5 Advanced Game Logic

Introduction

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

After completing this chapter, you are expected to:

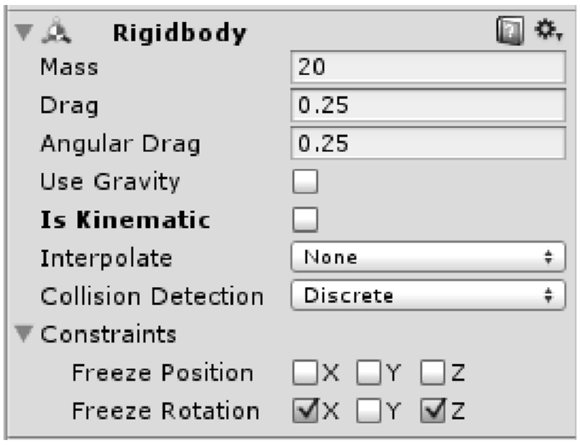
- Make doors, locks, and keys
- Program simple puzzles and unlock combinations
- Program player's health, lives, and score
- Program different types of weapons with ammo and reload mechanism

5.1 Doors, locks, and keys

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

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

Illustration 76: A room with a basic rotating door



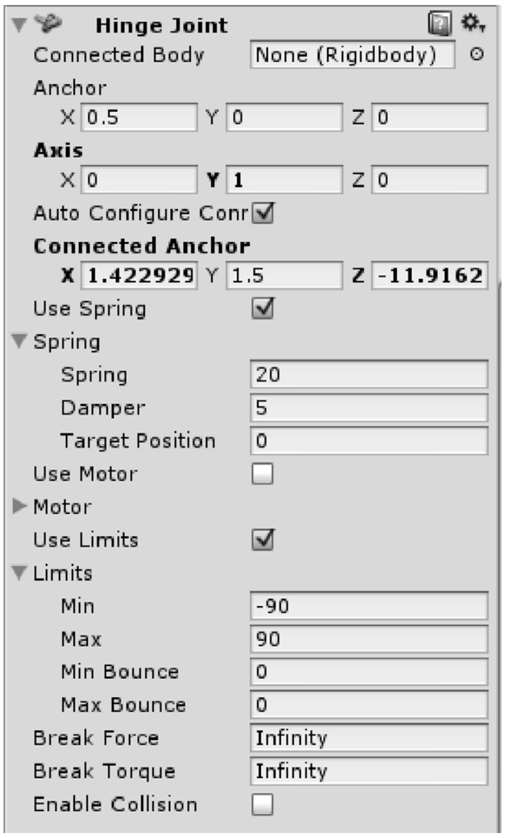
Rigidbody

Mass	20
Drag	0.25
Angular Drag	0.25
Use Gravity	☐
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	☐ X ☐ Y ☐ Z
Freeze Rotation	✓ X ☐ Y ✓ Z

Illustration 77: Rigid body configuration of the rotating door

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

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



Hinge Joint

Connected Body	None (Rigidbody)
Anchor	X 0.5 Y 0 Z 0
Axis	X 0 **Y 1** Z 0
Auto Configure Conr	✓
Connected Anchor	**X 1.422929** Y 1.5 **Z -11.9162**
Use Spring	✓
Spring	
Spring	20
Damper	5
Target Position	0
Use Motor	☐
Motor	
Use Limits	✓
Limits	
Min	-90
Max	90
Min Bounce	0
Max Bounce	0
Break Force	Infinity
Break Torque	Infinity
Enable Collision	☐

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

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

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

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

public class InventoryBox : MonoBehaviour {

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

	//What keys does the player have?
	public List<string> keys;

	void Start () {

	}

	void Update () {

	}
}
```

Listing 61: The modified version of *InventoryBox* script

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

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

```
using UnityEngine;
using System.Collections;

public class CollectableKey : MonoBehaviour {

     //Key to give to the player
     public string key;

     void Start () {

     }

     void Update () {

     }

     //Receive collect message
     public void Collect(GameObject owner){
          //Find the inventory box of the owner
          //Give the key to the owner by adding it
          //to the list of keys the owner has
          InventoryBox box = owner.GetComponent<InventoryBox>();
          if(box != null){
               box.keys.Add(key);
               //Finally, destroy the key object
               Destroy (gameObject);
          }
     }
}
```

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

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

```
using UnityEngine;
using System.Collections;

public class CollisionCollector : MonoBehaviour {

     void Start () {

     }

     void Update () {

     }

     //Collect the collectable on collision
     void OnCollisionEnter(Collision col){
          SendCollectMessage(col.gameObject);
     }

     //Collict on trigger hit
     void OnTriggerEnter(Collider col){
          SendCollectMessage(col.gameObject);
     }

     void SendCollectMessage(GameObject target){
          //Send collect message to the colliding object.
          //Provide self as owner of what is to be collected
          target.gameObject.SendMessage("Collect",
               gameObject, //owner
               SendMessageOptions.DontRequireReceiver);
     }
}
```

Listing 63: Collector script based on collisions

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

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

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

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

public class PhysicsKeyLock : MonoBehaviour {

      //Unique string to unlock this lock
      public string unlockKey;

      void Start () {
      Lock ();
      }

      void Update () {

      }

      //Lock the key by setting the rigid body to kinematic
      public void Lock(){
            rigidbody.isKinematic = true;
      }

      //Try to unlock using the provided keys
      public void Unlock(ICollection<string> keys){

            if(!rigidbody.isKinematic){
                  return;
            }

            //If one of the keys match, then unlock
            foreach(string key in keys){
                  if(unlockKey.Equals(key)){
                        //Tell other scripts that unlocked succeded
                        SendMessage("OnUnlock",
                              SendMessageOptions.DontRequireReceiver);

                        rigidbody.isKinematic = false;
                        return;
                  }
            }

            //Tell other scripts that unlocking has failed
            SendMessage("OnUnlockFail",
                  SendMessageOptions.DontRequireReceiver);
      }
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

public class TouchUnlocker : MonoBehaviour {

     void Start () {

     }

     void Update () {

     }

     //Send unlock message to colliding object
     void OnCollisionEnter(Collision col){
          //Get the inventory box
          InventoryBox box = GetComponent<InventoryBox>();

          //Try all keys in the inventory box with the lock
          col.gameObject.SendMessage("Unlock",
               box.keys, //Colletion of keys to try
               SendMessageOptions.DontRequireReceiver);

     }

}
```

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

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

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

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

public class GeneralDoor : MonoBehaviour {

	//Is the door initially open?
	public bool initiallyOpen = false;

	//Key to unlock the door
	public string unlockKey;

	//Internal state storage
	bool isOpen;

	//Internal state of lock
	bool locked;

	void Start () {
		//Lock the door if there is an unlock key provided
		locked = !string.IsNullOrEmpty(unlockKey);
		//Set the initial state of the door
		isOpen = initiallyOpen;
	}

	void Update () {

	}

	//Open the door if not locked
	public void Open(){
		if(!locked){
			isOpen = true;
		}
	}

	//Close the door if not locked
	public void Close(){
		if(!locked){
			isOpen = false;
		}
	}

