4 Physics Simulation

Introduction

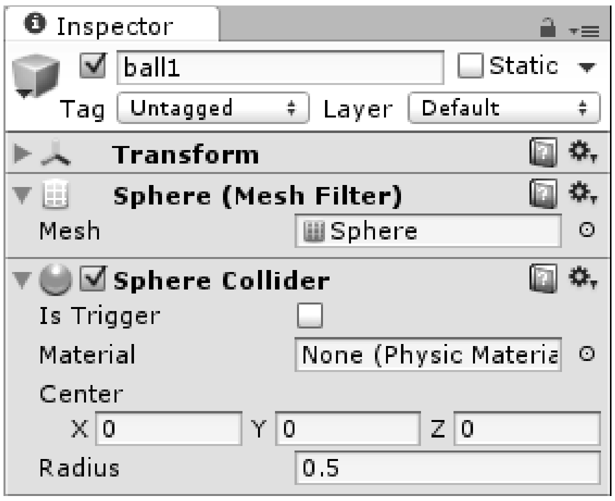
Up to now, we have been dealing with semi-static objects that do not move or rotate unless controlled by some script. In this chapter, we introduce physics simulation, an important function of any game engine. Physics simulation gives the objects realistic behavior and hence helps us making better, more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

After completing this chapter you should be able to:

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

4.1 Gravity and Collision Detection

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

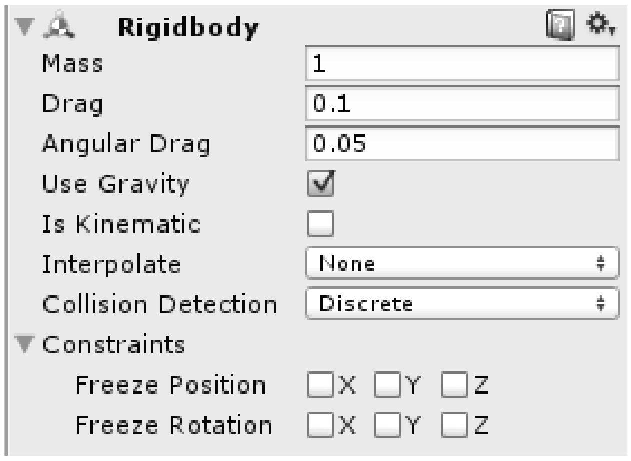


Inspector
ball1 — Static
Tag: Untagged — Layer: Default
Transform
Sphere (Mesh Filter)
Mesh: Sphere
Sphere Collider
Is Trigger: ☐
Material: None (Physic Materia
Center: X 0 Y 0 Z 0
Radius: 0.5

Illustration 43: The Collider component

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

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



Rigidbody

Mass	1
Drag	0.1
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	☐X ☐Y ☐Z
Freeze Rotation	☐X ☐Y ☐Z

Illustration 44: The Rigid Body component

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

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

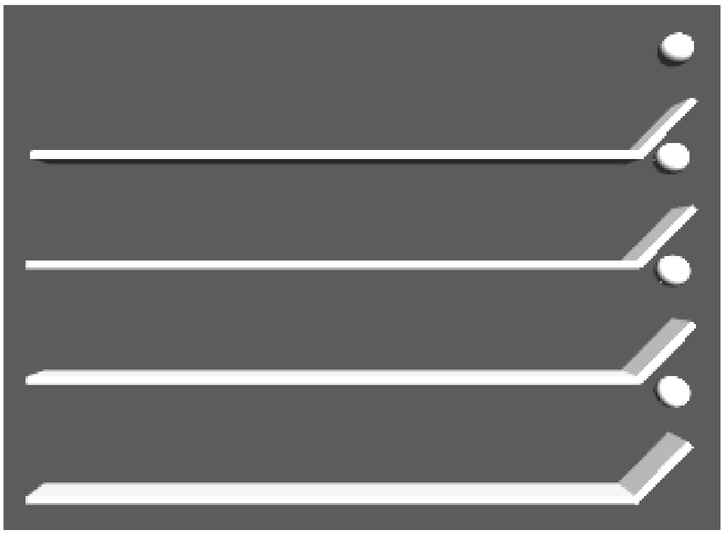


Illustration 45: A simple scene to demonstrate the properties of the rigid body

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

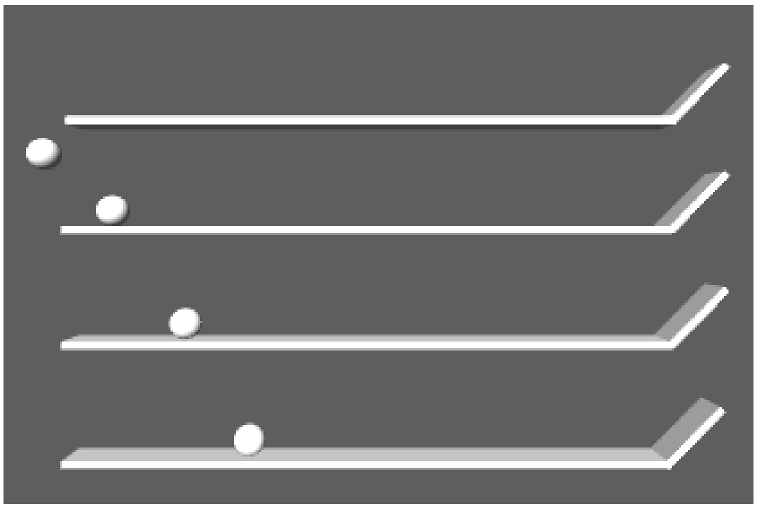


Illustration 46: The effect of the drag on the movement of the objects: upper balls have lower drag values

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

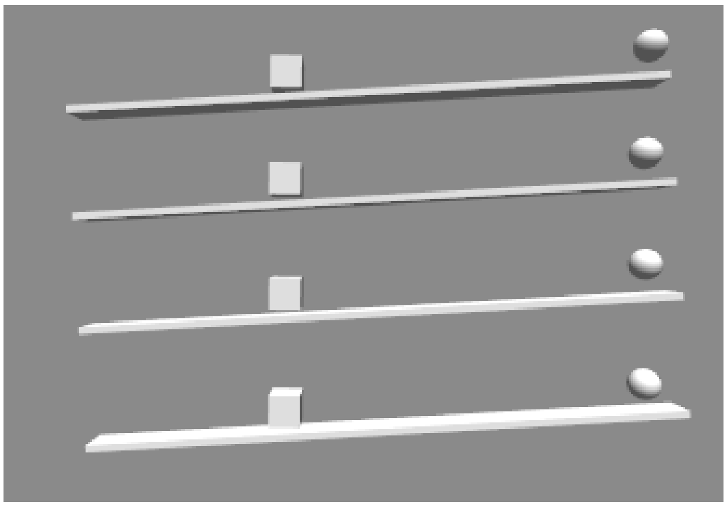
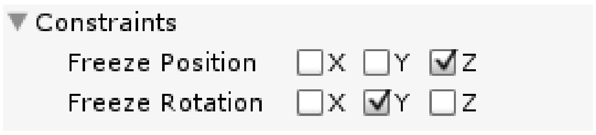


Illustration 47: A scene to demonstrate the effect of the mass of the rigid body

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



Constraints			
Freeze Position	☐ X	☐ Y	☑ Z
Freeze Rotation	☐ X	☑ Y	☐ Z

Illustration 48: Movement and rotation constraints of the rigid body

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

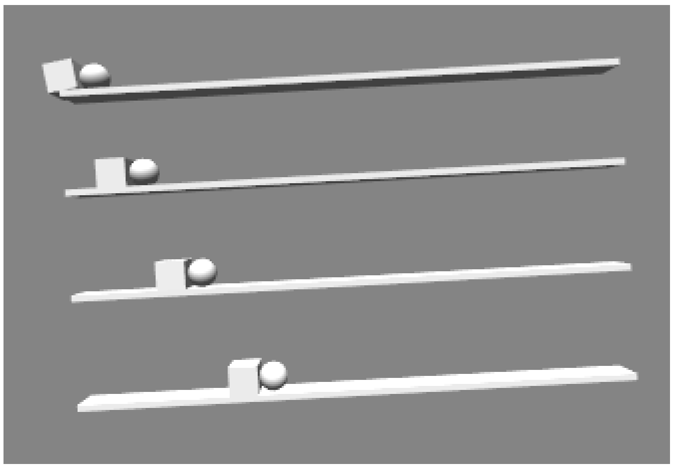


Illustration 49: The effect of the mass on the rigid body: upper balls have greater masses, and the cubes have equal masses

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

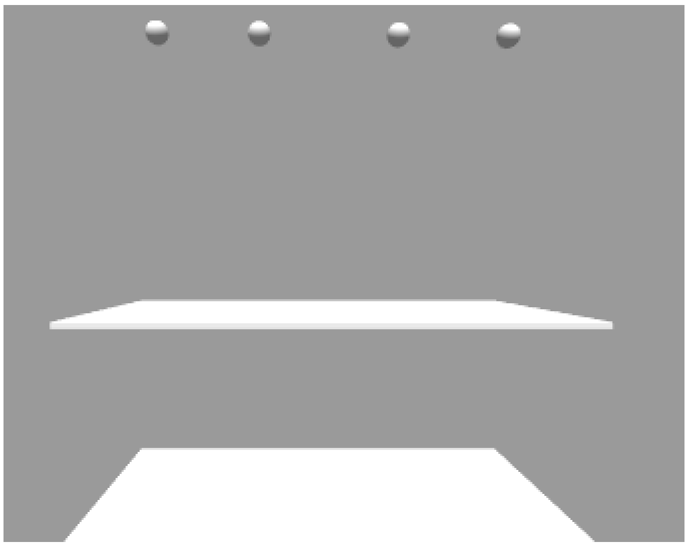


Illustration 50: A scene to demonstrate collision detection and handling in code

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

```
using UnityEngine;
using System.Collections;

public class ColorBall : MonoBehaviour {

	//Color of the ball
	public Color color;

	void Start () {
		renderer.material.color = color;
	}

	void Update () {

	}

	//To handle collision with solid colliders
	void OnCollisionEnter(Collision col){
		//access the colliding object and change its color
		col.collider.renderer.material.color = color;
	}

	//TO handle collision with triggers
	void OnTriggerEnter(Collider col){
		col.renderer.material.color = color;
	}
}
```

