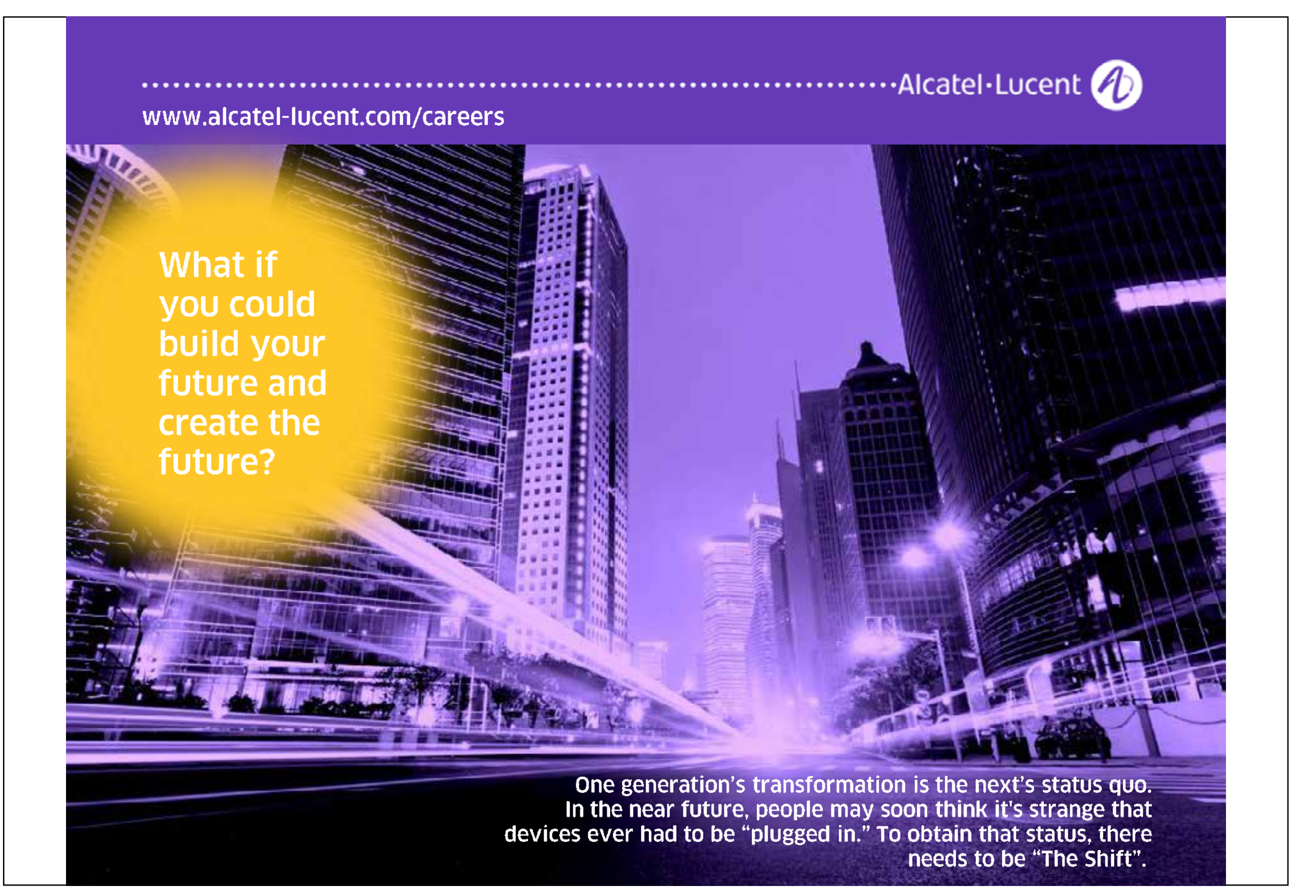
2 Handling User Input

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



Download free eBooks at bookboon.com

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.

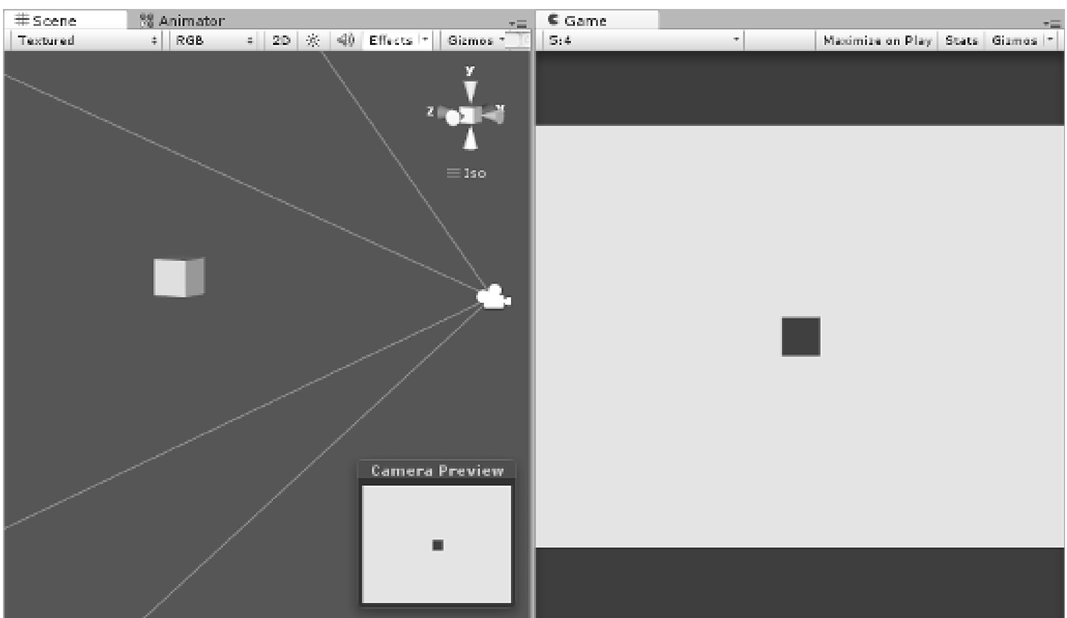


Illustration 16: A simple scene to demonstrate reading keyboard input

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.