	//Lock the door
	public void Lock(){
		locked = true;
	}

	//Try to unlock the door using provided keys
	public void Unlock(ICollection<string> keys){
		//Check if it already unlocked
		if(!IsLocked()){
			return;
		}
		//Try all keys to unlock the door
		foreach(string key in keys){
```

```
            if(key.Equals(unlockKey)){
                //Tell other scripts that the door has been unlocked
                SendMessage("OnUnlock",
                    SendMessageOptions.DontRequireReceiver);

                locked = false;
                return;
            }
        }
        //Tell other scripts that unlocking failed
        SendMessage("OnUnlockFail",
            SendMessageOptions.DontRequireReceiver);
    }

    //Is the door currently locked?
    public bool IsLocked(){
        return locked;
    }

    //Is the door currently open?
    public bool IsOpen(){
        return isOpen;
    }

    //Switch the state of the door
    public void Switch(){
        if(IsOpen()){
            Close();
        } else {
            Open();
        }
    }
}
```

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(GeneralDoor))]
public class SlidingDoor : MonoBehaviour {

    //Relative new position of door when opened
    public Vector3 slidingDirection = Vector3.up;
    //Speed of door movement
    public float speed = 2;
    //Store close and open positions
    Vector3 originalPosition, slidingPosition;
    //Reference to general door script
    GeneralDoor door;
    //Current state of the door
    SlidingDoorState state;

    void Start () {
        //Initialize the variables
        door = GetComponent<GeneralDoor>();
        originalPosition = transform.position;
        slidingPosition = transform.position + slidingDirection;
        state = SlidingDoorState.close;
    }

    void Update () {
        if(door.IsOpen()){
```

Excellent Economics and Business programmes at:
university of groningen
AACSB ACCREDITED
"The perfect start of a successful, international career."
CLICK HERE to discover why both socially and academically the University of Groningen is one of the best places for a student to be
www.rug.nl/feb/education

```
            //The door must be open
            if(state != SlidingDoorState.open){
                //The door is not open, move it
                //smoothly towards open position
                transform.position =
                            Vector3.Lerp(
                            transform.position,
                            slidingPosition,
                            Time.deltaTime * speed);

                float remaining =
                    Vector3.Distance(
                        transform.position, slidingPosition);

                //Check if door reached open position
                if(remaining < 0.01f){
                    //Open position reached:
                    //change state of the door
                    state = SlidingDoorState.open;
                    transform.position = slidingPosition;
                    //Inform other scripts about open completion
                    SendMessage("OnOpenComplete",
                        SendMessageOptions.DontRequireReceiver);

                } else if(state != SlidingDoorState.openning){
                    //Door just started to open,
                    //send a message to inform about that
                    SendMessage("OnOpenStart",
                        SendMessageOptions.DontRequireReceiver);

                    state = SlidingDoorState.openning;
                }
            }
        } else {
            //The door must be close
            if(state != SlidingDoorState.close){
                //The door is not close, move it
                //smoothly towards close position
                transform.position =
                    Vector3.Lerp(
                            transform.position,
                            originalPosition,
                            Time.deltaTime * speed);
                float remaining =
                    Vector3.Distance(
                        transform.position, slidingPosition);

                //Check if door reached close position
                if(remaining < 0.01f){
                    //Close position reached:
                    //change state of the door
                    state = SlidingDoorState.close;
                    transform.position = originalPosition;
                    //Inform other scripts about close completion
                    SendMessage("OnCloseComplete",
83.                         SendMessageOptions.DontRequireReceiver);
84.
85.                 } else if(state != SlidingDoorState.closing){
86.                     //Door just started to close,
87.                     //send a message to inform about that
88.                     SendMessage("OnCloseStart",
89.                         SendMessageOptions.DontRequireReceiver);
90.
91.                     state = SlidingDoorState.closing;
92.                 }
93.             }
94.         }
95.     }
96.
97.     void OnCollisionEnter(Collision col){
98.         if(state == SlidingDoorState.closing){
99.             //Something interrupted the door while closing
100.            //Inform about that
101.            SendMessage("OnCloseInterruption",
102.                col.gameObject,
103.                SendMessageOptions.DontRequireReceiver);
104.        }
105.    }
106.
107.    //Enumeration of different door states
108.    enum SlidingDoorState{
109.        open, close, openning, closing
110.    }
111. }
```

Listing 67: A script that implements sliding door functionality

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

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

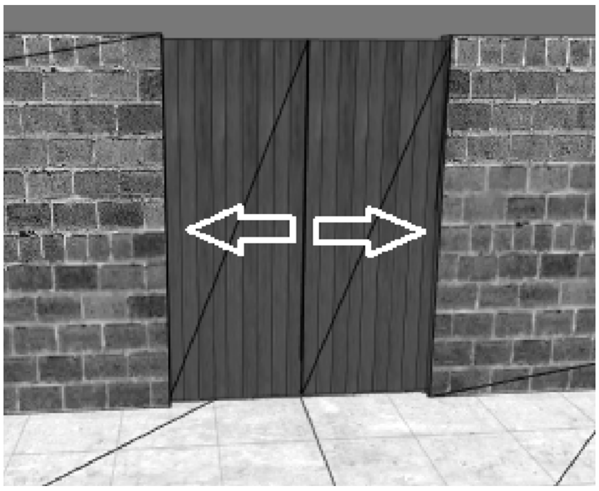


Illustration 80: A sliding door with two parts. Arrows indicate sliding direction of each part.

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

5.2 Puzzles and unlock combinations

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

```
using UnityEngine;
using System.Collections;

public class CentralLock : MonoBehaviour {

     //Door object(s)
     public GeneralDoor[] targetDoors;

     //Keys to unlock the doors
     public string[] keys;

     //Should the doors be opened after unlocking?
     public bool autoOpen = true;

     //Should the doors be closed before locking?
     public bool autoClose = true;

     void Start () {

     }

     void Update () {

     }

     //Locks all target doors
     public void LockAll(){
          foreach(GeneralDoor door in targetDoors){
               if(autoClose){
                    door.Close();
               }
               door.Lock();
          }
     }

     //Unlocks all target doors using available keys
     public void UnlockAll(){
          foreach(GeneralDoor door in targetDoors){
                door.Unlock(keys);
                if(autoOpen){
                     door.Open();
               }
          }
     }
}
```

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

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

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

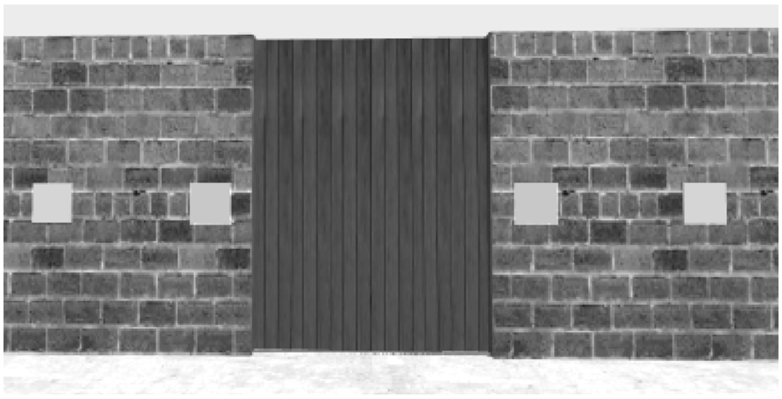


Illustration 81: The four buttons of unlock puzzle arranged around the sliding door