Listing 40: A script for handling collisions

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

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

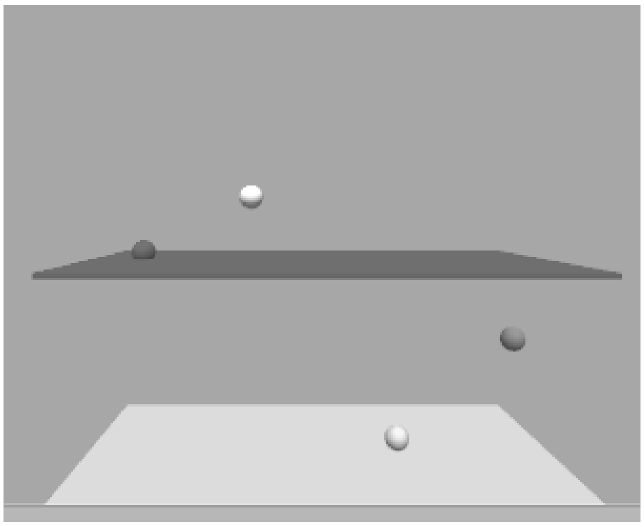


Illustration 51: Handling collisions and changing the colors of planes accordingly

4.2 Physical Vehicles

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

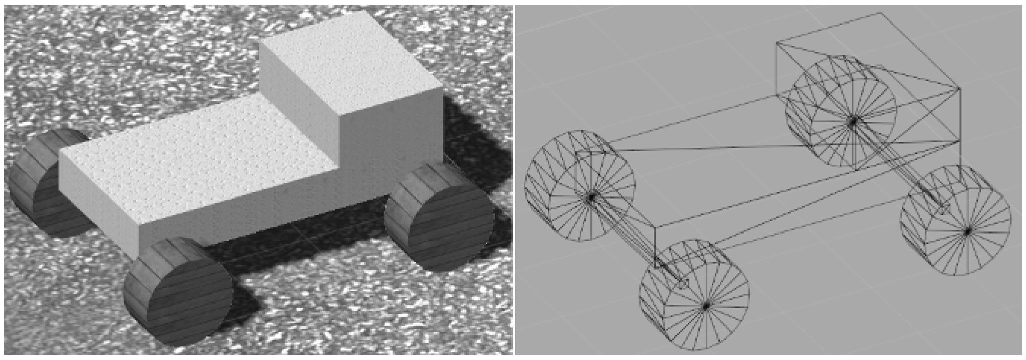
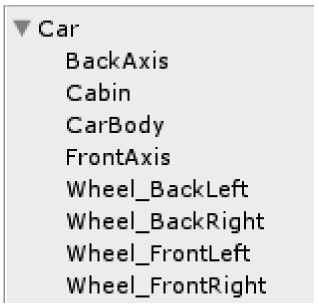


Illustration 52: A simple vehicle constructed using basic shapes

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



```
▼ Car
    BackAxis
    Cabin
    CarBody
    FrontAxis
    Wheel_BackLeft
    Wheel_BackRight
    Wheel_FrontLeft
    Wheel_FrontRight
```

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

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

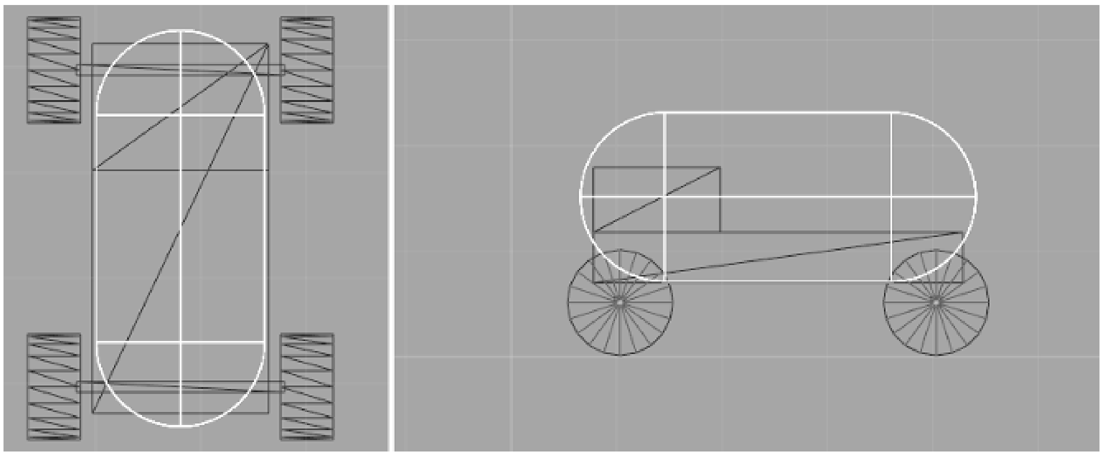
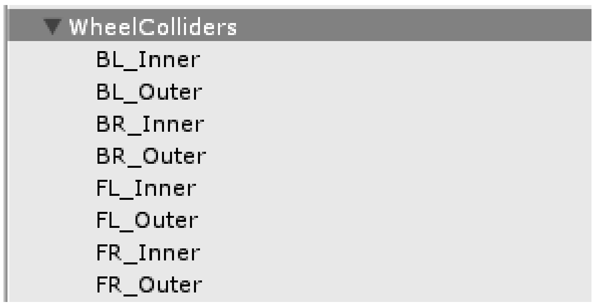


Illustration 54: Capsule collider added to the vehicle

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

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

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

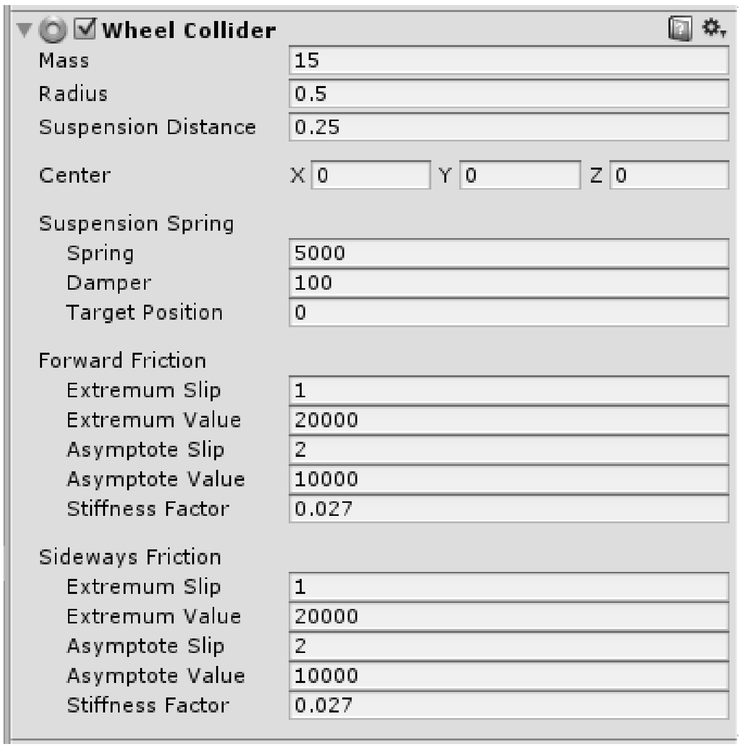


```
▼ WheelColliders
    BL_Inner
    BL_Outer
    BR_Inner
    BR_Outer
    FL_Inner
    FL_Outer
    FR_Inner
    FR_Outer
```

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

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

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



Wheel Collider

Mass	15
Radius	0.5
Suspension Distance	0.25
Center	X 0 Y 0 Z 0
Suspension Spring	
Spring	5000
Damper	100
Target Position	0
Forward Friction	
Extremum Slip	1
Extremum Value	20000
Asymptote Slip	2
Asymptote Value	10000
Stiffness Factor	0.027
Sideways Friction	
Extremum Slip	1
Extremum Value	20000
Asymptote Slip	2
Asymptote Value	10000
Stiffness Factor	0.027