Download free eBooks at bookboon.com

```
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.

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

Click on the ad to read more

```
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.

Download free eBooks at bookboon.com

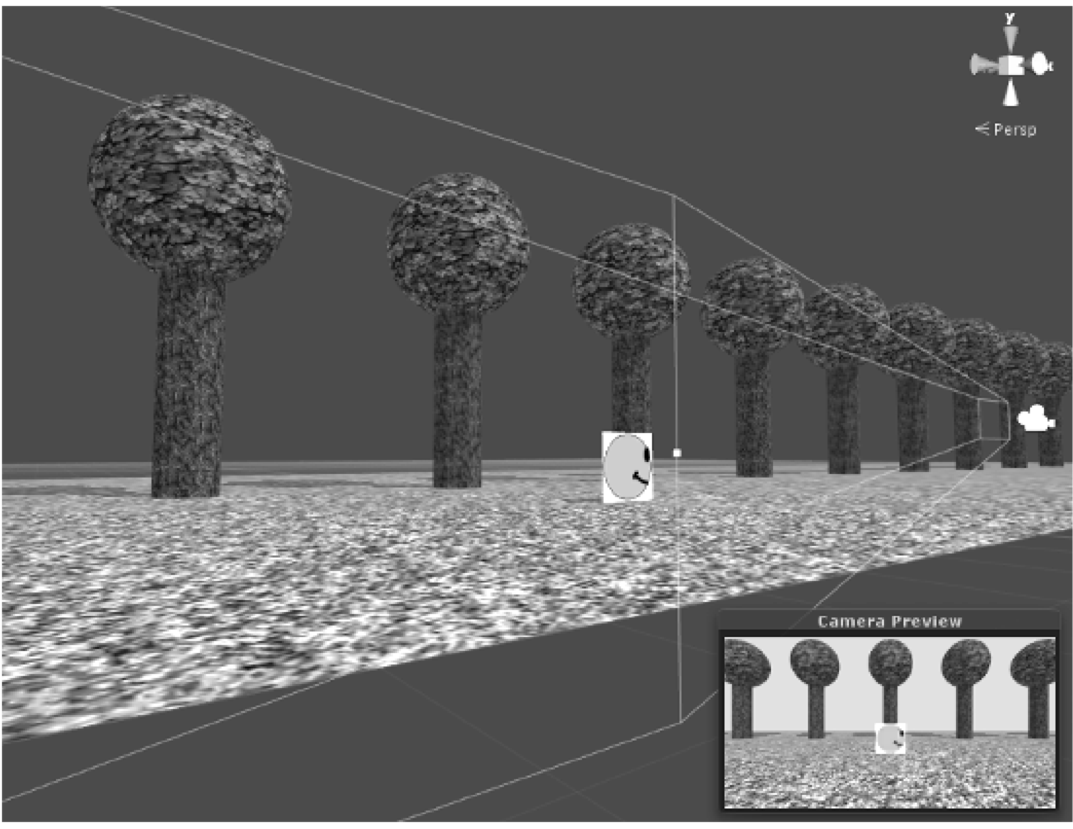


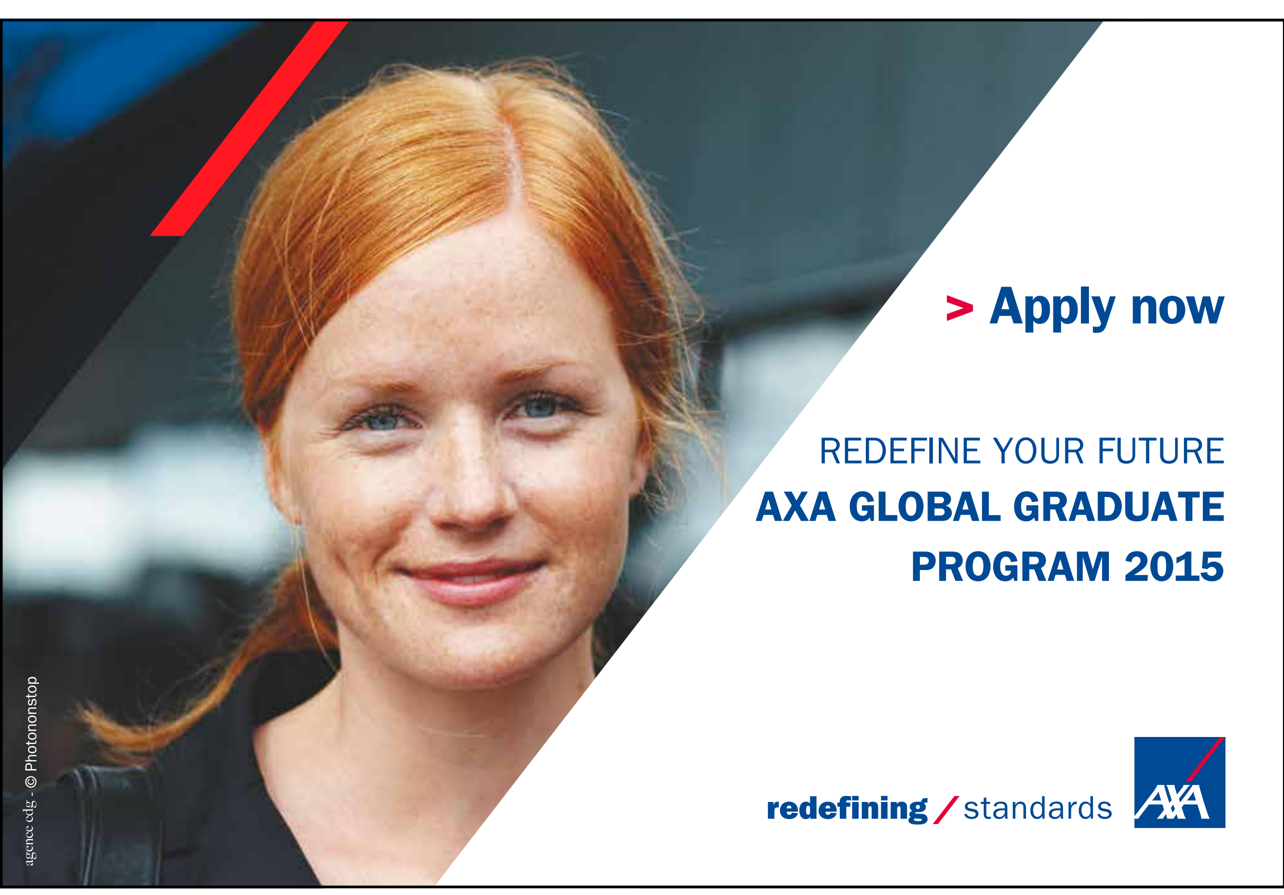
Illustration 17: The scene we are going to use to implement the platformer input system

