3 Basic Game Logic

Introduction

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

After completing this chapter, you are expected to:

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

3.1 Shooting

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

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

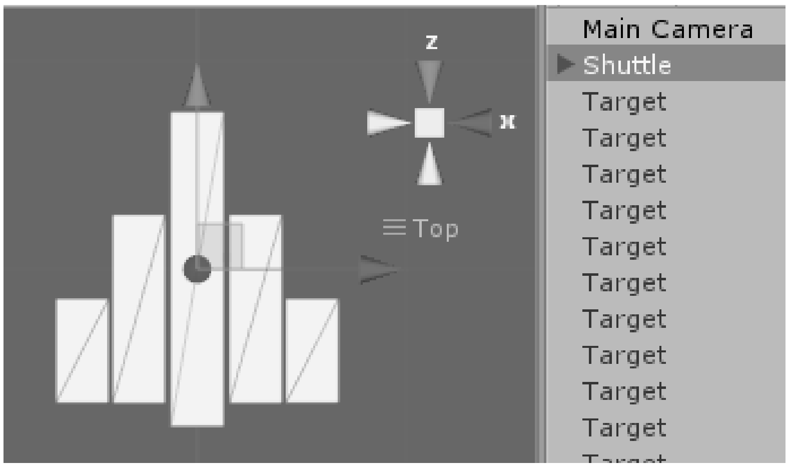


Illustration 30: The space shuttle to be used for shooting projectiles

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

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

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

```
using UnityEngine;
using System.Collections;

public class Target : MonoBehaviour {

     //The bullet sets this value to true when it hits the target
     public bool hit = false;

     //Set this flag to true after calling destroy
     bool destroyed = false;

     void Start () {

     }

     void Update () {
          if(hit){
               //The bullet has hit the target. Play destruction
               //animation, which is rotation and size reduction
               transform.Rotate(0, 720 * Time.deltaTime, 0);
               transform.localScale -= Vector3.one * Time.deltaTime;

               //If we have not called Destroy() yet, call it now
               if(!destroyed){
                      //Delay destruction for one second so the player
                      //can see the animation
                      Destroy(gameObject, 1);
                      //Set the destruction flag to prevent multiple
                      //Destroy() calls
                      destroyed = true;
               }
          }
     }
}
```

Listing 15: Script for targets to be shot

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

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

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

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

```
using UnityEngine;
using System.Collections;

public class AutoMover : MonoBehaviour {

      //Movement speed
      public Vector3 speed = new Vector3(0, 0, 0);

      void Start () {

      }

      void Update () {
            //Move the object with the specified speed
            transform.Translate(speed * Time.deltaTime);
      }
}
```

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

```
using UnityEngine;
using System.Collections;

public class Wrapper : MonoBehaviour {

     //When the target moves out of these bounds,
     //it should be wrapped around the scene
     public Vector3 limits = new Vector3(10, 0, 10);

     void Start () {

     }

     void Update () {
          //Get the current position
          Vector3 newPos = transform.position;

          if(transform.position.x > limits.x){
               //Object left from the right, return it from the left
               newPos.x = -limits.x;
          }

          if(transform.position.x < -limits.x){
               //Object left from the left, return it from the right
               newPos.x = limits.x;
          }

          if(transform.position.z > limits.z){
               ////Object left from the front, return it from the back
               newPos.z = -limits.z;
          }

          //Set the new position after the modifications
          transform.position = newPos;
     }
}
```

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

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

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

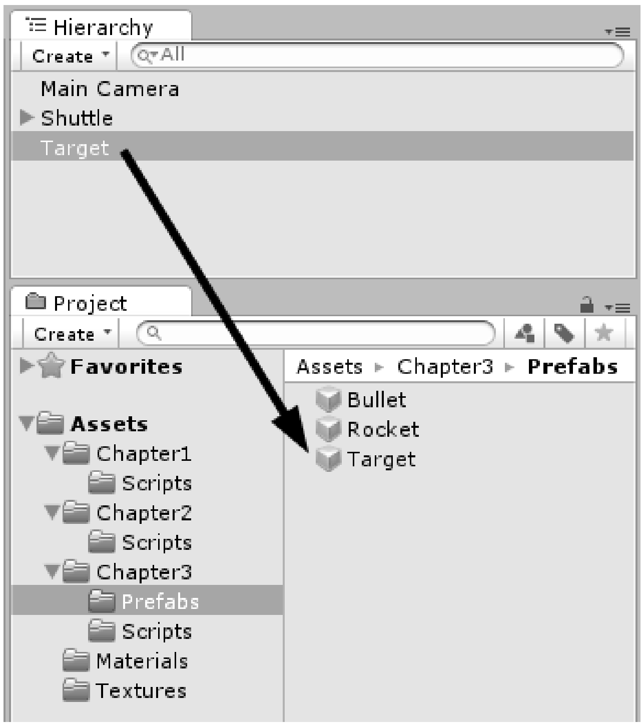


Illustration 31: Creating a new prefab from an object

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

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

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {

     //Movement speed of the projectile in m/s
     public float speed = 15;

     //How many meters can the projectile travel?
     public float range = 20;