Illustration 56: Setting the properties of the wheel colliders

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

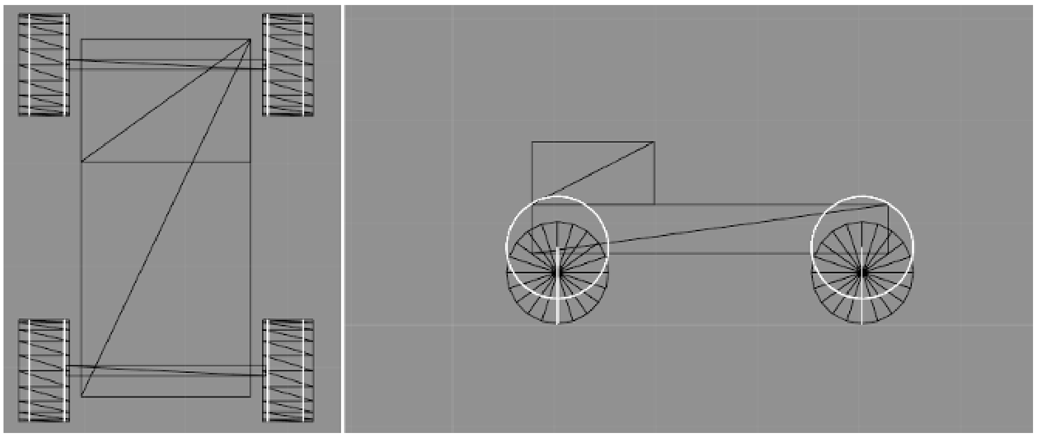
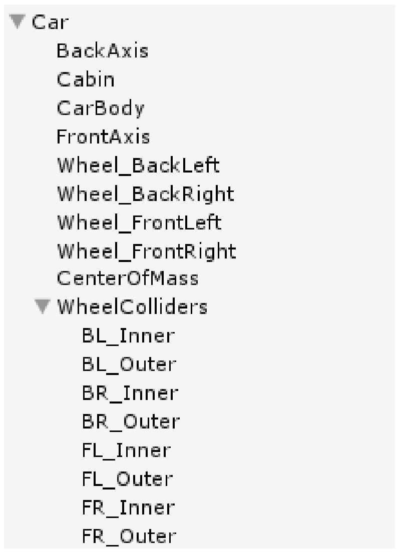


Illustration 57: The correct positioning of the wheel colliders in the vehicle. The colliders are shown in white color

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



```
▼ Car
    BackAxis
    Cabin
    CarBody
    FrontAxis
    Wheel_BackLeft
    Wheel_BackRight
    Wheel_FrontLeft
    Wheel_FrontRight
    CenterOfMass
  ▼ WheelColliders
      BL_Inner
      BL_Outer
      BR_Inner
      BR_Outer
      FL_Inner
      FL_Outer
      FR_Inner
      FR_Outer
```

Illustration 58: The complete hierarchy of the vehicle

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

```
using UnityEngine;
using System.Collections;

public class PhysicsCarDriver : MonoBehaviour {

    //All colliders of front wheels
    public WheelCollider[] frontWheels;

    //All colliders of back wheels
    public WheelCollider[] backWheels;

    //Car’s center of mass
    public Transform centerOfMass;

    //Max torque of the motor
    public float maxMotorTorque = 9500;

    //Braking power
    public float brakesTorque = 7500;

    //Angle to rotate wheels when steering
    public float maxSteeringAngle = 20;

    //steering rotation speed in degrees per second
    public float steeringSpeed = 30;

    //maximum car speed in km/h
    public float maxSpeed = 250;

    //maximum speed moving reverse
    public float maxReverseSpeed = 20;

    //current steering position
    float currensSteering = 0;

    //maximum car speed in rpm
    float maxRPM, maxReverseRPM;

    //Input flags
    bool accelerateForward,
        accelerateBackwards,
        brake, steerRight, steerLeft;

```

Listing 41: Variables declarations for *PhysicsCarDriver* script

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

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

```
    void Start () {
        //Convert max speed to rpm
        maxRPM =
            KmphToRPM(frontWheels[0], maxSpeed);
        maxReverseRPM =
            KmphToRPM(frontWheels[0], maxReverseSpeed);

        //Set the center of mass for better car turning
        rigidbody.centerOfMass =
            centerOfMass.localPosition;
    }

    //We use fixed update for physics
    void FixedUpdate () {
        //Update acceleration
        if(accelerateForward){
            foreach(WheelCollider wheel in frontWheels){
                UpdateWheelTorque(wheel, maxMotorTorque);
            }

            foreach(WheelCollider wheel in backWheels){
                UpdateWheelTorque(wheel, maxMotorTorque);
            }
            accelerateForward = false;
        } else if(accelerateBackwards){

            foreach(WheelCollider wheel in frontWheels){
                UpdateWheelTorque(wheel, -maxMotorTorque);
            }

            foreach(WheelCollider wheel in backWheels){
                UpdateWheelTorque(wheel, -maxMotorTorque);
            }
            accelerateBackwards = false;
        } else {
            foreach(WheelCollider wheel in frontWheels){
                UpdateWheelTorque(wheel, 0);
            }

            foreach(WheelCollider wheel in backWheels){
                UpdateWheelTorque(wheel, 0);
            }
        }

        //Update steering
        if(steerRight){
            UpdateSteering(steeringSpeed * Time.deltaTime);
            steerRight = false;
        } else if(steerLeft){
            UpdateSteering(-steeringSpeed * Time.deltaTime);
            steerLeft = false;
        } else {
```

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

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

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

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

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

```
    //Drive car forward
    public void AccelerateForward(){
         accelerateForward = true;
         accelerateBackwards = false;
    }

    //Drive backwards
    public void AccelerateBackwards(){
         accelerateBackwards = true;
         accelerateForward = false;
    }

    //Turn steering wheel to right
    public void SteerRight(){
         steerRight = true;
         steerLeft = false;
    }

    //Turn steering wheel to left
    public void SteerLeft(){
         steerLeft = true;
         steerRight = false;
    }

    //Apply braking to all wheels
    public void Brake(){
         brake = true;
    }

    //Applies torque to the wheel and checks RPM limits
    void UpdateWheelTorque(WheelCollider wheel, float torque){
         wheel.motorTorque = torque;
         if(wheel.rpm > maxRPM || wheel.rpm < -maxReverseRPM){
              wheel.motorTorque = 0;
         }
    }

    //Updates steering angle
    void UpdateSteering(float amount){
        if(amount != 0){
            currensSteering += amount;
        } else {
            //Steering is released,
            //return steering to straight
            //steering dead zone is
            //between -3 and 3 degrees
            if(currensSteering > 3){
                currensSteering -=
                        steeringSpeed * Time.deltaTime;
            } else if(currensSteering < -3){
                currensSteering +=
                        steeringSpeed * Time.deltaTime;
            } else {
                currensSteering = 0;
            }
        }
        //Apply max and min steering angles
        if(currensSteering > maxSteeringAngle){
            currensSteering = maxSteeringAngle;
        }

        if(currensSteering < -maxSteeringAngle){
        currensSteering = -maxSteeringAngle;
        }
        //Apply steering angle to front wheels only
        foreach(WheelCollider wheel in frontWheels){
            wheel.steerAngle = currensSteering;
        }
    }

    //Converts Km/h tp RPM based on
    //the radius of provided wheel
    float KmphToRPM(WheelCollider wheel, float speed){
        //Meters per hour
        float mph = speed * 1000;
        //Meters per minute
        float mpm = mph / 60;
        return mpm / (wheel.radius * 2 * Mathf.PI);
    }
}
```

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

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

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

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

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

```
using UnityEngine;
using System.Collections;

public class KeyboardCarController : MonoBehaviour {

     //Reference to car we are going to drive
     PhysicsCarDriver driver;

     void Start () {
          //Get the attached car driver
          driver = GetComponent<PhysicsCarDriver>();
     }

     void Update () {
          //Use up and down arrows for acceleration
          if(Input.GetKey(KeyCode.UpArrow)){
               driver.AccelerateForward();
          } else if(Input.GetKey(KeyCode.DownArrow)){
               driver.AccelerateBackwards();
          }

          //Use right and left arrows for steering
          if(Input.GetKey(KeyCode.RightArrow)){
               driver.SteerRight();
          } else if(Input.GetKey(KeyCode.LeftArrow)){
               driver.SteerLeft();
          }

          //Use space for braking
          if(Input.GetKey(KeyCode.Space)){
               driver.Brake();
          }
     }
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

public class CarSpeedMeasure : MonoBehaviour {

     public WheelCollider wheel;

     void Start () {

     }

     void Update () {
          //Print the speed in the console
          print (GetCarSpeed());
     }

     //Converts RPM to Km/h based on
     //the radius of the wheel
     float GetCarSpeed(){
          //Meters per minute
          float mpm = wheel.rpm * wheel.radius * 2 * Mathf.PI;
          //Meters per hour
          float mph = mpm * 60;
          //Kilometers per hour
          float kmph = mph / 1000;
          return kmph;
     }
}
```

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

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

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

```
using UnityEngine;
using System.Collections;

public class CarWheelAnimator : MonoBehaviour {

      //The actual wheel to read data from
      public WheelCollider wheel;

      //Axis for vertical rotaion
      public Transform steeringAxis;

      //Stores steering angle from last frame,
      //in order to be able to reset Y rotation
      //of the wheel
      float lastSteerAngle = 0;

      //To save the original position at the position
      Vector3 originalPos;

      void Start () {
            //Register the original position of the wheel
            originalPos = transform.localPosition;
      }

      void LateUpdate () {
            //Convert wheel speed in rpm to degrees per second
            float rotationsPerSecond = wheel.rpm / 60;
```

```
        float degreesPerSecond = rotationsPerSecond * 360;

        //Rotate around local y axis
        transform.Rotate(0, degreesPerSecond * Time.deltaTime, 0);

        //Steering axis exists in front wheels
        if(steeringAxis != null){
            //Reset the steering to zero by sbtracting
            //the steering value of last frame
            transform.RotateAround(
                                steeringAxis.position,
                                steeringAxis.up,
                                -lastSteerAngle);
            //Apply new steering value
            transform.RotateAround(
                                steeringAxis.position,
                                steeringAxis.up,
                                wheel.steerAngle);
            //Update last steering angle value for the next frame
            lastSteerAngle = wheel.steerAngle;
        }

        //Check if the wheel hits the ground
        WheelHit hit;

        if(wheel.GetGroundHit(out hit)){
            //Wheel hits the ground.
            //Move the wheel up by spring pressing distance
            //Use world space
            float colliderCenter = hit.point.y + wheel.radius;
            Vector3 wheelPosition = transform.position;
            wheelPosition.y = colliderCenter;
            transform.position = wheelPosition;
        } else {
            //No hit, smoothly return wheel to its original position
            Vector3 pos = transform.localPosition;
            pos = Vector3.Lerp(transform.localPosition,
                                           originalPos, Time.deltaTime);
            transform.localPosition = pos;
        }
    }
}
```