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.

Download free eBooks at bookboon.com

```
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
```



Download free eBooks at bookboon.com

Click on the ad to read more

```
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.

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

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.

Empowering People.
Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

Download free eBooks at bookboon.com

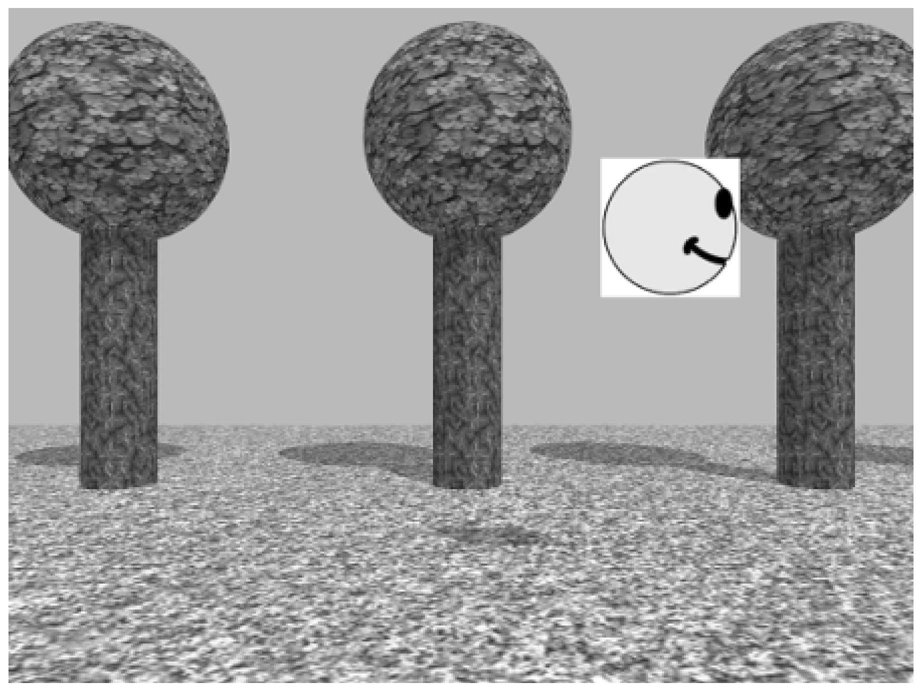


Illustration 18: Implementing the platformer input system and performing jump

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.

Download free eBooks at bookboon.com

```
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

Download free eBooks at bookboon.com

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.

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now

Go to www.helpmyassignment.co.uk for more info

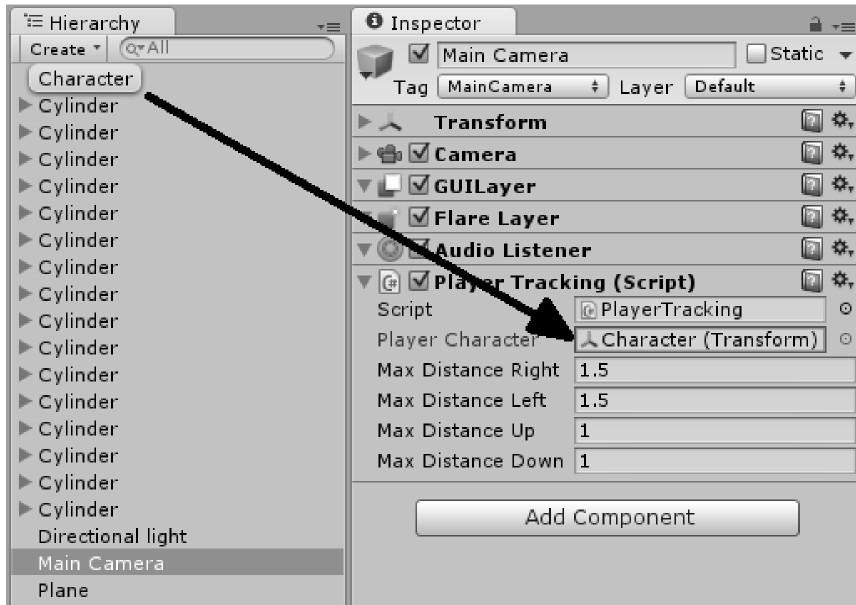


Illustration 19: Binding a game object from the scene with a variable in a script

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.

