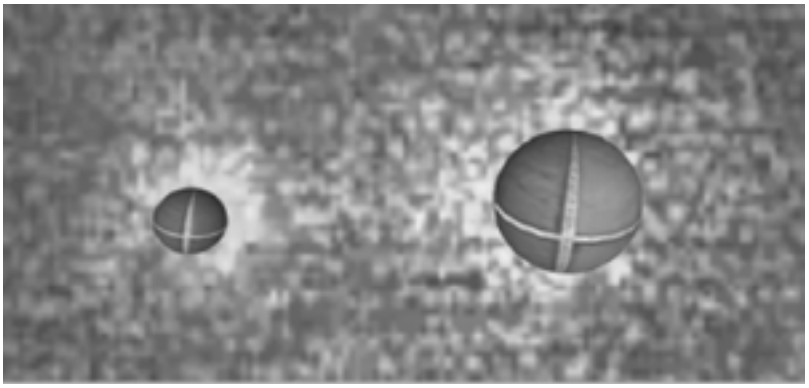


Illustration 84: Small green shining projectile (left) with 5 damage, and big red shining projectile with 10 damage

These two prefabs need to be added to *projectiles* array inside shooter prefab. As a result, all shooter scripts attached to the cannons in the scene are going to have these projectiles automatically added to their *projectiles* array. We have now completed the playground in which our game is going to be played, so the next step will be creating the player. Our player for this game is going to be a simple cube with *PhysicsCharacter* attached to it, in addition to *TopViewControl* shown in Listing 73, which allows us to control the character from a top view and move it in the four directions. Additionally, we have to disable jump by freezing the movement of player's rigid body on y axis. To prevent unwanted rotations, we have also to freeze rigid body rotation on x, y, and z axes.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(PhysicsCharacter))]
5.  public class TopViewControl : MonoBehaviour {
6.
7.       //Reference to the physics character
8.       PhysicsCharacter pc;
9.
10.      void Start () {
11.           //Get the attached physics character
12.           pc = GetComponent<PhysicsCharacter>();
13.      }
14.
15.      void Update () {
16.           //Use arrows to control the movement
17.           if(Input.GetKey(KeyCode.RightArrow)){
18.                pc.StrafeRight();
19.           } else if(Input.GetKey(KeyCode.LeftArrow)){
20.                pc.StrafeLeft();
21.           }
22.
23.           if(Input.GetKey(KeyCode.UpArrow)){
24.                pc.WalkForward();
25.           } else if(Input.GetKey(KeyCode.DownArrow)){
26.                pc.WalkBackwards();
27.           }
28.
29.      }
30. }
```

Listing 73: A script to control physics character from a top view. Jumping is not enabled in this controller

Since we are developing a game in which the player has multiple lives, it is important to save the cube that represents player as a prefab, which gives us the ability to destroy/regenerate player multiple times. In addition to control scripts, the prefab needs other scripts that specify player's health and how it can be reduced or increased. Therefore, we need *PlayerHealth* script, shown in Listing 74, which represent the health as integer value than can be changed through function calls.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PlayerHealth : MonoBehaviour {
5.
6.       //Health amount at the beginning
7.       public int initialHealth = 100;
8.
9.       //Max health limit
10.      public int maxHealth = 100;
11.
12.      //Current health
13.      int health;
14.
15.      //Internal dead flag
16.      bool dead = false;
17.
18.      void Start () {
19.           //Insure appropriate initial health
20.           health = Mathf.Min(initialHealth, maxHealth);
21.           //Player cannot start dead
22.           if(health < 0){
23.                health = 1;
24.           }
25.      }
26.
27.      void Update () {
28.           if(!dead){
29.                if(health <= 0){
30.                      //Player died
31.                      dead = true;
32.
33.                      //Tell other scripts about player's death
34.                      //and give them his final health
35.                      SendMessage("OnPlayerDeath",
36.                            health,
37.                            SendMessageOptions.DontRequireReceiver);
38.                }
39.           }
40.      }
41.
42.      //Deacrease health and inform other scripts about it
43.      public void DecreaseHealth(int amount){
44.           //Do nothing if the player is already dead
45.           if(IsDead()) return;
46.
47.           health -= amount;
48.           SendMessage("OnHealthDecrement",
49.                           health,
50.                           SendMessageOptions.DontRequireReceiver);
51.      }
52.
53.      //Increase health and inform other scripts about it
54.      public void IncreaseHealth(int amount){
55.           //Do nothing if the player is already dead
```

```
56.           if(IsDead()) return;
57.
58.           //Increase only of health is less than full
59.           if(health < maxHealth){
60.                //Do not allow the health to exceed maxHealth
61.                health = Mathf.Min(maxHealth, health + amount);
62.                SendMessage("OnHealthIncrement",
63.                            health,
64.                            SendMessageOptions.DontRequireReceiver);
65.           }
66.      }
67.
68.      //Returns whether player is dead
69.      public bool IsDead(){
70.           return dead;
71.      }
72.
73.      public int GetCurrentHealth(){
74.           return health;
75.      }
76. }
```

Listing 74: A script to handle the health of the player and alter it

When the script is attached to the player, it is possible to set the initial value of health through *initialHealth*. However, the script ensures that the actual start value is between *maxHealth* and 1. This prevents initial health from exceeding set limits as well as preventing player from starting dead. Player's death occurs when the value of the internal state *health* gets less than or equal to 0. This internal state can be altered only through *IncreaseHealth()* and *DecreaseHealth()* functions. These functions enforce minimum and maximum limits and send relevant messages upon each health state change. When the player dies, the script sends *OnPlayerDeath* message.

