Dragon Picker #Часть 2
создание игрового прототипа

Часть 1. Описание игры

Вспомните основную идею игры, которую мы разрабатываем. См. презентацию.

Часть 2. Основные игровые объекты

В первой части были добавлены базовые игровые объекты. Откройте проект, выполненный в предыдущей работе.

Часть 3. Программирование игровых объектов

Введение

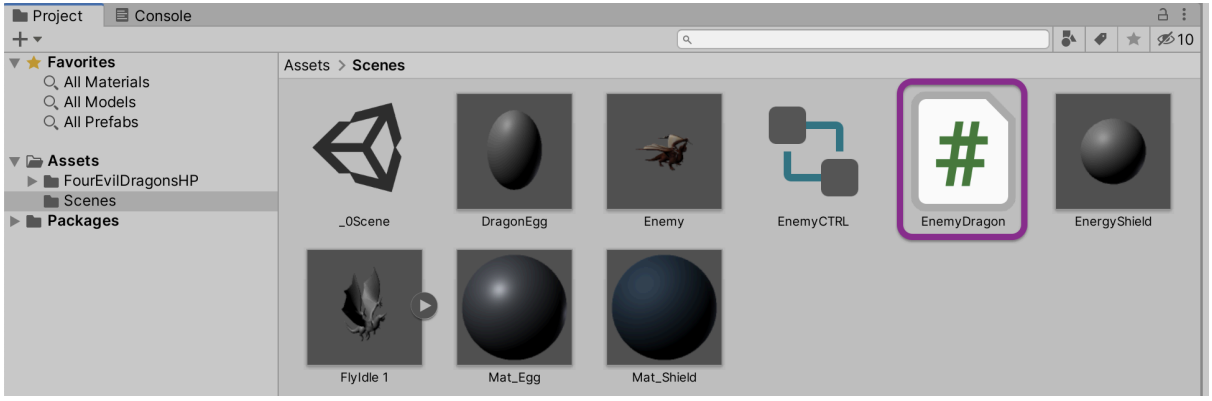
При написании сценариев мы будем пошагово добавлять строки кода в скрипт-файлы, давая описание команд по ходу создания нового функционала. Как уже было отмечено ранее, действия игровых объектов описываются в так называемых Script-файлах. Эти файлы содержат код на языке C# и имеют расширение .cs. При этом следует отметить, что предварительная компиляция файлов не требуется, Unity берет эту работу на себя в режиме реального времени в процессе работы в среде разработки.

3.1 Скрипт-файл EnemyDragon

В пункте 3.1 мы создадим скрипт-файл, который будет выполнять следующие действия:

- перемещать игрового персонажа Enemy влево и вправо;
- через случайные интервалы времени вероятность изменения направления движения будет изменяться;
- враг Enemy будет сбрасывать яйцо (DragonEgg) через случайные интервалы времени.

1. Создайте сценарий с именем EnemyDragon.cs в папке Assets - Scenes. Для этого кликните правой кнопкой мыши внутри содержимого окна Scene (так же как мы ранее создавали материалы и контроллер анимации) и из выпадающего меню выберите Create - C# Script, дайте имя файлу EnemyDragon:



2. Откройте сценарий EnemyDragon.cs, кликнув по нему два раза (файл автоматически откроется в среде разработки Microsoft Visual Studio) и введите следующий код:

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class EnemyDragon : MonoBehaviour
```

```
{
    [Header("Set in Inspector")]
    public GameObject dragonEggPrefab;
    public float speed = 1f;
    public float timeBetweenEggDrops = 1f;
    public float leftRightDistance = 10f;
    public float chanceDirections = 0.1f;
    void Start()
    {
    }
    void Update()
    {
    }
}
```

// End Code

Листинг дублируется ниже в виде скриншота из MS Visual Studio.

EnemyDragon.cs

No selection

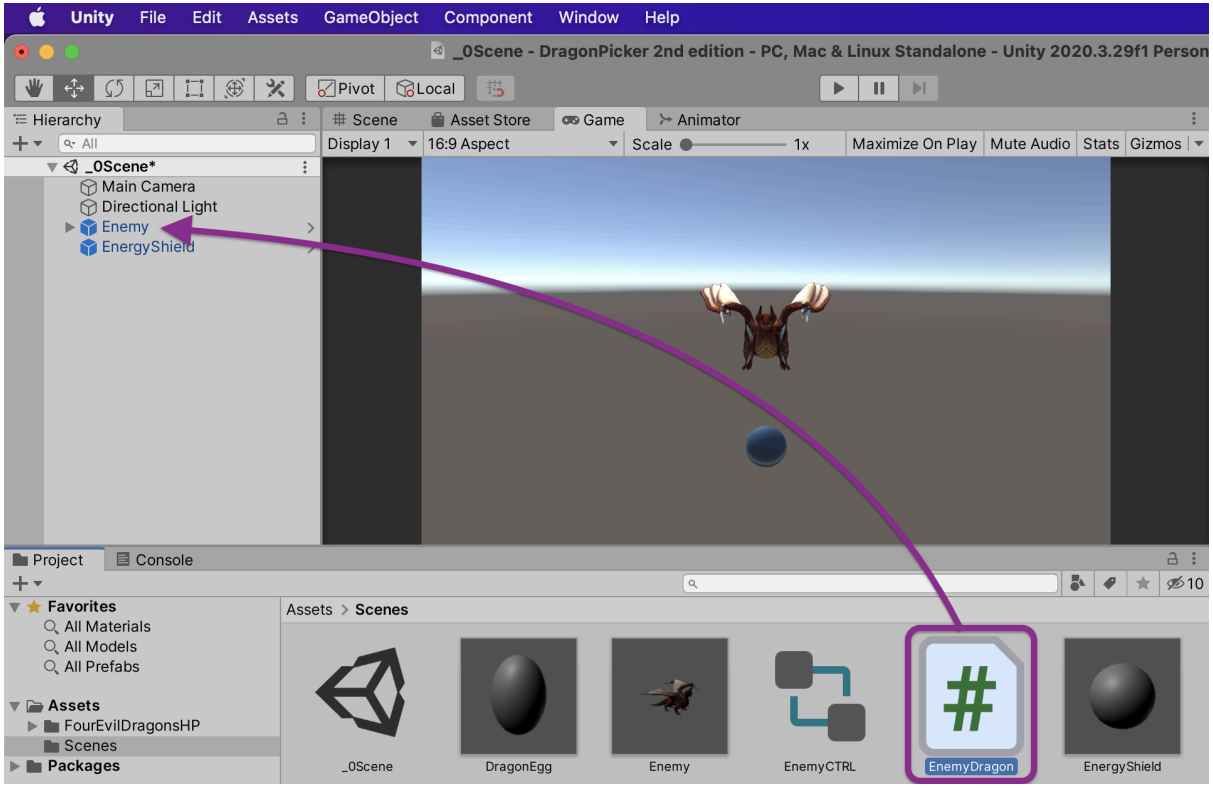
```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class EnemyDragon : MonoBehaviour
6   {
7       [Header("Set in Inspector")]
8       public GameObject dragonEggPrefab;
9       public float speed = 1f;
10      public float timeBetweenEggDrops = 1f;
11      public float leftRightDistance = 10f;
12      public float chanceDirections = 0.1f;
13
14      void Start()
15      {
16      }
17
18      void Update()
19      {
20      }
21  }
22
```

В листинге мы добавили следующие строки кода:
- с помощью команды [Header("Set in Inspector")] создализаголовок в окне Inspector;
- public GameObject dragonEggPrefab создает префаб для драконьего яйца DragonEgg;
– созданная переменная speed определяет скорость движения объекта DragonEgg;
– переменная timeBetweenEggDrops определяет скорость генерации объектов DragonEgg;

– переменная leftRightDistance определяет расстояние от края экрана, на котором меняется направление движения дракона;

- chanceDirections определяет вероятность изменения направления движения.

3. Подключите сценарий к игровому объекту Enemy. Сделать это можно так же, как и ранее - перетаскиванием скрипта EnemyDragon.cs, на объект Enemy:



4. Теперь, если выбрать объект Enemy в окне Hierarchy, то можно увидеть, что в инспекторе объекта (в правой части среды разработки) добавился компонент с соответствующим именем Enemy Dragon (Script). Обратите внимание, что компонент содержит поля, для которых мы создали переменные на

языке C#. Это значит, что в дальнейшем мы сможем изменять значения переменных прямо в этих полях, находясь в среде разработки Unity без лишнего обращения к коду.

5. Внутри скрипт-файла EnemyDragon.cs добавьте в метод Update следующие строки кода, выделенные жирным шрифтом:

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class EnemyDragon : MonoBehaviour
{
   [Header("Set in Inspector")]
   public GameObject dragonEggPrefab;
   public float speed = 1f;
   public float timeBetweenEggDrops = 1f;
   public float leftRightDistance = 10f;
   public float chanceDirections = 0.1f;
   void Start()
   {
   }
   void Update()
   {
      Vector3 pos = transform.position; //1
      pos.x += speed * Time.deltaTime; //2
      transform.position = pos; //3
   }
}
```

// End Code

Листинг дублируется ниже в виде скриншота из MS Visual Studio.