     //Compute distance moved so far. When total distance reaches
     //the range, the projectile must be destroyed
     float totalDistance = 0;

     void Start () {

     }

     void Update () {
          //Compute the distance to move in current frame
          float distance = speed * Time.deltaTime;
          //Move the projectile forward on its local z axis
          transform.Translate(0, 0, distance);

          //Add current frame distance to total distance
          totalDistance += distance;

          //When the projectile reaches its range distance
          //we have to destroy it
          if(totalDistance > range){
               Destroy(gameObject);
          }
     }
}
```

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

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

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

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

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

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

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

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

```
using UnityEngine;
using System.Collections;

public class TargetFollower : MonoBehaviour {

     //Target we are following
     Target currentTarget;

     void Start () {
          //At the beginning we find the nearest target
          Target[] allTargets = FindObjectsOfType<Target>();

          //Make sure there are targets in the scene
          if(allTargets.Length > 0){
               //Assume first target is the nearest
               Target nearest = allTargets[0];

               //Find the nearest target
               foreach(Target t in allTargets){
                      //We don't care about targets that have been
                      //already hit
                      if(!t.hit){
                            //Distance between projectile
                            //and current target
                            float distance =
                                   Vector3.Distance(
                                        transform.position,
                                        t.transform.position);

                            //Distance between projectile
                            //and nearest target
                            float minDistance =
                                   Vector3.Distance(
                                        transform.position,
                                        nearest.transform.position);

                            //Update the nearest target if necessary
                            if(distance < minDistance){
                                   nearest = t;
                            }
                      }
               }

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

Listing 20: Target following script of the rocket

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

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

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

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

```
using UnityEngine;
using System.Collections;

public class ShuttleControl : MonoBehaviour {

     //Shuttle movement speed
     public float speed = 7;

     void Start () {

     }

     void Update () {
          //Reads keyboard input and moves
          //the shuttle on local x and z axis
          if(Input.GetKey(KeyCode.UpArrow)){
               transform.Translate(0, 0, speed * Time.deltaTime);
          } else if(Input.GetKey(KeyCode.DownArrow)){
               transform.Translate(0, 0, -speed * Time.deltaTime);
          }

          if(Input.GetKey(KeyCode.RightArrow)){
               transform.Translate(speed * Time.deltaTime, 0, 0);
          } else if(Input.GetKey(KeyCode.LeftArrow)){
               transform.Translate(-speed * Time.deltaTime, 0, 0);
          }
     }
}
```

Listing 21: A script for controlling the shuttle

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

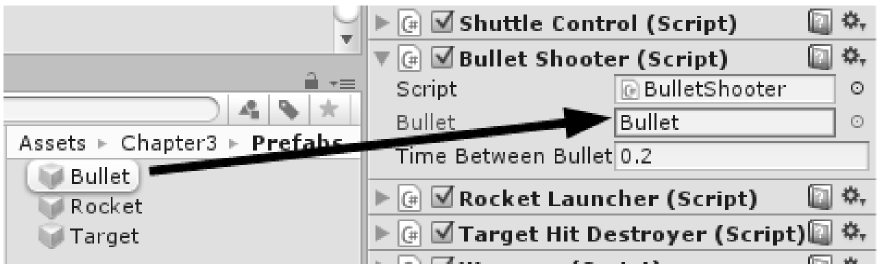
```
using UnityEngine;
using System.Collections;

public class BulletShooter : MonoBehaviour {

	//Prefab of the bullet the shuttle shoots
	public GameObject bullet;

	//How many seconds must pass between two consecutive bullets?
	public float timeBetweenBullets = 0.2f;

	//When did the shuttle shoot the last bullet?
	float lastBulletTime = 0;

	void Start () {

	}

	void Update () {
		//We use left control for bullet shooting
		if(Input.GetKey(KeyCode.LeftControl)){
			//Check if the time between bullets as already passed
			if(Time.time - lastBulletTime > timeBetweenBullets){
				//Create new bullet using the prefab.
				//The bullet is at the same position of the
				//shuttle and looks at the same direction of it
				Instantiate(bullet, //object to create
					transform.position, //position of the object
					transform.rotation);//rotation of the object

				//Register the time in which we shot the bullet
				lastBulletTime = Time.time;
			}
		}
	}
}
```

Listing 22: Bullet shooting script

```
using UnityEngine;
using System.Collections;

public class RocketLauncher : MonoBehaviour {

      //Prefab of the rockets the shuttle launches
      public GameObject rocket;

      void Start () {

      }

      void Update () {
            //We use spacebar (discrete presses) for rocket launching
            if(Input.GetKeyDown(KeyCode.Space)){

                  //How many rockets there are in the scene?
                  TargetFollower[] rockets =
                           FindObjectsOfType<TargetFollower>();

                  //Do not allow more than one rocket at the same time
                  if(rockets.Length == 0){
                           //Create new rocket at the position of the
                           //shuttle and at the same rotation of it
                           Instantiate(rocket,
                                   transform.position, transform.rotation);
                  }
            }
      }
}
```

