Game Programming Using Unity 3D

Lesson 17: Weapons Part 2

Ferdinand Joseph Fernandez

Chief Technological Officer, Dreamlords Digital Inc.

Admin and Co-founder, Unity Philippines Users Group

December 2011

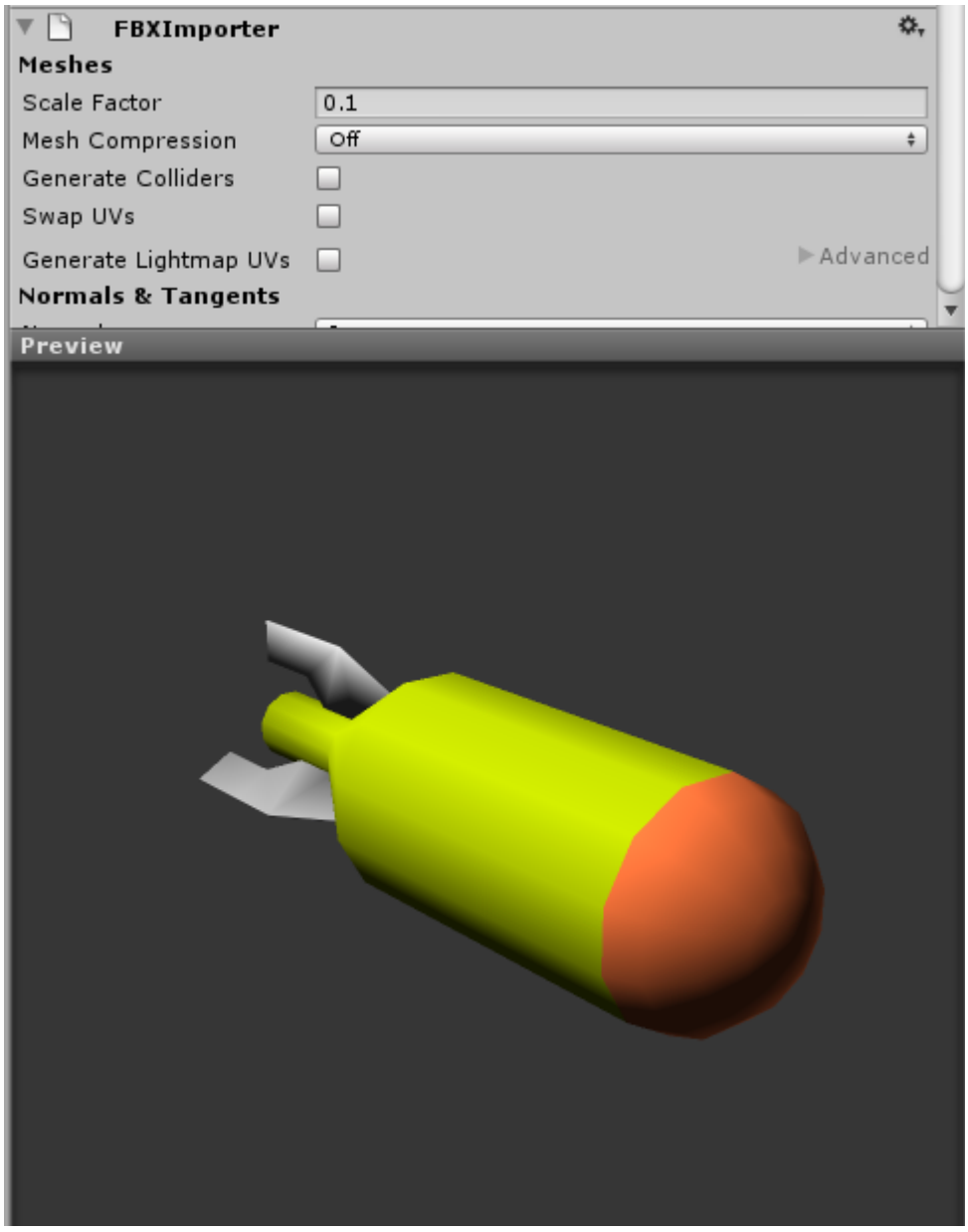
Overview

In this lesson we'll add a new weapon: a missile launcher. This weapon will have a projectile that travels in the world, and damaging whatever it collides with.