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

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

```
using UnityEngine;
using System.Collections;

public class ColorCycler : MonoBehaviour {

     //Color to cycle between
     public Color[] colors;

     //index of the current color
     public int currentColor = 0;

     void Start () {
          renderer.material.color = colors[currentColor];
     }

     void Update () {

     }

```

```
	//Cycles to next color in the array
	public void CycleColor(){
		if(colors.Length > 0){
			currentColor++;

			if(currentColor == colors.Length){
				currentColor = 0;
			}

			renderer.material.color = colors[currentColor];
		}
	}
}
```

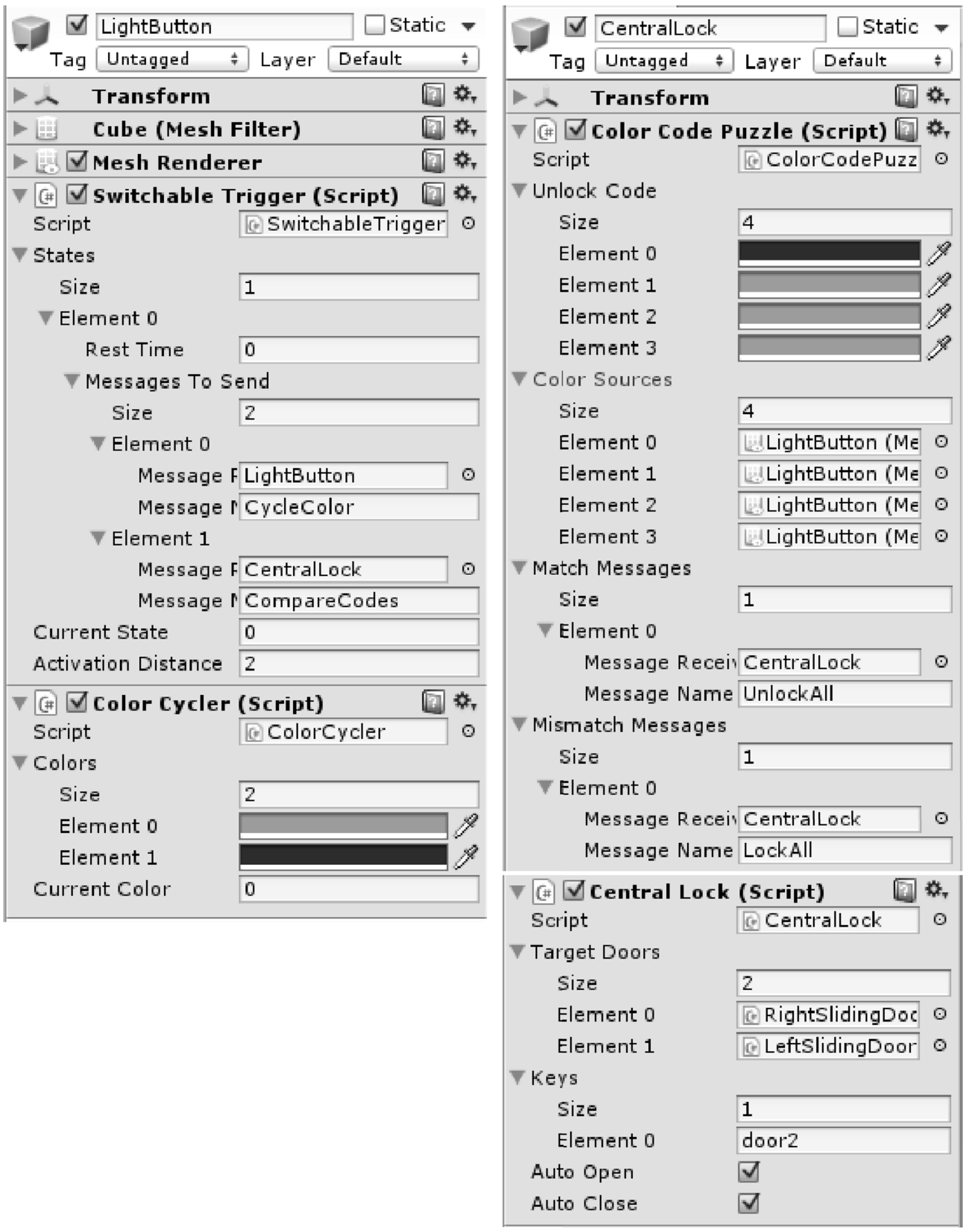
Listing 69: A script that cycles the color of the object

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

```
using UnityEngine;
using System.Collections;

public class ColorCodePuzzle : MonoBehaviour {

	//Unlock sequence
	public Color[] unlockCode;

	//Where to get input code
	public Renderer[] colorSources;

	//Message to send upon code match
	public TriggerMessage[] matchMessages;

	//Messages to send upon code mismatch
	public TriggerMessage[] mismatchMessages;

	void Start () {

	}

	void Update () {

    }

    //Compare both codes
    public void CompareCodes(){
        //Assume combinations match
        bool match = true;

        //Get combination from sources and compare it with unlock code
        for(int i = 0; i < colorSources.Length; i++){
            //One difference is enough to deny match
            if(!colorSources[i].material.color.Equals(unlockCode[i])){
                match = false;
            }
        }

        TriggerMessage[] toSend;

        //If the code matches combination
        //Then send match messages
        if(match){
            toSend = matchMessages;
        } else {
            //else send mismatch messages
            toSend = mismatchMessages;
        }

        //Send messages
        foreach(TriggerMessage msg in toSend){
            if(msg.messageReceiver != null){
                msg.messageReceiver
                          .SendMessage(
                             msg.messageName,
                             SendMessageOptions.RequireReceiver);

            }
        }
    }
}
```

Listing 70: The script of color combination unlock puzzle

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



```
LightButton   Static
Tag Untagged   Layer Default
Transform
Cube (Mesh Filter)
Mesh Renderer
Switchable Trigger (Script)
Script                  SwitchableTrigger
States
  Size                  1
  Element 0
    Rest Time           0
    Messages To Send
      Size              2
      Element 0
        Message F       LightButton
        Message N       CycleColor
      Element 1
        Message F       CentralLock
        Message N       CompareCodes
Current State           0
Activation Distance     2
Color Cycler (Script)
Script                  ColorCycler
Colors
  Size                  2
  Element 0
  Element 1
Current Color           0
```

```
CentralLock   Static
Tag Untagged   Layer Default
Transform
Color Code Puzzle (Script)
Script                  ColorCodePuzz
Unlock Code
  Size                  4
  Element 0
  Element 1
  Element 2
  Element 3
Color Sources
  Size                  4
  Element 0             LightButton (Me
  Element 1             LightButton (Me
  Element 2             LightButton (Me
  Element 3             LightButton (Me
Match Messages
  Size                  1
  Element 0
    Message Recei       CentralLock
    Message Name        UnlockAll
Mismatch Messages
  Size                  1
  Element 0
    Message Recei       CentralLock
    Message Name        LockAll
Central Lock (Script)
Script                  CentralLock
Target Doors
  Size                  2
  Element 0             RightSlidingDoc
  Element 1             LeftSlidingDoor
Keys
  Size                  1
  Element 0             door2
Auto Open               ✓
Auto Close              ✓
```