Listing 23: Rocket launching script

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



Script: BulletShooter
Bullet: Bullet
Time Between Bullet: 0.2

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

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

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

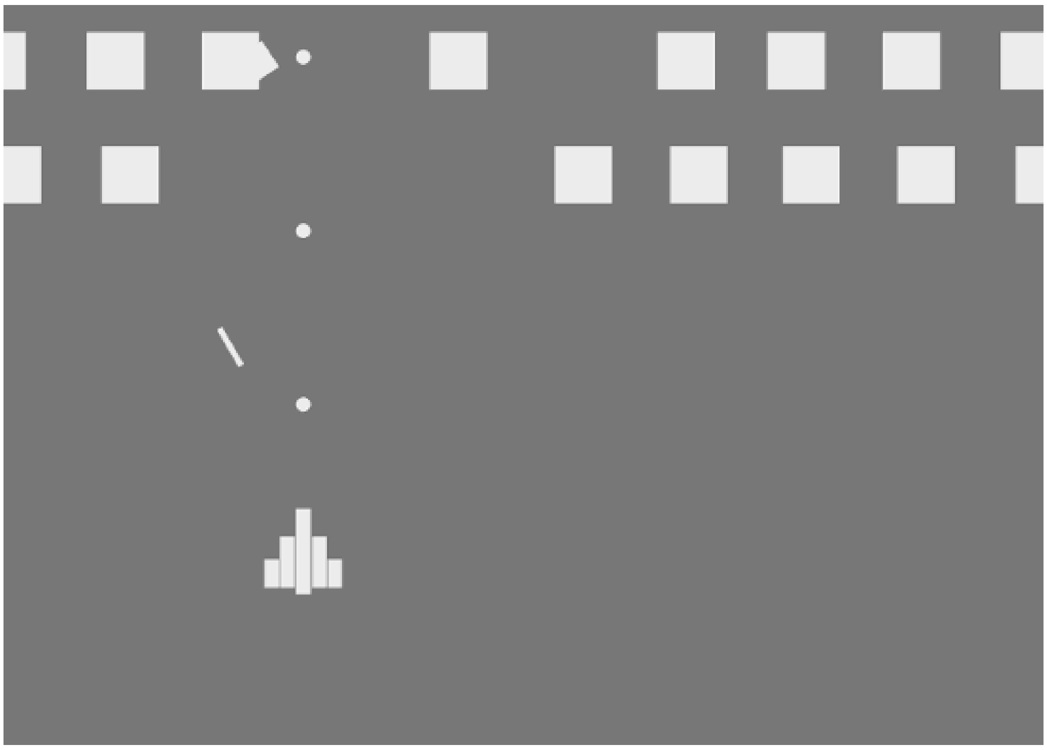


Illustration 33: A screen shot of space shuttle demo

3.2 Collectables

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

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

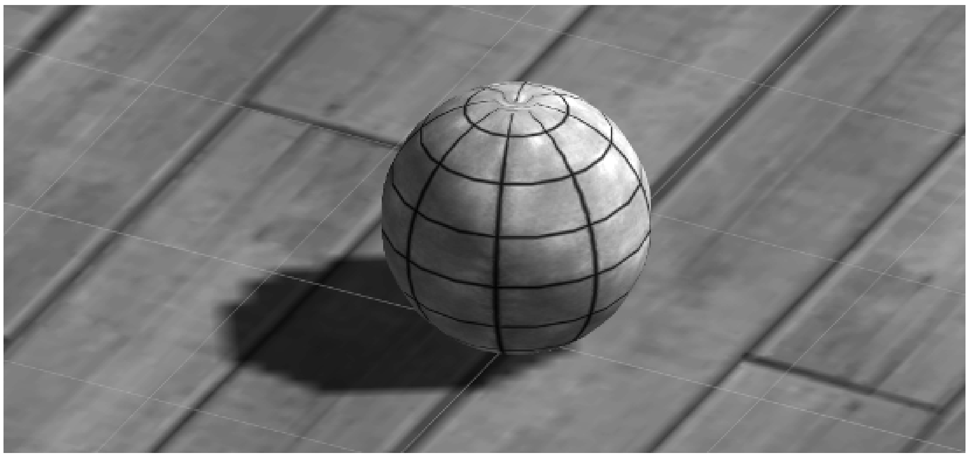


Illustration 34: Glass ball that serves as player character

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

```
using UnityEngine;
using System.Collections;

public class BallRoller : MonoBehaviour {

     //Movement speed on x and z axes
     public float moveSpeed = 5;

     //Rolling speed of the ball
     public float rollSpeed = 360;

     void Start () {

     }

     void Update () {
          //Move along global x axis and roll around global z axis
          if(Input.GetKey(KeyCode.UpArrow)){
               transform.Translate(0, 0,
                      moveSpeed * Time.deltaTime, Space.World);

               transform.Rotate(rollSpeed * Time.deltaTime,
                                      0, 0, Space.World);
          } else if(Input.GetKey(KeyCode.DownArrow)){
               transform.Translate(0, 0,
                      -moveSpeed * Time.deltaTime, Space.World);

               transform.Rotate(-rollSpeed * Time.deltaTime,
                                      0, 0, Space.World);
          }

          //Move along global z axis and roll around global x axis
          if(Input.GetKey(KeyCode.RightArrow)){
               transform.Translate(moveSpeed * Time.deltaTime,
                                      0, 0, Space.World);

               transform.Rotate(0, 0,
                             -rollSpeed * Time.deltaTime, Space.World);
          } else if(Input.GetKey(KeyCode.LeftArrow)){
               transform.Translate(-moveSpeed * Time.deltaTime,
                                      0, 0, Space.World);
               transform.Rotate(0, 0,
                             rollSpeed * Time.deltaTime, Space.World);
          }
     }
}
```