Creating The Missile

Your Lesson Assets folder will have a MinimalMissile folder. Copy that whole folder into the Models folder of your Project View.

It will have a “MinimalMissile1” fbx file. Drag it into your scene. You may want to change the Scale Factor. Try 0.1.



We'll need a new script that will handle a live missile. Create a new C# script and name it “Missile”.

Add this code:

```
01 using UnityEngine;
```

```
02 using System.Collections;
03
04 public class Missile : MonoBehaviour
05 {
06     [SerializeField]
07     float _speed = 50.0f;
08
09     void Start()
10     {
11         rigidbody.AddRelativeForce(new Vector3(0, 0, _speed), ForceMode.VelocityChange);
12     }
13 }
```

This will simply move the missile at its forward direction with the specified force. The force is 10.0 newtons by default.

Attach this script to the MinimalMissile1 game object. Add a rigidbody to it too so the code will work.

Test the game. The missile should move on its own.

Launching Missiles From Your Weapon

We'll make an equivalent of the RifleWeapon script for missiles. We'll use the RifleWeapon script as a template.

Duplicate the RifleWeapon script. You can do this by selecting the RifleWeapon script from your Project View. Then press Ctrl + D.

Rename it to “MissileWeapon”. Open it and rename the class name to “MissileWeapon” as well.

Change the code to this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MissileWeapon : MonoBehaviour
05 {
06     [SerializeField]
07     GameObject _missilePrefab;
08
09     [SerializeField]
10     Transform _muzzle;
11
12     void Start()
13     {
14         Screen.lockCursor = true;
15     }
16
17     void Update()
18     {
19         if (Input.GetKey(KeyCode.Escape))
20         {
21             Screen.lockCursor = false;
22         }
23
24         if (Input.GetButtonDown("Fire1"))
```