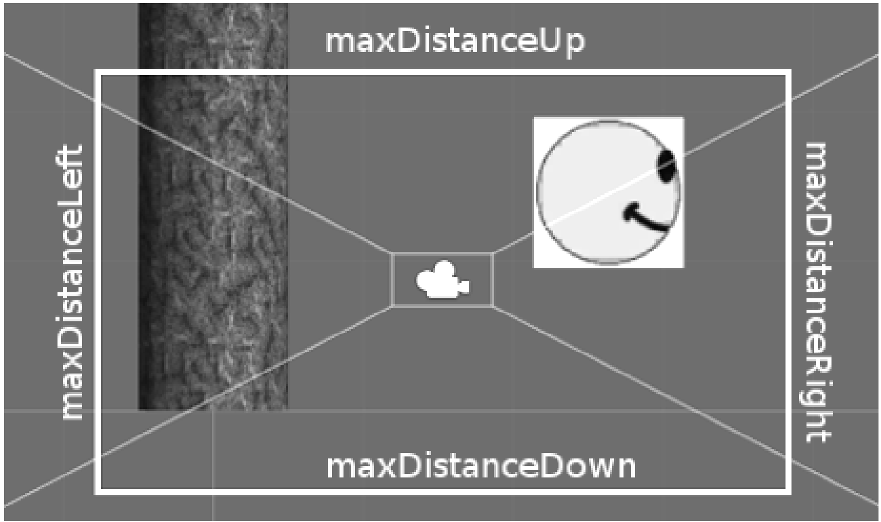


Illustration 20: The area in which the character can move without moving the camera

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

Click on the ad to read more

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.

Download free eBooks at bookboon.com

```
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 () {
```

Click on the ad to read more

```
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.

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

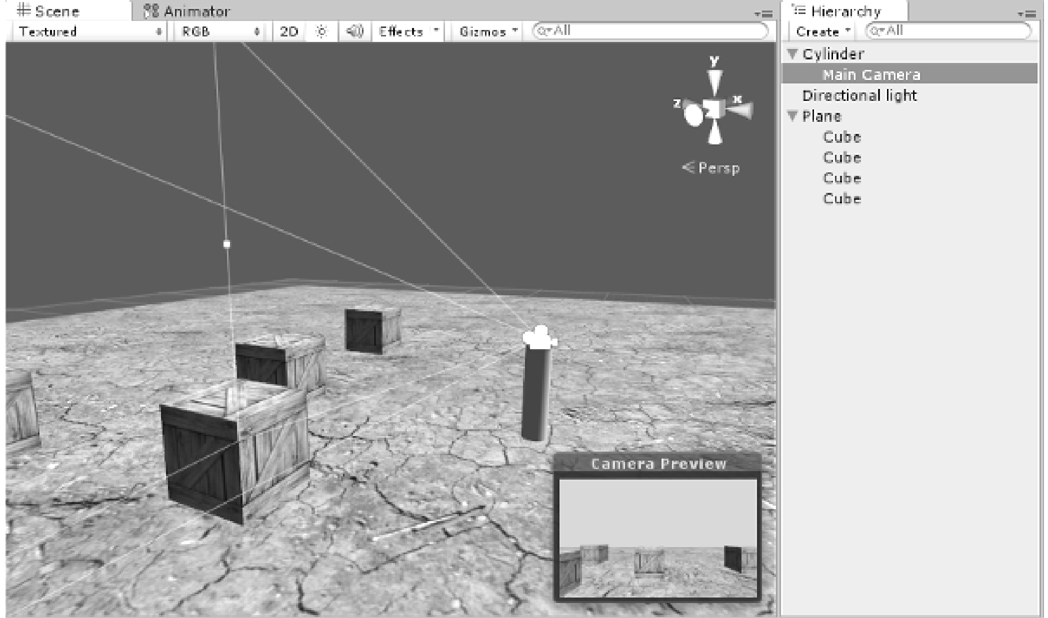
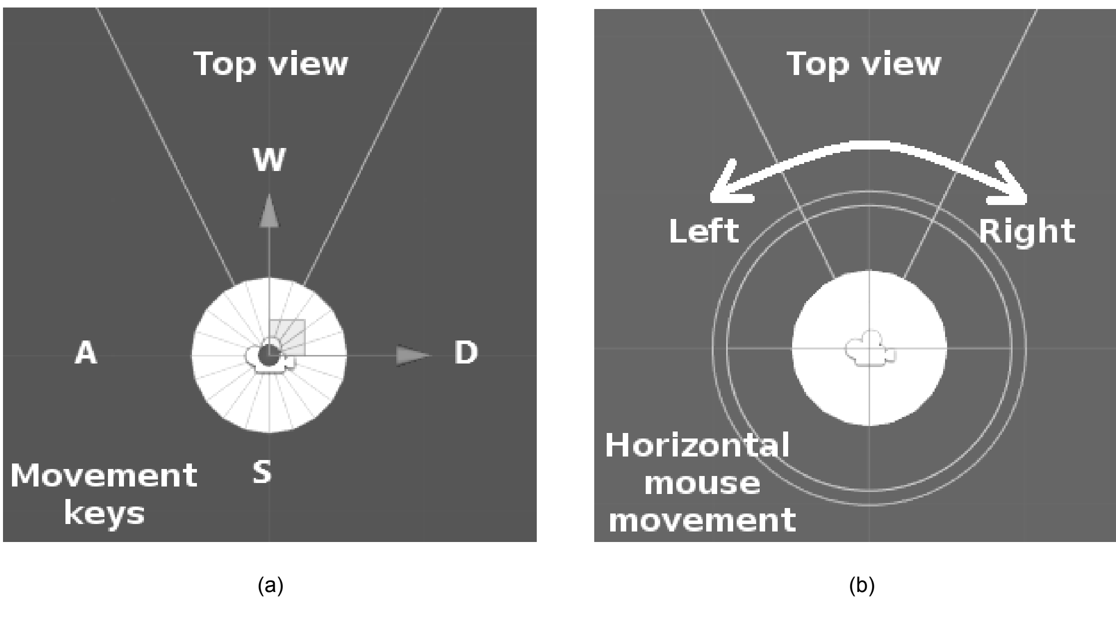


Illustration 21: The cylinder object used to create the first person input system

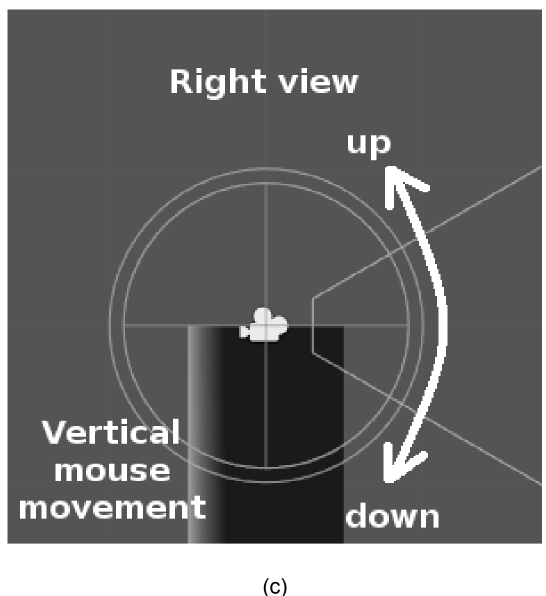
Notice that the camera is added as a child to the cylinder and it resides at the top of the cylinder upper surface. The mouse look mechanism is going to be as follows: when the player moves the mouse horizontally, we rotate the cylinder around the y axis leading to right or left rotation of the cylinder, 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.

Download free eBooks at bookboon.com



(a) (b)



(c)

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.

Download free eBooks at bookboon.com

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;
```