EnemyDragon.cs

No selection

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class EnemyDragon : MonoBehaviour
6   {
7       [Header("Set in Inspector")]
8       public GameObject dragonEggPrefab;
9       public float speed = 1f;
10      public float timeBetweenEggDrops = 1f;
11      public float leftRightDistance = 10f;
12      public float chanceDirections = 0.1f;
13
14      void Start()
15      {
16      }
17
18      void Update()
19      {
20          Vector3 pos = transform.position; //1
21          pos.x += speed * Time.deltaTime; //2
22          transform.position = pos; //3
23      }
24  }
25
```

Каждому закомментированному номеру соответствует следующее:

- Vector3 pos – это локальная переменная для хранения текущий позиции объекта (//1);

- x увеличивается на произведение скорости и временного интервала Time.deltaTime (//2). Это встроенная переменная, которая будет изменять свое значение при разной частоте кадров. Если говорить простыми словами, то она отвечает за плавность игры при разном FPS;

- изменение значения pos записывается обратно в transform.position (//3).

6. Сохраните сценарий и нажмите Play в Unity. Теперь дракон (игровой объект Enemy) должен медленно начать двигаться по экрану. Можете самостоятельно подобрать такое значение скорости Speed в окне Inspector, которое покажется вам наиболее подходящим.

7. Чтобы дракон не улетал за край экрана следует добавить в метод Update строки кода с условием, при наступлении которого, направление движения будет меняться:

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class EnemyDragon : MonoBehaviour
{
   [Header("Set in Inspector")]
   public GameObject dragonEggPrefab;
   public float speed = 1f;
   public float timeBetweenEggDrops = 1f;
   public float leftRightDistance = 10f;
   public float chanceDirections = 0.1f;
   void Start()
   {
   }
```

```
    void Update()
    {
        Vector3 pos = transform.position;
        pos.x += speed * Time.deltaTime;
        transform.position = pos;
        if (pos.x < -leftRightDistance) //1
        {
            speed = Mathf.Abs(speed);
        }
        else if (pos.x > leftRightDistance) //2
        {
            speed = -Mathf.Abs(speed);
        }
    }
}
// End Code
```

Листинг дублируется ниже в виде скриншота из MS Visual Studio.

EnemyDragon.cs

EnemyDragon ▸ No selection

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class EnemyDragon : MonoBehaviour
6   {
7       [Header("Set in Inspector")]
8       public GameObject dragonEggPrefab;
9       public float speed = 1f;
10      public float timeBetweenEggDrops = 1f;
11      public float leftRightDistance = 10f;
12      public float chanceDirections = 0.1f;
13
14      void Start()
15      {
16      }
17
18      void Update()
19      {
20          Vector3 pos = transform.position;
21          pos.x += speed * Time.deltaTime;
22          transform.position = pos;
23
24          if (pos.x < -leftRightDistance) //1
25          {
26              speed = Mathf.Abs(speed);
27          }
28          else if (pos.x > leftRightDistance) //2
29          {
30              speed = -Mathf.Abs(speed);
31          }
32      }
33  }
```

Каждому закомментированному номеру соответствует следующее:

- если значение pos.x оказалось меньше значения leftRightDistance - переменная speed имеет положительное значение (движение вправо) (//1);

- иначе если значение pos.x оказалось больше значения leftRightDistance - переменной speed присваивается отрицательное значение (движение влево) (//2).

8. Сохраните сценарий EnemyDragon.cs в Unity и нажмите Play. Проверьте корректность движения игрового объекта Enemy, согласно написанному сценарию. Теперь он должен менять направление движения при приближении к краям экрана.

9. Для добавления случайного изменения направления перемещения дракона, создайте под методом Update() новый метод с именем FixedUpdate(). FixedUpdate вызывается точно 50 раз в секунду и в отличие от Update() не зависит от быстродействия компьютера (а точнее – от частоты кадров, FPS).

// Start Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class EnemyDragon : MonoBehaviour
{
   [Header("Set in Inspector")]
   public GameObject dragonEggPrefab;
   public float speed = 1f;
   public float timeBetweenEggDrops = 1f;
   public float leftRightDistance = 10f;
   public float chanceDirections = 0.1f;
   void Start()
   {
   }
```

```
    void Update()
    {
        Vector3 pos = transform.position;
        pos.x += speed * Time.deltaTime;
        transform.position = pos;
        if (pos.x < -leftRightDistance)
        {
            speed = Mathf.Abs(speed);
        }
        else if (pos.x > leftRightDistance)
        {
            speed = -Mathf.Abs(speed);
        }
    }
    private void FixedUpdate()
    {
        if (Random.value < chanceDirections)
        {
            speed *= -1;
        }
    }
}
// End Code
```

Так как листинг достаточно длинный, то для удобства на скриншоте приводится лишь его нижняя часть, включая уже созданный ранее метод Update и новый метод FixedUpdate.

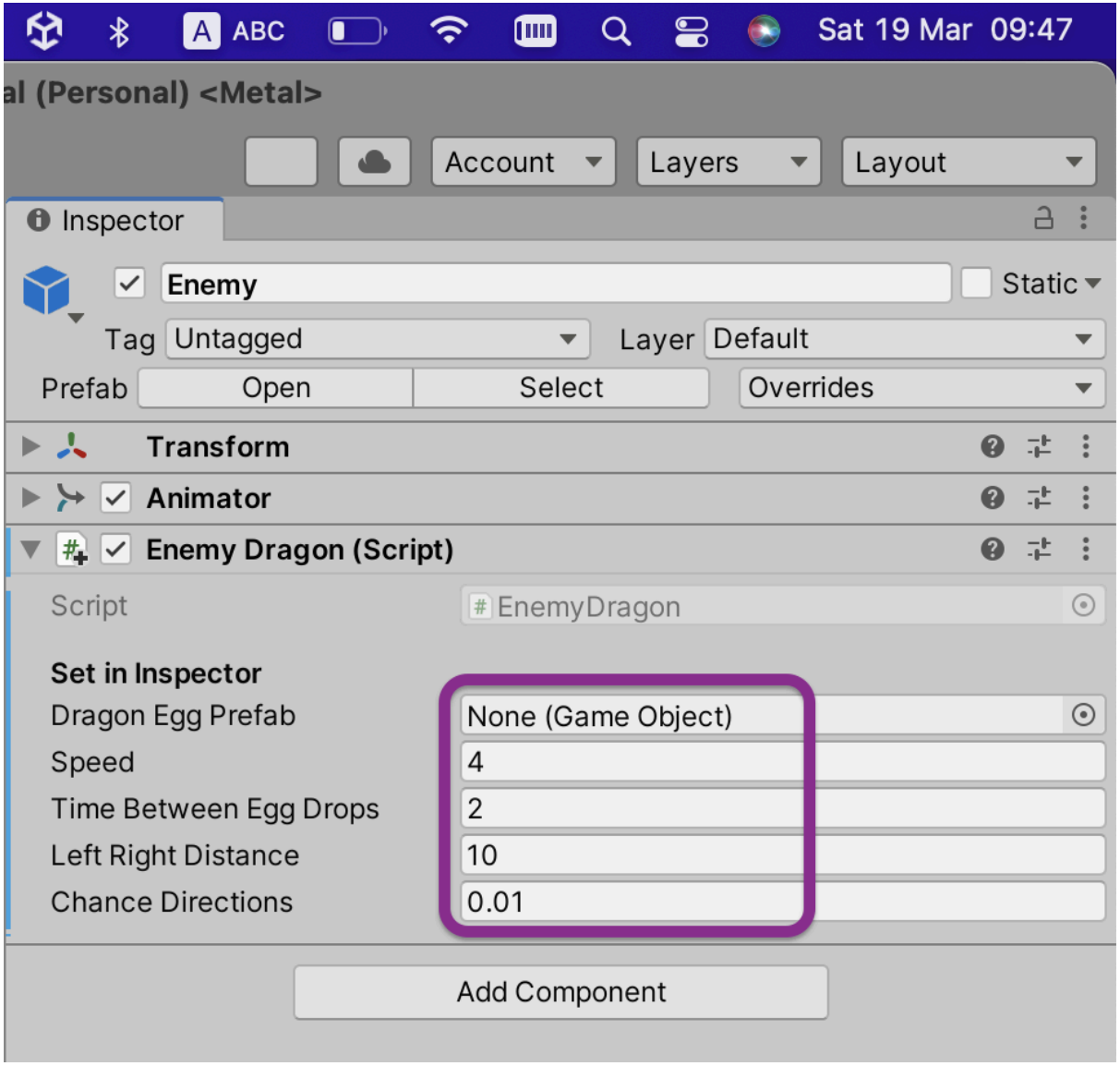
EnemyDragon.cs

EnemyDragon ▸ No selection

```
17
18      void Update()
19      {
20          Vector3 pos = transform.position;
21          pos.x += speed * Time.deltaTime;
22          transform.position = pos;
23
24          if (pos.x < -leftRightDistance)
25          {
26              speed = Mathf.Abs(speed);
27          }
28          else if (pos.x > leftRightDistance)
29          {
30              speed = -Mathf.Abs(speed);
31          }
32      }
33
34      private void FixedUpdate()
35      {
36          if (Random.value < chanceDirections)
37          {
38              speed *= -1;
39          }
40      }
41  }
```