PlayerHealth gives us the ability to manage player's health and detect his death. In addition to that, it allows us to change health value by calling appropriate functions. It does not, however, say anything about what causes the health to increase/decrease. Therefore, we need another script which can detect hits that player receives and consequently decreases health. This script is *PainfulDamageTaker*, shown in Listing 75.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(PlayerHealth))]
5.  public class PainfulDamageTaker : MonoBehaviour {
6.
7.       //Reference to player health
8.       PlayerHealth playerHealth;
9.
10.      void Start () {
11.           playerHealth = GetComponent<PlayerHealth>();
12.      }
13.
14.      void Update () {
15.
16.      }
17.
18.      //Handle painful hit by decreasing
19.      //player's health by the provided amount (damage)
20.      void OnPainfulHit(int amount){
21.           playerHealth.DecreaseHealth(amount);
22.      }
23. }
```

Listing 75: A script to receive painful hit and consequently reduce player's health

Since everything regarding player's health is handled trough *PlayerHealth* script, all we have to do in this script is to receive the message *OnPainfulHit* along with provided damage amount. This amount is then used as input value when calling *DecreaseHealth()* function of *PlayerHealth*. If you run the game now with the player character inside the room, you should be able to control the cube and try to avoid the projectiles that cannons shoot. Illustration 85 shows a screen shot during game play.

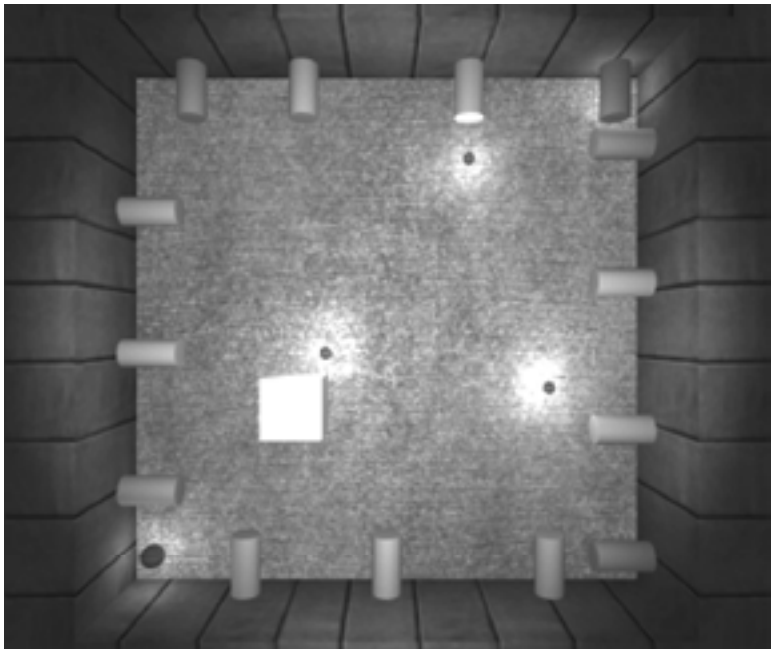


Illustration 85: A screen shot of game play with player cube and projectiles

What we need to do now is to visually inform the player about his health status. Managing the status internally is enough to know how much health the player still has and whether he is dead or not. However, it is necessary to share this information with the player as well. One option is to textually represent health amount, but there are unlimited other options. For this example we are going to use color-coded health display. The color of the cube should vary between red and green depending on the current health. When the health is full then the color of the cube must be green, and it gets closer to red as health value drops. This effect can be achieved through *HealthColorChanger* script shown in Listing 76.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(PlayerHealth))]
5.  public class HealthColorChanger : MonoBehaviour {
6.
7.       //Color when health is full
8.       public Color fullHealth = Color.green;
9.
10.      //Color when player is dead
11.      public Color zeroHealth = Color.red;
12.
13.      //Reference to player health script
14.      PlayerHealth playerHealth;
15.
16.      void Start () {
17.           playerHealth = GetComponent<PlayerHealth>();
18.           UpdateColor(playerHealth.GetCurrentHealth());
19.      }
20.
21.      void Update () {
22.
23.      }
24.
25.      void OnHealthIncrement(int amount){
26.           UpdateColor(amount);
27.      }
28.
29.      void OnHealthDecrement(int amount){
30.           UpdateColor(amount);
31.      }
32.
33.      //Vary color between full and death colors depending on
34.      //the value of new player health
35.      void UpdateColor(int newHealth){
36.           //Convert integer health to float value
37.           //between 0 (death) and 1 (full health)
38.           float val = (float)newHealth /
39.                       (float) playerHealth.maxHealth;
40.
41.           //Apply the new color
42.           renderer.material.color =
43.                Color.Lerp(zeroHealth, fullHealth, val);
44.      }
45. }
```

Listing 76: A script to interpolate cube color between two values based on player's health

What does this script do is simply handle *OnHealthIncrement* and *OnHealthDecrement* messages by taking the new health value as interpolate value between *zeroHealth* and *fullHealth* colors. Since the health is represented as integer value, it must be converted to a float between zero (*minHealth*) and one (*maxHealth – minHealth*). Finally, *Color.Lerp()* is used to set the new color value. If you play the game after attaching this script to player's prefab, you can notice that the cube starts in green. As the player receives hits and the health drops, the color changes to yellow, orange, and then red.

The next question to answer is: what happens when the player dies? Currently nothing, since we do not do anything when the health of the player reaches zero (or less). However, what needs to be done is to take one life from the player and regenerate it again with full health. Therefore, we need to appropriately handle *OnPlayerDeath* that *PlayerHealth* sends when the player dies. This must be handled by an external script that counts player's lives and manages game state accordingly. In other words, when the remaining lives reach zero, the game is over and no further regeneration is possible. This script is *LivesManager*, and it must be attached to a permanent object in the scene. From a logical point of view, it is not possible to attach this script to the player cube game object, since the destruction and regeneration of this object causes stored values to be lost. One good option is to attach this script to the main camera. *LivesManager* script is shown in Listing 77.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class LivesManager : MonoBehaviour {
5.
6.       //Initial number of lives
7.       public int startLives = 3;
8.
9.       //internal counter
10.      int lives;
11.
12.      void Start () {
13.           //Enforce at least one life initially
14.           if(startLives > 0){
15.                lives = startLives;
16.           } else {
17.                lives = 1;
18.           }
19.      }
20.
21.      void Update () {
22.
23.      }
24.
25.      public void GiveLife(){
26.           lives++;
27.           SendMessage("OnLifeGained",
28.                            lives, //New number of lives
29.                            SendMessageOptions.DontRequireReceiver);
30.      }
31.
32.      public void TakeLife(){
33.           lives--;
34.           if(lives == 0){
35.                //Last live lost
36.                //Someone has to take care about that
37.                SendMessage("OnAllLivesLost",
38.                      SendMessageOptions.DontRequireReceiver);
39.           } else {
40.                //A life has been lost
41.                //Handle this elsewhere
42.                SendMessage("OnLifeLost",
43.                      lives, //Remaining lives
44.                      SendMessageOptions.DontRequireReceiver);
45.           }
46.      }
47.
48. }
```

Listing 77: A script to control the number of lives for the player. This script must be attached to a permanent game object in the scene

The script handles the number of lives in a similar way to that *PlayerHealth* uses to handle the health: we have an initial value that is enforced to be at least 1 at the beginning, and the internal value can be later altered through *GiveLife()* and *TakeLife()* functions. Notice that *TakeLife()* can send two messages when called: if the player still has more lives it sends *OnLifeLost* message. However, if the last life has just been lost, *OnAllLivesLost* message is sent. We need now to a mechanism to call *TakeLife()* when the player's health reaches zero or less. In other words, *OnPlayerDeath* message needs to be forwarded as *TakeLife* message. This mechanism is fairly simple and can be achieved through *PlayerDeathReporter* script shown in Listing 78.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PlayerDeathReporter : MonoBehaviour {
5.
6.       //Reports player's death to lives manager
7.       LivesManager manager;
8.
9.       void Start () {
10.           manager = FindObjectOfType<LivesManager>();
11.      }
12.
13.      void Update () {
14.
15.      }
16.
17.      //Handle player's death event by taking a life
18.      void OnPlayerDeath(int deathHealth){
19.           if(manager != null){
20.                manager.TakeLife();
21.           }
22.      }
23. }
```

Listing 78: A script to report player's death event to lives manager in order to take a life from player

