# Стрельба и разрушение объектов

Для того, чтобы реализовать стрельбу какими-то объектами (например, пульками или огненными шарами), необходимо создать и настроить два вида объектов – сами снаряды, и разрушаемые объекты. Более того, нам ещё понадобится создать заготовку для «осколков», когда вместо разрушаемого объекта будет создаваться массив разлетающихся осколков.

Для иллюстрации работы у нас уже должен быть создан проект с каким-нибудь объектом в качестве «цели», а также игрок с возможностью перемещения по сцене и управления направлением взгляда с помощью мыши.

## Создание заготовки «осколка»

Осколком можно сделать простой объект, например, кубик Cube. Создаём один куб: GameObject – 3D Object – Cube, переименовываем в Shard, настраиваем размеры (должны быть относительно небольшими, чтобы один разрушаемый объект по размеру равнялся примерно сотне осколков). Положение на сцене не важно – всё равно потом будем явно указывать при создании осколков.

Создаём для осколка «Материал»: на панели проекта ПКМ – Create – Material, сразу называем Blood (или как-то поудобней), меняем цвет на красный, да и вообще экспериментируем с настройками цвета. Материал перетаскиваем на объект Shard прямо на сцене, должен изменить цвет.

Самое главное! Чтобы наши осколки двигались более-менее реалистично, надо добавить им компонент «физическое тело» – Rigidbody. Иными словами, мы как бы говорим Unity, что наши осколки обладают свойствами физического тела, и для них нужно выполнять расчёты движения под действием силы тяжести, столкновений и тому подобного. Для этого выбираем Shard, и на панели свойств добавляем компонент Rigidbody (справа на панели): AddComponent – Physics – Rigidbody. Можно попробовать сбросить галочку «Use Gravity» (тогда на объект не будет действовать гравитация).

## Таймер исчезновения объекта

Нам нужно, чтобы осколки автоматически уничтожались через некоторое время, причём время жизни не должно быть у них одинаковым, а случайно выбирается из некоторого диапазона. В этом нам поможет простой скрипт, который при создании объекта устанавливает случайное время жизни для объекта, и по прошествии этого периода автоматически уничтожает объект. Создаём новый скрипт, называем его Suicide, и модифицируем его вот таким образом:

using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class Suicide : MonoBehaviour {  
  
    *//  Переменные для установления нижней и верхней границ*  
    [SerializeField] private float shortestLife = 3.0f;  
    [SerializeField] private float longestLife = 10.0f;  
  
    *//  Время исчезновения объекта*  
    private float endOfLife = 0f;  
    *// Вызывается один раз при создании объекта*  
    void Start () {  
        endOfLife = Time.time + Random.Range (shortestLife, longestLife);  
    }  
      
    *// Вызывается для каждого кадра*  
    void Update () {  
        if (Time.time > endOfLife)  
            Destroy (this.gameObject);  
    }

}

Этот скрипт нужно добавить на «осколок» Shard, и мы использовали атрибут SerializeField для того, чтобы верхнюю и нижнюю границу времени жизни можно было устанавливать прямо в редакторе свойств, а не только в коде.

## Создание шаблона «осколка»

Замечание: для того, чтобы наложить на осколки текстуру, можно найти картинку текстуры (например, в Интернете найти бесшовную текстуру кирпичной стены или какого-нибудь камня), и желательно создать отдельные заготовки материала для осколков и для разрушаемого объекта. Добавление файла текстуры в проект выполняется элементарно: нужный файл можно просто разместить в подпапке Assets папки проекта, она тогда появится в проекте автоматически, и её можно перетянуть на нужный объект.

Теперь заготовка Shard готова к тому, чтобы её превратить в шаблон – Prefab. Просто перетащите объект с панели объектов на панель Assets проекта, и переименуйте в ShardPrefab. После этого с панели объектов его можно удалить (исчезнет со сцены, но останется в Assets).

## Тестирование разрушения объектов

Нам нужно проверить работу осколков – как они создаются, движутся и исчезают, а также уничтожение «родительского» объекта. Такие действия мы планируем реализовать как реакцию на попадание другого объекта – «снаряда», однако пока что попробуем для тестирования просто разрушать какой-то объект по щелчку мыши. Для этого на сцене у нас должен находиться кубик (Cube), который будем разрушать. Обратите внимание на его размеры – они должны быть больше размера осколка в несколько раз.