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

```
using UnityEngine;
using System.Collections;

public class ObjectTracker : MonoBehaviour {

      //Object to track
      public Transform objectToTrack;

      void Start () {

      }

      void Update () {
            //Set (x, z) position to (x, z)
            //position of the tracked object
            Vector3 newPos = transform.position;
            newPos.x = objectToTrack.position.x;
            newPos.z = objectToTrack.position.z;
            transform.position = newPos;
      }
}
```

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

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

```
using UnityEngine;
using System.Collections;

public class Collectable : MonoBehaviour {

     //Distance between the center and
     //the external collision surface
     public float radius = 0.5f;

     void Start () {

     }

     void Update () {

     }

          //This function is going to be invoked by the collector
          //When it touches this collectable
          public void Collect(Collector owner){
               //Tell all other scripts in the collectable object
               //to run collection logic if they have
               SendMessage("Collected",
                      owner, SendMessageOptions.RequireReceiver);
     }
}
```

Listing 26: Script for collectable objects

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

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

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

```
using UnityEngine;
using System.Collections;

public class Collector : MonoBehaviour {

      //Radius of collision with collectables
      public float radius = 0.5f;

      void Start () {

      }

      void Update () {
            //Get all collectables in the scene
            Collectable[] allCollectables =
                                    FindObjectsOfType<Collectable>();

            //Check for collision with collectables
            //Use the radii to determine collision distance
            foreach(Collectable col in allCollectables){
                  float distance =
                        Vector3.Distance(transform.position,
                                                col.transform.position);

                  //If the distance is less than the sum of the radii
                  //Then we can try to collect this collectable
                  if(distance < col.radius + radius){
                        //Tell the collectable that this collector
                        //is trying to collect it
                        col.Collect (this);
                  }
            }
      }
}
```

Listing 27: Script for collecting collectable objects

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

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

```
using UnityEngine;
using System.Collections;

public class YRotator : MonoBehaviour {

     //Rotation speed in degree/second
     public float speed = 180;

     //Should the angle be randomized at the beginning?
     public bool randomStartAngle = true;

     void Start () {
          if(randomStartAngle){
               //Rotate with a random angle between 0 and 180
               transform.Rotate(
                      0, Random.Range(0, 180), 0, Space.World);
          }
     }

     void Update () {
          //Simply rotate around global y
          transform.Rotate(
               0, speed * Time.deltaTime, 0, Space.World);
     }
}
```

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

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

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

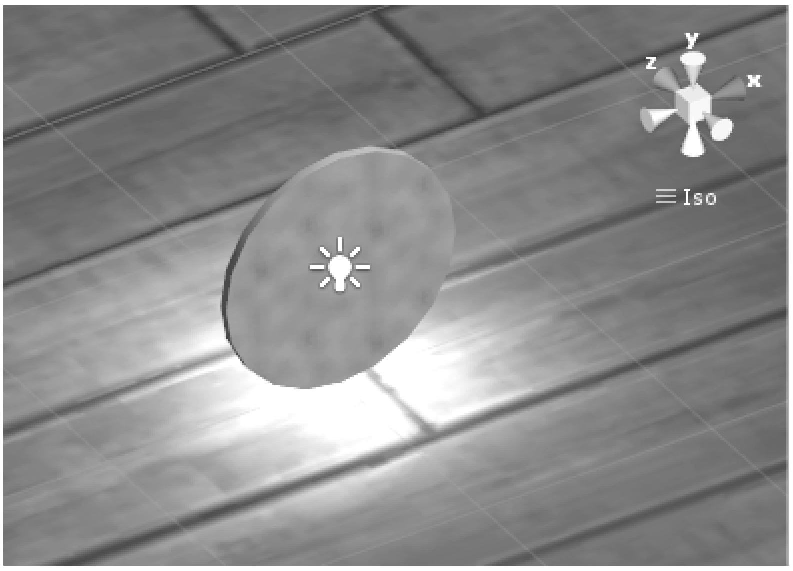


Illustration 35: The coin we are going to use

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

```
using UnityEngine;
using System.Collections;

public class InventoryBox : MonoBehaviour {

	//How much money does the player have?
	public int money = 0;

	void Start () {

	}

	void Update () {

	}
}
```

Listing 29: The inventory box for the collector

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

```
using UnityEngine;
using System.Collections;

public class Coin : MonoBehaviour {

     //This value is going to be added to the
     //money of the inventory box upon collection
     public int coinValue = 1;

     void Start () {

     }

     void Update () {

     }

     //We declare this function to receive Collectable's
     //command to run collection logic
     public void Collected(Collector owner){
          //Check if the collector has an inventory box
          InventoryBox box = owner.GetComponent<InventoryBox>();
          if(box != null){
               //Inventory box exists,
               //so increase money by coin value
               box.money += coinValue;

               //Done, destroy coin object
               Destroy(gameObject);
          }
     }
}
```