As you see, the script is fairly simple and self explanatory. Remember that we have created *LivesManager* script to handle the event of player's death by reducing a life. However, we still need to handle the case when all lives are lost. Up to now, nothing really happens when a life is lost other than decreasing an internal counter that has no effect. Therefore, the next step is going to create a generation and destruction mechanism for player's character (the cube). Remember that we have already counted for this, hence created a cube prefab with all necessary scripts attached to it. What we need to do now is to remove the cube from the scene and delegate generation and destruction functions to *PlayerSpawn* script. This script controls when to destroy an existing player cube and instantiate a new one. Listing 79 shows this script.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class PlayerSpawn : MonoBehaviour {
5.
6.       //Prefab of the player object
7.       public GameObject playerPrefab;
8.
9.       //Seconds to wait between death and respawn
10.      public int spawnDelay = 3;
11.
12.      //Reference to current player
13.      GameObject currentInstance;
14.
15.      void Start () {
16.           SpawnPlayer();
17.      }
18.
19.      void Update () {
20.
21.      }
22.
23.      //A life has been lost
24.      void OnLifeLost(int remainingLives){
25.           //Regenerate Player, delay spawn
26.           Destroy(currentInstance);
27.           Invoke("SpawnPlayer", spawnDelay);
28.      }
29.
30.      //Game Over
31.      void OnAllLivesLost(){
32.           Destroy(currentInstance);
33.      }
34.
35.      void SpawnPlayer(){
36.           currentInstance =
37.                (GameObject) Instantiate(playerPrefab);
38.      }
39. }
```

Listing 79: A script to handle destruction and instantiation of player's character based on lives

This script needs a prefab to instantiate, in addition to a time delay to wait between death and next spawn. The script starts by calling *SpawnPlayer()*, which instantiates the prefab of the character and keeps an internal reference to it in *currentInstance*. This reference is necessary to destroy the player when a life is lost. Therefore, there is a need to handle *OnLifeLost* message sent by *LivesManager*, which is done by *OnLifeLost()* function. This function destroys the current instance, and then calls *SpawnPlayer()* with the delay predefined in *spawnDelay*. The lost of last life is handled through *OnAllLivesLost()* function, which destroys *currentInstance*, but this time without calling *SpawnPlayer()*.

Just like what we have done with player's health, we need a visual representation of the number of lives the player has. The simplest way is through a textual representation. For this purpose, we are going to use a new game object, which is *3D Text*. This object can be placed anywhere in the scene, and can render the given text as 3D characters that can be viewed from different angles. However, for this example we need the text to be directly on front of the camera.

To add a 3D text to the scene, go to Game Object > Create Other > 3D Text. After that, position the text just like you do with any other game object. You can switch to game view to make sure it is positioned correctly in front in the camera and visible to the player.

Illustration 86 shows 3D Text properties as they appear in the inspector.

Illustration 86: Properties of 3D Text displayed in the inspector

Most of the properties are clear, and they are familiar to anyone who deals with text in computer. Our focus will be on *Text* property, which we need to access through a script and modify it. To begin with lives display, we have first to add a 3D Text and position it in an appropriate position. For example, we can position it in the top of game view. It is a good idea to give the text an initial value, such as 0. After that we have to write a script that reads the number of lives the player has and updates the displayed text correspondingly. This script is *LivesCounterHandler*, shown in Listing 80.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(LivesManager))]
5.  public class LivesCounterHandler : MonoBehaviour {
6.
7.       //Reference to 3D Text object
8.       public TextMesh display;
9.
10.      //Reference to lives manager
11.      LivesManager lManager;
12.
13.      void Start () {
14.           lManager = GetComponent<LivesManager>();
15.           //Start by displaying startLives
16.           display.text = lManager.startLives.ToString();
17.      }
18.
19.      void Update () {
20.
21.      }
22.
23.      void OnLifeGained(int newLives){
24.           //Number of lives changed, update
25.           display.text = newLives.ToString();
26.      }
27.
28.      void OnLifeLost(int remainingLives){
29.           //Number of lives changed, update
30.           display.text = remainingLives.ToString();
31.      }
32.
33.      void OnAllLivesLost(){
34.           //All lives have been lost,
35.           //display 'Game Over' text
36.           display.text = "Game Over";
37.      }
38. }
```

Listing 80: A script that updates a 3D Text to display the number of lives the player currently has

Notice that the variable type we use to reference a 3D Text object is called *TextMesh*. This script requires *LivesManager*, so it has also to be attached to the main camera. After adding it, we need to drag the 3d text object we are going to use as lives count display from the hierarchy to *display* variable. The initial value of *display.text* is set to *startLives*, which is the initial number of lives according to *LivesManager*. *display.text* is a string that sets the displayed text of 3D Text, and, since it is string, we need to convert the integer value of *startLives* to string by calling *ToString()* function as in line 16. After setting the initial text value, all we have to do is to monitor any changes on the number of lives by handling *OnLifeLost* and *OnLifeGained* messages. Upon each change, we read the provided new number of lives and update *display.text* according to it. However, when *OnAllLivesLost* message is received, we display the message "Game Over".