Создаём скрипт CrushableObject, который будет заставлять разрушаться родительский объект. Этот скрипт «навешивается» на любой объект и приводит к его разрушению, но нужно уметь указывать класс осколков, объекты которого порождаются при разрушении исходного объекта – нам потребуется сериализуемое поле объекта в скрипте.

using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class CrushableObject : MonoBehaviour {  
  
    [SerializeField] private GameObject shardPrefab = null;  
    [SerializeField] public float veloc = 8.0f;  
  
    *// Use this for initialization*  
    void Start () {  
    }  
  
    *// Update is called once per frame*  
    void Update () {  
        *//  Если нажата кнопка мыши, то уничтожаем с разлётом осколков*  
        if (Input.GetMouseButtonDown (0)) Die();  
    }  
  
    private void Die() {  
        *//  Вывод отладочной информации - можно и отключить*  
        Debug.Log ("I'm crushed!");  
        *//  Порождаем массив осколков до того, как уничтожим тело*  
        *//  это добавит осколкам красивое ускорение, т.к. они создаются внутри другого тела*  
        for(int k = -3; k <= 3; ++k)  
            for(int j = -3; j <= 3; ++j)  
                for (int i = -3; i <= 3; ++i) {  
                    var piece = Instantiate (shardPrefab) as GameObject;  
                    var rig = piece.GetComponent<Rigidbody> ();  
                    if (rig != null)  
                        rig.velocity = new Vector3 (0, veloc, 0);  
                    piece.transform.position = new Vector3 (transform.position.x + 0.2f \* i, transform.position.y + 0.2f \* j + 2.0f, transform.position.z + 0.2f \* k);  
                }  
        Destroy (this.gameObject);  
    }  
}

Обсудите этот скрипт с преподавателем, подумайте, как и где создаются осколки, как увеличить их количество (только не перестарайтесь, это сильно влияет на производительность).

Этот скрипт пока что бесполезен – ему надо указать вид объектов (класс) для создания осколков. Внутри скрипта порождаемые объекты называются shardPrefab, и имеют тип GameObject – то есть фрагментами могут быть любые виды игровых объектов, нам же надо указать, что это именно ShardPrefab, который мы создали раньше. Для этого закройте редактор кода, выберите скрипт CrushableObject, и в панели его свойств найдите поле Shard Prefab со значением None, перетащите туда заготовку осколка (шаблон ShardPrefab). Теперь можно запустить и протестировать анимацию разрушения объектов.

Обратите внимание на очень важную особенность: осколки создаются с перекрытием! Данный скрипт тестировался с кубиками размера 0.3x0.3x0.3, а создаются они с шагом 0.2 по всем координатам. Это приводит к тому, что осколки отталкиваются друг от друга, так как обладают свойством Rigidbody и «не любят» взаимопересечений. Также в этом сценарии сначала создаются осколки, а потом уничтожается исходное тело – это делается «одномоментно» (в одном вызове функции) и особого эффекта не имеет, однако в качестве упражнения предлагается попробовать уничтожать исходный объект не «сразу», а спустя несколько мгновений (например, 0.2 с) – так, как мы сделали это с осколками в скрипте Suicide, только отсчёт времени вести не от создания объекта, а от нажатия на кнопку мыши. Изменится ли что-то?

Задание: приведите в порядок этот скрипт – сделайте константы полями класса, поэкспериментируйте с количеством объектов их начальным положением, а также начальной скоростью осколков.

В результате у нас получился скрипт CrushableObject, который можно добавлять к любым объектам – они будут разрушаться, а вместо них будет создаваться красивое множество осколков, исчезающих со временем.

## Создание шаблона снаряда

Просто так разрушать объекты не интересно – добавим разрушение при попадании «снаряда». Для этого сначала нужно реализовать шаблон снаряда, и предусмотреть стрельбу по щелчку мыши. Есть два различных подхода, с помощью которых можно реализовать движение снаряда. Во-первых, можно добавить к нему компонент Rigidbody, установить начальную скорость, а остальное система сделает сама с помощью физического «движка». Второй вариант – это управление движением снаряда «самостоятельно», но это потребует написания скрипта и работы с переменными. В принципе, оба варианта имеют свои плюсы и минусы, первый немного проще и удобнее для создания объектов, которые «отскакивают» от различных предметов.