Listing 30: The script of a collectable coin

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

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

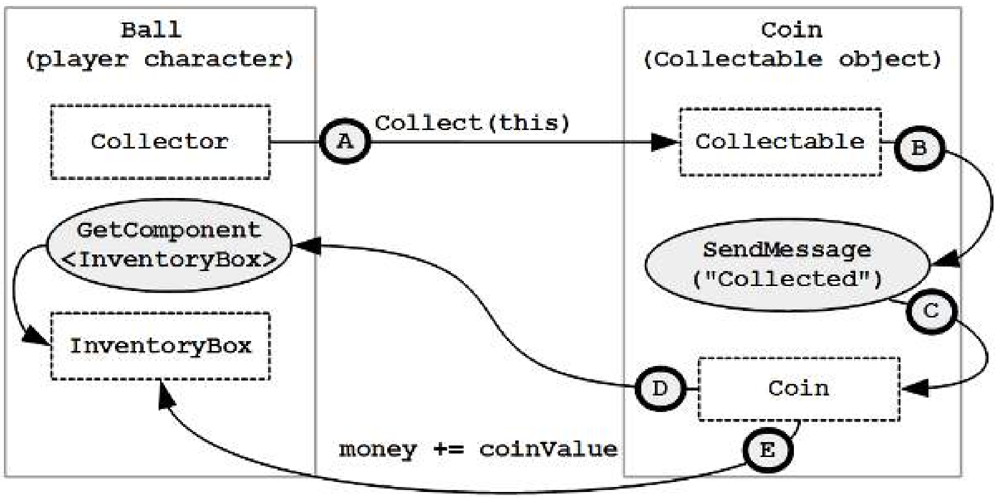


Illustration 36: Coin collection mechanism with the interactions among all involved scripts

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

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

```
using UnityEngine;
using System.Collections;

public class SizeChanger : MonoBehaviour {

    //current size of the object
    float currentSize = 0;

    //Reference to Collector script
    Collector col;

    void Start () {
    col = GetComponent<Collector>();
    }

    void Update () {

    }

    //Called by Food script to try to increase the size
    public bool IncreaseSize(float amount, float duration){
        //Size can be increased only if it is zero
        if(currentSize == 0){
            //Set increment amount and increase the scale
            currentSize = amount;
            transform.localScale = Vector3.one * currentSize;
            //Call DecreaseSpeed() function after duration
            Invoke("DecreaseSize", duration);

            //If there is a collector, increase its radius
            if(col != null){
            col.radius = col.radius * currentSize;
            }
            //Return true = food has been taken
            return true;
        }
        //Return false = food has NOT been taken
        return false;
    }

    //Resets size to one
    public void DecreaseSize(){
        transform.localScale = Vector3.one;
        //If there is a collector, restore its original radius
        if(col != null){
            col.radius = col.radius / currentSize;
        }
        //Set current size back to zero
        currentSize = 0;
    }
}
```

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

```
using UnityEngine;
using System.Collections;

public class Food : MonoBehaviour {

    //Size increment for the food taker
    public float sizeIncrementAmount = 2;

    //How long can this food increase size (seconds)?
    public float incrementDuration = 5;

    void Start () {

    }

    void Update () {

    }

    //We declare this function to receive Collectable's
    //command to run collection logic
    public void Collected(Collector owner){
        //Collector must have a size changer to take food
        SizeChanger changer = owner.GetComponent<SizeChanger>();
        if(changer != null){
            //Size changer found, try to take food
            bool canTake =
                changer.IncreaseSize(
                    sizeIncrementAmount, incrementDuration);
            //Has the food been taken?
            if(canTake){
                //canTake = true, so the food has been taken
                Destroy(gameObject);
            }
        }
    }
}
```

Listing 32: A script for collectable food

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

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

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

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

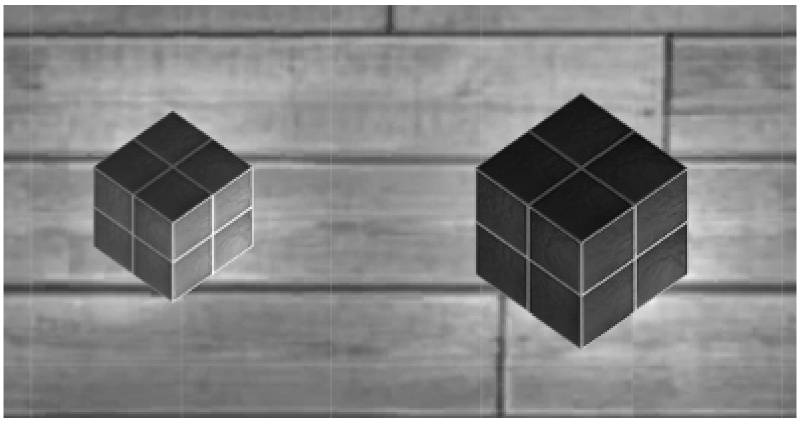


Illustration 37: Red food object (right) and green food object

3.3 Holding and releasing objects

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

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

```
using UnityEngine;
using System.Collections;

public class Holdable : MonoBehaviour {

     //Radius of the holdable object
     public float radius = 1.5f;

     void Start () {

     }

     void Update () {

     }
}
```