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

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

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

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

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

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

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

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

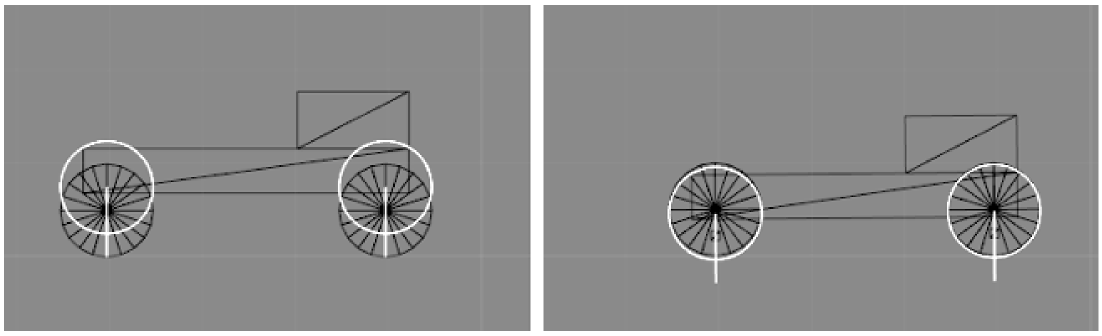


Illustration 59: Suspension springs at rest state (left) and while pressed (right). When springs are pressed wheel colliders sink into the ground and the car body is lowered, while visual wheels are kept on the ground.

4.3 Physical player character

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

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

```
using UnityEngine;
using System.Collections;

public class PhysicsCharacter : MonoBehaviour {

	//maximum jump height in meters
	public float jumpHeight = 2;

	//Horizontal movement speed
	public float movementSpeed = 8;

	//Position of the character's feet
	public Transform feet;

	//Input flags
	bool walkForward, walkBackwards,
		strafeRight, strafeLeft, jump;

	void Start () {

	}

	public void WalkForward(){
		walkForward = true;
		walkBackwards = false;
	}

	public void WalkBackwards(){
		walkBackwards = true;
		walkForward = false;
	}

	public void StrafeRight(){
		strafeRight = true;
		strafeLeft = false;
	}

	public void StrafeLeft(){
		strafeLeft = true;
		strafeRight = false;
	}

	public void Jump(){
		jump = true;
	}

	public void Turn(float amount){
		transform.RotateAround(Vector3.up, amount);
	}

	//Fixed update is better with physics
	void FixedUpdate () {
		//Player can direct character only
		//when it is on the ground
		//or stuck somewhere with near-zero vertical velocity
```

```
        Vector3 velocity = rigidbody.velocity;
        if(OnGround() || (velocity.y >= 0 && velocity.y < 0.1f)){
            //Reset velocity on x and z to zero
            velocity.x = velocity.z = 0;

            //update movement
            if(strafeLeft){
                //Move left
                velocity += -transform.right * movementSpeed;
                strafeLeft = false;
            } else if(strafeRight){
                //Move right
                velocity += transform.right * movementSpeed;
                strafeRight = false;
            }

            if(walkForward){
                //Move forward
                velocity += transform.forward * movementSpeed;
                walkForward = false;
            } else if(walkBackwards){
                //Move backwards
                velocity += -transform.forward * movementSpeed;
                walkBackwards = false;
            }
        }

        rigidbody.velocity = velocity;

        //Read jump input
        if(jump && OnGround()){
            //v2^2 - v1^2 = 2as
            //v2 is zero (at max height)
            //v1^2 = -2as
            //v1 = sqrt(-2as)
            float v1 =
                Mathf.Sqrt(-2 * Physics.gravity.y * jumpHeight);

            //Use momentum formula p=mv with up as velocity direction
            rigidbody.AddForce(
                Vector3.up * v1 * rigidbody.mass,
                ForceMode.Impulse);

            jump = false;
        }

    }

    //Checks whether the character is on the ground
    public bool OnGround(){
        //Cast a ray from feet position downwards.
        //The length of the ray is 10 cm.
```

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

Listing 47: Character controller based on physics simulation

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

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

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

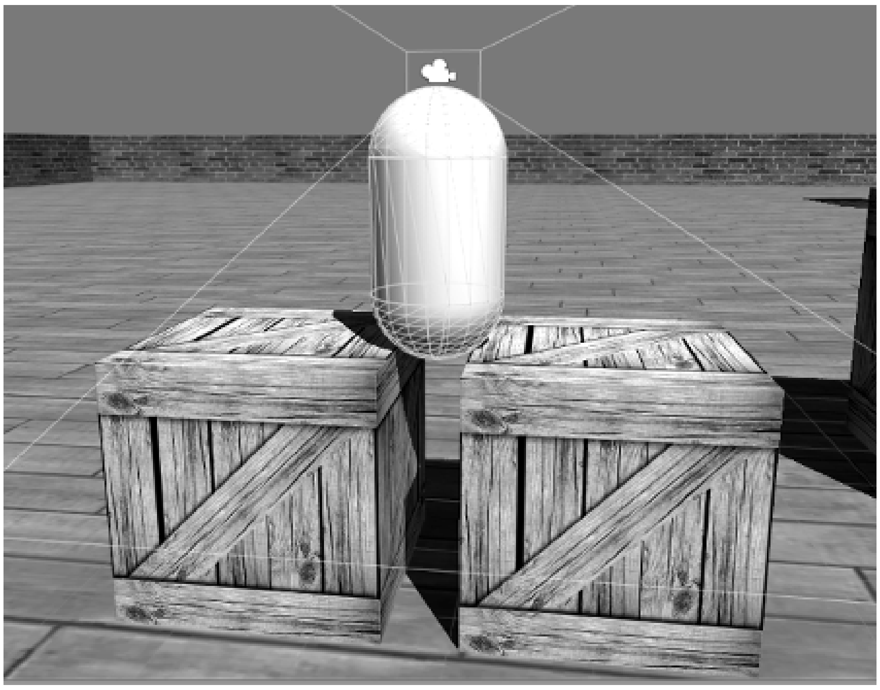


Illustration 60: The character in this case is not grounded. However, it must still be controllable

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

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

We use the projectile formula, $v_2^2 - v_1^2 = 2as$, where v_2 is the velocity of the objects when it reaches its maximum height, v_1 is the initial velocity of the object when it leaves the ground, *a* is the acceleration, and *s* is the maximum height the object reaches. In our case, v_1 is the sole unknown we need to work out for. At the maximum height, the velocity of the object reaches zero before it starts to fall, hence $v_2 = 0$. *s* in our case is the value of *jumpHeight*, which is also known to us. As the object goes up, it loses its velocity due to the acceleration of its weight, which is the acceleration of the gravity ($9.8 m/s^2$). By working out for v_1, we get $v_1 = \sqrt{-2as}$, which is expressed in lines 91 and 92 in the script. Since jump force is a momentum that is applied once, we can use the momentum formula $p = mv$ to compute the amount of the force we need to add to the rigid body of the object. This formula is expressed in line 96. We call *rigidbody.AddForce()* at line 95, which is used to add a force to the rigid body. After adding the appropriate jump force, we reset *jump* flag to *false*.

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

```
using UnityEngine;
using System.Collections;

public class FPSInput : MonoBehaviour {

     //Mouse look speed on both axis
     public float horizontalMouseSpeed = 0.9f;
     public float verticalMouseSpeed = 0.5f;

     //Max allowed cam vertical angle
     public float maxVerticalAngle = 60;

     //Mouse position in previous frame,
     //important to measure mouse displacement
     private Vector3 lastMousePosition;

     //Store camera transform
     private Transform camera;

     //The character to control
     PhysicsCharacter character;

     void Start () {
          character = GetComponent<PhysicsCharacter>();
          lastMousePosition = Input.mousePosition;
          //Find camera object in children
          camera = transform.FindChild("Main Camera");
     }

     void Update () {
          //Step 1: rotate cylinder around global Y
          //axis based on horizontal mouse displacement
          Vector3 mouseDelta = Input.mousePosition - lastMousePosition;

          character.Turn(mouseDelta.x *
               horizontalMouseSpeed *
               Time.deltaTime);

          //Get current vertical camera rotation
          float currentRotation = camera.localRotation.eulerAngles.x;

          //Convert vertical camera rotation from range [0, 360]
          //to range [-180, 180]
          if(currentRotation > 180){
               currentRotation = currentRotation - 360;
          }

          //Calculate rotation amout for current frame
          float ang =
               -mouseDelta.y * verticalMouseSpeed * Time.deltaTime;

          //Step 2: rotate camera around it’s local X
          //axis based on vertical mouse displacement
          //First check allowed limits
```

```
        if((ang < 0 && ang + currentRotation > -maxVerticalAngle) ||
            (ang > 0 && ang + currentRotation < maxVerticalAngle)){
            camera.RotateAround(camera.right, ang);
        }

        //Update last mouse position for next frame
        lastMousePosition = Input.mousePosition;

        if(Input.GetKey(KeyCode.A)){
            character.StrafeLeft();
        } else if(Input.GetKey(KeyCode.D)){
            character.StrafeRight();
        }

        if(Input.GetKey(KeyCode.W)){
            character.WalkForward();
        } else if(Input.GetKey(KeyCode.S)){
            character.WalkBackwards();
        }

        if(Input.GetKeyDown(KeyCode.Space)){
            character.Jump();
        }
    }
}
```