```
25          {
26              Screen.lockCursor = true;
27
28              Ray mouseRay = Camera.main.ViewportPointToRay(new  Vector3(0.5f, 0.5f, 0));
29              RaycastHit hitInfo;
30
31              if  (Physics.Raycast(mouseRay, out hitInfo))
32              {
33                  Vector3 direction = hitInfo.point - _muzzle.position;
34                  direction.Normalize();
35
36                  GameObject m = Instantiate(_missilePrefab, _muzzle.position, _muzzle.rotation) as
                        GameObject;
37                  m.transform.forward = direction;
38              }
39          }
40      }
41 }
```

Instead of immediately damaging any Health instance found, we instantiate a new missile from the "_muzzle". Then we make it face the proper direction.

Assigning a new value to a Transform's forward property effectively changes its rotation.

Now we'll test this.

Turn the MinimalMissile1 into a prefab first. Drag it into the Prefabs folder in your Project View to turn it into a prefab.

Attach this MissileWeapon script to your Player game object. Disable the RifleWeapon for the moment by unchecking the checkbox beside its name.

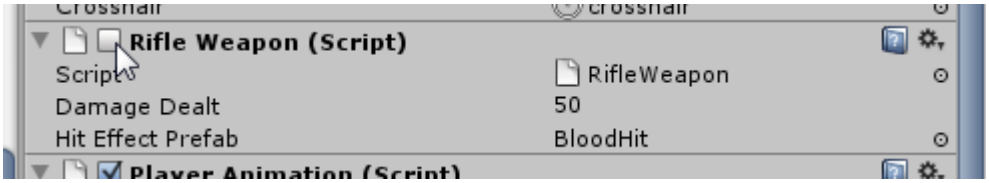


Illustration 1: The RifleWeapon script disabled.

This will prevent the code in RifleWeapon from being executed. We do this so we can test the MissileWeapon script properly.

Go to your Player's MissileWeapon script component. Now assign the MinimalMissile1 prefab to the "Missile Prefab" property. As for the Muzzle, we'll need to make that first.

Create a new empty game object. Name it "Muzzle". We'll need to parent this Muzzle game object to the Player model's gun.

Click on your player 3d model. Click on its gun. The gun should highlight its wireframe to blue. If you look at the Hierarchy, the game object "main_weapon001" will be selected. This is where you need to put the Muzzle game object.

Illustration 2: Select the gun's 3d model to find where to put the Muzzle.

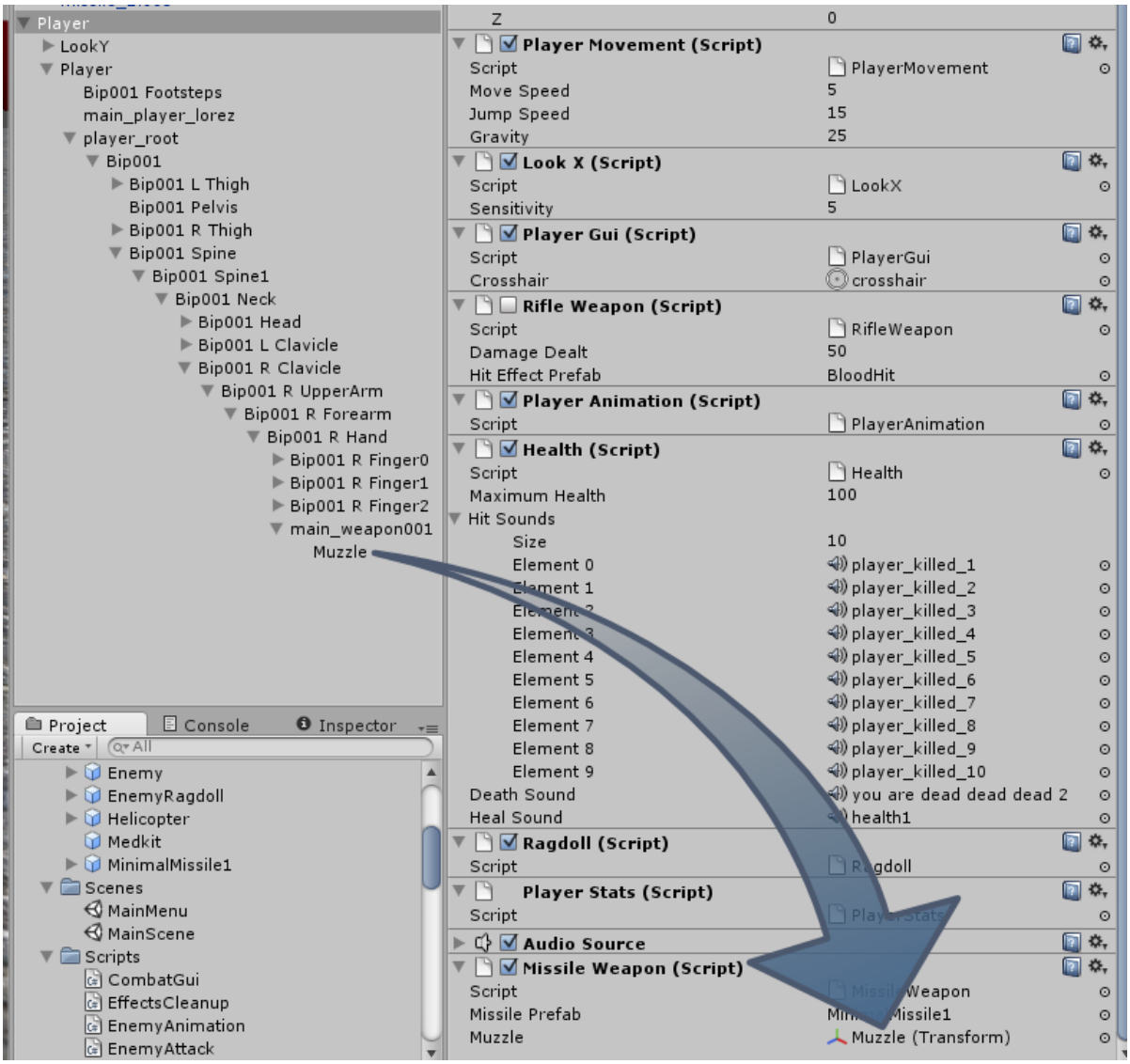
Drag the Muzzle game object to the “main_weapon001” game object. Position it somewhere near the “mouth” of the gun. This is otherwise known as a muzzle.

Make sure the blue arrow is facing in the gun's forward direction. That will be the direction where the missiles will launch. Try setting the Muzzle game object's rotation to (270, 0, 0).

Now that you have the Muzzle in the right place, assign it to your MissileWeapon. Go back to the Player game object where the MissileWeapon is. Drag the Muzzle game object (its entry in the Hierarchy View) into your Muzzle property in the MissileWeapon.

This is needed so the MissileWeapon script knows where to put the missile at the start, when it's launched.

Now test the game. Target a zombie and press the left mouse button. It should now launch a missile instead.



Shooting A Missile Even Without A Target In The Crosshair

Right now shooting in empty air won't launch a missile. This is because our code doesn't take that into account.

It needs something to hit before it can compute the proper direction for the missile to face to.

We can make some assumptions when shooting empty-air. Change the code in MissileWeapon to this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MissileWeapon : MonoBehaviour
05 {
```

```
06      [SerializeField]
07      GameObject _missilePrefab;
08
09      [SerializeField]
10      Transform _muzzle;
11
12      void  Start()
13      {
14           Screen.lockCursor = true;
15      }
16
17      void  Update()
18      {
19           if  (Input.GetKey(KeyCode.Escape))
20           {
21                Screen.lockCursor = false;
22           }
23
24           if  (Input.GetButtonDown("Fire1"))
25           {
26                Screen.lockCursor = true;
27
28                Ray mouseRay = Camera.main.ViewportPointToRay(new  Vector3(0.5f, 0.5f, 0));
29                RaycastHit hitInfo;
30
31                Vector3 direction;
32
33                if  (Physics.Raycast(mouseRay, out hitInfo))
34                {
35                     direction = hitInfo.point - _muzzle.position;
36                     direction.Normalize();
37                }
38                else
39                {
40                     direction = Camera.main.ViewportToWorldPoint(new  Vector3(0.5f, 0.5f, 50)) -
                           _muzzle.position;
41                     direction.Normalize();
42                }
43
44                GameObject m = Instantiate(_missilePrefab, _muzzle.position, _muzzle.rotation) as
                       GameObject;
45                m.transform.forward = direction;
46           }
47      }
48 }
```