The last topic in this section is player score. Up to now we have developed a fully functional game with lives and health. What remains is to evaluate player's performance by a score value. Since we are talking about a survival game, the best thing to use as score is the number of seconds the player was able to survive. First of all, we need a 3D Text to display the score, and we are going to add it to the bottom of the screen. Additionally, we need to write a script that increments the score every second. Like *LivesManager*, score script needs a permanent game object to be attached to, so we will use the main camera again for this purpose. Listing 81 shows *ScoreCounter* script, which counts and displays player's score.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(LivesManager))]
5.  public class ScoreCounter : MonoBehaviour {
6.
7.       //Where to show score? (optional)
8.       public TextMesh display;
9.
10.      LivesManager lManager;
11.
12.      //Internal score counter
13.      int score = 0;
14.
15.      void Start () {
16.           lManager = GetComponent<LivesManager>();
17.           //Increase 1 point every second
18.           InvokeRepeating("IncrementScore", 1, 1);
19.      }
20.
21.      void Update () {
22.
23.      }
24.
25.      void IncrementScore(){
26.           score++;
27.           if(display != null){
28.                display.text = score.ToString();
29.           }
30.      }
31.
32.      void OnAllLivesLost(){
33.           //Game over, stop counting
34.           CancelInvoke("IncrementScore");
35.      }
36. }
```

Listing 81: A script to increment player's score every second and display it

The core function of this script is *IncrementScore()*, which increments internal score counter by 1 every time it is called. It also updates the text on *display* if a text mesh is provided. In *Start()*, we call *InvokeRepeating()* function and ask it to keep calling *IncrementScore()* one time every second. As a result, the score will be incremented by 1 every second, and the 3D Text that displays the score updates continuously as well. When *OnAllLivesLost* message is received, we know that the game is over. Therefore, we have to stop incrementing score by stopping the repetitive calling of *IncrementScore()*. To stop calling a function we use *CancelInvoke()* function and give it the name of the target function. The final look of the game with lives and score counter is shown in Illustration 87. The complete demo is available in *scene21* in the accompanying project.

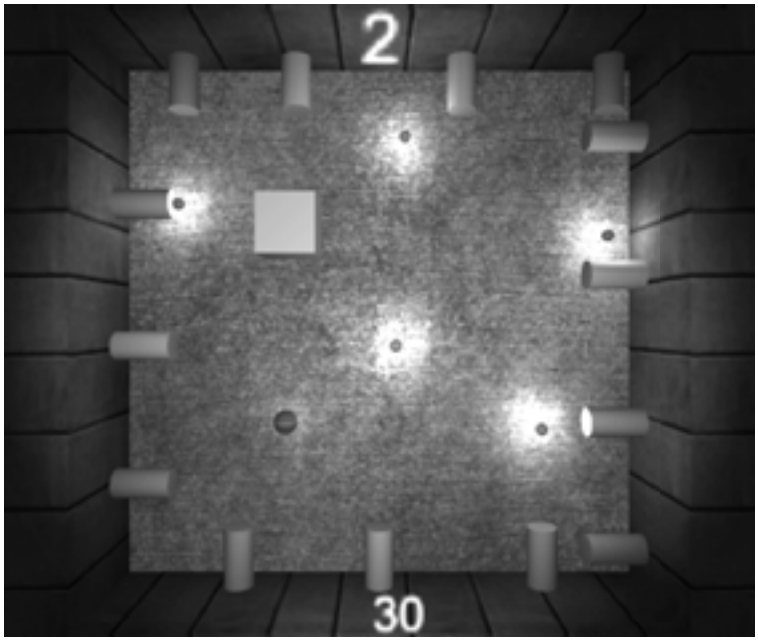


Illustration 87: A screen shot of the final game. The upper digit is the number of lives left, and the lower number is the score.

5.4 Weapons, ammunition, and reload

In many games that contain shooting mechanic, it is possible the player owns a number of weapons and can switch between them in order to deal with different situations. Additionally, most weapons have a form of finite ammunition that need to be refilled from time to time. This ammunition is sometimes represented as a number of magazines that need to be replaced when they are empty, which results in reload mechanic known to most first person shooter players. In this section we are going to learn how to implement all these weapon functions. So let's begin with a scene with a fixed camera like in Illustration 88. In this scene, we are not going to move, but will be able to aim and shoot with mouse pointer, we can also use the keyboard to switch between different weapons. Notice that 3D text are used to draw a simple GUI for the user, which we are going to use to show the number of remaining ammo as well as reload progress.

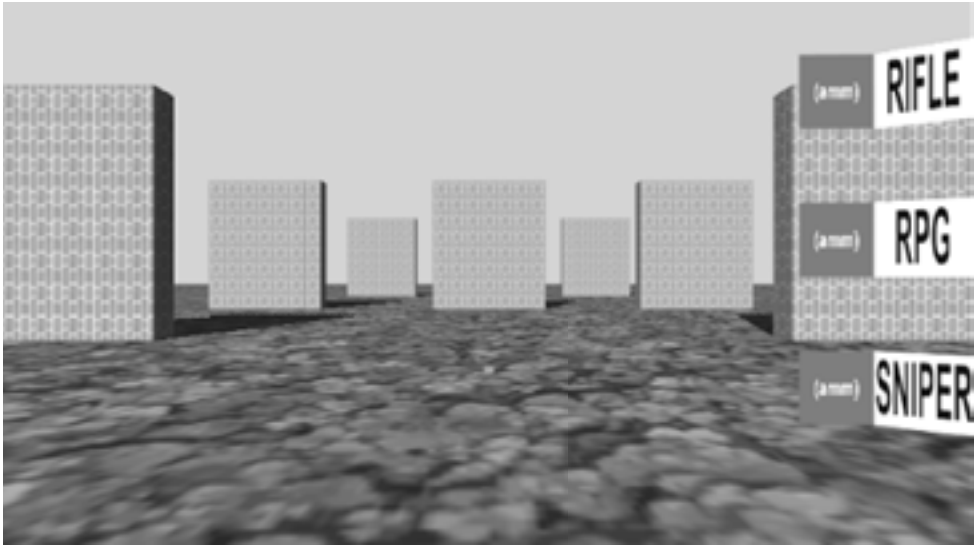


Illustration 88: Basic scene constructed to test different weapons

The walls you see in Illustration 88 are constructed using destructible building blocks similar to those we used in section 4.6. However, we are going to make some modifications on these building blocks, therefore we need to create a new prefab other than the one used in section 4.6. One good thing about prefabs is ability to modify hundreds of objects from a single place, so we are going just to make a copy of the original *ReturnableBrick* prefab, rename it to *ShootableBrick*, and use it to build these walls. We are going to come back later to our prefab to modify it. Now we need to create a new object and name it *player*, and position it in the same position of the camera. This object is going to be used as aiming and shooting point, which means that it must initially look forward towards the scene (the positive z axis of the object must point to the same direction of the camera). Additionally, we need to add three empty children to this object, which are the weapons to be used by the players. These objects should be named after the weapons they represent: *Rifle*, *RPG*, and *Sniper*.

The three different weapons (rifle, RPG, and sniper) have common properties such as the ability to fire them, their need to ammo, and so on. On the other hand, each one of them has its own implementation to "fire weapon": the sniper shoots single accurate bullet, the rifle shoots a large number of bullets in short time, and the RPG shoots one rocket. Therefore, we need to separate general functions that reflects common properties among all weapons from specific implementation of weapon firing. These general functions are implemented in *GeneralWeapon* script in Listing 82. This script must be added to all weapons.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class GeneralWeapon : MonoBehaviour {
5.
6.        //How many magazines remaining
7.        public int magazineCount = 3;
8.
9.        //Ammunation count per full magazine
10.       public int magazineCapacity = 30;
11.
12.       //Ammunation remaining in current magazine
13.       public int magazineSize = 30;
14.
15.       //How many seconds needed to reload?
16.       public float reloadTime = 3;
17.
18.       //How many times can the weapon shoot in one second?
19.       public float fireRate = 3;
20.
21.       //If true, there is no need to release
22.       //the trigger between firings
23.       public bool automatic = false;
24.
25.       //Ammunation lost per firing
26.       //Must not exceed magazine capacity
27.       public int ammoPerFiring = 1;
28.
29.       //Is this weapon currently hold by the player?
30.       public bool inHand = false;
31.
32.       //Internal timer for fire rate
33.       float lastFiringTime = 0;
34.
35.       //Internal state for reload progress
36.       float reloadProgress = 0;
37.
38.       //Internal storage of trigger state
39.       bool triggerPulled = false;
40.
41.       void Start () {
42.
43.       }
44.
45.       void Update () {
46.            //If the weapon is currently in hand and it reloads,
47.            //then advance reload progress with time
48.            if(inHand && reloadProgress > 0){
49.                 reloadProgress += Time.deltaTime;
50.                 if(reloadProgress >= reloadTime){
51.                       //Reloading completed
52.                       //Discard the current magazine
53.                       //and install a new one
54.                       magazineSize = magazineCapacity;
```

```
55.                     magazineCount--;
56.                     reloadProgress = 0;
57.                     SendMessage("OnReloadComplete",
58.                           SendMessageOptions.DontRequireReceiver);
59.               }
60.          }
61.     }
62.
63.     public void Fire(){
64.          //Make sure the weapon is in hand and not
65.          //currently reloading, enforce time gap between firings,
66.          //and make sure that the weapon is either automatic
67.          //or the trigger has been released after last firing
68.          if(inHand && reloadProgress == 0 &&
69.               (automatic || !triggerPulled) &&
70.               Time.time - lastFiringTime > 1 / fireRate){
71.               //Do we have enough ammo in the current magazine?
72.               if(magazineSize >= ammoPerFiring){
73.                     //Yes, fire by reducing ammo,
74.                     //setting fire timer,
75.                     //and sending OnWeaponFire message
76.                     magazineSize -= ammoPerFiring;
77.                     lastFiringTime = Time.time;
78.                     triggerPulled = true;
79.                     SendMessage("OnWeaponFire",
80.                           SendMessageOptions.DontRequireReceiver);
81.
82.                     //if the remaining ammo is not enough, then reload
83.                     if(magazineSize < ammoPerFiring){
84.                          Reload();
85.                     }
86.
87.               } else {
88.                     //No, reload
89.                     Reload();
90.               }
91.          }
92.     }
93.
94.     public void ReleaseTrigger(){
95.          triggerPulled = false;
96.     }
97.
98.     public void Reload(){
99.          //Make sure there is no reloading in progress
100.         if(reloadProgress == 0){
101.              //Make sure there is enough magazines
102.              //and the current magazine isn't full
103.              if(magazineCount > 0 &&
104.                    magazineSize < magazineCapacity){
105.                    //Initialize reloading progress
106.                    reloadProgress = Time.deltaTime;
107.                    SendMessage("OnReloadStart",
108.                          SendMessageOptions.DontRequireReceiver);
109.              }
```

```
110.          }
111.      }
112.
113.      //returns current reload progress percentage
114.      public float GetReloadProgress(){
115.          return reloadProgress / reloadTime;
116.      }
117.  }
```