Listing 33: Holdable script

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

```
using UnityEngine;
using System.Collections;

public class Holder : MonoBehaviour {

     //Radius of the holder
     public float radius = 0.5f;

     Holdable objectInHand;

     void Start () {

     }

     void Update () {
          if(Input.GetKeyDown(KeyCode.E)){
               //If there is no object in hand,
               //look for one and try to hold it
               if(objectInHand == null){
                    //Get all holdables in the scene
                    Holdable[] allHoldables =
                         FindObjectsOfType<Holdable>();
                    foreach(Holdable holdable in allHoldables){
                         //Find distance between
                         //holder and holdable
                         float distance =
                              Vector3.Distance(
                                        transform.position,
                                        holdable.transform.position);
30.
31.                     //Holdable must be close enough
32.                     bool close =
33.                         distance < radius + holdable.radius;
34.
35.                     //Player must be facing the holdable
36.                     Vector3 dVector;
37.                     dVector = holdable.transform.position
38.                                 - transform.position;
39.
40.                     float ang =
41.                         Vector3.Angle(dVector,
42.                             transform.forward);
43.
44.                     if(close && ang < 90){
45.                         //Now we can hold it
46.                         //1. Set it as object in hands
47.                         objectInHand = holdable;
48.
49.                         //2. Add it as a child to move with
50.                         //the holdable
51.                         holdable.transform.parent = transform;
52.
53.                         return;
54.                     }
55.
56.                 }
57.             } else {
58.                 //There is an object already in hands
59.                 //Now we have to release it
60.                 objectInHand.transform.parent = null;
61.                 objectInHand = null;
62.             }
63.         }
64.     }
65. }
```

Listing 34: Holder script

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

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

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

3.4 Triggers and usable objects

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

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

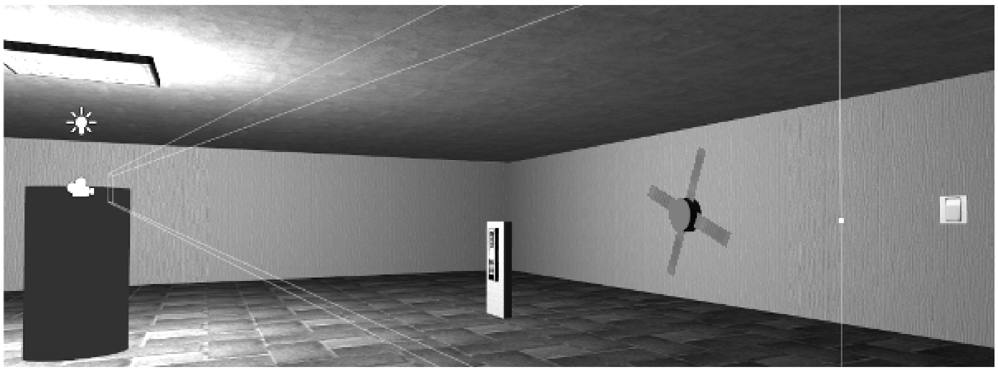


Illustration 38: The scene we are going to use to illustrate triggers and usable objects

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

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

```
using UnityEngine;
using System.Collections;

public class SwitchableTrigger : MonoBehaviour {

      //The trigger switches between these states
      public TriggerState[] states;

      //Index of the current state
      public int currentState = 0;

      //Minimum distance to intarct with this trigger
       public float activationDistance = 3;

      //Last time the current state changed
      float lastSwitchTime = 0;

      void Start () {

      }

      void Update () {

      }

       //Tries to switch the state of the trigger
      //Returns true if switching was successful
      public bool SwitchState(){
           //If states array is empty we do nothing
           if(states.Length == 0){
                      return false;
           }

           //Get the current state
           TriggerState current = states[currentState];

           //Check if the rest time of current state is over
           if(Time.time - lastSwitchTime > current.restTime){
                //It is over, we can switch to next state
                currentState += 1;

                //If we are in the last state, return to the first
                 if(currentState == states.Length){
                      currentState = 0;
                 }

                //Get the new state
                TriggerState newState = states[currentState];

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

Listing 35: Switchable trigger script

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

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

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

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

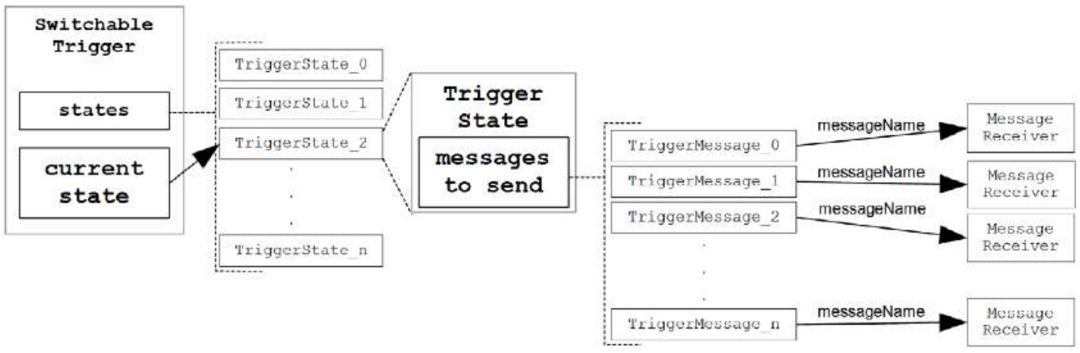


Illustration 39: Triggering mechanism: when the state changes, the new state sends all messages stored in messagesToSend array

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