Download free eBooks at bookboon.com

```
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.            }
```

Download free eBooks at bookboon.com

```
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.   }
```

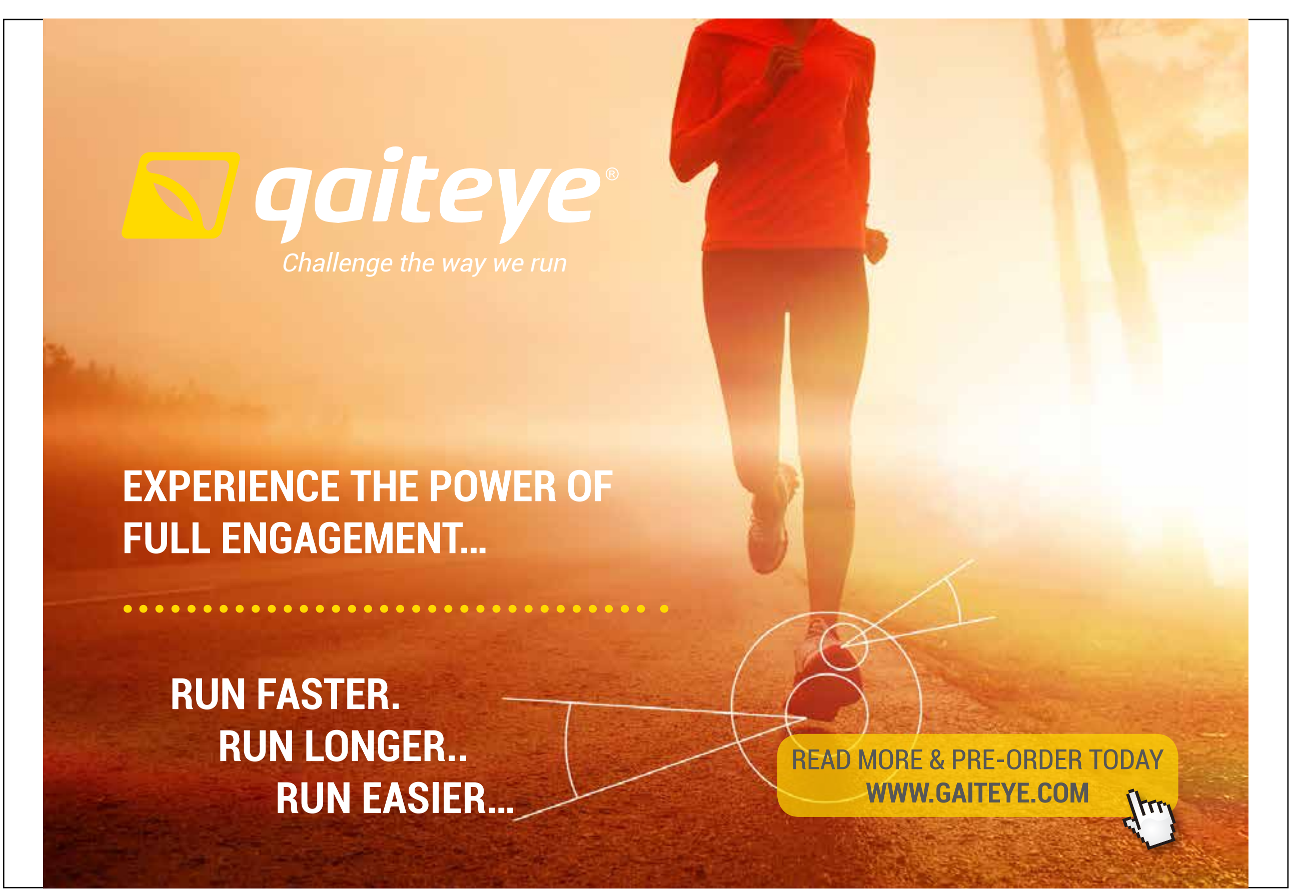
Listing 9: Implementing the first person input system

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.

Download free eBooks at bookboon.com

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.



Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

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.