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

5.3 Health, lives, and score

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

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

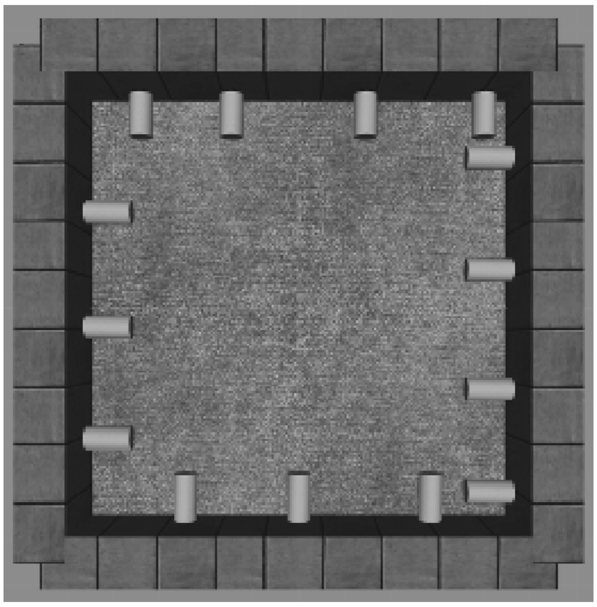


Illustration 83: Top view of the play room and the shooters surrounding it

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

```
using UnityEngine;
using System.Collections;

public class PhysicsShooter : MonoBehaviour {

	//Prefabs of projectiles to shoot
	public GameObject[] projectils;

	//min and max time between shots
	public float minTime = 1, maxTime = 6;

	void Start () {
	ShootRandomly();
	}

	void Update () {

	}

	void ShootRandomly(){
		//Shoot after random time
		float randomTime = Random.Range(minTime, maxTime);
		Invoke("Shoot", randomTime);
	}

	void Shoot(){
		//Select random projectile
		int index = Random.Range(0, projectils.Length);
		GameObject prefab = projectils[index];
		GameObject projectile = (GameObject)Instantiate(prefab);

		//Shoot the projectile
		projectile.transform.position = transform.position;
		projectile.rigidbody.AddForce
				(transform.up * 6, ForceMode.Impulse);

		//Reshoot after random time
		ShootRandomly();
	}
}
```

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

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

```
using UnityEngine;
using System.Collections;

public class PainfulProjectile : MonoBehaviour {

    //Damage caused by the projectile
    public int damage;

    void Start () {

    }

    void Update () {

    }

    void OnCollisionEnter(Collision col){
        //Tell other object about painful hit
        col.gameObject.SendMessage("OnPainfulHit",
                                   damage,
                                   SendMessageOptions.DontRequireReceiver);

        //Destroy the projectile
        Destroy(gameObject);
    }
}
```

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

Alcatel·Lucent
www.alcatel-lucent.com/careers
What if you could build your future and create the future?
One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

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

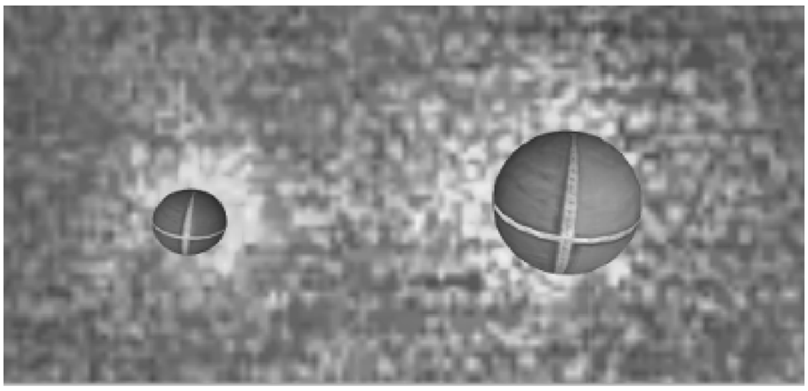


Illustration 84: Small green shining projectile (left) with 5 damage, and big red shining projectile with 10 damage

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(PhysicsCharacter))]
public class TopViewControl : MonoBehaviour {

     //Reference to the physics character
     PhysicsCharacter pc;

     void Start () {
          //Get the attached physics character
          pc = GetComponent<PhysicsCharacter>();
     }

     void Update () {
          //Use arrows to control the movement
          if(Input.GetKey(KeyCode.RightArrow)){
               pc.StrafeRight();
          } else if(Input.GetKey(KeyCode.LeftArrow)){
               pc.StrafeLeft();
          }

          if(Input.GetKey(KeyCode.UpArrow)){
               pc.WalkForward();
          } else if(Input.GetKey(KeyCode.DownArrow)){
               pc.WalkBackwards();
          }

     }
}
```

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

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

```
using UnityEngine;
using System.Collections;

public class PlayerHealth : MonoBehaviour {

     //Health amount at the beginning
     public int initialHealth = 100;

     //Max health limit
     public int maxHealth = 100;

     //Current health
     int health;

     //Internal dead flag
     bool dead = false;

     void Start () {
          //Insure appropriate initial health
          health = Mathf.Min(initialHealth, maxHealth);
          //Player cannot start dead
          if(health < 0){
               health = 1;
          }
     }

     void Update () {
          if(!dead){
               if(health <= 0){
                    //Player died
                    dead = true;

                    //Tell other scripts about player’s death
                    //and give them his final health
                    SendMessage("OnPlayerDeath",
                          health,
                          SendMessageOptions.DontRequireReceiver);
               }
          }
     }

     //Deacrease health and inform other scripts about it
     public void DecreaseHealth(int amount){
          //Do nothing if the player is already dead
          if(IsDead()) return;

          health -= amount;
          SendMessage("OnHealthDecrement",
                          health,
                          SendMessageOptions.DontRequireReceiver);
     }

     //Increase health and inform other scripts about it
     public void IncreaseHealth(int amount){
          //Do nothing if the player is already dead
```

```
        if(IsDead()) return;

        //Increase only of health is less than full
        if(health < maxHealth){
            //Do not allow the health to exceed maxHealth
            health = Mathf.Min(maxHealth, health + amount);
            SendMessage("OnHealthIncrement",
                      health,
                      SendMessageOptions.DontRequireReceiver);
        }
    }

    //Returns whether player is dead
    public bool IsDead(){
        return dead;
    }

    public int GetCurrentHealth(){
        return health;
    }
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(PlayerHealth))]
public class PainfulDamageTaker : MonoBehaviour {