```
using UnityEngine;
using System.Collections;

public class LightControl : MonoBehaviour {

     //The light we are going to control
     Light toControl;

     void Start () {
          toControl = GetComponent<Light>();
     }

     void Update () {

     }

     //Receives "SwitchOn" message
     public void SwitchOn(){
          toControl.enabled = true;
     }

     //Receives "SwitchOff" message
     public void SwitchOff(){
          toControl.enabled = false;
     }
}
```

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

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

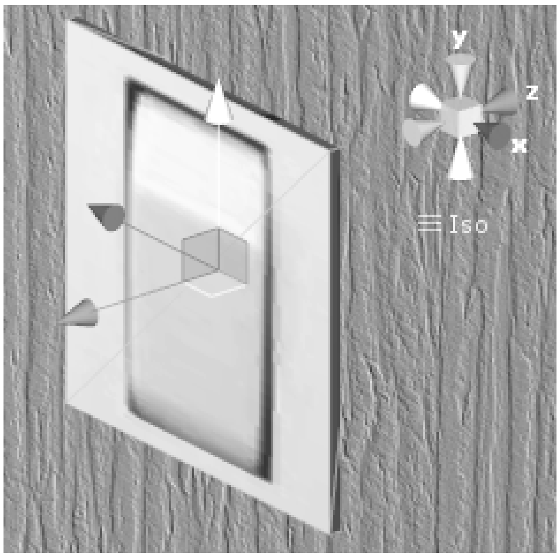


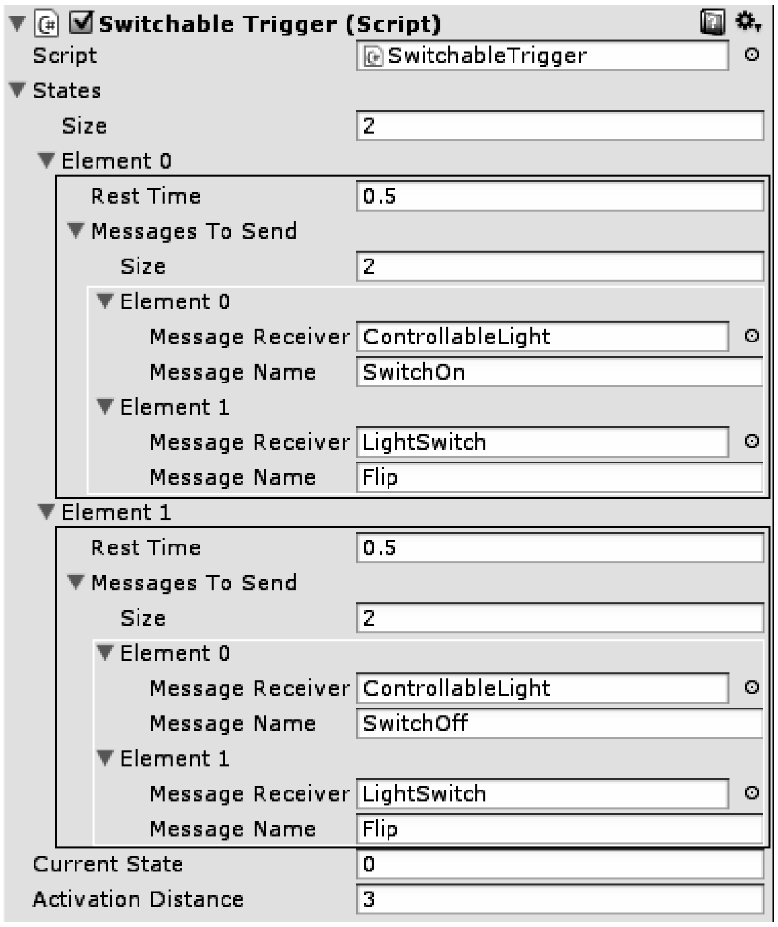
Illustration 40: Power switch object

ZFlipper script is shown in Listing 37.