We'll just make it that the missile will target the point 50 meters away from the camera, as shown in line 40.

Making The Missile Deal Damage

Right now the missile just bumps objects it collides with. Let's make it explode!

Add this code to the Missile script:

```
01 using  UnityEngine;
02 using  System.Collections;
03
04 public class  Missile : MonoBehaviour
05 {
06       [SerializeField]
07       float  _speed = 10.0f;
08
09       void  Start()
10       {
11            rigidbody.AddRelativeForce(new  Vector3(0, 0, _speed), ForceMode.VelocityChange);
12       }
13
14       void  OnCollisionEnter(Collision c)
15       {
16            Destroy(gameObject);
17       }
18 }
```

This will delete the missile when it hits something.

Test the game now. The missiles should now disappear when they hit objects.

Now all we need is to check if what the missile hit has a Health instance, then call its Damage function if found. Add this code:

```
01 using  UnityEngine;
02 using  System.Collections;
03
04 public class  Missile : MonoBehaviour
```

```
{
    [SerializeField]
    float _speed = 50.0f;

    [SerializeField]
    int _damageDealt = 100;

    void Start()
    {
        rigidbody.AddRelativeForce(new Vector3(0, 0, _speed), ForceMode.VelocityChange);
    }

    void OnCollisionEnter(Collision c)
    {
        Health h = c.transform.root.GetComponent<Health>();
        if (h != null)
        {
            h.Damage(_damageDealt);
        }

        Destroy(gameObject);
    }
}
```

That will deal damage to whatever the missile hits.

Preventing Damaging The Player

You may find that sometimes you die from the missile that gets launched from your own weapon. Sometimes when it just launched out of the weapon! This is because the missile collider is hitting the player's collider as the missile just appeared in the world.

We can prevent the damaging by disallowing the missile from damaging the player.

Add this code:

```
using UnityEngine;
using System.Collections;

public class Missile : MonoBehaviour
{
    [SerializeField]
    float _speed = 50.0f;

    [SerializeField]
    int _damageDealt = 100;

    void Start()
    {
        rigidbody.AddRelativeForce(new Vector3(0, 0, _speed), ForceMode.VelocityChange);
    }

    void OnCollisionEnter(Collision c)
    {
        if (c.transform.root.tag == "Player")
```

```
20          {
21              return;
22          }
23
24          Health h = c.transform.root.GetComponent<Health>();
25          if (h != null)
26          {
27              h.Damage(_damageDealt);
28          }
29
30          Destroy(gameObject);
31      }
32 }
```

We simply check if what we hit is the player. If so, we abort the whole code.