     //Reference to player health
     PlayerHealth playerHealth;

     void Start () {
          playerHealth = GetComponent<PlayerHealth>();
     }

     void Update () {

     }

     //Handle painful hit by decreasing
     //player's health by the provided amount (damage)
     void OnPainfulHit(int amount){
          playerHealth.DecreaseHealth(amount);
     }
}
```

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

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

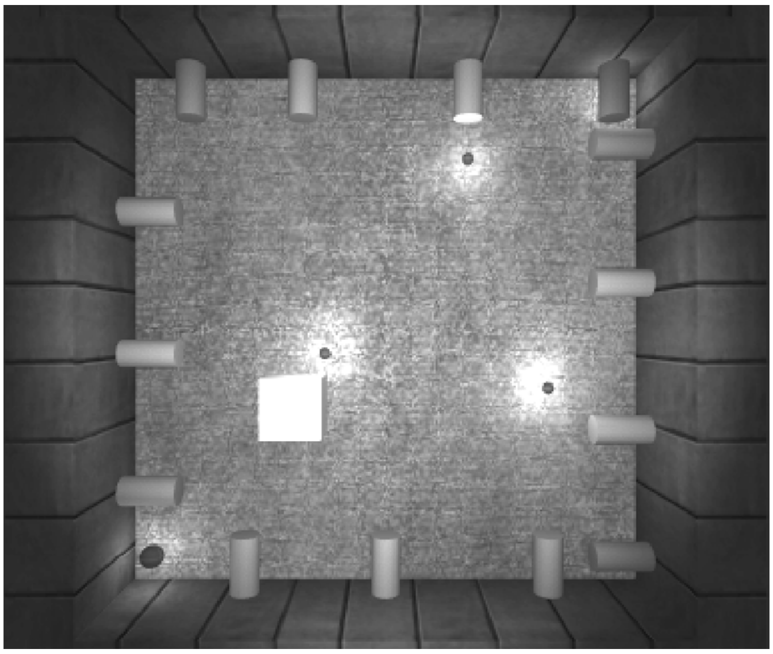


Illustration 85: A screen shot of game play with player cube and projectiles

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(PlayerHealth))]
public class HealthColorChanger : MonoBehaviour {

	//Color when health is full
	public Color fullHealth = Color.green;

	//Color when player is dead
	public Color zeroHealth = Color.red;

	//Reference to player health script
	PlayerHealth playerHealth;

	void Start () {
		playerHealth = GetComponent<PlayerHealth>();
		UpdateColor(playerHealth.GetCurrentHealth());
	}

	void Update () {

	}

	void OnHealthIncrement(int amount){
		UpdateColor(amount);
	}

	void OnHealthDecrement(int amount){
		UpdateColor(amount);
	}

	//Vary color between full and death colors depending on
	//the value of new player health
	void UpdateColor(int newHealth){
		//Convert integer health to float value
		//between 0 (death) and 1 (full health)
		float val = (float)newHealth /
				(float) playerHealth.maxHealth;

		//Apply the new color
		renderer.material.color =
			Color.Lerp(zeroHealth, fullHealth, val);
	}
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

public class LivesManager : MonoBehaviour {

     //Initial number of lives
     public int startLives = 3;

     //internal counter
     int lives;

     void Start () {
          //Enforce at least one life initially
          if(startLives > 0){
               lives = startLives;
          } else {
               lives = 1;
          }
     }

     void Update () {

     }

     public void GiveLife(){
          lives++;
          SendMessage("OnLifeGained",
                           lives, //New number of lives
                           SendMessageOptions.DontRequireReceiver);
     }

     public void TakeLife(){
          lives--;
          if(lives == 0){
               //Last live lost
               //Someone has to take care about that
               SendMessage("OnAllLivesLost",
                     SendMessageOptions.DontRequireReceiver);
          } else {
               //A life has been lost
               //Handle this elsewhere
               SendMessage("OnLifeLost",
                     lives, //Remaining lives
                     SendMessageOptions.DontRequireReceiver);
          }
     }

}
```

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

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

```
using UnityEngine;
using System.Collections;

public class PlayerDeathReporter : MonoBehaviour {

      //Reports player’s death to lives manager
      LivesManager manager;

      void Start () {
            manager = FindObjectOfType<LivesManager>();
      }

      void Update () {

      }

      //Handle player’s death event by taking a life
      void OnPlayerDeath(int deathHealth){
            if(manager != null){
                  manager.TakeLife();
            }
      }
}
```

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

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

```
using UnityEngine;
using System.Collections;

public class PlayerSpawn : MonoBehaviour {

	//Prefab of the player object
	public GameObject playerPrefab;

	//Seconds to wait between death and respawn
	public int spawnDelay = 3;

	//Reference to current player
	GameObject currentInstance;

	void Start () {
		SpawnPlayer();
	}

	void Update () {

	}

	//A life has been lost
	void OnLifeLost(int remainingLives){
		//Regenerate Player, delay spawn
		Destroy(currentInstance);
		Invoke("SpawnPlayer", spawnDelay);
	}

	//Game Over
	void OnAllLivesLost(){
		Destroy(currentInstance);
	}

	void SpawnPlayer(){
		currentInstance =
			(GameObject) Instantiate(playerPrefab);
	}
}
```

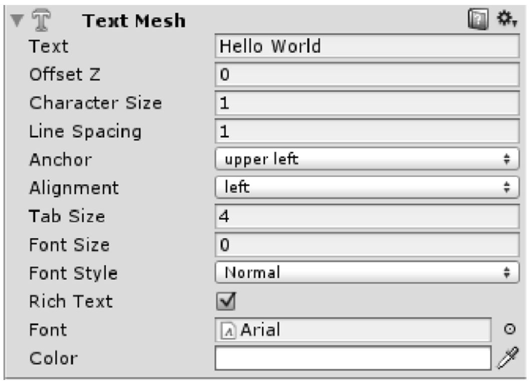
Listing 79: A script to handle destruction and instantiation of player's character based on lives

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

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

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

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



Text Mesh

Text	Hello World
Offset Z	0
Character Size	1
Line Spacing	1
Anchor	upper left
Alignment	left
Tab Size	4
Font Size	0
Font Style	Normal
Rich Text	✓
Font	Arial
Color	

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(LivesManager))]
public class LivesCounterHandler : MonoBehaviour {

      //Reference to 3D Text object
      public TextMesh display;

      //Reference to lives manager
      LivesManager lManager;

      void Start () {
            lManager = GetComponent<LivesManager>();
            //Start by displaying startLives
            display.text = lManager.startLives.ToString();
      }

      void Update () {

      }

      void OnLifeGained(int newLives){
            //Number of lives changed, update
            display.text = newLives.ToString();
      }

      void OnLifeLost(int remainingLives){
            //Number of lives changed, update
            display.text = remainingLives.ToString();
      }

      void OnAllLivesLost(){
            //All lives have been lost,
            //display ‘Game Over’ text
            display.text = "Game Over";
      }
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(LivesManager))]
public class ScoreCounter : MonoBehaviour {

	//Where to show score? (optional)
	public TextMesh display;

	LivesManager lManager;