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

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

4.4 Ray cast shooting

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

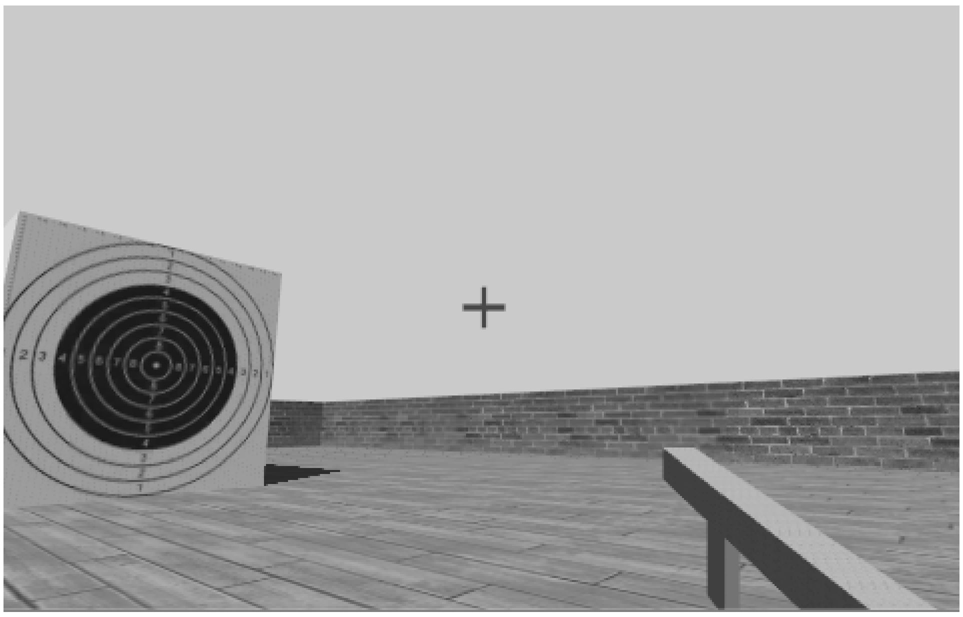


Illustration 61: A simple gun and a crosshair made of basic shapes and added to the first person camera

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

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

```
using UnityEngine;
using System.Collections;

public class RaycastShooter : MonoBehaviour {

     //How far can each bullet travel
     public float maxRange = 100;

     //how many seconds to wait between two consecutive shoots
     public float shootRate = 0.1f;

     //vertical inaccuracy in degrees
     public float verticalInaccuracy = 1;

     //horizontal inaccuracy in degrees
     public float horizontalInaccuracy = 1;

     //Bullet damage from zero-distance
     public float power = 100;

     //Possition and direction of casted rays
     public Transform muzzle;

     //Last time shooting is performed
     float lastShootTime = 0;

     //Store last inaccuracy vector
     Vector2 inaccuracyVector;

     void Start () {

     }

     void Update () {

     }

     //Shoot a bullet using ray cast
     //return true if a bullet has been shot
     public void Shoot(){
          if(Time.time - lastShootTime > shootRate){
               //Get some random values for inaccuracy
               inaccuracyVector.y = Random.Range(
                                            -horizontalInaccuracy,
                                            horizontalInaccuracy);

               inaccuracyVector.x = Random.Range(
                                            -verticalInaccuracy,
                                            verticalInaccuracy);

               //Rotate the muzzle to apply inaccuracy
               muzzle.Rotate(inaccuracyVector.x, 0, 0);
               muzzle.Rotate(0, inaccuracyVector.y, 0);

               //A variable to store hit data
```

```
            RaycastHit hit;

            //Perform the ray cast
            if(Physics.Raycast(
                    new Ray(muzzle.position, muzzle.forward),
                    out hit, maxRange)){

                //Overwrite hit.distance
                //with the value of scaled damage

                hit.distance =
                    power * (1 – (hit.distance / maxRange));
                hit.transform.SendMessage(
                        "OnRaycastHit", hit,
                        SendMessageOptions.DontRequireReceiver);

            }

            //return muzzle to its original rotation
            muzzle.Rotate(-inaccuracyVector.x, 0, 0);
            muzzle.Rotate(0, -inaccuracyVector.y, 0);

            //Register last shooting time
            lastShootTime = Time.time;

            //Inform other scripts that shooting happened
            SendMessage("OnRaycastShoot",
                    SendMessageOptions.DontRequireReceiver);
        }
    }

    //Get last inaccuracy
    public Vector2 GetLastInaccuracyVector(){
        return inaccuracyVector;
    }
}
```

Listing 49: Ray cast shooting scrip

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

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

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

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

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

```
using UnityEngine;
using System.Collections;

public class RaycastShooterAnimator : MonoBehaviour {

	//Distance to move backwards when animating
	public float zDistance = 0.15f;

	//Reference to shooter
	RaycastShooter shooter;

	//Original position
	Vector3 originalPosition;

	//Original rotation
	Quaternion originalRotation;

	void Start () {
		shooter = GetComponent<RaycastShooter>();
		originalPosition = transform.localPosition;
		originalRotation = transform.localRotation;
	}

	void LateUpdate () {
		//Slowly return to original position and rotation
		transform.localPosition =
			Vector3.Lerp(transform.localPosition,
				originalPosition, Time.deltaTime * 10);

		transform.localRotation =
			Quaternion.Lerp(transform.localRotation,
				originalRotation, Time.deltaTime * 10);
	}

	void OnRaycastShoot(){
		//Shooting happend: animate based on in
		Vector2 rotation =
			shooter.GetLastInaccuracyVector();
		transform.Rotate(rotation.x, 0, 0);
		transform.Rotate(0, rotation.y, 0);
		transform.Translate(0, 0, -zDistance);
	}
}
```

Listing 50: A script to animate the gun during shooting

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

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

```
using UnityEngine;
using System.Collections;

public class GunInput : MonoBehaviour {

     //If true, the player don't have to release
     //the mouse button between shoots
     public bool continuous = true;

     void Start () {

     }

     void Update () {
          //Send Shoot message on mouse click
          if(continuous){
               if(Input.GetMouseButton(0)){
                    SendMessage("Shoot");
               }
          } else {
               if(Input.GetMouseButtonDown(0)){
                    SendMessage("Shoot");
               }
          }
     }
}
```

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

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

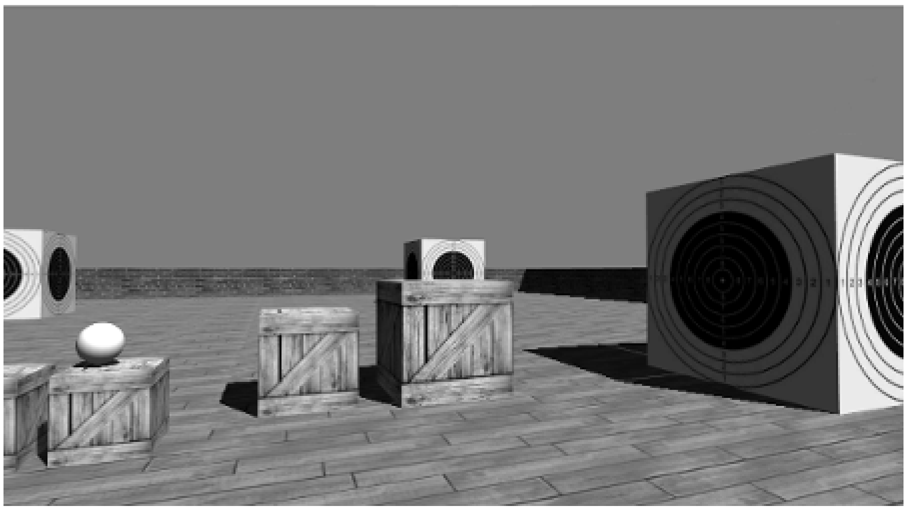


Illustration 62: A scene to test ray cast shooting

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

```
using UnityEngine;
using System.Collections;

public class BulletHoleMaker : MonoBehaviour {

      //Object to instantiate as hole
      public GameObject holePrefab;

      //Seconds to wait before destroying hole object
      public float holeLife = 15;

      void Start () {

      }

      void Update () {

      }

      //Receive OnRaycastHit message and generate hole at hit position
      void OnRaycastHit(RaycastHit hit){
           GameObject hole = (GameObject) Instantiate(holePrefab);
           hole.transform.position = hit.point;
           //If there is a rigid body, add the hole as a child to it
           //This makes the hole move along with the hit object
           if(hit.rigidbody != null){
                hole.transform.parent = hit.transform;
           }
           //Since we use a quad, we rotate it towards inside
           //This might be different if you use another shape
           hole.transform.LookAt(hit.point - hit.normal);
           //move it away from hit surface with a tiny amount
           //This ensures that the hole is always on top
           hole.transform.Translate(0, 0, -0.0125f);

           //Invoke destruction after a while
           Destroy(hole, holeLife);

           //Report some data
           print (name + " took damage of " + hit.distance);
      }
}
```