But we don't want the player to be completely immune to a missile. So here's what we do, we just make that check only for a short period.

Add this code:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Missile : MonoBehaviour
05 {
06      [SerializeField]
07      float _speed = 50.0f;
08
09      [SerializeField]
10      int _damageDealt = 100;
11
12      float _playerImmunityDuration = 0.5f;
13      float _timePlayerCanBeDamaged = 0.0f;
14
15      void Start()
16      {
17          rigidbody.AddRelativeForce(new Vector3(0, 0, _speed), ForceMode.VelocityChange);
18          _timePlayerCanBeDamaged = Time.time + _playerImmunityDuration;
19      }
20
21      void OnCollisionEnter(Collision c)
22      {
23          if (c.transform.root.tag == "Player" && Time.time <= _timePlayerCanBeDamaged)
24          {
25              return;
26          }
27
28          Health h = c.transform.root.GetComponent<Health>();
29          if (h != null)
30          {
31              h.Damage(_damageDealt);
32          }
33
34          Destroy(gameObject);
35      }
36 }
```

We make use of Time.time to find out whether 0.5 seconds have passed. If so, the player's immunity is disregarded.

That 0.5 seconds is enough time for the missile to leave the gun's muzzle and any complications of the missile colliding with the player upon launch.

Fixing Redundant Code

Right now, we have a little bit of duplicate code in RifleWeapon and MissileWeapon. What we can do here is create a parent class for both of them that does the basic functionality present in both weapons.

Create a new C# script. Name it Weapon. This Weapon script will have the code that is general to both RifleWeapon and MissileWeapon, then those two classes will just be sub-classed from this Weapon class.

The idea is we move code that is common to both weapons in the Weapon class. Then the two weapons will only need to have the code that is unique to both of them.

The ability to shoot is the common thing among the two weapons. But what they shoot out is different for both. So that is how we will code the Weapon class. The Weapon class will have code that allows it to shoot. What it does shoot out, is something we won't specify there.

In other words, the weapon class is incomplete, and is meant to be improved upon by other classes. That's where the RifleWeapon and MissileWeapon comes in. Those two weapons will “inherit” the code in class Weapon (namely the ability to shoot). All that's left to do is to code exactly what it shoots out.

Add this code to the Weapon script. Most of this code is similar to MissileWeapon with a few changes:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Weapon : MonoBehaviour
05 {
06     [SerializeField]
07     protected Transform _muzzle;
08
09     void Start()
10     {
11         Screen.lockCursor = true;
12     }
13
14     void Update()
15     {
16         if (Input.GetKey(KeyCode.Escape))
17         {
18             Screen.lockCursor = false;
19         }
20
21         if (Input.GetButtonDown("Fire1"))
22         {
23             Screen.lockCursor = true;
24
25             Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
```

```
26                  RaycastHit hitInfo;
27
28                  Vector3 direction;
29
30                  if (Physics.Raycast(mouseRay, out hitInfo))
31                  {
32                        direction = hitInfo.point - _muzzle.position;
33                        direction.Normalize();
34                  }
35                  else
36                  {
37                        direction = Camera.main.ViewportToWorldPoint(new Vector3(0.5f, 0.5f, 50)) -
                              _muzzle.position;
38                        direction.Normalize();
39                  }
40
41                  Shoot(hitInfo, direction);
42            }
43      }
44
45      virtual protected void Shoot(RaycastHit hitInfo, Vector3 direction)
46      {
47      }
48 }
```

We now have a virtual function Shoot, which will have specific code when used as a MissileWeapon or RifleWeapon.

When a function is virtual, it means that function can then be “overridden” later on.

Having our _muzzle variable set to protected means that variable can be accessed by the sub-classes (the RifleWeapon and MissileWeapon classes). If we didn't do that, Unity will generate an error.