	//Internal score counter
	int score = 0;

	void Start () {
		lManager = GetComponent<LivesManager>();
		//Increase 1 point every second
		InvokeRepeating("IncrementScore", 1, 1);
	}

	void Update () {

	}

	void IncrementScore(){
		score++;
		if(display != null){
			display.text = score.ToString();
		}
	}

	void OnAllLivesLost(){
		//Game over, stop counting
		CancelInvoke("IncrementScore");
	}
}
```

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

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

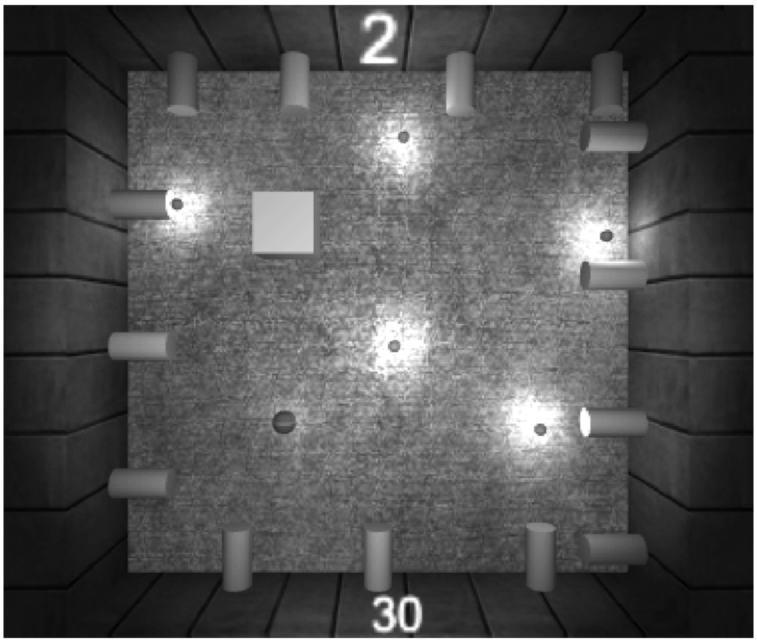


Illustration 87: A screen shot of the final game. The upper digit is the number of lives left, and the lower number is the score.

5.4 Weapons, ammunition, and reload

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

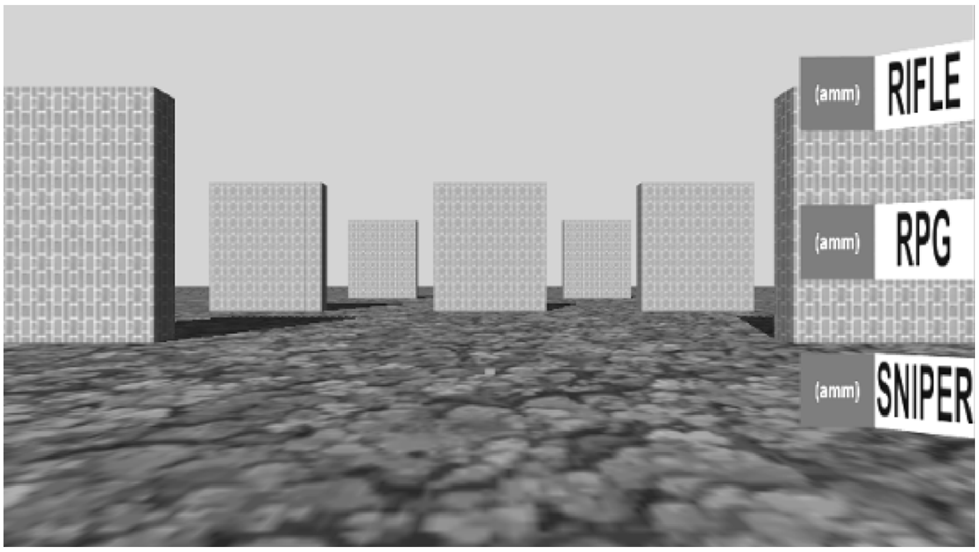


Illustration 88: Basic scene constructed to test different weapons

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

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

```
using UnityEngine;
using System.Collections;

public class GeneralWeapon : MonoBehaviour {

     //How many magazines remaining
     public int magazineCount = 3;

     //Ammunation count per full magazine
     public int magazineCapacity = 30;

     //Ammunation remaining in current magazine
     public int magazineSize = 30;

     //How many seconds needed to reload?
     public float reloadTime = 3;

     //How many times can the weapon shoot in one second?
     public float fireRate = 3;

     //If true, there is no need to release
     //the trigger between firings
     public bool automatic = false;

     //Ammunation lost per firing
     //Must not exceed magazine capacity
     public int ammoPerFiring = 1;

     //Is this weapon currently hold by the player?
     public bool inHand = false;

     //Internal timer for fire rate
     float lastFiringTime = 0;

     //Internal state for reload progress
     float reloadProgress = 0;

     //Internal storage of trigger state
     bool triggerPulled = false;

     void Start () {

     }

     void Update () {
          //If the weapon is currently in hand and it reloads,
          //then advance reload progress with time
          if(inHand && reloadProgress > 0){
               reloadProgress += Time.deltaTime;
               if(reloadProgress >= reloadTime){
                    //Reloading completed
                    //Discard the current magazine
                    //and install a new one
                    magazineSize = magazineCapacity;
```

```
                magazineCount--;
                reloadProgress = 0;
                SendMessage("OnReloadComplete",
                    SendMessageOptions.DontRequireReceiver);
            }
        }
    }

    public void Fire(){
        //Make sure the weapon is in hand and not
        //currently reloading, enforce time gap between firings,
        //and make sure that the weapon is either automatic
        //or the trigger has been released after last firing
        if(inHand && reloadProgress == 0 &&
            (automatic || !triggerPulled) &&
            Time.time - lastFiringTime > 1 / fireRate){
            //Do we have enough ammo in the current magazine?
            if(magazineSize >= ammoPerFiring){
                //Yes, fire by reducing ammo,
                //setting fire timer,
                //and sending OnWeaponFire message
                magazineSize -= ammoPerFiring;
                lastFiringTime = Time.time;
                triggerPulled = true;
                SendMessage("OnWeaponFire",
                    SendMessageOptions.DontRequireReceiver);

                //if the remaining ammo is not enough, then reload
                if(magazineSize < ammoPerFiring){
                    Reload();
                }

            } else {
                //No, reload
                Reload();
            }
        }
    }

    public void ReleaseTrigger(){
        triggerPulled = false;
    }

    public void Reload(){
        //Make sure there is no reloading in progress
        if(reloadProgress == 0){
            //Make sure there is enough magazines
            //and the current magazine isn't full
            if(magazineCount > 0 &&
                magazineSize < magazineCapacity){
                //Initialize reloading progress
                reloadProgress = Time.deltaTime;
                SendMessage("OnReloadStart",
                    SendMessageOptions.DontRequireReceiver);
            }
        }
    }