Создайте заготовку – это может быть капсула, или сфера, или любой другой простой объект, настройте материал, не забудьте добавить к нему компонент Rigidbody, после этого превратите объект в шаблон (prefab) с названием bulletPrefab.

Теперь нужно научиться «стрелять» этими объектами. Идея довольно проста: надо создавать такой объект при щелчке мыши, и задавать для него направление и скорость движения, остальное физический движок Unity сделает самостоятельно. Так как мы хотим стрелять в направлении взгляда игрока, то создавать объект нужно в том скрипте, в котором доступна информация о направлении взгляда игрока – в нашем случае это основная камера, прикреплённая к игроку (у нас на сцене только камера «знает» своё направление взгляда). Для этого нужно либо создать новый скрипт и привязать его к камере, либо модифицировать любой существующий скрипт для камеры. Откроем скрипт Spin и изменим его таким образом:

using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class Spin : MonoBehaviour {  
  
    [SerializeField] public GameObject bulletObj = null;  
  
    public enum RotationAxes {  
        MouseXAndY = 0,  
        MouseX = 1,  
        MouseY = 2  
    }  
    public RotationAxes axes = RotationAxes.MouseXAndY;  
  
    public float sensitivityVert = 9.0f;  
    public float minimumVert = -45.0f;  
    public float maximumVert = 45.0f;  
    private float \_rotationX = 0;  
  
    *// Use this for initialization*  
    void Start () {  
          
    }  
  
    void OnGUI() {  
        int size = 32;  
        Camera me = GetComponent<Camera>();

        float posX = me.pixelWidth / 2 - size / 4;  
        float posY = me.pixelHeight / 2 - size / 2;  
        GUI.Label (new Rect (posX, posY, size, size), "\*");  
    }  
  
    *// Update is called once per frame*  
    void Update () {  
        \_rotationX -= Input.GetAxis ("Mouse Y") \* sensitivityVert;  
        \_rotationX = Mathf.Clamp (\_rotationX, minimumVert, maximumVert);  
        transform.localEulerAngles = new Vector3 (\_rotationX, 0, 0);  
  
        if (Input.GetMouseButtonDown (0)) {  
            var  poleno = Instantiate (bulletObj) as GameObject;  
            poleno.transform.position = transform.TransformPoint (Vector3.forward \* 1.5f);  
            var rig = poleno.GetComponent<Rigidbody> ();  
            rig.velocity = transform.TransformVector (Vector3.forward \* 25f);  
            rig.angularVelocity = new Vector3 (5f, 0f, 0f);  
        }  
  
    }  
}

Здесь точно так же вводится сериализуемое поле bulletObj, содержащее информацию о типе снаряда:

[SerializeField] public GameObject bulletObj = null;

После этого вносим изменения в метод Update(): проверяем, была ли нажата левая кнопка мыши, и если так, то создаём экземпляр шаблона bulletObj с именем «poleno». Обратите внимание, мы пока что не указали скрипту конкретный шаблон создаваемых объектов (планируем BulletPrefab), но это можно сделать и позже. Далее указываем положение объекта в пространстве. В рамках скрипта мы работаем с относительной координатной системой камеры, а нам надо указать положение снаряда в глобальной системе координат, для этого воспользуемся методом transform.TransformPoint (Vector3.forward \* 1.5f) – как бы «отступаем» немного вперёд от камеры, и координаты полученной точки превращаем в глобальные. Осталось указать начальную скорость, но это сложнее – у объектов нет такого свойства, оно есть только у компонента Rigidbody нашего снаряда, поэтому сначала нужно «найти» этот компонент с помощью метода GetComponent<Rigidbody>, а уже потом указывать для него скорость – velocity, причём вектор скорость тоже надо преобразовывать из локальной в глобальную систему координат. Можно добавить угловую скорость (будет небольшое вращение объекта). Фрагмент создания и настройки снаряда выглядит следующим образом:

if (Input.GetMouseButtonDown (0)) {  
            var  poleno = Instantiate (bulletObj) as GameObject;  
            poleno.transform.position = transform.TransformPoint (Vector3.forward \* 1.5f);  
            var rig = poleno.GetComponent<Rigidbody> ();  
            rig.velocity = transform.TransformVector (Vector3.forward \* 25f);  
            rig.angularVelocity = new Vector3 (5f, 0f, 0f);  
        }

После того, как всё это сделали, можно вернутся в среду разработки Unity, и перетащить шаблон BulletPrefab на поле Bullet Obj скрипта камеры (выберите камеру на панели слева в иерархии объектов, после чего найдите на правой панели скрипт Spin, у него будет поле Bullet Obf, и на это поле перетащите шаблон BulletPrefab).