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

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

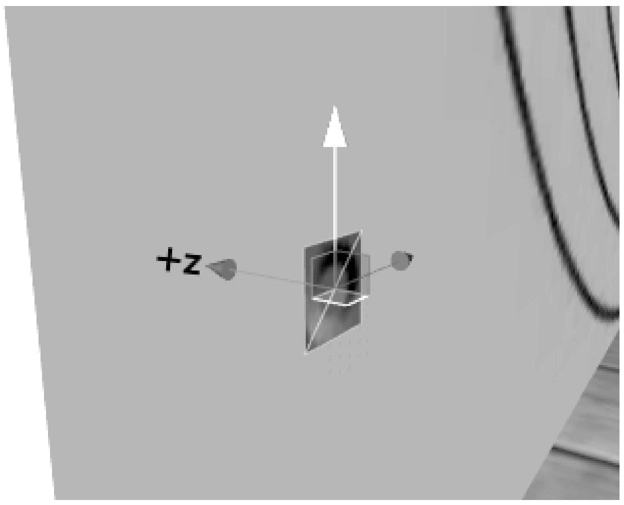


Illustration 63: Hole object made of quad, the positive z axis is looks inside so that the rendered face looks outside

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

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

```
using UnityEngine;
using System.Collections;

public class BulletForceReceiver : MonoBehaviour {

     void Start () {

     }

     void Update () {

     }

     //Receive OnRaycastHit message and apply an impulse force
     void OnRaycastHit(RaycastHit hit){
          rigidbody.AddForceAtPosition(
               -hit.normal * hit.distance, hit.point, ForceMode.Impulse);
     }
}
```

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

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

4.5 Physics projectiles

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

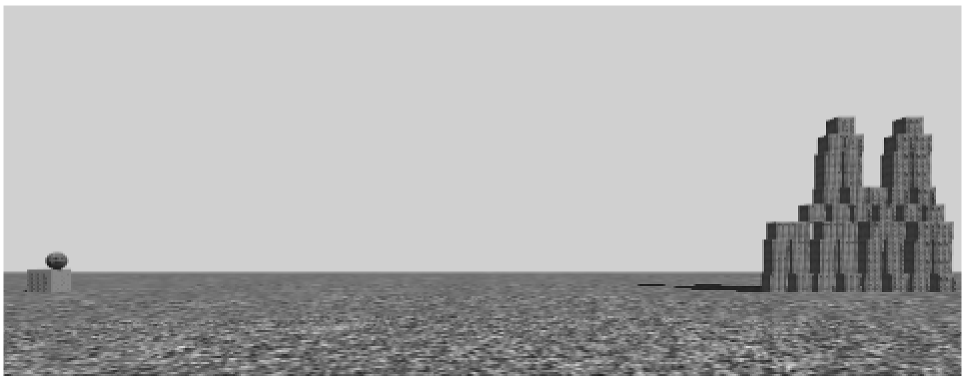
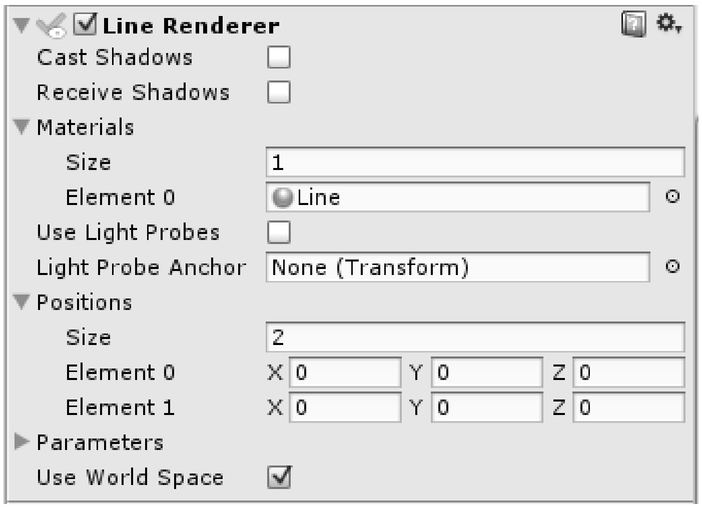


Illustration 64: A scene to demonstrate physics projectiles

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



Line Renderer

Cast Shadows	☐
Receive Shadows	☐
Materials	
Size	1
Element 0	Line
Use Light Probes	☐
Light Probe Anchor	None (Transform)
Positions	
Size	2
Element 0	X 0 Y 0 Z 0
Element 1	X 0 Y 0 Z 0
Parameters	
Use World Space	✓

Illustration 65: Line renderer that draws launch direction indicator

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

```
using UnityEngine;
using System.Collections;

public class PhysicsProjectile : MonoBehaviour {

	//Launch force multiplier
	public float launchPower = 300;

	//How long seconds to keep the projectile alive after launching?
	//-1 = keep for infinity
	public float lifeTime = 7;

	//Has the player hold the mouse down on the object?
	bool mousePressed = false;

	//Has the projectile been already launched?
	bool launched = false;

	//Position to generate launch power from
	Vector3 launchPosition;

	//Line to show launch direction
	LineRenderer line;

	//Reference to main camera,
	//necessary for launch point specification with the mouse
	Camera cam;

	void Start () {
		//Get the attached line and find the camera
		line = GetComponent<LineRenderer>();
		cam = Camera.main;
	}

	void Update () {
		//We draw line after pressing the mouse on the projectile,
		//and before launching the projectile
		if(!launched && mousePressed){
			//Create a ray that goes from
			//camera position into the screen,
			//and passes throug mouse pointer position
			Ray cameraRay = cam.ScreenPointToRay(Input.mousePosition);

			//Find the distance between the camera and the projectile
			float dist = Vector3.Distance(
					cam.transform.position, transform.position);
```

```

            //Set the launch position to the point on the ray that
            //has the same distance from the camera as the projectile
            launchPosition = cameraRay.GetPoint (dist);

            //Update line start and end positions
            //Line starts at the position of the projectile
            line.SetPosition(0, transform.position);

            //Line ends at the launch position
            line.SetPosition(1, launchPosition);

            //After drawing the line, make sure that projectile
            //and lanunch position has the same z position
            launchPosition.z = transform.position.z;
        }
    }

    //Called when a mouse button is pressed on the object
    void OnMouseDown(){
        mousePressed = true;
    }

    //Called when a mouse button is released
    //and the same button has been already pressed in the object
    void OnMouseUp(){
        //Projectile must not has been alread launched
        if(!launched){
            //Set launched to true and destroy the line component
            launched = true;
            Destroy(line);

            //Destroy object after its lifetime
            Destroy(gameObject, lifeTime);

            //Apply a force that is directly proportional with
            //the distance between launch position and
            //the position of the projectile
            Vector3 forceDirection =
                transform.position - launchPosition;
            forceDirection = forceDirection * launchPower;
            rigidbody.AddForce(forceDirection, ForceMode.Impulse);
        }
    }
}
```

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

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

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

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

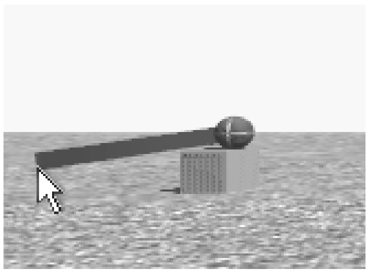


Illustration 66: Indication line drawn between the projectile and the mouse pointer

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

```
using UnityEngine;
using System.Collections;

public class ProjectileGenerator : MonoBehaviour {

    //Prefab to generate
    public GameObject projectile;

    void Start () {
        //Try to generate a projectile once every second
        InvokeRepeating("Generate", 0, 1);
    }

    void Update () {

    }

    void Generate(){
        //If there are no projectiles in the scene, generate one
        PhysicsProjectile[] prjectiles =
            FindObjectsOfType<PhysicsProjectile>();
        if(prjectiles.Length == 0){
            Instantiate(projectile,
                          transform.position,
                          transform.rotation);
        }
    }
}
```

Listing 55: A script to continuously generate projectiles

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

4.6 Explosions and destruction

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

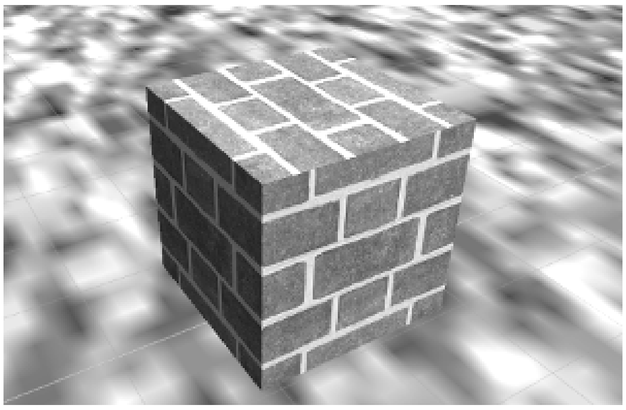


Illustration 68: Brick-textured cube to be used as building block for the destructible building