Listing 82: A script that handles common functions of all weapons

The first three variables are used to manage ammunition. The difference between *magazineCapacity* and *magazineSize* is that the first one is constant and tells us the number of maximum bullets in a single magazine. However, *magazineSize* is variable and is reduced by *ammoPerFiring* whenever the weapon is fired. In addition to ammunition management, we have other variables to manage timing. For example, *reloadTime* is the number of seconds needed to reload the weapon when the current magazine becomes empty. Additionally, *fireRate* decides how many times the weapon can be fired in one second. *lastFiringTime* and *reloadProgress* are used in together with *fireRate* and *reloadTime* to compute the timing correctly. The third important aspect we need to manage is whether the weapon is automatic, which means it has the ability to continuously fire bullets while the trigger is pulled. This property is managed through *automatic* and *triggerPulled* flags. Finally, we have *inHand* flag, which affects all other functions: if the weapon is not currently held in hands, it can not be neither fired nor reloaded.

Fire() and *ReleaseTrigger()* functions are related. If the weapon is not *automatic*, *ReleaseTrigger()* must be called each time after *Fire()*. *Fire()* on the other hand must ensure that

- the weapon is currently in hand,
- is not reloading,
- either automatic or the trigger is currently released, and
- there are enough bullets in the magazine.

If all of these conditions are met, it sends *OnWeaponFire* message, sets *triggerPulled* flag, and reduces bullet count in the current magazine. If the number of bullets remaining in the magazine is less than the number needed to fire, *Reload()* is automatically called. *Reload()* is responsible for *initiating* reload process rather than instantly reloading the weapon. The variable *reloadProgress* represent the time passed since the last time *Reload()* has been called. If *reloadProgress* is zero, this means the weapon is not currently reloading. Therefore, *Reload()* must check that *reloadProgress* equals zero before initiating reload process. It is also necessary to have at least one additional magazine in order for reloading to take place. Therefore, *Reload()* checks the value of *magazineCount* in addition to *magazineCapacity* and *magazineSize*, to make sure that the magazine we are trying to replace is not full. If all conditions are met, we set the value of *reloadProgress* to *Time.deltaTime*. As a result, the value of reload progress will accumulate through *Update()* calls by adding *Time.deltaTime* during each frame. When the value of *reloadProgress* exceeds *reloadTime*, reloading is completed by decrementing the count of magazines remaining and setting the size of the current magazine to magazine capacity.

The last function we are going to cover in this script is *GetReloadProgress()*. If the weapon is currently reloading, it returns reloading progress as a float value between 0 and 1. If this function returns zero, it means that the weapon is not currently reloading. The returned value can be used to interpolate some animations or control progress bars etc. Illustration 89 shows the *GeneralWeapon* for the three weapons and how the values vary between them.

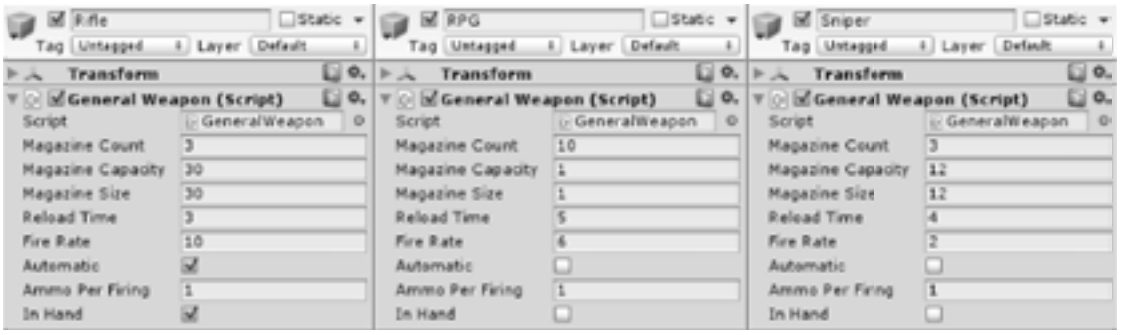


Illustration 89: Setting the properties of GeneralWeapon for rifle, RPG, and sniper