Change MissileWeapon to this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MissileWeapon : Weapon
05 {
06      [SerializeField]
07      GameObject _missilePrefab;
08
09      override protected void Shoot(RaycastHit hitInfo, Vector3 direction)
10      {
11            GameObject m = Instantiate(_missilePrefab, _muzzle.position, _muzzle.rotation) as
                  GameObject;
12            m.transform.forward = direction;
13      }
14 }
```

MissileWeapon now gets sub-classed from Weapon. Class Weapon, in turn, gets sub-classed from MonoBehaviour. So MissileWeapon is still a MonoBehaviour type of class.

Remember, when a script is sub-classed, its as if it has the code of the parent class. So MissileWeapon will have the Start and Update functions of class Weapon.

And if you look at the Weapon class' code again, it calls the Shoot function. The Shoot function here

in MissileWeapon has been overridden with new code. Its like you are changing the behavior of the Shoot function. So this code will get called when the user presses the left mouse button, instead of the empty Shoot function declared in class Weapon.

We'll do the same thing for the RifleWeapon. Change RifleWeapon to this:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class RifleWeapon : Weapon
05 {
06     [SerializeField]
07     int _damageDealt = 10;
08
09     [SerializeField]
10     GameObject _hitEffectPrefab;
11
12     override protected void Shoot(RaycastHit hitInfo, Vector3 direction)
13     {
14         if (hitInfo.transform == null)
15         {
16             return;
17         }
18
19         Health enemyHealth = hitInfo.transform.GetComponent<Health>();
20         if (enemyHealth != null)
21         {
22             enemyHealth.Damage(_damageDealt);
23
24             Vector3 hitEffectPosition = hitInfo.point;
25             Quaternion hitEffectRotation = Quaternion.FromToRotation(Vector3.forward,
                   hitInfo.normal);
26             Instantiate(_hitEffectPrefab, hitEffectPosition, hitEffectRotation);
27         }
28     }
29 }
```

Remember, since RifleWeapon is now sub-classed from Weapon, it now has the code of the Weapon class. The Weapon class has a _muzzle variable. So your RifleWeapon script component needs to have its Muzzle property assigned to. Use the same Muzzle game object that we used for the MissileWeapon for this one.

Class RifleWeapon also overridded the Shoot function, but it uses a different piece of code. This is the main point of having a parent class.

The parent class Weapon will hold parts of code that are common to both MissileWeapon and RifleWeapon. All you did in the MissileWeapon and RifleWeapon files, is coded their correspondingly unique parts: one shoots out missiles, while the other shoots bullets.

Now any functionality that you want to be present in both weapons, you just add it in the Weapon class.

Fixing Shooting Behind The Player

You may find that you can shoot behind the player. This is because of an error in our code in the two

Weapon scripts. It makes a raycast starting from the camera, not from the gun's muzzle. We just have to check if the found target is behind the player. Add this code to the Weapon script:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Weapon : MonoBehaviour
05 {
06     [SerializeField]
07     protected Transform _muzzle;
08
09     void Start()
10     {
11         Screen.lockCursor = true;
12     }
13
14     void Update()
15     {
16         if (Input.GetKey(KeyCode.Escape))
17         {
18             Screen.lockCursor = false;
19         }
20
21         if (Input.GetButtonDown("Fire1"))
22         {
23             Screen.lockCursor = true;
24
25             Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
26             RaycastHit hitInfo;
27
28             Vector3 direction;
29             bool foundTarget = false;
30
31             if (Physics.Raycast(mouseRay, out hitInfo) &&
                   (Camera.main.WorldToScreenPoint(hitInfo.point).z >=
                   Camera.main.WorldToScreenPoint(_muzzle.position).z))
32             {
33                 foundTarget = true;
34                 direction = hitInfo.point - _muzzle.position;
35                 direction.Normalize();
36             }
37             else
38             {
39                 direction = Camera.main.ViewportToWorldPoint(new Vector3(0.5f, 0.5f, 50)) -
                       _muzzle.position;
40                 direction.Normalize();
41             }
42
43             RaycastHit gunHitInfo;
44             if (Physics.Raycast(_muzzle.position, direction, out gunHitInfo))
45             {
46                 foundTarget = true;
47             }
48
49             Shoot(foundTarget, gunHitInfo, direction);
50         }
51     }
```

```
52
53      virtual protected  void  Shoot(bool  foundTarget, RaycastHit hitInfo, Vector3 direction)
54      {
55      }
56 }
```

Now, in line 31, it checks the muzzle and the found target, if the target is farther than the muzzle, then its valid; the target is in front of the player. If not, then the target must be behind the player.

You'll need to change both RifleWeapon and MissileWeapon to take that extra argument into account.

RifleWeapon:

```
01 using  UnityEngine;
02 using  System.Collections;
03
04 public class  RifleWeapon : Weapon
05 {
06      [SerializeField]
07      int  _damageDealt = 10;
08
09      [SerializeField]
10      GameObject _hitEffectPrefab;
11
12      override protected  void  Shoot(bool  foundTarget, RaycastHit hitInfo, Vector3 direction)
13      {
14           if  (!foundTarget)
15           {
16                return;
17           }
18
19           Health enemyHealth = hitInfo.transform.GetComponent<Health>();
20           if  (enemyHealth != null)
21           {
22                enemyHealth.Damage(_damageDealt);
23
24                Vector3 hitEffectPosition = hitInfo.point;
25                Quaternion hitEffectRotation = Quaternion.FromToRotation(Vector3.forward,
                      hitInfo.normal);
26                Instantiate(_hitEffectPrefab, hitEffectPosition, hitEffectRotation);
27           }
28      }
29 }
```

MissileWeapon:

```
01 using  UnityEngine;
02 using  System.Collections;
03
04 public class  MissileWeapon : Weapon
05 {
06      [SerializeField]
07      GameObject _missilePrefab;
08
09      override protected  void  Shoot(bool  foundTarget, RaycastHit hitInfo, Vector3 direction)
10      {
11           GameObject m = Instantiate(_missilePrefab, _muzzle.position, _muzzle.rotation) as
                 GameObject;
12           m.transform.forward = direction;
```

```
    }
}
```

Adding Rapid Fire Functionality

We'll change the code so you can hold down the left mouse button and the weapon will keep on firing. Same with how we regulate the spawn rate of enemies, we'll regulate this one with a firing rate of sorts.

Add this code to the Weapon script:

```
using UnityEngine;
using System.Collections;

public class Weapon : MonoBehaviour
{
    [SerializeField]
    protected Transform _muzzle;

    [SerializeField]
    float _fireDelay = 0.3f;

    float _nextFireTimeAllowed = -1.0f;

    void Start()
    {
        Screen.lockCursor = true;
    }

    void Update()
    {
        if (Input.GetKey(KeyCode.Escape))
        {
            Screen.lockCursor = false;
        }

        if (Input.GetButton("Fire1") && Time.time >= _nextFireTimeAllowed)
        {
            _nextFireTimeAllowed = Time.time + _fireDelay;

            Screen.lockCursor = true;

            Ray mouseRay = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
            RaycastHit hitInfo;

            Vector3 direction;
            bool foundTarget = false;

            if (Physics.Raycast(mouseRay, out hitInfo) &&
                (Camera.main.WorldToScreenPoint(hitInfo.point).z >=
                Camera.main.WorldToScreenPoint(_muzzle.position).z))
            {
                foundTarget = true;
                direction = hitInfo.point - _muzzle.position;
```

```
42                  direction.Normalize();
43              }
44              else
45              {
46                  direction = Camera.main.ViewportToWorldPoint(new  Vector3(0.5f, 0.5f, 50)) -
                        _muzzle.position;
47                  direction.Normalize();
48              }
49
50              RaycastHit gunHitInfo;
51              if  (Physics.Raycast(_muzzle.position, direction, out gunHitInfo))
52              {
53                  foundTarget = true;
54              }
55
56              Shoot(foundTarget, gunHitInfo, direction);
57          }
58      }
59
60      virtual protected  void  Shoot(bool  foundTarget, RaycastHit hitInfo, Vector3 direction)
61      {
62      }
63 }
```

Firstly, we now use Input.GetButton() instead of Input.GetButtonDown(). This will allow us to just hold down the button and that new function will keep on returning true. The previous function only returned true once when the user presses the left mouse button.

Like our code in EnemySpawnManager, we regulate the firing rate by checking the current time elapsed with Time.time, and comparing that with our timed value for firing rate.

Try This

Try adding a shoot sound, played whenever the player shoots his weapon.

In Conclusion...

You've learned how to structure your code better by making parent classes and avoiding duplicate code. With that, any code you want present for the variations, you just put it in the parent class.