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

```
using UnityEngine;
//We have to import this library
using System.Collections.Generic;

public class Destructible : MonoBehaviour {

      //We scan from the center of the object towards this direction
      //in order to find a dependency
      public Vector3 scanDirection;

      //Desructibles that depend on this one
      List<Destructible> dependents = new List<Destructible>();

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

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

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

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

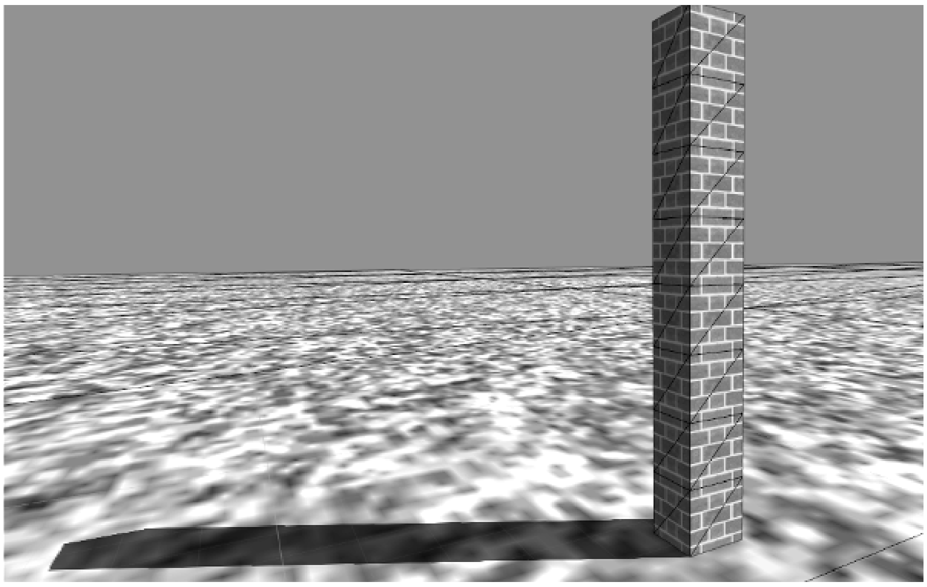


Illustration 69: A pillar made of 8 destructible building blocks

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

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

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

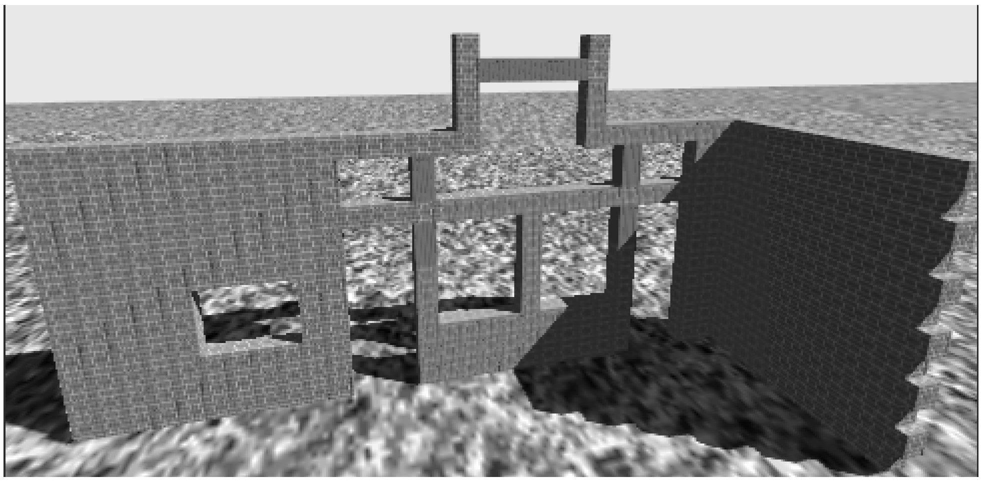


Illustration 70: A building constructed completely from destructible building blocks

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

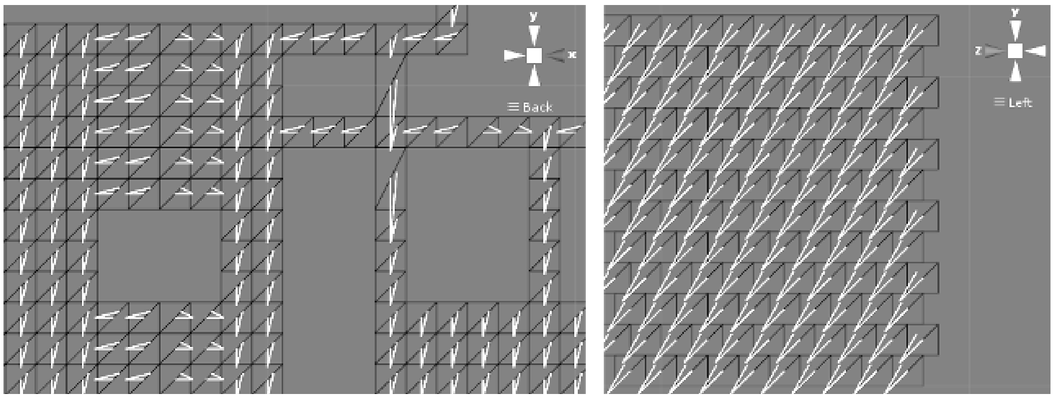


Illustration 71: Comparing block arrangement between left and right walls: the blocks of left wall scan down, left, or right, while the blocks of the right wall scan towards bottom left

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

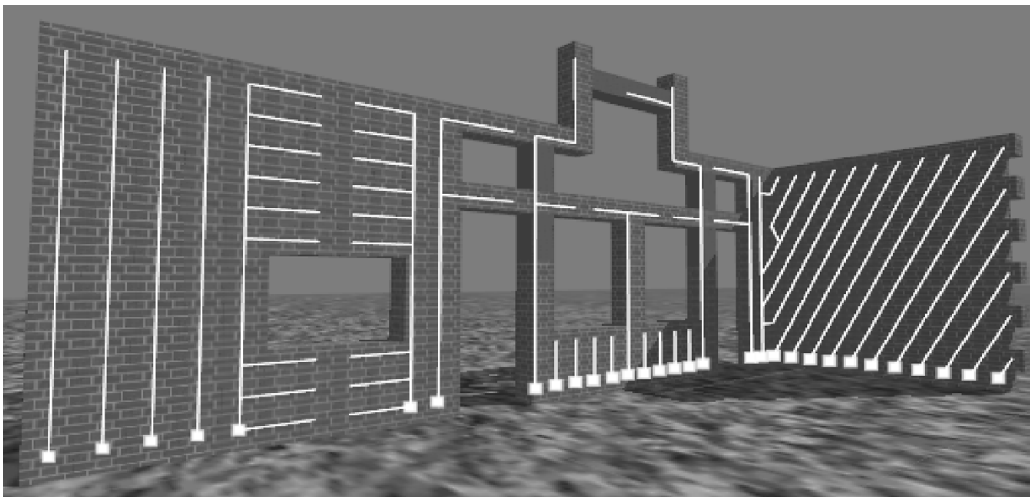


Illustration 72: Dependency trees between building blocks: when the root destructs, the whole tree must eventually be destructed. Roots are displayed as small squares

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

```
using UnityEngine;
using System.Collections;

public class MouseExploder : MonoBehaviour {

	//How strong is the explosion?
	public float explosionForce = 40000;

	//Range of explsion effect
	public float explosionRadius = 5;

	//Position of the explosion
	//Relative to the position of the object
	//Expressed in world space
	public Vector3 explosionPosition = new Vector3(-1, 0, -1);

	void Start () {

	}

	void Update () {

	}

	//Perform explosion on mouse click
	void OnMouseDown(){
		//Get all rigid bodies
		Rigidbody[] allBodies = FindObjectsOfType<Rigidbody>();

		//Calculate the position of the explosion
		Vector3 explosionPos = transform.position;
		explosionPos += explosionPosition;

		//Find bodies within radius, send destruction
		//message in case they are destructible,
		//and finally apply explosion force to them
		foreach(Rigidbody body in allBodies){
			float dist =
				Vector3.Distance(
					body.transform.position,
					explosionPos);

			if(dist < explosionRadius){
				body.SendMessage("Destruct",
					SendMessageOptions.DontRequireReceiver);

				body.AddExplosionForce(
					explosionForce, //Explosion strength
					explosionPos, //Explosion position
					explosionRadius);//effective radius
			}
		}
	}
}
```

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

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

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

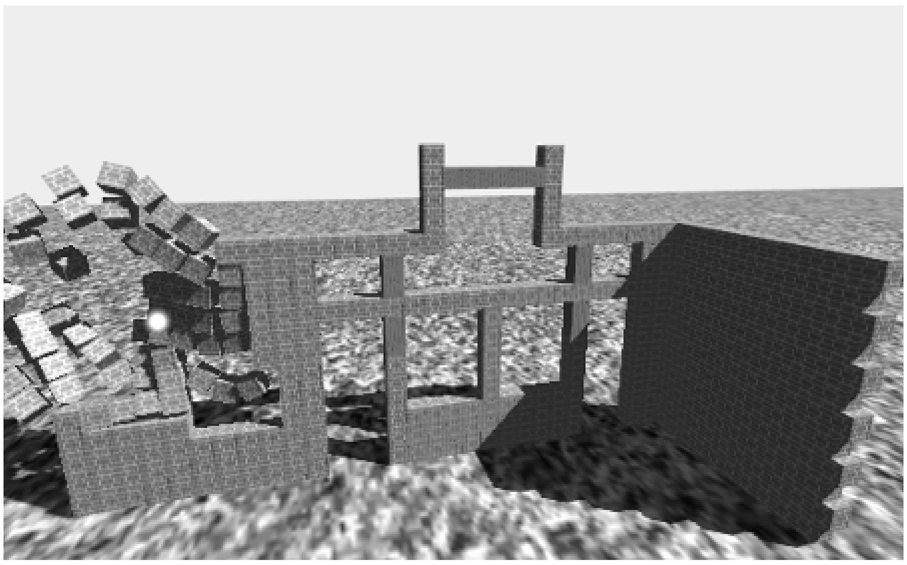


Illustration 73: Effect of the explosion force on destructible blocks: the blocks are pushed away from the position of the explosion, which is shown as bright point

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