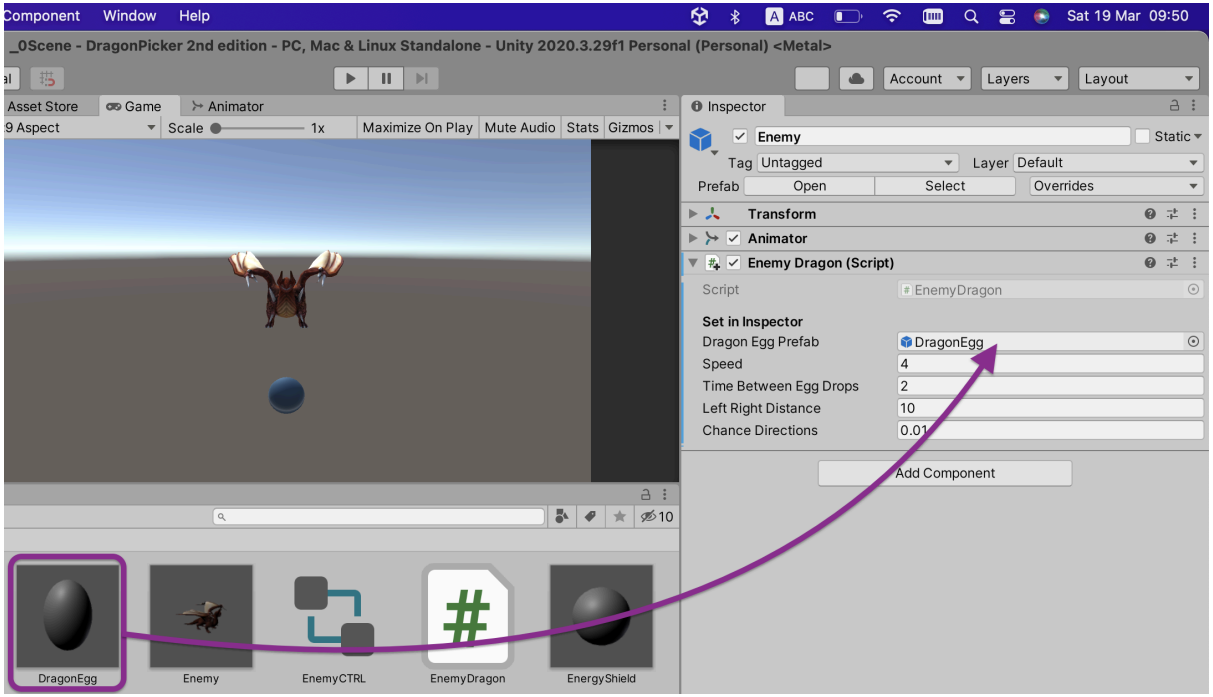
10. Попробуйте самостоятельно поэкспериментировать со значением переменных созданного скрипт-файла в окне инспектора. Например, подберите наиболее подходящее на ваш взгляд значение переменной chanceDirections таким образом, чтобы объект Enemy случайно менял направление в среднем 1 раз в секунду. Подобрать подходящее значение поможет формула: 50 вызовов FixedUpdate в секунду * вероятность = среднее число срабатываний в секунду. Напомним, что подбирать подходящие

значения можно в окне Inspector (сразу после сохранения скрипта). Мне, например, показались наиболее подходящими следующие значения переменных скрипт-файла:


Sat 19 Mar 09:47
al (Personal) <Metal>
Account
Layers
Layout
Inspector
Enemy
Static
Tag
Untagged
Layer
Default
Prefab
Open
Select
Overrides
Transform
Animator
Enemy Dragon (Script)
Script
EnemyDragon
Set in Inspector
Dragon Egg Prefab
None (Game Object)
Speed
4
Time Between Egg Drops
2
Left Right Distance
10
Chance Directions
0.01
Add Component

11. Запустите игру, нажав Run. Проверьте корректность срабатывания вероятности изменения направления движения дракона Enemy.

12. Далее мы реализуем сбрасывание объектов DragonEgg из дракона Enemy. Для этого выберите игровой объект Enemy и в окне Inspector найдите поле для добавления префаба (переменная -DragonEgg.prefab), и добавьте в него префаб игрового объекта DragonEgg. Это можно сделать также, как мы это делали ранее - перетаскиванием, либо нажав мишень и выбрать нужный из доступных префабов:



13. После того как префаб DragonEgg подключен, добавим необходимые строки кода для их генерации на сцене. Для этого в скрипт-файле нам нужно создать новый метод DropEgg() и команду вызова этого метода внутри метода Start(). Метод DropEgg будет создавать экземпляр драконьего яйца DragonEgg в точке нахождения дракона Enemy. Из метода Start будет запускаться генерация объекта-яйцо через заданное количество секунд:

// Start Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class EnemyDragon : MonoBehaviour
{
   [Header("Set in Inspector")]
   public GameObject dragonEggPrefab;
   public float speed = 1f;
   public float timeBetweenEggDrops = 1f;
   public float leftRightDistance = 10f;
   public float chanceDirections = 0.1f;
   void Start()
   {
      Invoke("DropEgg", 2f); // 1
   }
   void DropEgg() // 2
   {
      Vector3 myVector = new
      Vector3(0.0f, 5.0f, 0.0f);
      GameObject egg =
      Instantiate<GameObject>(dragonEggPrefab);
      egg.transform.position =
      transform.position + myVector;
      Invoke("DropEgg", timeBetweenEggDrops);
   }
```

```
    void Update()
    {
        Vector3 pos = transform.position;
        pos.x += speed * Time.deltaTime;
        transform.position = pos;
        if (pos.x < -leftRightDistance)
        {
            speed = Mathf.Abs(speed);
        }
        else if (pos.x > leftRightDistance)
        {
            speed = -Mathf.Abs(speed);
        }
    }
    private void FixedUpdate()
    {
        if (Random.value < chanceDirections)
        {
              speed *= -1;
        }
    }
}
```

// End Code

На скриншоте листинга показаны первые строки, включающие новые команды внутри методов Start() и DropEgg().

Также я бы хотел обратить ваше внимание на следующий момент: слишком длинные строки кода в листинге выше были разбиты на несколько строк (сделано это для обеспечения требований верстки

печатного издания). Эта особенность языка C#, которую вам полезно будет знать. В языке C# нет требований к тому, чтобы операторы располагались в одной строке. Пробелы и символы переноса строки игнорируются компилятором, поэтому мы и можем себе позволить перенос строк. В скриншоты листинга ниже мы приводим программный код без переноса строк.

EnemyDragon.cs

EnemyDragon ▸ Update()

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class EnemyDragon : MonoBehaviour
6   {
7       [Header("Set in Inspector")]
8       public GameObject dragonEggPrefab;
9       public float speed = 1f;
10      public float timeBetweenEggDrops = 1f;
11      public float leftRightDistance = 10f;
12      public float chanceDirections = 0.1f;
13
14      void Start()
15      {
16          Invoke("DropEgg", 2f); // 1
17      }
18
19      void DropEgg() // 2
20      {
21          Vector3 myVector = new Vector3(0.0f, 5.0f, 0.0f);
22          GameObject egg = Instantiate<GameObject>(dragonEggPrefab);
23          egg.transform.position = transform.position + myVector;
24          Invoke("DropEgg", timeBetweenEggDrops);
25      }
26
27      void Update()
28      {
```

1

2

- функция Invoke в методе Start() выше вызывает функцию, заданную именем через указанное число секунд. В данном случае - вызывается функция DropEgg(), с параметром 2f, т.е. с ожиданием 2 секунды перед каждым новым вызовом. Таким образом, мы можем установить наиболее подходящие

значения для частоты генерации объектов. Можете самостоятельно подобрать наиболее подходящее значение.

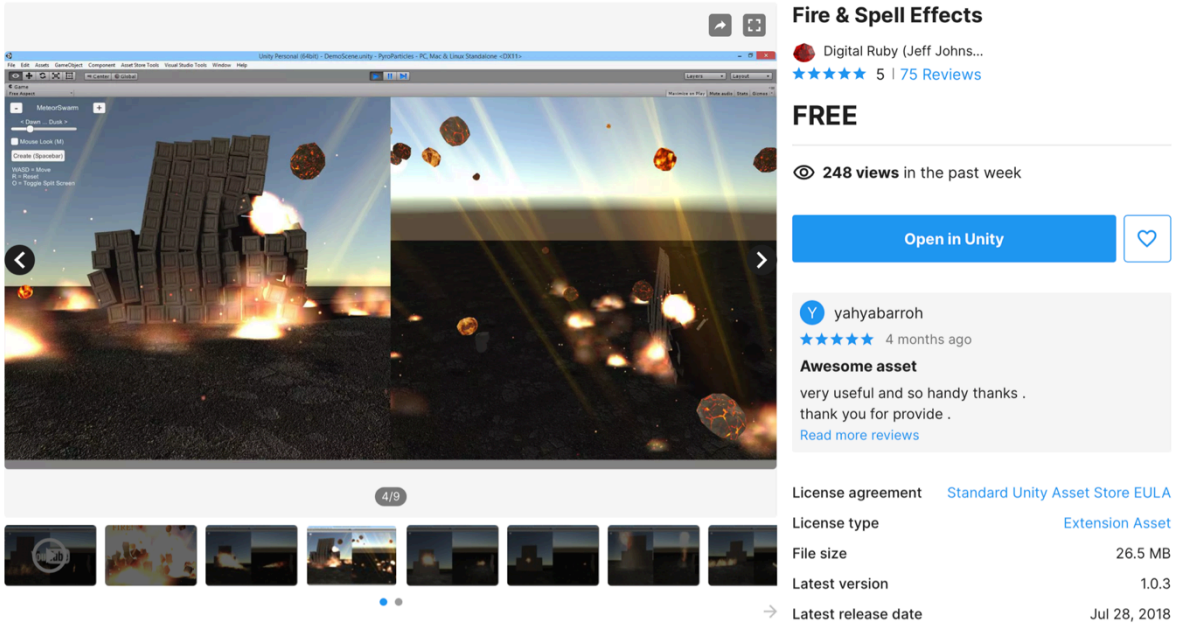
- функция DropEgg() создает экземпляр GameObject с именем dragonEggPrefab и присваивает его переменной egg. Далее изменяется положение egg, и функция вызывает саму себя снова через интервалы времени, равные timeBetweenEggDrops.