Можно попробовать запустить и посмотреть, как работает стрельба снарядами. При этом разрушение кубика может нам мешать, поэтому его можно отключить – просто закомментируйте реакцию на нажатие кнопки мышки в скрипте CrushableObject:

*// Update is called once per frame*  
    void Update () {  
        *//  Если нажата кнопка мыши, то уничтожаем с разлётом осколков*  
*//if (Input.GetMouseButtonDown (0)) Die();*    }

Снаряды летают и всё хорошо, только цель теперь не разрушается.

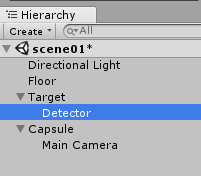
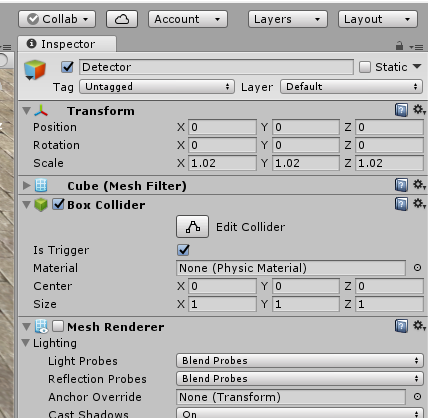
## Разрушение цели снарядом

Мы писали скрипт, который разрушает куб-цель, но только по щелчку мыши, теперь же нам надо сделать так, чтобы это происходило при попадании снаряда. Для этого воспользуемся механизмом триггеров – это аналог событий, связанный с компонентом Collider, и срабатывает такое событие при определённых условиях, как правило – при вхождении в область «коллайдера» другого объекта, имеющего компонент Collider.

Если некоторый 3D-объект сцены имеет в своём составе компонент Collider, то этот компонент можно сделать триггером (галочка Is Trigger в панели свойств компонента Collider). В этом случае данный компонент начинает реагировать на столкновения с другими 3D-объектами, тоже имеющими в своём составе компонент Collider, причём для других объектов не нужно включать флажок Is Trigger. Точнее, если внутри области коллайдера-триггера находится другой объект-коллайдер, то генерируются соответствующие события (нам нужно событие OnTriggerEnter коллайдера-триггера), и в обработчике таких событий можно реагировать на столкновение с другими объектами.

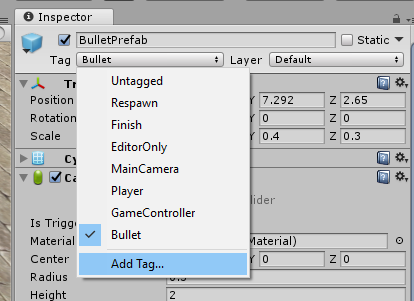
Мы сделаем куб-цель коллайдером с регистрацией столкновений, а снаряды у нас и так уже имеют компонент Collider, и напишем скрипт для разрушения объекта при столкновении. Есть только одна существенная проблема: если коллайдер становится триггером (то есть начинает реагировать на другие физические объекты, попавшие в его область), то Unity для него перестаёт обрабатывать столкновения – другие «физические» объекты не будут от него отскакивать, а будут проходить сквозь, вызывая срабатывание триггера. Это не страшно, но как только мы запустим такую игру – цель просто провалится сквозь пол! Поэтому попробуем сделать всё аккуратно – так, чтобы объект и сохранил физические свойства, и реагировал на столкновения с другими объектами, вызывая срабатывание триггера.

Для этого добавим к цели (напоминаю, что у нас это куб) дочерний объект-куб, который будет выполнять роль детектора столкновений (а куб-родитель будет выполнять роль физического объекта-цели), переименовываем его в . Создаём новый 3D-объект на сцене (он должен иметь такую же форму, как и родитель – куб), и перетаскиваем его на панели объектов на цель, делая дочерним. Настраиваем размеры – дочерний должен быть немного больше родительского, чтобы выступать за границы, поэтому в примере для куба указаны размеры 1.02 по всем координатным осям. Включаем флажок Is Trigger, и выключаем Mesh Rendering (теперь объект будет невидимым):

Теперь осталось только написать скрипт, который будет реагировать на вхождение в область Detector другого объекта, обладающего компонентом Collider. И тут возникнет очередная проблема: мы хотим реагировать на попадания только снарядов, и игнорировать другие объекты (например, пол или осколки). Реализовать это можно довольно просто: к «снарядам» добавим тег (это строка, связанная с объектом) «Bullet», а цель научим разрушаться под воздействием любого объекта с тегом Bullet.