```
using UnityEngine;
using System.Collections;

public class Returner : MonoBehaviour {

     //Original position of the object
     Vector3 originalPos;
     //Original rotation of the object
     Quaternion originalRot;
     //Is the object currently moving to its original position?
     bool returning = false;

     void Start () {
          //Record original position and rotation
          originalPos = transform.position;
          originalRot = transform.rotation;
     }

     void Update () {

          //When space is pressed, initialize returning
          if(Input.GetKeyDown(KeyCode.Space)){
          Return();
          }

     if(returning){
          //During return freese all constraints to prevent
          //external forces from affecting the rigid body
          if(rigidbody.constraints !=
               RigidbodyConstraints.FreezeAll){
               //Clear any linear or angular
               // velocities to stop object
               rigidbody.velocity = Vector3.zero;
               rigidbody.angularVelocity = Vector3.zero;
               //Now freeze the rigid body
               rigidbody.constraints =
                     RigidbodyConstraints.FreezeAll;
          }

          //Smoothly return position and rotation
          //to their original values
          transform.position =
               Vector3.Lerp(transform.position, //from
                                 originalPos, //to
                                 Time.deltaTime * 3);//amount
```

```

            transform.rotation =
                Quaternion.Lerp(
                                transform.rotation, //from
                                originalRot, //to
                                Time.deltaTime * 3);//amount

            //If the object is too near, manually set original values,
            //and set returning to false
            float remaining =
                    Vector3.Distance(transform.position, originalPos);
                if(remaining < 0.01f){
                    transform.position = originalPos;
                    transform.rotation = originalRot;
                    returning = false;
                }
            }
        }

        //Return the object to its original position
        public void Return(){
            returning = true;
        }
}
```

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

We do not reinvent the wheel we reinvent light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA

Click on the ad to read more

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

4.7 Breakable objects

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

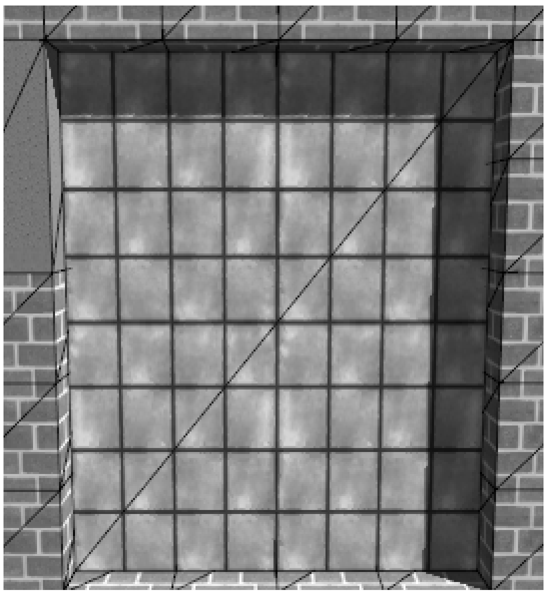


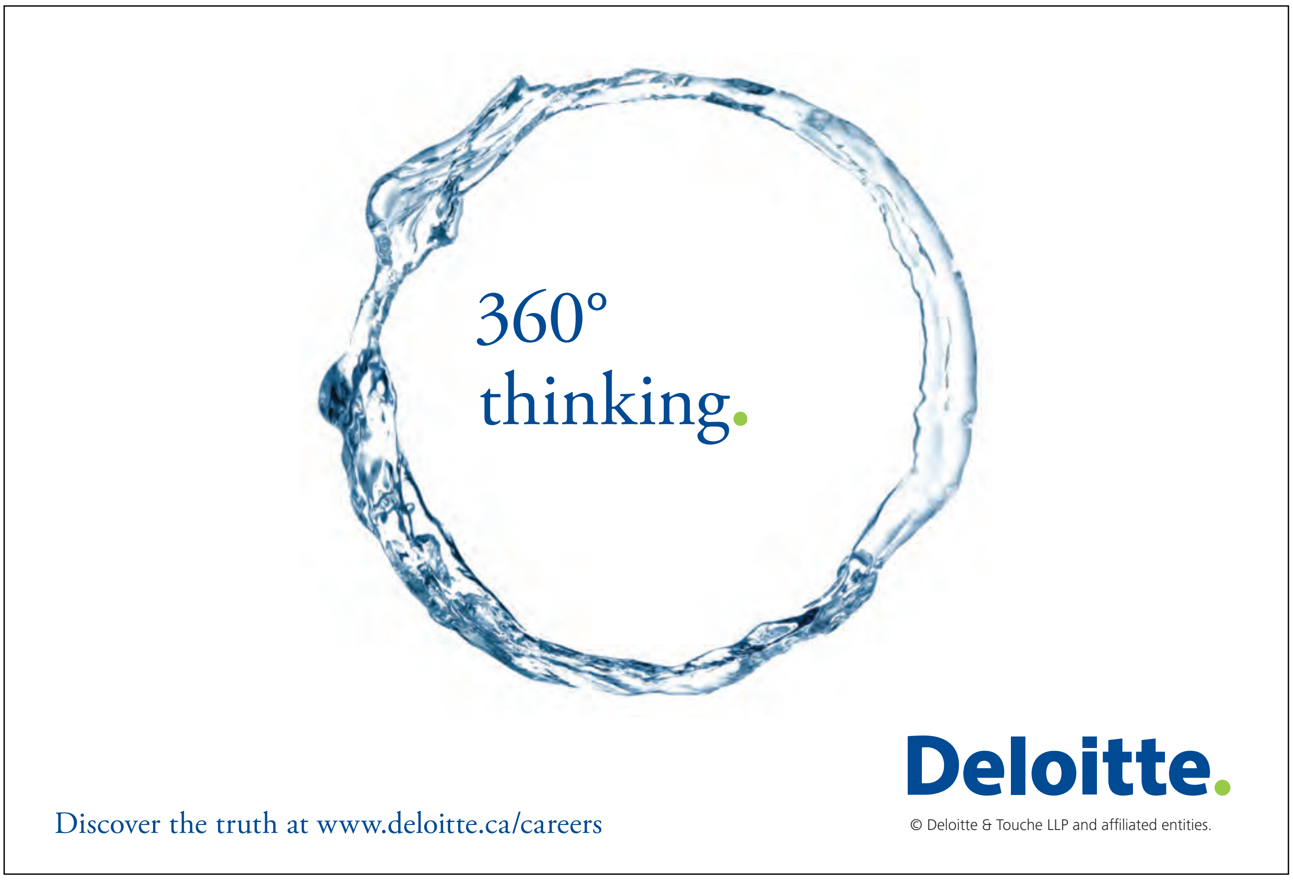
Illustration 74: Glass block to be used as breakable window

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

```
using UnityEngine;
using System.Collections;

public class Breakable : MonoBehaviour {

      //Objects to generate when broken (pieces)
      public GameObject[] pieces;

      //How many times each piece should be generated?
      public int pieceCopies = 5;

      //How strong this breakable explodes?
      public float explosionPower = 10;

      //How long to keep pieces?
      public float pieceLifetime = 10;

      void Start () {

      }

      void Update () {

      }

      //Break the breakable
      public void Break(){
            //Generate the pieces with spiecified count
            foreach(GameObject piecePrefab in pieces){
                  for(int i = 0; i < pieceCopies; i++){
                        GameObject piece =
                              (GameObject)Instantiate(piecePrefab);

                        //Get a random position within
                        //the borders of the object
                        Vector3 borders = transform.localScale;
                        Vector3 randPos;

                        randPos.x =
                         Random.Range(-borders.x * 0.5f, borders.x * 0.5f);

                        randPos.y =
                         Random.Range(-borders.y * 0.5f, borders.y * 0.5f);

                        randPos.z =
                         Random.Range(-borders.z * 0.5f, borders.z * 0.5f);

                        //Place the piece at the random position
                        piece.transform.position =
                              transform.position + randPos;
```

```

                Vector3 explosionPos = transform.position;
                float explosionRad = transform.localScale.magnitude;

                piece.rigidbody.AddExplosionForce(explosionPower,
                                                  explosionPos,
                                                  explosionRad);

                //If life time is spiecified,
                //destroy the piece after it
                if(pieceLifetime > 0){
                    Destroy(piece, pieceLifetime);
                }
            }
        }

        //Inform other scripts that the object has been broken
        SendMessage("OnBreak", SendMessageOptions.DontRequireReceiver);

        //Finally, destroy the breakable
        Destroy(gameObject);

    }
}
```

Listing 59: A script to make an object breakable


360°
thinking.
Deloitte.
Discover the truth at www.deloitte.ca/careers
© Deloitte & Touche LLP and affiliated entities.

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

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

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

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

```
using UnityEngine;
using System.Collections;

public class BreakOnDestruct : MonoBehaviour {

     void Start () {

     }

     void Update () {

     }

     //Simply receive destruction message
     //and send break message
     void OnDestruction(){
          SendMessage("Break", SendMessageOptions.DontRequireReceiver);
     }
}
```

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

Exercises

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

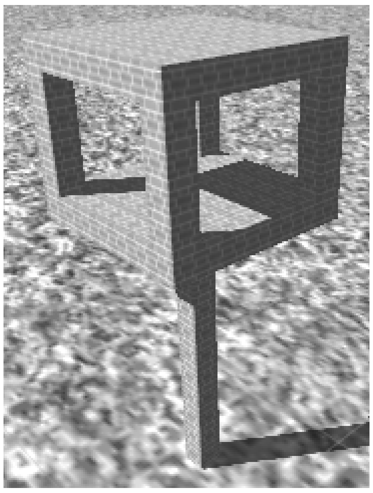


Illustration 75: The tower of Exercise 6

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