14. Теперь если нажать кнопку Play, то движущийся по экрану дракон будет сбрасывать драконье яйцо через равные интервалы времени.

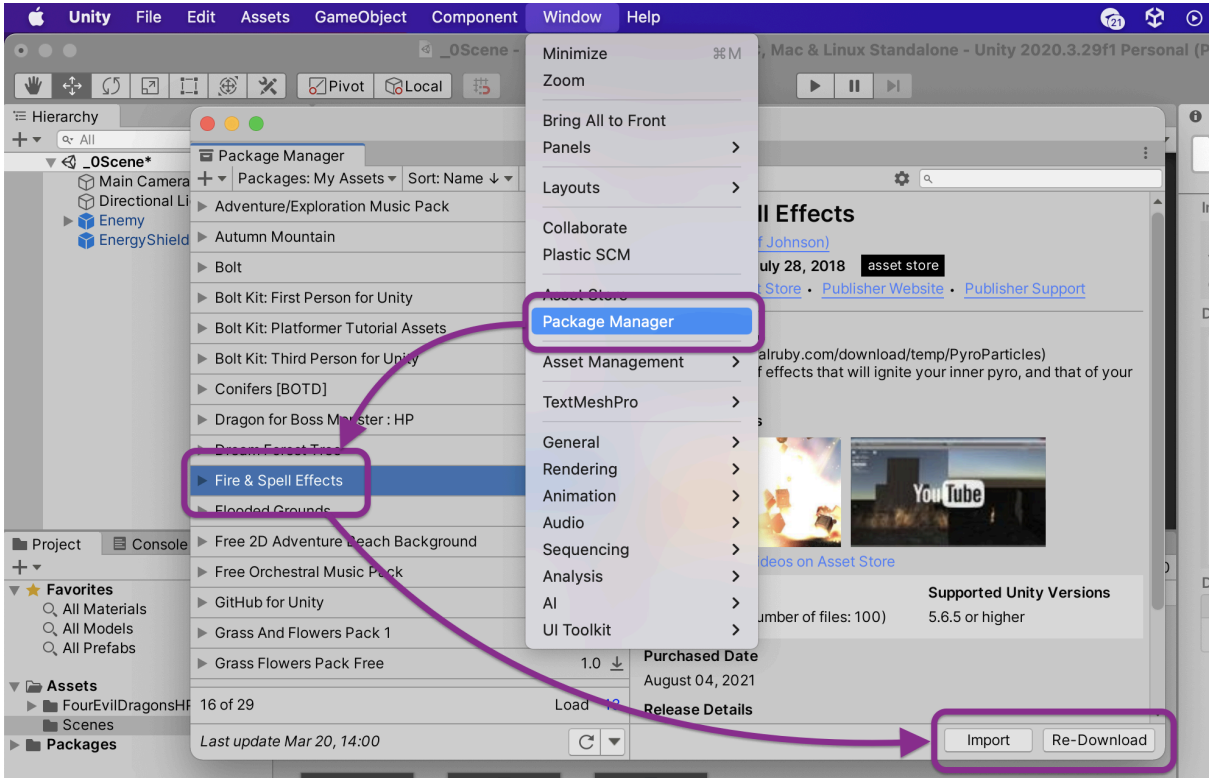
3.2 Скрипт-файл DragonEgg

В этом разделе мы опишем работу драконьего яйца. Перед тем как открыть файлы со сценариями и перейти к программированию, давайте несколько изменим оформление нашей сцены. В частности, добавим несколько игровых объектов и элементов визуального оформления. Мы сделаем это для того, чтобы нам стало более наглядно работать, к тому же в этом разделе мы добавим спецэффекты, а работать с ними конечно же эффективней, когда можно визуально оценить их работу.

1. Дополнительные визуальные эффекты мы возьмем из уже знакомого Asset Store. Зайдите в assetstore.unity.com и используя строку поиска найдите Asset с именем Fire & Spell Effects. Добавьте его в свой профиль и вернитесь в среду разработки Unity.



2. В среде разработки Unity откройте Window - Package Manager, в списке Package: My Assets найдите добавленный ассет с именем Fire & Spell Effects (если вы его не видите возможно потребуется обновить список, нажмите Refresh List в нижней части окна Package Manager). Далее нажмите Download и после этого – Import (и в новом всплывающем окне еще раз Import). Описанная последовательность действий изображена на скриншоте ниже:



3. После того как импорт будет завершен, в окне Project, в папке Assets/PyroParticles появится добавленный набор текстур и эффектов.

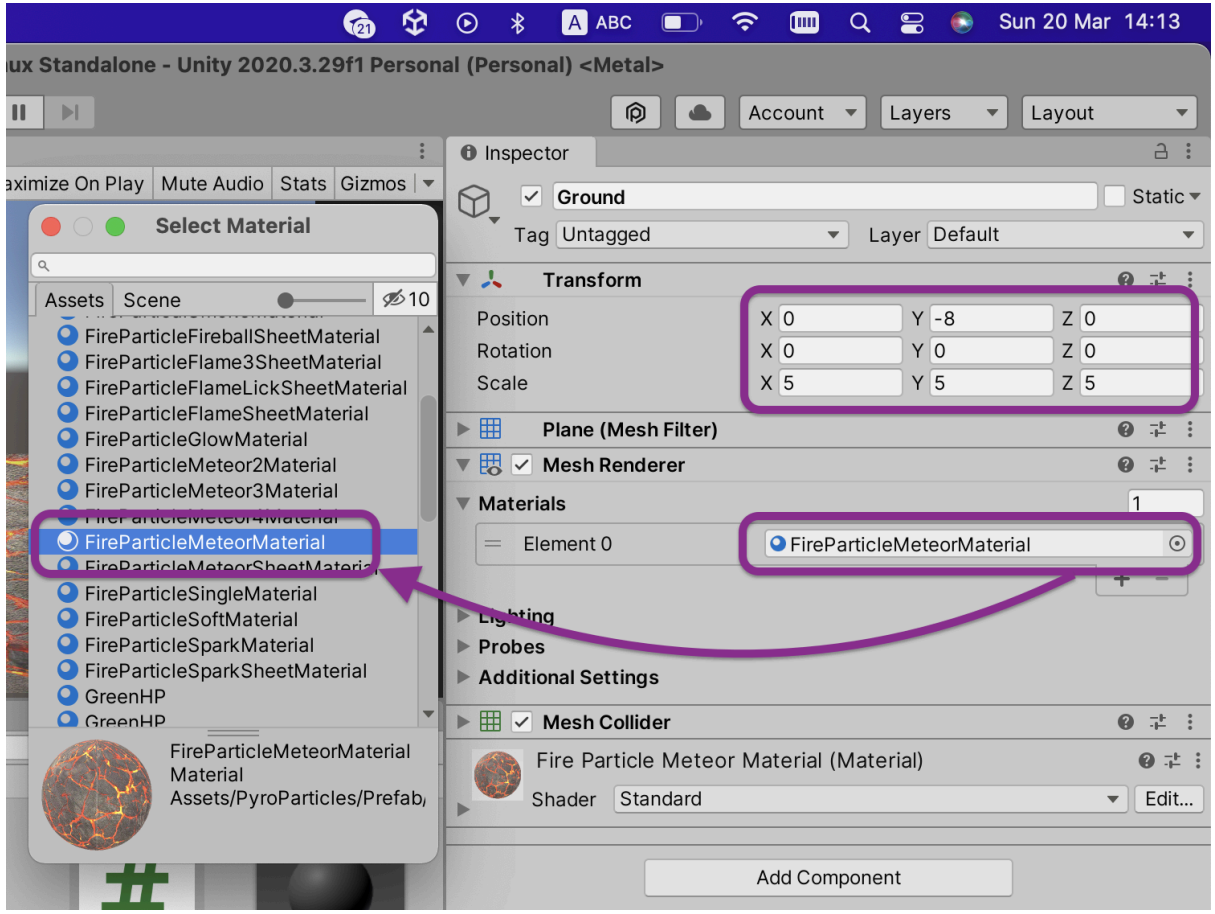
4. Далее мы добавим плоскость Plane, которая будет играть роль поверхности земли. Также мы оформим ее в виде раскаленной лавы, падая на которую яйцо дракона будут взрываться. Чтобы добавить плоскость на сцену, в верхнем меню выберите Windows - GameObject – Plane.