If you consider *GeneralWeapon* as the processing core of shooting mechanism, then you would recognize that it needs both input and output handlers. The input handler is responsible for aiming the weapon and calling *Shoot()* function based on player input. On the other side we need an output handle which receives *OnWeaponFire* message and translates it to actual effect on the scene. Let's begin with the input handler, since it is common among the three weapons, unlike output handlers. Remember that we are using the mouse to aim at targets and shoot them. Therefore, we need a script that looks at the position of the mouse pointer. This script is *MousePointerFollower* shown in Listing 83. This script must be added to player game object, which is the parent of the three weapons.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class MousePointerFollower : MonoBehaviour {
5.
6.       //A marker that can be placed on the current target
7.       public Transform targetMarker;
8.
9.       void Start () {
10.           //Hide the marker behind the player
11.           targetMarker.position =
12.                transform.position - Vector3.forward;
13.      }
14.
15.      void Update () {
16.           //Try to find the point where mouse points
17.           Ray camToMouse =
18.                Camera.main.ScreenPointToRay (Input.mousePosition);
19.
20.           RaycastHit hit;
21.           if(Physics.Raycast(camToMouse, out hit, 500)){
22.                //An object has been found, look at it
23.                transform.LookAt(hit.point);
24.                //Move the marker to the hit point
25.                targetMarker.position = hit.point;
26.                //Move the marker a little bit towards us
27.                targetMarker.LookAt(transform.position);
28.                targetMarker.Translate(0, 0, 0.1f);
29.           } else {
30.                //No object under the mosue pointer,
31.                //look far away
32.                transform.LookAt(camToMouse.GetPoint(500));
33.                //Hide the marker
34.                targetMarker.position =
35.                     transform.position - Vector3.forward;
36.           }
37.      }
38.
39. }
```

Listing 83: A script to make the object always look at the position of the mouse pointer

For our example, we are going to use a small red light with low radius and high intensity to mark the current target. This marker can be referenced from the script via *targetMarker*. At the beginning, we hide this marker by positioning it behind the player (remember that the positions of the player and the camera are the same). During each frame update, we get the ray that starts from the camera and passes through the mouse pointer. We then use this ray in a ray cast test to check if there is an object under the mouse pointer. If such object is found, we position the marker at the point where the ray hits the object. Notice that we make the pointer look to our object and move it forward a little bit to make it visible. We also make our object (the player) look at hit point. Since all weapons are children of the player and have the same position and rotation of it, they are going to be targeted towards the hit point as well. If ray casting did not detect any object under mouse pointer, we take a far point (500 meters away) along the ray and look at it. In that case, the marker is positioned again behind the player to make it invisible.

After pointing the player (and consequently all weapons) correctly, we need to handle other input commands: weapon switching, firing, and reloading. All these functions are handled through *WeaponController*, which must be also added to player's game object. This script is shown in Listing 84.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class WeaponController : MonoBehaviour {
5.
6.       //Array of available weapons
7.       public GeneralWeapon[] weapons;
8.
9.       //Index of initially hold weapon
10.      public int initialWeapon = -1;
11.
12.      //Index of currently hold weapon
13.      int currentWeapon;
14.
15.      void Start () {
16.           //Set current weapon to the initial value
17.           //selected from the inspector
18.           currentWeapon = initialWeapon;
19.           //Update inHand variable of all weapons
20.           RefreshInHandValues();
21.      }
22.
23.      void Update () {
24.           UpdateSwitching();
25.           UpdateShooting();
26.      }
27.
28.      void UpdateSwitching(){
29.           //Convert the value of "1" key in alphabit
30.           //section of the keyboard to int
31.           int keyCode = (int)KeyCode.Alpha1;
32.           for(int i = 0; i < weapons.Length; i++){
33.                //Each weapon takes the number keyCode + weapon index
34.                if(Input.GetKeyDown((KeyCode) keyCode + i)){
35.                      currentWeapon = i;
36.                      RefreshInHandValues();
37.                }
38.           }
39.      }
40.
41.      void UpdateShooting(){
42.           //Mouse button down: Fire
43.           if(Input.GetMouseButton(0)){
44.                weapons[currentWeapon].Fire();
45.           }
46.           //Mouse button up: ReleaseTrigger
47.           if(Input.GetMouseButtonUp(0)){
48.                weapons[currentWeapon].ReleaseTrigger();
49.           }
50.           //Right click: Reload
51.           if(Input.GetMouseButtonDown(1)){
52.                weapons[currentWeapon].Reload();
53.           }
54.      }
```

```
55.
56.       //Change weapon
57.       public void SetCurrentWeapon(int newIndex){
58.             weapons[currentWeapon].ReleaseTrigger();
59.             currentWeapon = newIndex;
60.             RefreshInHandValues();
61.       }
62.
63.       void RefreshInHandValues(){
64.             foreach(GeneralWeapon gw in weapons){
65.                   //inHand must be true only for the
66.                   //currently hold weapon
67.                   gw.inHand = weapons[currentWeapon] == gw;
68.             }
69.       }
70. }
```

Listing 84: A script to handle weapon switching, firing, and reloading

All weapons we use must be added to *weapons* array, so that they can be accessed by the script and hence the player has the ability to switch between them. By default, *initialWeapon* is set to -1. After adding the script to player's game object and adding our three weapons to *weapons* array, we can set *initialWeapon* to 0, 1, or 2. The currently hold weapon is managed by the script internally through *currentWeapon*, so the only way to change the current weapon is by calling *SetCurrentWeapon()* function. This is necessary to make sure that *RefreshInHandValues()* is called each time we switch the weapon. The importance of this function is that it guarantees having only one weapon that has *true* value for *inHand*. This weapon is in fact the one in the index *currentWeapon* in *weapons* array. During each frame update, *UpdateSwitching()* and *UpdateShooting()* are invoked.

UpdateSwitching() scans keyboard keys starting from *KeyCode.Alpha1*. *KeyCode.Alpha1* is the key with digit 1 found on the the upper left corner of the keyboard. If we convert *KeyCode.Alpha1* to integer and add 1 to it, we get an integer value equal to *KeyCode.Alpha2*. This fact is useful for us in scanning all numeric keys using *for* loop instead of writing a specific *if* statement for each key. As a result, the key with digit 1 matches the weapon in index 0 in *weapons* and so on. On the other hand, *UpdateShooting()* reads input from mouse buttons. It calls *Fire()* function from the current weapon when the left mouse button is pressed, and calls *ReleaseTrigger()* from the same weapon when the left button is released. Additionally, it performs reload when the right mouse button is clicked. Our player object should now look like Illustration 90.