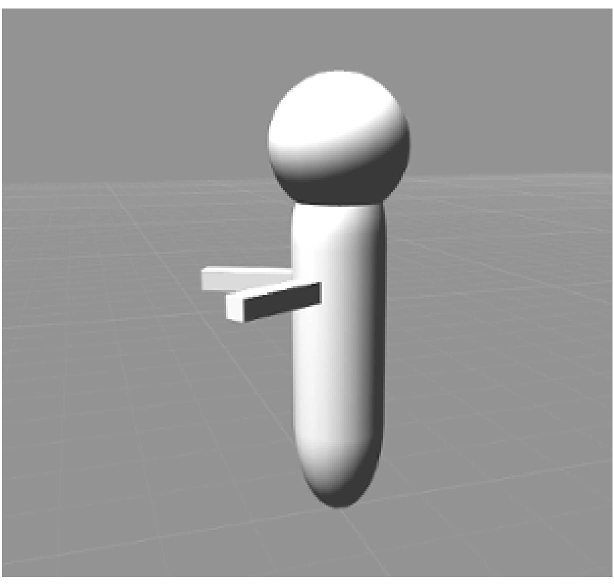


Illustration 23: A simple character we are going to use for the third person input system

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.

Download free eBooks at bookboon.com

```
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;
```

Download free eBooks at bookboon.com

Click on the ad to read more

```
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.

Download free eBooks at bookboon.com

```
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.
```

Download free eBooks at bookboon.com

```
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. }
```

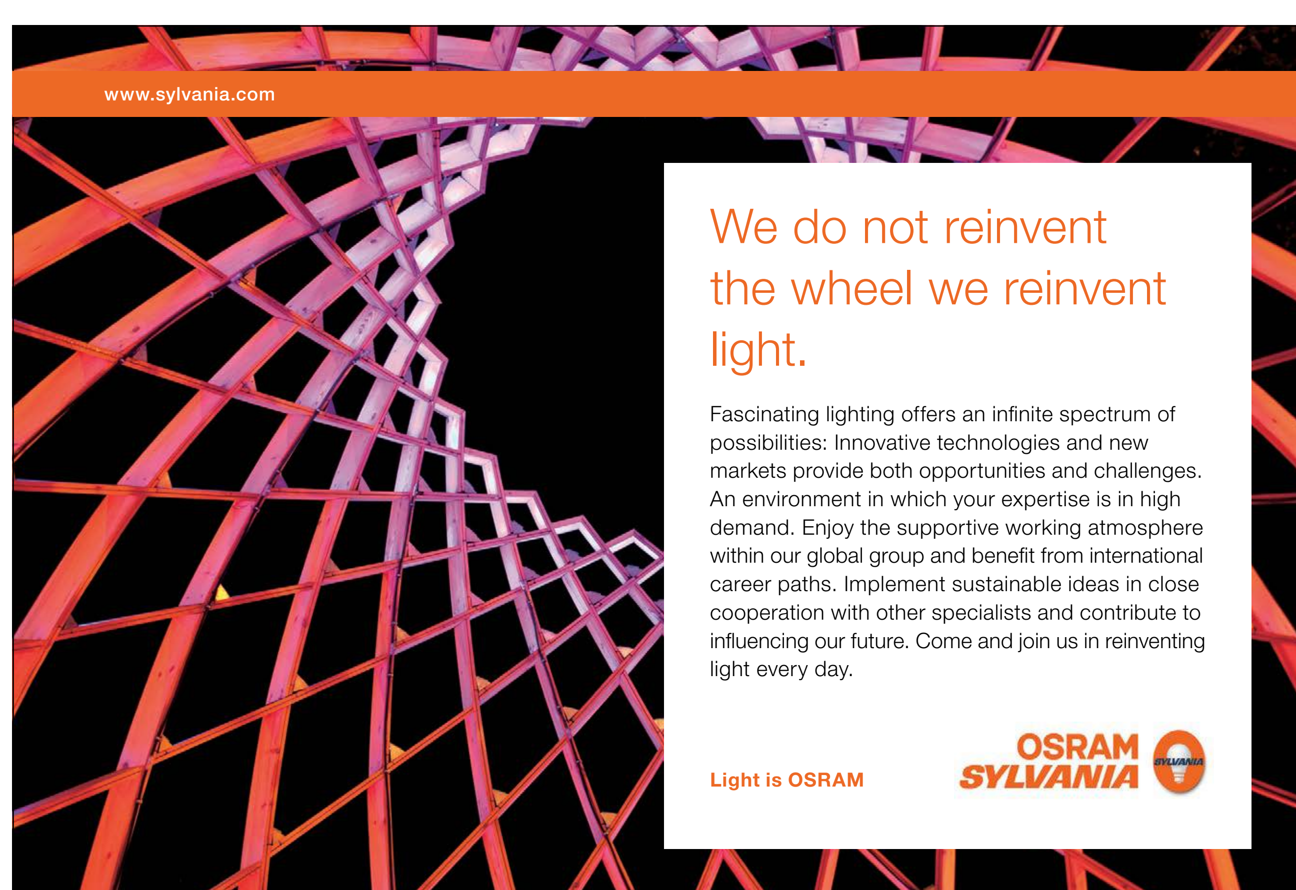
Listing 11: Camera script for the third person input system

Download free eBooks at bookboon.com

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.



Download free eBooks at bookboon.com

Click on the ad to read more

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.

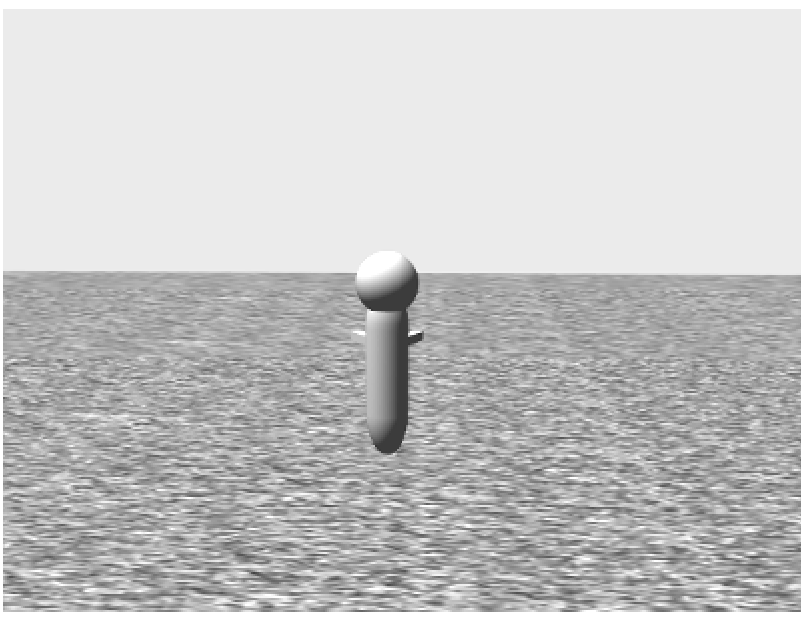


Illustration 24: A simple scene to demonstrate the third person input system

Download free eBooks at bookboon.com

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.

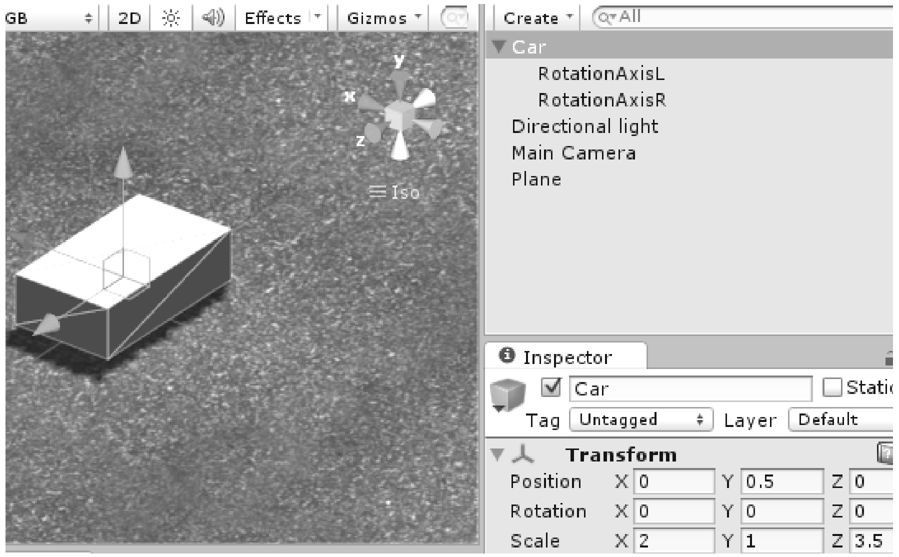


Illustration 25: A scene to demonstrate the car racing input system

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.

Download free eBooks at bookboon.com

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;
```