5. Переименуйте объект Plane в Ground. Установите следующие размеры и положение объекта Ground:

- Position: 0, -8, 0;
- Rotation: 0, 0, 0;

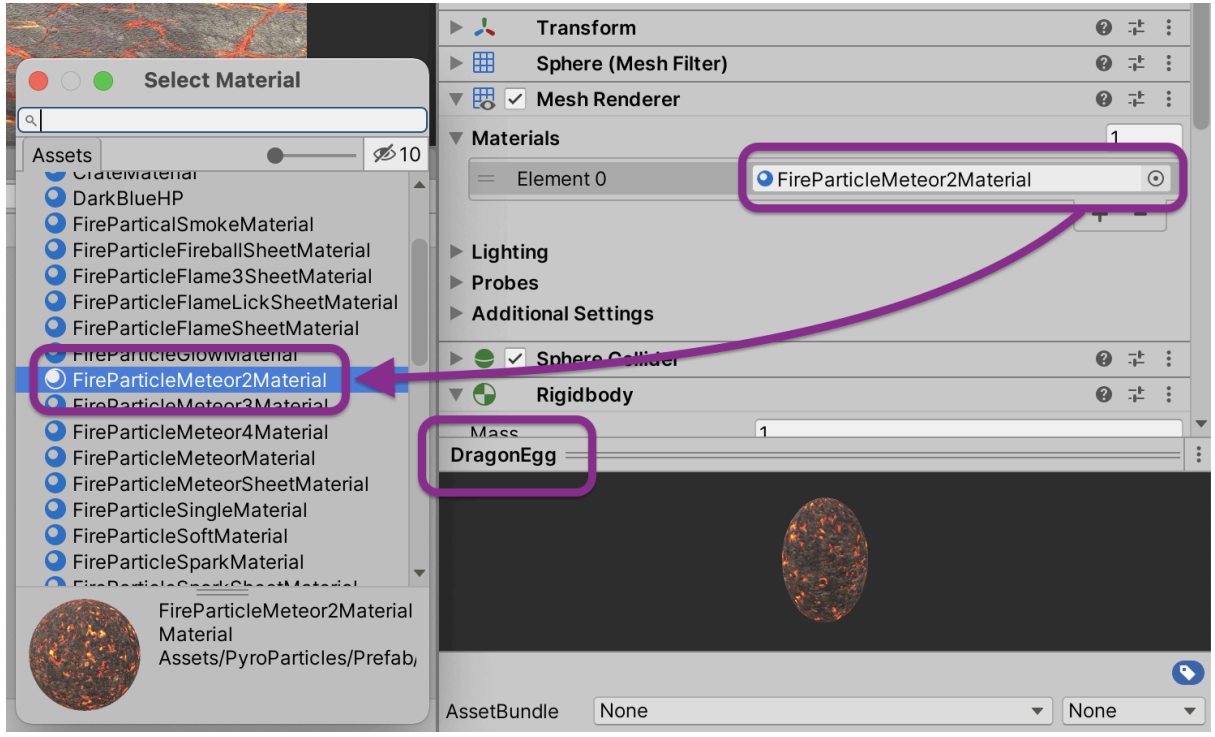
- Scale: 5, 5, 5;

6. Также изменим визуальное оформление элемента Ground. В окне Inspector (в правой части) разверните компонент Mesh Renderer (), нажмите на “мишень” и из появившегося меню Select Material выберите материал FireParticleMeteorMaterial.

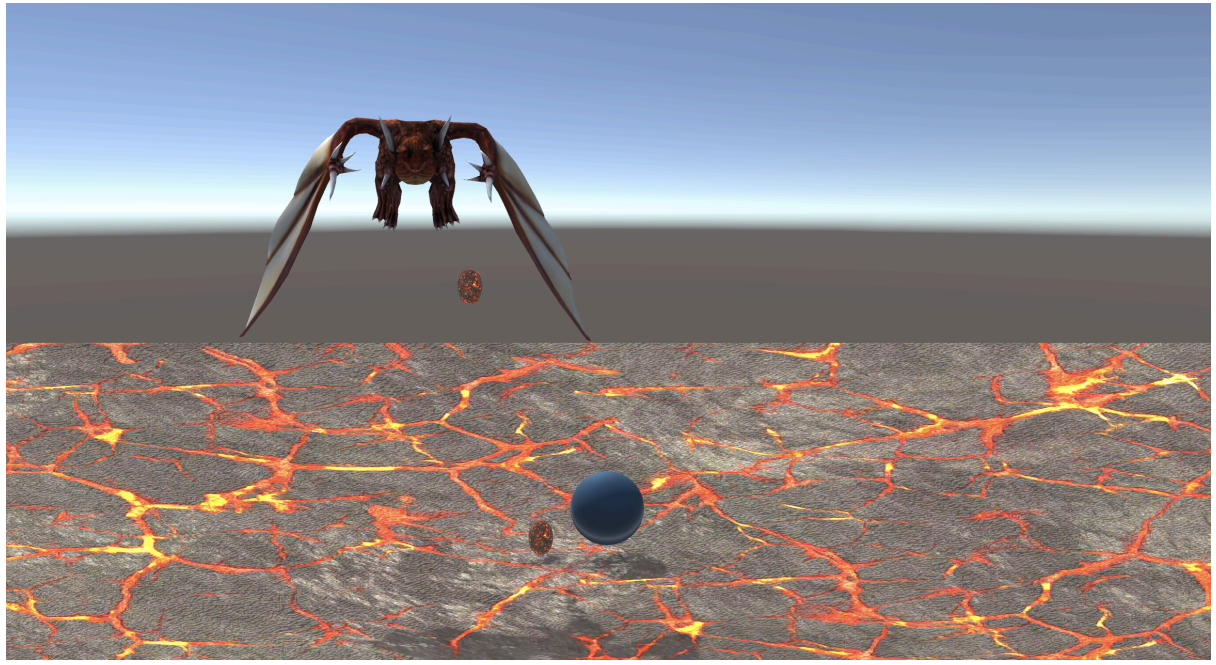


7. В качестве альтернативного варианта наложения текстуры на объект, можете найти нужную текстуру в папке скачанного ассет-пака и перетащить FireParticleMeteorMaterial в поле выбора материала, напротив надписи Element 0.

8. Текстуры из скачанного ассет-пака можно использовать и для других объектов. Например, вместо созданного вручную материала драконьего яйца Mat_Egg, можно также использовать готовую текстуру из Asset Store. Для этого выберите префаб DragonEgg из папки Scenes и внутри компонента MeshRenderer выберите подходящую текстуру, например FireParticleMeteor2Material:



9. Теперь можете запустить сцену и проверить ее работу. Если вы выполнили все пункты выше, то сцена должно выглядеть так, как изображено на рисунке ниже:



10. Рядом со скрипт-файлом EnemyDragon создайте новый сценарий (клик ПКМ – Create – C# Script) и назовите его DragonEgg.cs. Подключите сценарий DragonEgg.cs к игровому объекту DragonEgg.prefab. Этот игровой объект дракон сбрасывает через равные временные интервалы. Далее в скрипт-файл мы добавим следующий функционал:

- При столкновении с плоскостью Ground мы создадим эффект взрыва яйца. А именно, будет запускаться Particle System — это компонент, который может имитировать генерацию частиц разного вида (используется при добавлении взрывов, эффектов дыма, молний, тумана и т. д.). В нашем случае мы будем использовать Particle System для имитации взрыва.

- Генерируемые объекты DragonEgg все время существуют на сцене после создания. Со временем их количество будет увеличиваться, поэтому будет правильнее удалять объекты DragonEgg после того, как они будут уходить за пределы сцены.

11. Настройку визуальных эффектов мы можем разделить на два этапа:

- написание кода в скрипт-файл в DragonEgg.cs;
- настройка свойств Particle System, в окне Inspector элемента DragonEgg.prefab.

12. Приступим к написанию кода. Для этого добавьте в скрипт-файл DragonEgg.cs следующие строки кода:

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class DragonEgg : MonoBehaviour
{
   public static float bottomY = -30f;
   void Start()
   {
   }
   private void OnTriggerEnter(Collider other)
   {
      ParticleSystem ps = GetComponent<ParticleSystem>();
      var em = ps.emission;
      em.enabled = true;
      Renderer rend;
      rend = GetComponent<Renderer>();
      rend.enabled = false;
   }
   void Update()
   {
      if (transform.position.y < bottomY)
```

```
        {
            Destroy(this.gameObject);
        }
    }
}
```

// End Code

Как и ранее, скриншот листинга покажем ниже.

EnemyDragon.cs | DragonEgg.cs

No selection

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class DragonEgg : MonoBehaviour
6   {
7       public static float bottomY = -30f;
8
9       void Start()
10      {
11      }
12
13      private void OnTriggerEnter(Collider other)
14      {
15          ParticleSystem ps = GetComponent<ParticleSystem>();
16          var em = ps.emission;
17          em.enabled = true;
18
19          Renderer rend;
20          rend = GetComponent<Renderer>();
21          rend.enabled = false;
22      }
23
24      void Update()
25      {
26          if (transform.position.y < bottomY)
27          {
28              Destroy(this.gameObject);
29          }
30      }
31  }
```