    //returns current reload progress percentage
    public float GetReloadProgress(){
        return reloadProgress / reloadTime;
    }
}
```

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

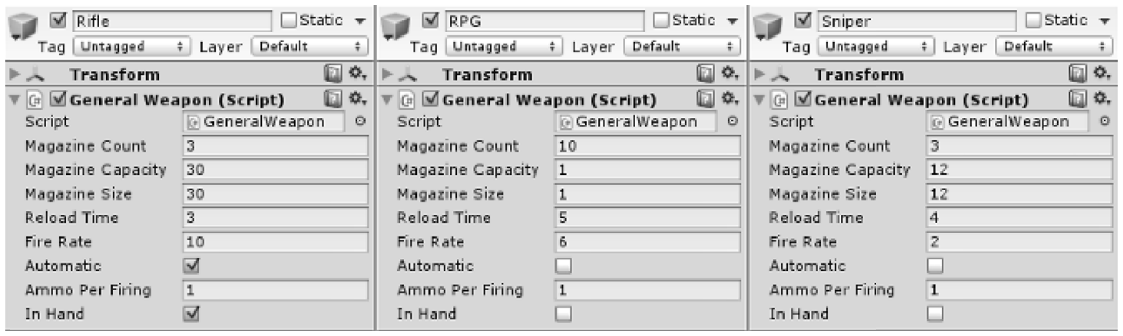
The first three variables are used to manage ammunition. The difference between *magazineCapacity* and *magazineSize* is that the first one is constant and tells us the number of maximum bullets in a single magazine. However, *magazineSize* is variable and is reduced by *ammoPerFiring* whenever the weapon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

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

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

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

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



	Rifle	RPG	Sniper
Tag	Untagged	Untagged	Untagged
Layer	Default	Default	Default
Script	GeneralWeapon	GeneralWeapon	GeneralWeapon
Magazine Count	3	10	3
Magazine Capacity	30	1	12
Magazine Size	30	1	12
Reload Time	3	5	4
Fire Rate	10	6	2
Automatic	✓	☐	☐
Ammo Per Firing	1	1	1
In Hand	✓	☐	☐

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

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

```
using UnityEngine;
using System.Collections;

public class MousePointerFollower : MonoBehaviour {

	//A marker that can be placed on the current target
	public Transform targetMarker;

	void Start () {
		//Hide the marker behind the player
		targetMarker.position =
			transform.position - Vector3.forward;
	}

	void Update () {
		//Try to find the point where mouse points
		Ray camToMouse =
			Camera.main.ScreenPointToRay (Input.mousePosition);

		RaycastHit hit;
		if(Physics.Raycast(camToMouse, out hit, 500)){
			//An object has been found, look at it
			transform.LookAt(hit.point);
			//Move the marker to the hit point
			targetMarker.position = hit.point;
			//Move the marker a little bit towards us
			targetMarker.LookAt(transform.position);
			targetMarker.Translate(0, 0, 0.1f);
		} else {
			//No object under the mosue pointer,
			//look far away
			transform.LookAt(camToMouse.GetPoint(500));
			//Hide the marker
			targetMarker.position =
				transform.position - Vector3.forward;
		}
	}

}
```

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

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

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

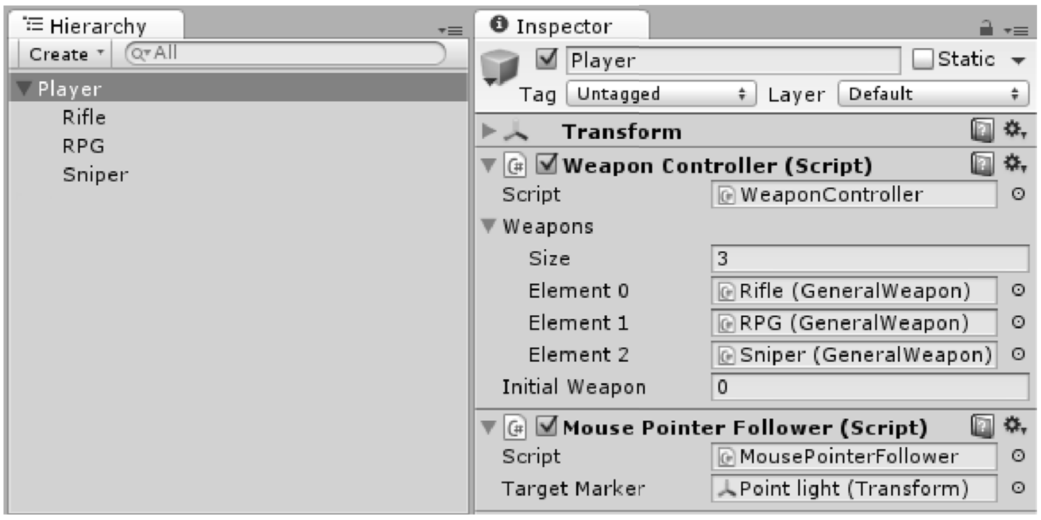
```
using UnityEngine;
using System.Collections;

public class WeaponController : MonoBehaviour {

     //Array of available weapons
     public GeneralWeapon[] weapons;

     //Index of initially hold weapon
     public int initialWeapon = -1;

     //Index of currently hold weapon
     int currentWeapon;

     void Start () {
          //Set current weapon to the initial value
          //selected from the inspector
          currentWeapon = initialWeapon;
          //Update inHand variable of all weapons
          RefreshInHandValues();
     }

     void Update () {
          UpdateSwitching();
          UpdateShooting();
     }

     void UpdateSwitching(){
          //Convert the value of "1" key in alphabit
          //section of the keyboard to int
          int keyCode = (int)KeyCode.Alpha1;
          for(int i = 0; i < weapons.Length; i++){
               //Each weapon takes the number keyCode + weapon index
               if(Input.GetKeyDown((KeyCode) keyCode + i)){
                    currentWeapon = i;
                    RefreshInHandValues();
               }
          }
     }

     void UpdateShooting(){
          //Mouse button down: Fire
          if(Input.GetMouseButton(0)){
               weapons[currentWeapon].Fire();
          }
          //Mouse button up: ReleaseTrigger
          if(Input.GetMouseButtonUp(0)){
               weapons[currentWeapon].ReleaseTrigger();
          }
          //Right click: Reload
          if(Input.GetMouseButtonDown(1)){
               weapons[currentWeapon].Reload();
          }
     }
```

```

    //Change weapon
    public void SetCurrentWeapon(int newIndex){
         weapons[currentWeapon].ReleaseTrigger();
         currentWeapon = newIndex;
         RefreshInHandValues();
    }

    void RefreshInHandValues(){
         foreach(GeneralWeapon gw in weapons){
              //inHand must be true only for the
              //currently hold weapon
              gw.inHand = weapons[currentWeapon] == gw;
         }
    }
}
```

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

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

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



Player
Static
Tag: Untagged
Layer: Default

Transform

Weapon Controller (Script)

Script	WeaponController
Weapons	
Size	3
Element 0	Rifle (GeneralWeapon)
Element 1	RPG (GeneralWeapon)
Element 2	Sniper (GeneralWeapon)
Initial Weapon	0

Mouse Pointer Follower (Script)

Script	MousePointerFollower
Target Marker	Point light (Transform)

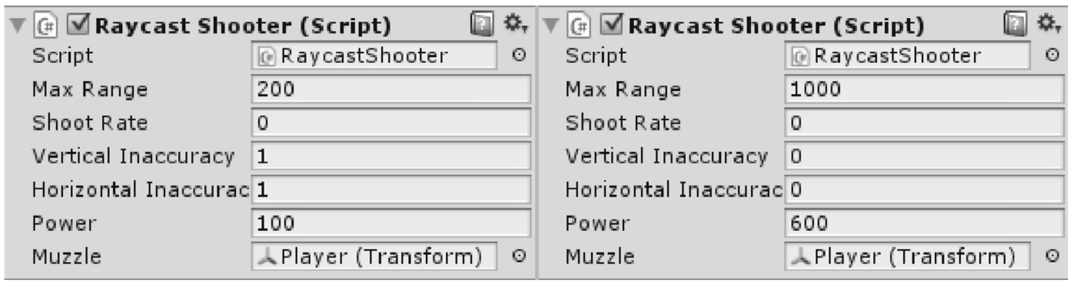
Illustration 90: Player game object configured completely

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(GeneralWeapon))]
[RequireComponent(typeof(RaycastShooter))]
public class WeaponToRaycast : MonoBehaviour {

	RaycastShooter shooter;

	void Start () {
		shooter = GetComponent<RaycastShooter>();
	}

	void Update () {

	}

	void OnWeaponFire(){
		shooter.Shoot();
	}
}
```