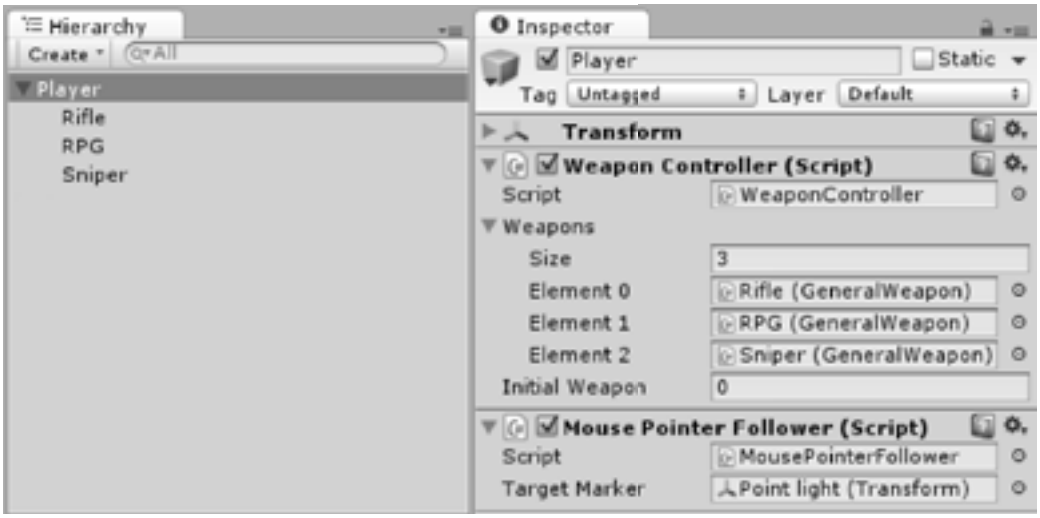


Illustration 90: Player game object configured completely

With this configuration of player's game object, the input system of our shooting mechanism is complete. We have now to deal with the output. Both rifle and sniper weapons can be implemented using ray casting. Therefore, it is reasonable to reuse our *RaycastShooter* script in Listing 49 (page 128). All we have to do is to add *RaycastShooter* to the game objects of rifle and sniper, set its properties (range, inaccuracy, power), and make a "bridge" script that receives *OnWeaponFire* message from *GeneralWeapon* and eventually call *Shoot()* function of *RaycastShooter*. This is a very simple script called *WeaponToRaycast*, shown in Listing 85.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(GeneralWeapon))]
5.  [RequireComponent(typeof(RaycastShooter))]
6.  public class WeaponToRaycast : MonoBehaviour {
7.
8.       RaycastShooter shooter;
9.
10.      void Start () {
11.           shooter = GetComponent<RaycastShooter>();
12.      }
13.
14.      void Update () {
15.
16.      }
17.
18.      void OnWeaponFire(){
19.           shooter.Shoot();
20.      }
21. }
```

Listing 85: Simple script that bridges between *RaycastShooter* and *GeneralWeapon*

Obviously, the script depends on both *RaycastShooter* and *GeneralWeapon*, which makes sense as its job is to link these scripts together. Illustration 91 shows different *RaycastShooter* configurations we need to set for both rifle and sniper.

Illustration 91: Different configurations of RaycastShooter for rifle (left) and sniper (right)

It is time to move back to our building blocks which we have used to build the walls in our scene. In addition to being destructible, we need these blocks to be affected by ray cast bullets. First of all, we need to create bullet holes on these walls. Therefore, we have to attach *BulletHoleMaker* script (Listing 52 page 133) to our *ShootableBrick* prefab, and provide the script with the prefab of the bullet hole we have created earlier. Additionally, we have to remove *MouseExploder* script from the building block, because we don't want to have an explosion with each mouse click on the block.

To remove a component from a game object, click the gear icon on the upper left corner of the component and select *Remove Component* from the menu.

The next script we need to attach to our block is *BulletForceReceiver* (Listing 53 page 134), which allows our weapons (rifle and sniper) to affect the block by moving it. Unfortunately, our block already has *Destructible* script attached, which means that *BulletForceReceiver* is not going to have any effect unless the block is destructed. Therefore, we need a script that destructs the block based on a ray cast hit with enough power. The script we need is *DestructOnHitDamage*, which is shown in Listing 86.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(Destructible))]
5.  public class DestructOnHitDamage : MonoBehaviour {
6.
7.       //Minimum damage to destruct
8.       public float destructionDamage = 250;
9.
10.      Destructible dest;
11.
12.      void Start () {
13.           dest = GetComponent<Destructible>();
14.      }
15.
16.      void Update () {
17.
18.      }
19.
20.      void OnRaycastHit(RaycastHit hit){
21.           //Hit damage is stored in distance
22.           //if damage more than destructionDamage,
23.           //then destruct
24.           if(hit.distance > destructionDamage){
25.                dest.Destruct();
26.           }
27.      }
28. }
```

Listing 86: A script to receive ray cast hit and eventually destruct the attached destructible

All we have to do is to specify the minimum amount of damage that destructs the block. By calling *Destruct()*, we remove all constraints that limit the movement of the block. As a result, *AddForceAtPosition()* which is called by *BulletForceReceiver* is going to have its proper effect on the rigid body of the block and move it. Our building block is now ready and the walls are affected by sniper and rifle bullets.

The last thing we need to take care about regarding output are RPG rockets. In this case we have to create a rocket prefab that is launched when the RPG is fired. When this rocket hits a wall, it must cause an explosion and consequently destroy the wall. To launch the rocket we need two parts: the rocket itself as prefab, and the launcher as a script that responds to *FireWeapon()* message by instantiating the prefab. Let's begin with *RPG* script, which is responsible for instantiating the rocket. This script must be added to the RPG weapon game object. Listing 87 shows *RPG* script.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class RPG : MonoBehaviour {
5.
6.       //Prefab of rocket to launch
7.       public GameObject rocketPrefab;
8.
9.       void Start () {
10.
11.      }
12.
13.      void Update () {
14.
15.      }
16.
17.      //Simply receive the message and instantiate a rocket
18.      //The rocket has initially the same position and rotation
19.      //of the launcher
20.      void OnWeaponFire(){
21.           GameObject rocket = (GameObject)Instantiate(rocketPrefab);
22.           rocket.transform.position = transform.position;
23.           rocket.transform.rotation = transform.rotation;
24.      }
25. }
```

Listing 87: RPG script to launch rockets based on general weapon

To represent RPG rocket, we are going to use a sphere that is stretched along its z axis so it look like ellipsoid. For this example I am going to use the dimensions (0.2, 0.2, 0.75). We can give this ellipsoid an arbitrary texture, and we have also to create a prefab out of it. Once we have the rocket prefab ready, we set the value of *rocketPrefab* in *RPG* script to that prefab. This prefab needs, of course, a number of components and scripts in order to behave as we wish. First of all, we need to add a rigid body component to it. Now we have to think about what does the rocket do: 1) it is launched with a specific impulse force, then hits the target. Once it hits the target, 2) it explodes and 3) blows the target as well. If the target is destructible, it must be 4) destructed as well. Therefore, we need four scripts to perform these four tasks. So let's begin with the first task: launching and moving the rocket. This task is performed by *RPGRocket* script shown in Listing 88.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(Rigidbody))]
5.  public class RPGRocket : MonoBehaviour {
6.
7.       //Force to apply upon launch
8.       public float launchForce = 100;
9.
10.      //Number of seconds to keep rocket alive
11.      //in case it hits nothing
12.      public float lifeTime = 7;
13.
14.      //To calculate life time
15.      float launchTime;
16.
17.      //Internal state tracking
18.      //Necessary to prevent collision detection
19.      //between the rocket and its pieces
20.      bool destroyed = false;
21.
22.      void Start () {
23.           rigidbody.AddForce(transform.forward * launchForce,
24.                                        ForceMode.VelocityChange);
25.
26.           launchTime = Time.time;
27.      }
28.
29.      void Update () {
30.           if(!destroyed && Time.time - launchTime > lifeTime){
31.                Destroy(gameObject);
32.           }
33.      }
34.
35.      void OnCollisionEnter(Collision col){
36.           if(!destroyed){
37.                destroyed = true;
38.                //Inform other scripts on the rocket about the hit
39.                //and provide a reference to the colliding object
40.                SendMessage("OnRocketHit",
41.                      col.collider,
42.                      SendMessageOptions.DontRequireReceiver);
43.
44.                //Destroy rocket object
45.                Destroy(gameObject);
46.           }
47.      }
48. }
```

Listing 88: A script to launch the rocket and detect its collision with other objects

The script starts by giving the rigid body of the rocket a force with enough magnitude to launch it. After that it starts to compute the life time of the rocket as set in the inspector. However, if the rocket hits an object during its movement, it destroys immediately after sending *OnRocketHit* other scripts and providing a reference to the colliding object. Notice that we count for the case of multiple collisions, and hence use the internal *destroyed* flag. By doing this, we guarantee that *OnRocketHit* is sent only once. After hitting the target, the rocket must explode into pieces. For this purpose, we can reuse *Breakable* script (Listing 59 page 150) with a custom piece prefab. However, we need to set a high value, such as 1000, for *explosionPower* variable. The reason for that is the fact that the rocket does not simply *break*, but rather *explode*. This means that its pieces must be scattered appropriately to mimic an explosion, which needs a force with high magnitude.