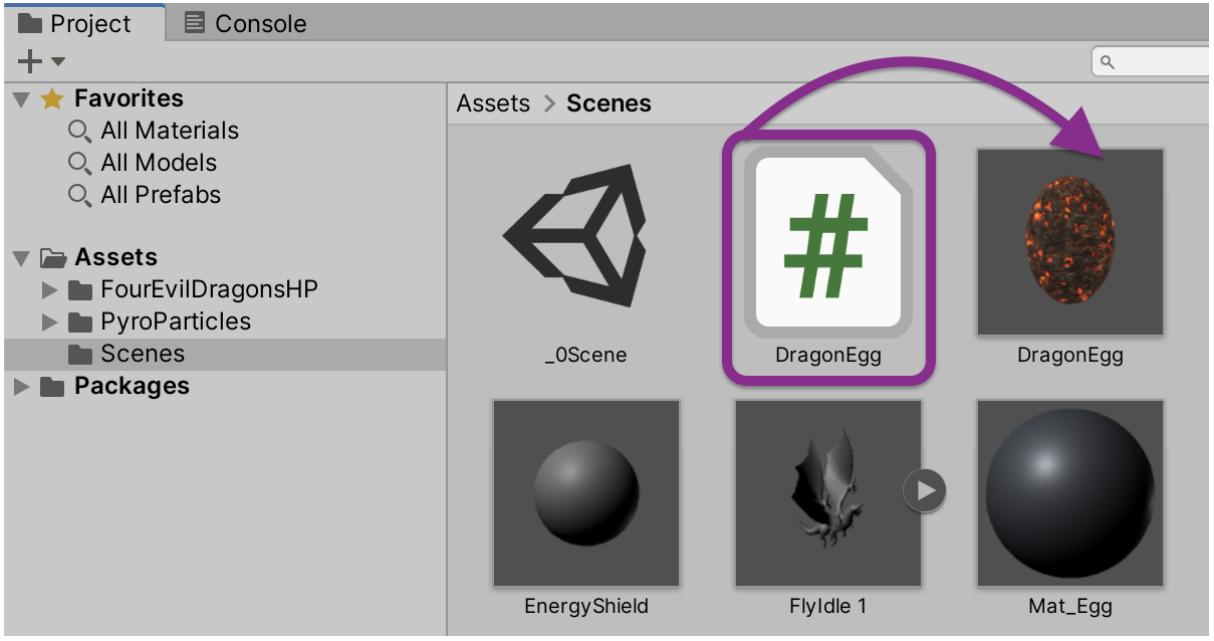
В скрипт файле основное функциональные блоки выполняют следующие действия:

- создается переменная bottomY, которая будет указывать, на каком расстоянии Y нужно будет удалять объекты DragonEgg.

- По названию метода OnTriggerEnter (Collision Other) можно понять, что он начинает работу, когда происходит пересечение с объектом, который работает как Trigger. В нашем случае таким объектом-триггером будет выступать плоскость Plane. При срабатывании триггера запускается проигрывание Particle System (em.enabled = true), и яйцо становится невидимым (rend.enabled = false).

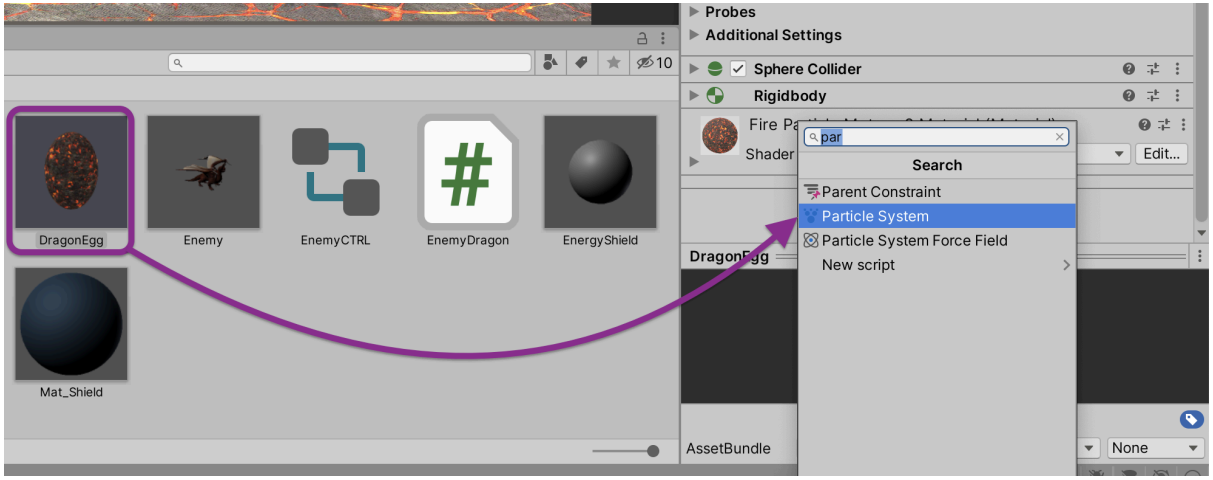
- внутри метода Update происходит уничтожение объекта DragonEgg (Destroy(this.gameObject)), если яйцо падает ниже уровня -10f. Это позволяет избежать накопления ненужных объектов за пределами игровой сцены.

13. Сохраните сценарий DragonEgg.cs и подключите его к префабу (шаблону) DragonEgg.prefab в панели Project. Для этого вы можете просто перетащить файл DragonEgg.cs на префаб DragonEgg.prefab:

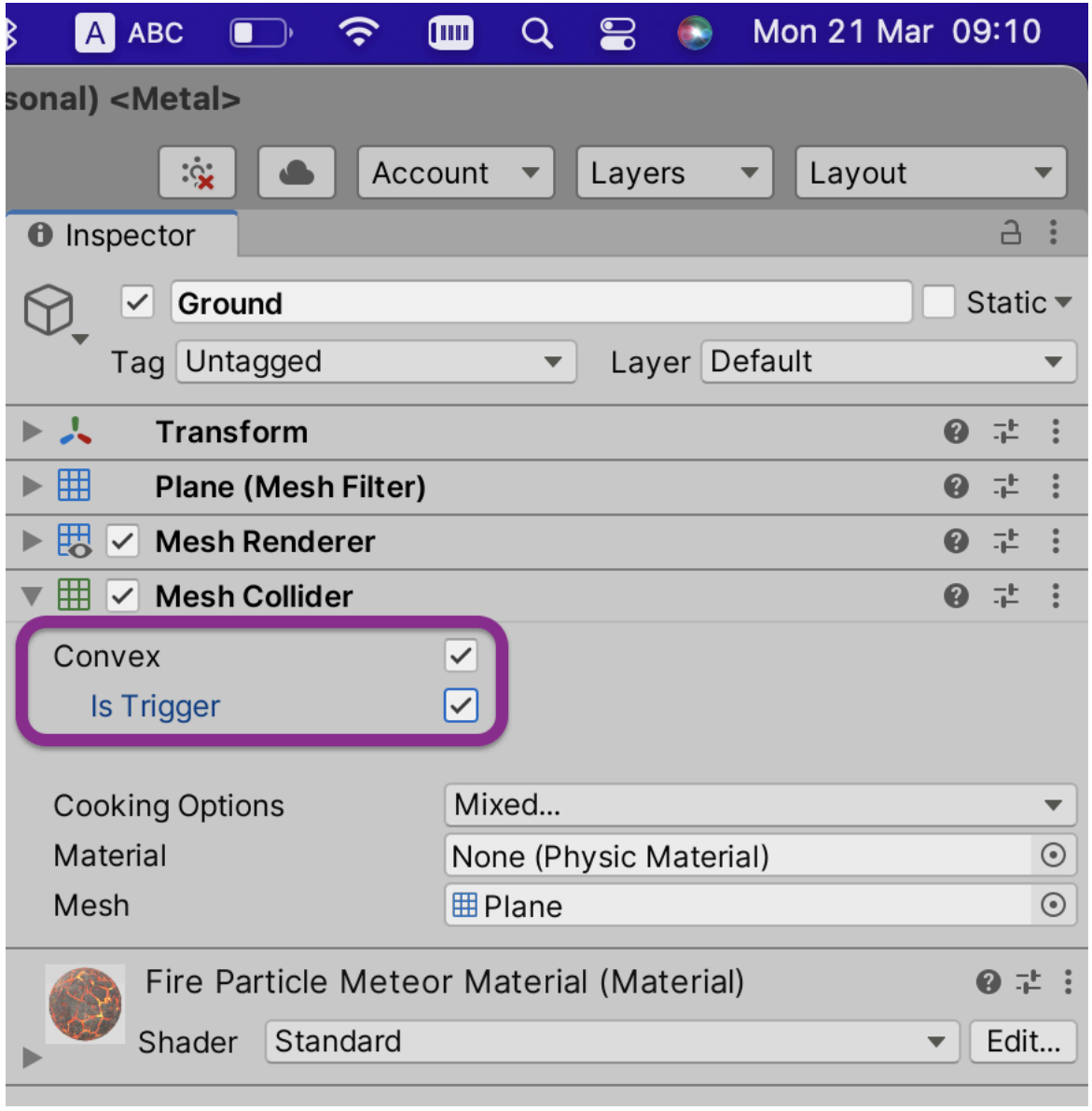


14. После этого можно убедиться, что в окне инспектора Inspector появится компонент с подключенным Script-файлом.