Download free eBooks at bookboon.com

Click on the ad to read more

```
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.      }
```

Download free eBooks at bookboon.com

```
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

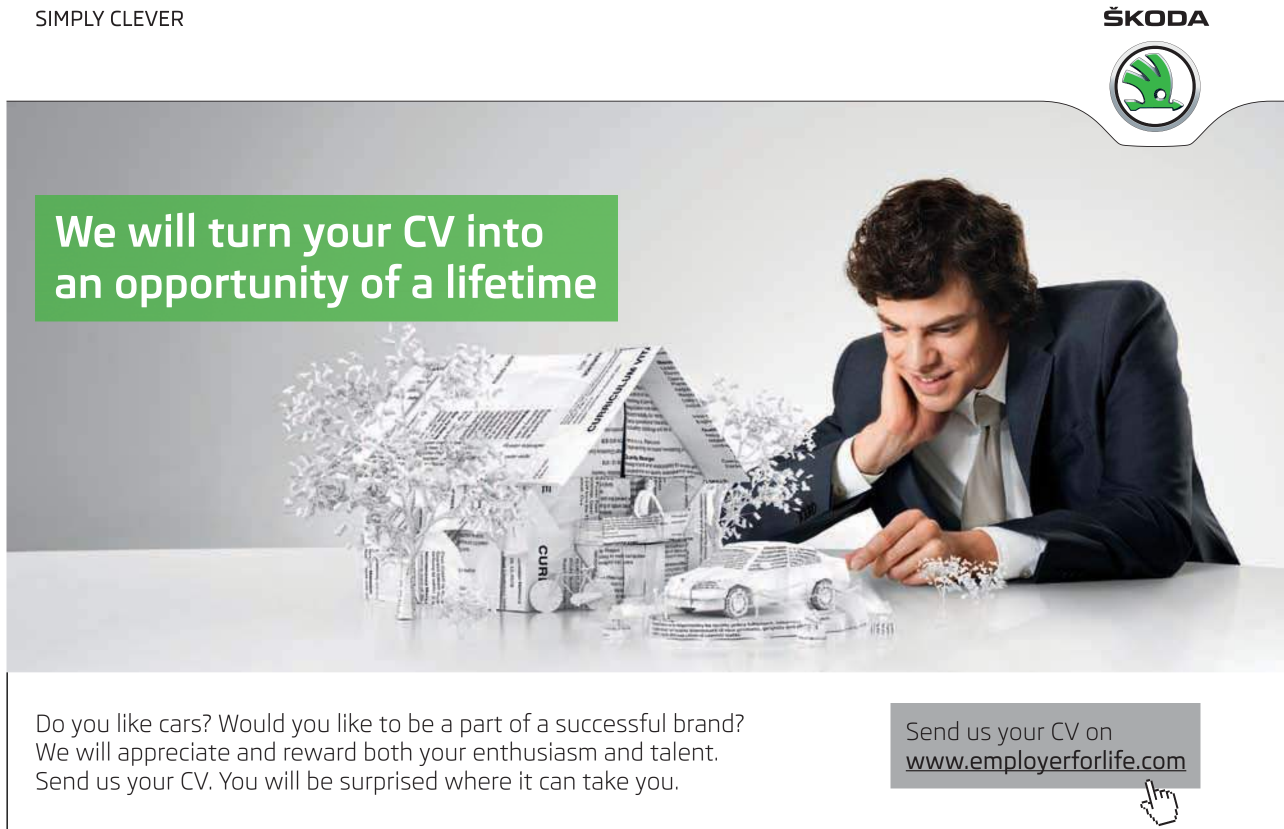
In lines 6 through 19 we declare few variables related to the car speed, including max speed of the car *maxSpeed*, increment of the speed with the time *acceleration*, decrement of the speed with the time *deceleration*, decrement of the speed with the brakes *braking*, and decrement of the speed when 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.

Download free eBooks at bookboon.com

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()*.



Download free eBooks at bookboon.com

Click on the ad to read more

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.

Download free eBooks at bookboon.com

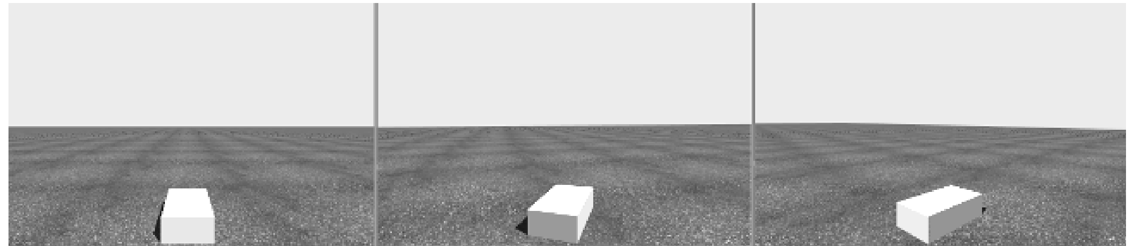


Illustration 26: Car rotation in front of the camera during turning, and the apparition of car side