```
using UnityEngine;
using System.Collections;

public class ZFlipper : MonoBehaviour {

     void Start () {

     }

     void Update () {

     }
     //Rotates 180 degree around local z axis
     public void Flip(){
          transform.Rotate(0, 0, 180);
     }
}
```

Listing 37: A simple script to flip switch object

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



```
▼ Switchable Trigger (Script)
  Script                              SwitchableTrigger
▼ States
    Size                              2
  ▼ Element 0
      Rest Time                       0.5
    ▼ Messages To Send
        Size                          2
      ▼ Element 0
          Message Receiver            ControllableLight
          Message Name                SwitchOn
      ▼ Element 1
          Message Receiver            LightSwitch
          Message Name                Flip
  ▼ Element 1
      Rest Time                       0.5
    ▼ Messages To Send
        Size                          2
      ▼ Element 0
          Message Receiver            ControllableLight
          Message Name                SwitchOff
      ▼ Element 1
          Message Receiver            LightSwitch
          Message Name                Flip
  Current State                       0
  Activation Distance                 3
```

Illustration 41: Light switch trigger completely configured

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

```
using UnityEngine;
using System.Collections;

public class TriggerSwitcher : MonoBehaviour {

     void Start () {

     }

     void Update () {
          if(Input.GetKeyDown(KeyCode.E)){
               //Find all switchable triggers in the scene
               SwitchableTrigger[] allST =
                      FindObjectsOfType<SwitchableTrigger>();

               //search for suitable trigger
               //Suitable = near + facing
               foreach(SwitchableTrigger st in allST){
                      float dist =
                             Vector3.Distance(transform.position,
                                               st.transform.position);

                      //If distance less than activationDistance,
                      //of the trigger, then it is close.
                      if(dist < st.activationDistance){
                             Vector3 distVector =
                                    st.transform.position
                                           - transform.position;

                             float angle =
                                    Vector3.Angle(distVector,
                                                   transform.forward);
                             //If angle < 90, it is facing
                             if(angle < 90){
                                    //facing trigger = we can use it
                                    st.SwitchState();
                             }
                      }
               }
          }
     }
}
```

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

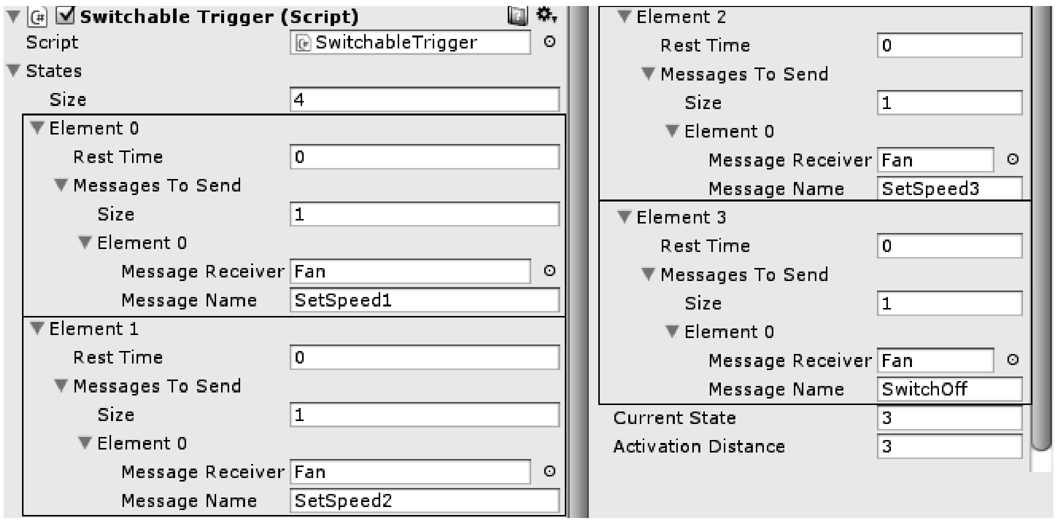
When the player presses E key, the script searches for all switchable triggers in the scene, and finds the distance between the player and each one. If the distance is less than activation distance set in the trigger, and the player is facing the trigger with an angle less than 90, *SwitchState()* function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

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

```
using UnityEngine;
using System.Collections;

public class ControllableFan : MonoBehaviour {

     public float speed1 = 20;
     public float speed2 = 40;
     public float speed3 = 60;

     float currentSpeed = 0;

     void Start () {

     }

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

     public void SetSpeed1(){
          currentSpeed = speed1;
     }

     public void SetSpeed2(){
          currentSpeed = speed2;
     }

     public void SetSpeed3(){
          currentSpeed = speed3;
     }

     public void SwitchOff(){
          currentSpeed = 0;
     }
}
```

Listing 39: Fan script

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



```
Switchable Trigger (Script)
Script                      SwitchableTrigger
States
  Size                      4
  Element 0
    Rest Time               0
    Messages To Send
      Size                  1
      Element 0
        Message Receiver    Fan
        Message Name        SetSpeed1
  Element 1
    Rest Time               0
    Messages To Send
      Size                  1
      Element 0
        Message Receiver    Fan
        Message Name        SetSpeed2
  Element 2
    Rest Time               0
    Messages To Send
      Size                  1
      Element 0
        Message Receiver    Fan
        Message Name        SetSpeed3
  Element 3
    Rest Time               0
    Messages To Send
      Size                  1
      Element 0
        Message Receiver    Fan
        Message Name        SwitchOff
Current State               3
Activation Distance         3
```

Illustration 42: Configured switchable trigger for the fan switch

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

Exercises

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