Указываем у шаблона (префаба) BulletPrefab тег Bullet – нужно добавить его в список тегов:



После этого создаём скрипт CollisionDetect с очень простым содержимым:

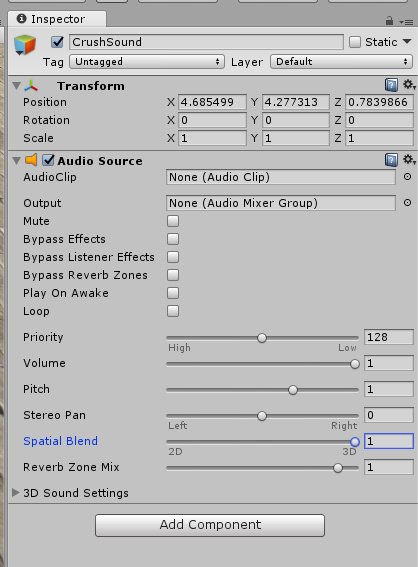
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class CollisionDetect : MonoBehaviour {  
  
    *// Use this for initialization*  
    void Start () {  
    }  
      
    *// Update is called once per frame*  
    void Update () {  
    }  
  
    void OnTriggerEnter(Collider other) {  
        if(other.tag == "Bullet")  
            transform.parent.SendMessage("Die");  
    }  
}

Здесь написан только метод OnTriggerEnter, который будет автоматически запускаться при вхождении в область триггера объекта. Мы проверяем, имеет ли объект тег «Bullet», и если это так, то отправляем сообщение нашему родительскому объекту с запросом на вызов метода Die – этот скрипт мы привяжем к объекту-триггеру, для которого куб-цель является родителем, и разрушать надо именно родителя. Скрипт CollisionDetect нужно добавить к коллайдеру Detector, и удостовериться, что у цели есть метод Die (мы его писали). Вот и всё, проверяем – ура, работает!

# Добавление звука разрушения объекта

Для озвучивания разрушения объекта нам нужен источник звука – объект AudioSource на сцене. Попробуем пока что обойтись одним таким источником (теоретически можно для разных объектов создавать различные источники звука). Нужно найти какой-нибудь файл со звуком выстрела в интернете, и сохранить его в папку проекта – откройте Проводник, перейдите в папку проекта и создайте в папке Assets подпапку Audio, сохраните туда нужный mp3-файл. Вернитесь в Unity – папка появится на панели Assets.

Добавьте на сцену объект AudioSource: меню GameObject – Audio – AudioSource. Переименуйте в CrushSound, положение настраивать не обязательно – потом в скрипте явно зададим. Самое главное – снять галочку Play On Awake (чтобы отключить проигрывание автоматически при создании объекта), а также настроить глубину 3D звучания ползунком Spatial Blend (по умолчанию звук не пространственный, а просто воспроизводится через колонки как 2D):



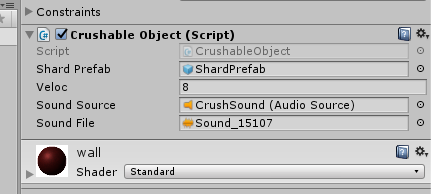
Теперь откройте скрипт CrushableObject, и добавьте описание двух сериализуемых полей:

    [SerializeField] private AudioSource soundSource = null;  
    [SerializeField] private AudioClip soundFile = null;

Внутри метода Die добавьте такие строки:

        if (soundSource != null) {  
            soundSource.transform.position = transform.TransformPoint (Vector3.zero);  
            soundSource.PlayOneShot (soundFile);  
        }

Здесь мы переносим источник звука в центр разрушаемого объекта, после чего воспроизводим звуковой файл. Сохраните скрипт, после чего выберите объект Target, найдите на панели его свойств поля скрипта CrusableObject, и укажите в них источник звука CrushSound (просто перетащите с панели на это поле), а также файл взрыва (нажмите на точку рядом с полем и выберите нужный файл).



Всё готово, можно тестировать!