15. Далее мы добавим эффект взрыва в момент падения на землю драконьего яйца. Для этого в папке Assets/Scenes нужно выбрать префаб DragonEgg.prefab и в окне Inspector добавить ему компонент (Add component) Particle System:



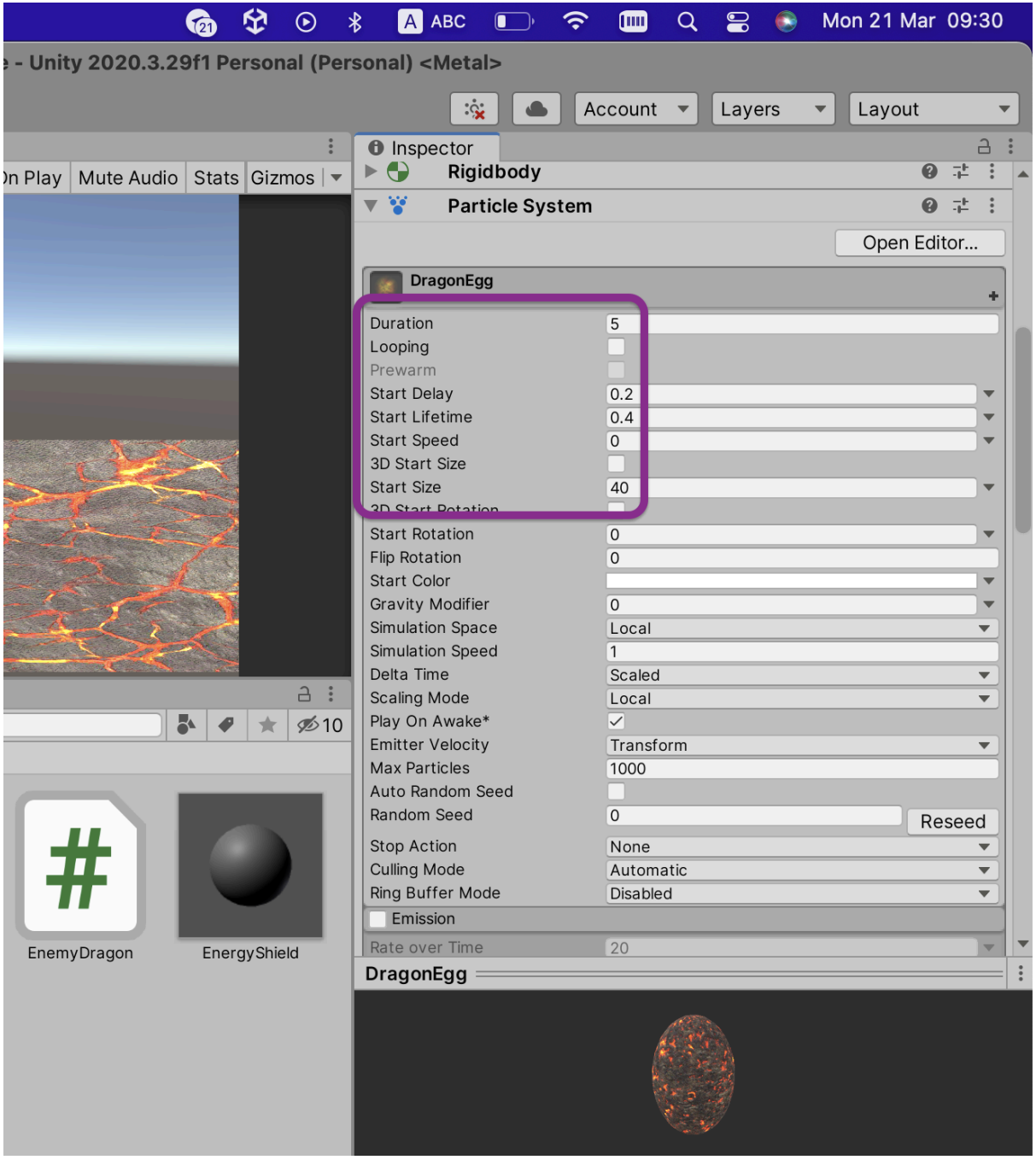
16. Теперь настроим объект Ground, чтобы он работал в качестве триггера. Сделать это можно достаточно просто, в окне Hierarchy выберите объект Ground и после этого в окне Inspector поставьте две галочки в компоненте Mesh Collider - Convex, Is Trigger:


ABC
Mon 21 Mar 09:10
sonal) <Metal>
Account
Layers
Layout
Inspector
Ground
Static
Tag
Untagged
Layer
Default
Transform
Plane (Mesh Filter)
Mesh Renderer
Mesh Collider
Convex
Is Trigger
Cooking Options
Mixed...
Material
None (Physic Material)
Mesh
Plane
Fire Particle Meteor Material (Material)
Shader
Standard
Edit...

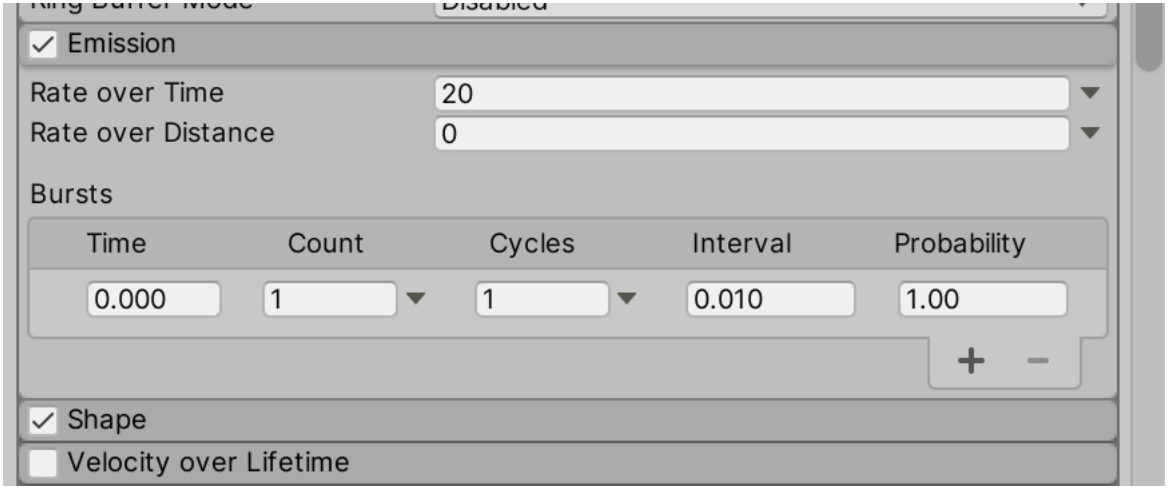
17. Убедитесь, что при нажатии кнопки Play игровые объекты DragonEgg летят вниз с генерацией элементов Particle System и исчезают из иерархии объектов после того, как достигнут нижнего края (заданного уровня в листинге выше -30).

18. В данный момент вид элементов Particle System не настроен, поэтому они отображаются в виде стандартных фиолетовых частиц, вылетающих из объекта DragonEgg.

19. Изменим настройки Particle System, чтобы они имели внешний вид, похожий на взрыв. Для этого выберите элемент DragonEgg, в окне Inspector разверните компонент Particle System. Он содержит большое количество дополнительных настроек, ниже выделены основные, на которые следует обратить внимание. Настройте компонент так, как показано на рисунке ниже:


Mon 21 Mar 09:30
- Unity 2020.3.29f1 Personal (Personal) <Metal>
Account
Layers
Layout
Inspector
On Play
Mute Audio
Stats
Gizmos
Rigidbody
Particle System
Open Editor...
DragonEgg
Duration 5
Looping
Prewarm
Start Delay 0.2
Start Lifetime 0.4
Start Speed 0
3D Start Size
Start Size 40
Start Rotation 0
Flip Rotation 0
Start Color
Gravity Modifier 0
Simulation Space Local
Simulation Speed 1
Delta Time Scaled
Scaling Mode Local
Play On Awake*
Emitter Velocity Transform
Max Particles 1000
Auto Random Seed
Random Seed 0
Reseed
Stop Action None
Culling Mode Automatic
Ring Buffer Mode Disabled
Emission
Rate over Time 20
EnemyDragon
EnergyShield
DragonEgg