To break the rocket upon collision with another object (the target), we need a third script to link *RPGRocket* and *Breakable*. The script has to receive *OnRocketHit* message and eventually send *Break* message to the breakable. This script is *BreakOnRocketHit* shown in Listing 89. This is a straightforward script that does nothing other than receiving a message and sends another one.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(RPGRocket))]
5.  [RequireComponent(typeof(Breakable))]
6.  public class BreakOnRocketHit : MonoBehaviour {
7.
8.        void Start () {
9.
10.       }
11.
12.       void Update () {
13.
14.       }
15.
16.       void OnRocketHit(Collider hitObject){
17.            GetComponent<Breakable>().Break();
18.       }
19. }
```

Listing 89: A script to link *RPGRocket* and *Breakable*

The last script we have to add to rocket prefab is in fact the explosive material which does the real destruction and causes explosions. When *OnRocketHit* message is received, all destructible blocks in explosion range must be destructed and an explosion force must be added to it. The script that performs this task is *DestructOnRocketHit*, which is shown is Listing 90.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class DestructOnRocketHit : MonoBehaviour {
5.
6.       //Radius of rocket explosion
7.       public float explosionRadius = 3;
8.
9.       //Force of the explosion
10.      public float explosionForce = 50000;
11.
12.      void Start () {
13.
14.      }
15.
16.      void Update () {
17.
18.      }
19.
20.      void OnRocketHit(Collider target){
21.           //Destruct all destructible blocks in range
22.           //and add explosion force to them
23.           Destructible[] all = FindObjectsOfType<Destructible>();
24.
25.           Vector3 explosionPos = transform.position;
26.
27.           foreach(Destructible dest in all){
28.                if(Vector3.Distance
29.                      (explosionPos, dest.transform.position)
30.                                          < explosionRadius){
31.
32.                      dest.Destruct();
33.
34.                      dest.rigidbody.
35.                            AddExplosionForce(explosionForce,
36.                                              explosionPos,
37.                                              explosionRadius);
38.                }
39.           }
40.      }
41. }
```

Listing 90: A script to destruct and explode nearby destructible objects upon rocket hit

You might have noticed similarities between this script and *MouseExploder* script (Listing 57 page 145). What is special in *DestructOnRocketHit* that it takes the position of the rocket as the position of the explosion.

All weapons are now functional and can be controlled by the mouse. Additionally, it is possible to switch between these weapons using keyboard number keys 1, 2, and 3. The final function we need to implement is the display of ammo count and reload progress. If you refer to Illustration 88, you will notice a text mesh that says "(amm)" next to each weapon name. We are going to use each one of these to display data about its weapon. If the weapon is not currently in hand, the text "XXX" must be displayed. This method informs the player directly which weapon he is currently holding in hand. However, if the weapon is currently in hand, the number of remaining magazines as well as magazine size and magazine capacity must be displayed. For instance, we can use the format *magazineSize/magazineCapacity (x magazineCount)*. Finally, if the weapon is reloading, the progress must be displayed as percentage.

To control the display, we need to attach a script to each weapon that continuously reads ammo data and updates the display accordingly. This script is *AmmoDisplay*, which is shown in Listing 91.

```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  [RequireComponent(typeof(GeneralWeapon))]
5.  public class AmmoDisplay : MonoBehaviour {
6.
7.       //Where to display data
8.       public TextMesh display;
9.
10.      //The weapon to display data for
11.      GeneralWeapon weapon;
12.
13.      void Start () {
14.           weapon = GetComponent<GeneralWeapon>();
15.      }
16.
17.      void LateUpdate () {
18.           //Do not show if weapon is not in hand
19.           if(!weapon.inHand){
20.                display.text = "XXX";
21.                return;
22.           }
23.
24.           float reloadProgress = weapon.GetReloadProgress();
25.           //Show number of bullets and magazines remaining
26.           if(reloadProgress == 0){
27.                display.text = weapon.magazineSize + "/" +
28.                                        weapon.magazineCapacity + " (x" +
29.                                        weapon.magazineCount + ")";
30.           } else {
31.                //If reloading, show reload progress
32.                int progress = (int)(reloadProgress * 100);
33.                display.text = "RLD " + progress + "%";
34.           }
35.      }
36. }
```

Listing 91: A script to display weapon data in text format

It is important to notice that we use *GetReloadProgress()* function to know whether the weapon is reloading. If this function returns zero, it means that no reloading is currently in progress. In this case, we display the ammo. However, if this function returns a value other than zero, this value is in fact the progress of reloading expressed in a value between 0 and 1. If we multiply the returned value by 100 then convert it to integer, we get a progress value that is neat to display. Illustration 92 shows a screen shot of the final scene. A complete demo can be found in *scene22* in the accompanying project.

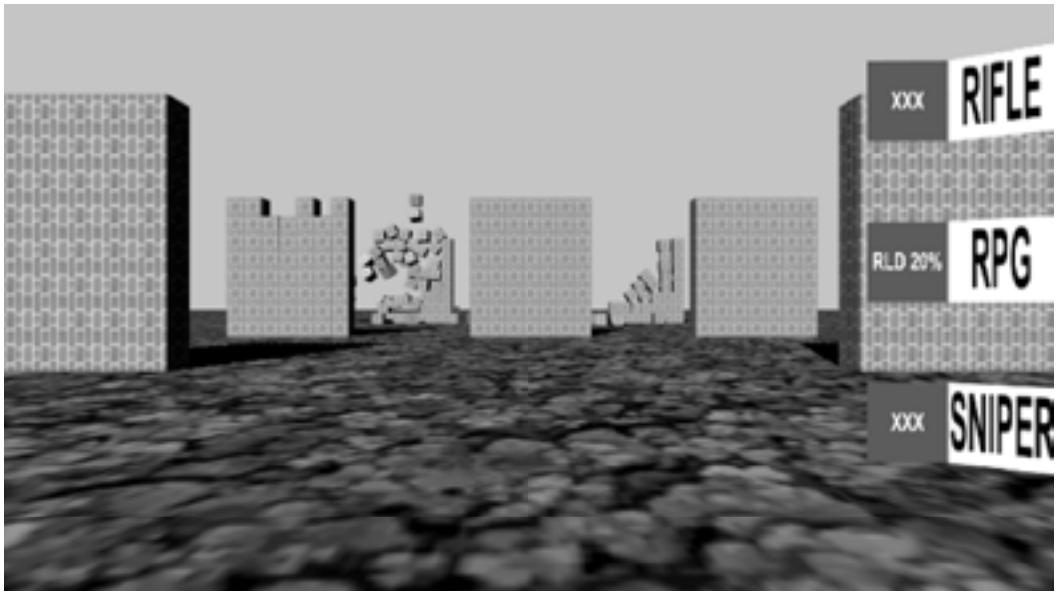


Illustration 92: Weapon switching and reloading demo

Exercises

1. Create a physics door (using hinge joint) that needs two collectable keys to be opened. You can either modify existing scripts used in section 5.1 or write your own scripts to implement the functionality.
2. Create an unlock puzzle that depends on placing three boxes at specific positions on the ground. When the player pushes these boxes to their correct positions, a sliding door opens automatically.
3. Write a script that randomly drops health packs for the player in the game we developed in section 5.3. Each health pack increases player's health by 15, and the player must touch the health pack to collect it. However, if a projectile hits the health pack it must be destroyed immediately.
4. Add grenade weapon to the demo in section 5.4. When the player clicks the mouse, a grenade must be thrown to the direction where the mouse points. This grenade must explode after 7 seconds and destruct any destructible objects in its specified range of effect.