```
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);
```

Download free eBooks at bookboon.com

```
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

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com
Click on the ad to read more

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.

Download free eBooks at bookboon.com

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.

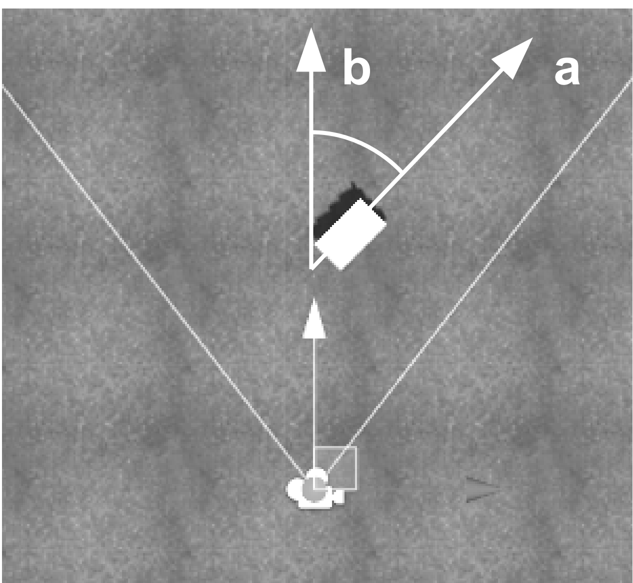


Illustration 27: Angle difference between the car front (a) and the camera direction (b)

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.

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com
Click on the ad to read more

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.

Download free eBooks at bookboon.com

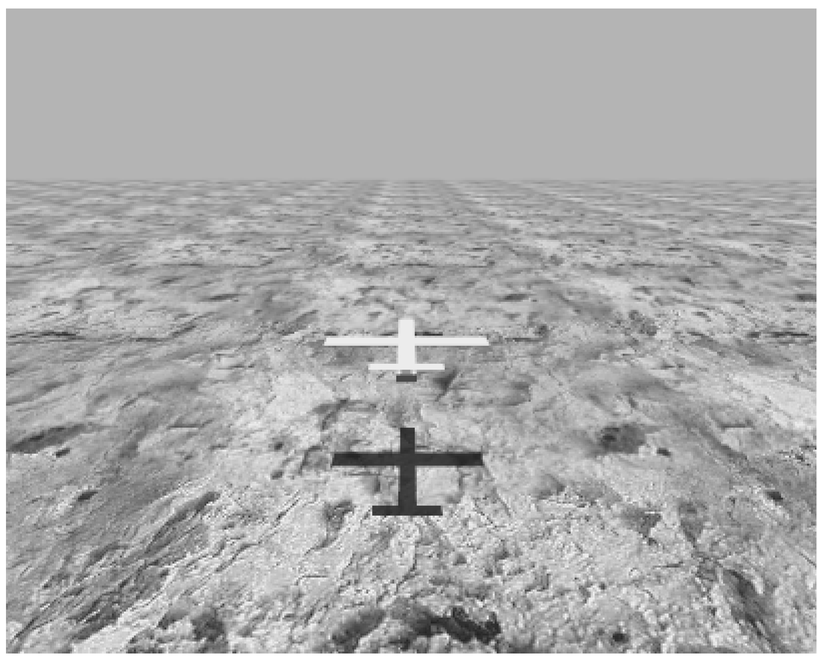


Illustration 29: The plane as seen by the camera during play

```
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;
```

Download free eBooks at bookboon.com

```
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.

Download free eBooks at bookboon.com

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.

Download free eBooks at bookboon.com

Click on the ad to read more