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

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



Raycast Shooter (Script)	
Script	RaycastShooter
Max Range	200
Shoot Rate	0
Vertical Inaccuracy	1
Horizontal Inaccurac	1
Power	100
Muzzle	Player (Transform)

Raycast Shooter (Script)	
Script	RaycastShooter
Max Range	1000
Shoot Rate	0
Vertical Inaccuracy	0
Horizontal Inaccurac	0
Power	600
Muzzle	Player (Transform)

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

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

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Destructible))]
public class DestructOnHitDamage : MonoBehaviour {

	//Minimum damage to destruct
	public float destructionDamage = 250;

	Destructible dest;

	void Start () {
		dest = GetComponent<Destructible>();
	}

	void Update () {

	}

	void OnRaycastHit(RaycastHit hit){
		//Hit damage is stored in distance
		//if damage more than destructionDamage,
		//then destruct
		if(hit.distance > destructionDamage){
			dest.Destruct();
		}
	}
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

public class RPG : MonoBehaviour {

      //Prefab of rocket to launch
      public GameObject rocketPrefab;

      void Start () {

      }

      void Update () {

      }

      //Simply receive the message and instantiate a rocket
      //The rocket has initially the same position and rotation
      //of the launcher
      void OnWeaponFire(){
           GameObject rocket = (GameObject)Instantiate(rocketPrefab);
           rocket.transform.position = transform.position;
           rocket.transform.rotation = transform.rotation;
      }
}
```

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Rigidbody))]
public class RPGRocket : MonoBehaviour {

	//Force to apply upon launch
	public float launchForce = 100;

	//Number of seconds to keep rocket alive
	//in case it hits nothing
	public float lifeTime = 7;

	//To calculate life time
	float launchTime;

	//Internal state tracking
	//Necessary to prevent collision detection
	//between the rocket and its pieces
	bool destroyed = false;

	void Start () {
		rigidbody.AddForce(transform.forward * launchForce,
		                              ForceMode.VelocityChange);

		launchTime = Time.time;
	}

	void Update () {
		if(!destroyed && Time.time - launchTime > lifeTime){
			Destroy(gameObject);
		}
	}

	void OnCollisionEnter(Collision col){
		if(!destroyed){
			destroyed = true;
			//Inform other scripts on the rocket about the hit
			//and provide a reference to the colliding object
			SendMessage("OnRocketHit",
			      col.collider,
			      SendMessageOptions.DontRequireReceiver);

			//Destroy rocket object
			Destroy(gameObject);
		}
	}
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(RPGRocket))]
[RequireComponent(typeof(Breakable))]
public class BreakOnRocketHit : MonoBehaviour {

     void Start () {

     }

     void Update () {

     }

     void OnRocketHit(Collider hitObject){
          GetComponent<Breakable>().Break();
     }
}
```

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

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

```
using UnityEngine;
using System.Collections;

public class DestructOnRocketHit : MonoBehaviour {

      //Radius of rocket explosion
      public float explosionRadius = 3;

      //Force of the explosion
      public float explosionForce = 50000;

      void Start () {

      }

      void Update () {

      }

      void OnRocketHit(Collider target){
            //Destruct all destructible blocks in range
            //and add explosion force to them
            Destructible[] all = FindObjectsOfType<Destructible>();

            Vector3 explosionPos = transform.position;

            foreach(Destructible dest in all){
                  if(Vector3.Distance
                        (explosionPos, dest.transform.position)
                                          < explosionRadius){

                        dest.Destruct();

                        dest.rigidbody.
                              AddExplosionForce(explosionForce,
                                                explosionPos,
                                                explosionRadius);
                  }
            }
      }
}
```

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

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

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

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

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(GeneralWeapon))]
public class AmmoDisplay : MonoBehaviour {

      //Where to display data
      public TextMesh display;

      //The weapon to display data for
      GeneralWeapon weapon;

      void Start () {
            weapon = GetComponent<GeneralWeapon>();
      }

      void LateUpdate () {
            //Do not show if weapon is not in hand
            if(!weapon.inHand){
                  display.text = "XXX";
                  return;
            }

            float reloadProgress = weapon.GetReloadProgress();
            //Show number of bullets and magazines remaining
            if(reloadProgress == 0){
                  display.text = weapon.magazineSize + "/" +
                                             weapon.magazineCapacity + " (x" +
                                             weapon.magazineCount + ")";
            } else {
                  //If reloading, show reload progress
                  int progress = (int)(reloadProgress * 100);
                  display.text = "RLD " + progress + "%";
            }
      }
}
```

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

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

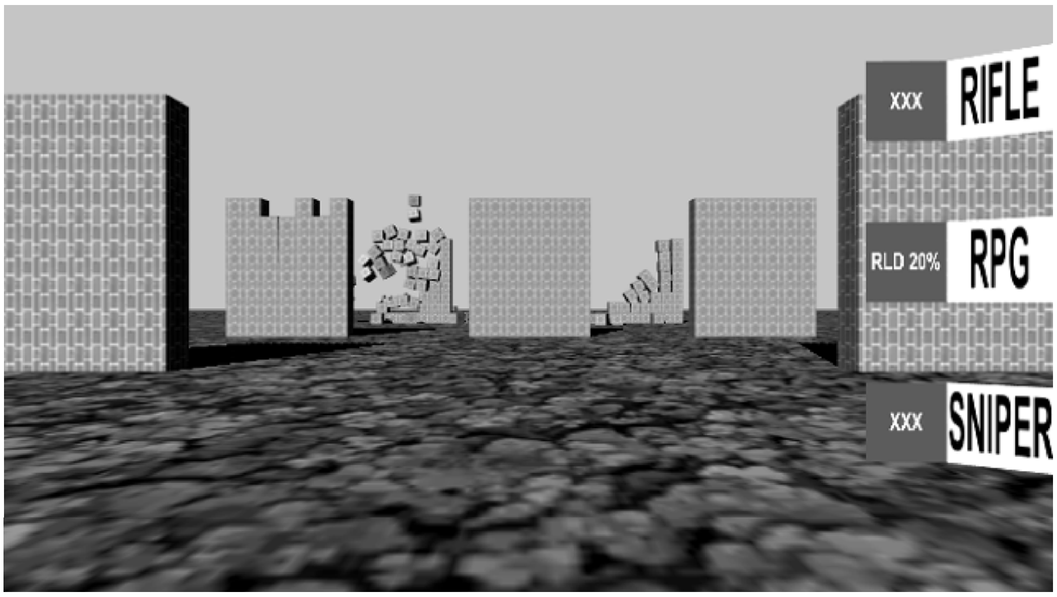


Illustration 92: Weapon switching and reloading demo

Exercises

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