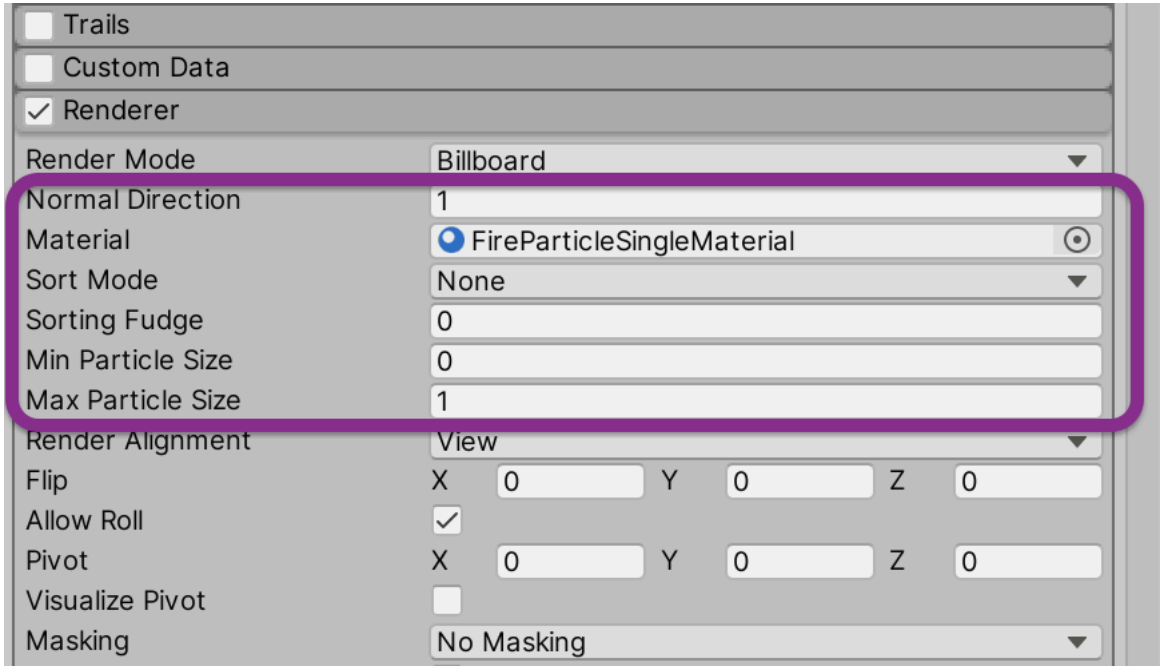
20. Там же в настройках Particle System поставьте галочку возле свойства Emission и введите параметры, показанные на рисунке ниже (чтобы добавить новую линию с параметрами “всплесков” нажмите “+”).



21. Теперь снимите галочку с Emission, чтобы элемент стал не активным. Это делается для того, чтобы при запуске сцены этот параметр был выключен и объект не излучал Particles с эффектами взрыва сразу после появления на сцене (он должен начинать работать только после срабатывания триггера, т. е. касания земли).

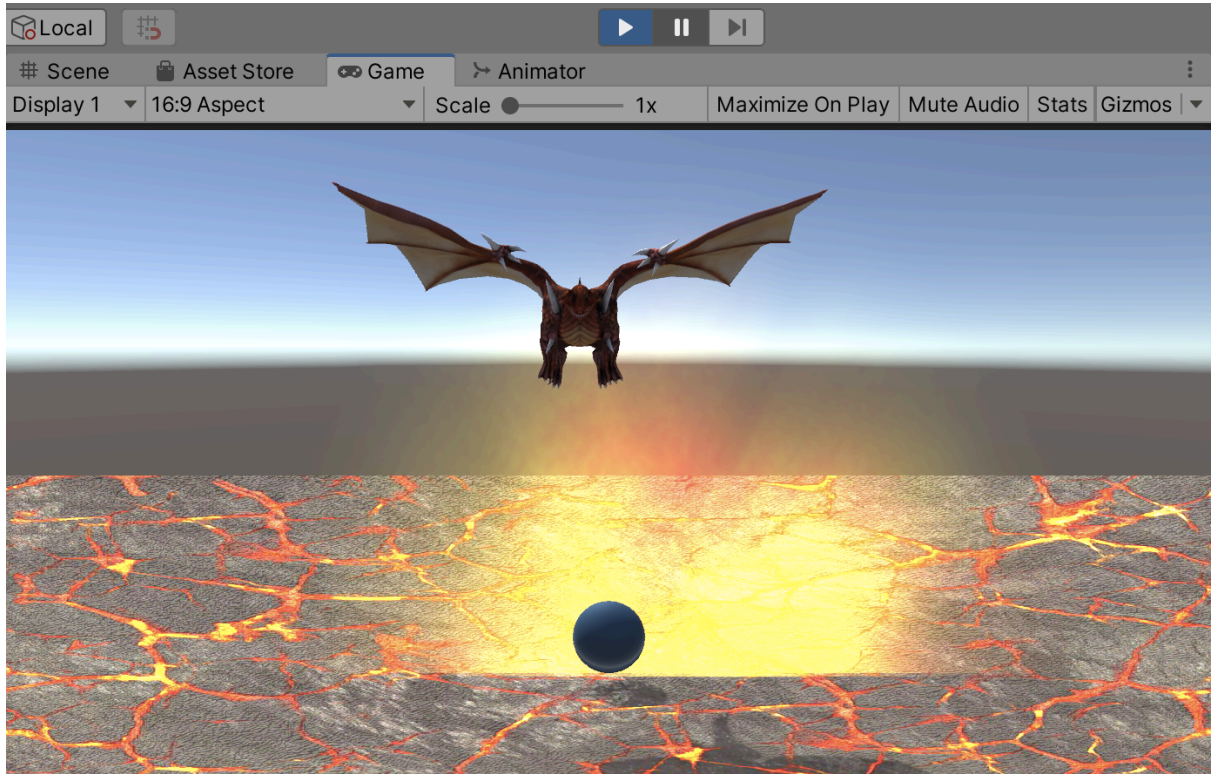
22. Поставьте галочку напротив “Size over Lifetime”, и выберите вид кривой, которая растет от минимума слева до максимума справа. Эта кривая означает, что элемент Particle будет распространяться от центра в разные стороны, как это происходит во время взрыва. Если обратите внимание, что существует несколько типовых видов кривых, которые позволяют создавать совершенно разные эффекты.

23. Последний элемент, с которым мы поработаем внутри Particle System — это Renderer, в нем назначается текстура частицы, которая генерируется при запуске. Разверните свойство Renderer, выберите в качестве текстуры материал с названием FireParticleSingleMaterial, эта текстура также находится в скачанном ассет-паке в папке PyroParticles. Также установите значение Min Particle Size – 0 и Max Particle Size – 1.



24. Теперь вы можете запустить сцену и проверить, что при столкновении драконьего яйца с плоскостью земли, создается эффект взрыва. Но если быть точным, то на самом деле происходит следующее:

- при столкновении объекта DragonEgg с Ground срабатывает триггер, после этого яйцо перестает отображаться (rend.enabled = false);
- запускается работа Particle System, имеющей визуальное оформление взрыва (em.enabled = true);
- объект DragonEgg уничтожается при падении ниже уровня y = -30f.



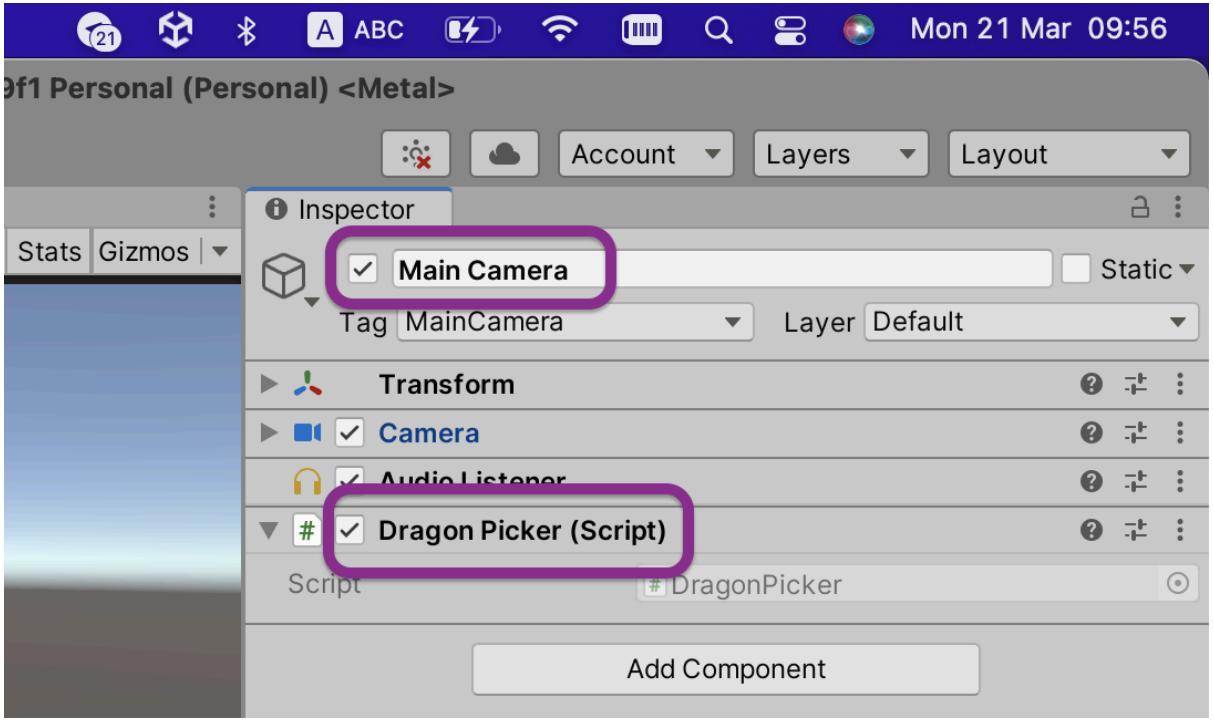
Таким образом, в пункте 3.2 мы проделали достаточно большую работу, улучшив визуальное оформление нашей игровой сцены. Также мы написали функционал объекта DragonEgg, научились создавать эффект взрыва, скрывать и удалять объекты при наступлении различных событий.

3.3 Скрипт-файл DragonPicker

Практически во всех играх существует сценарий, который управляет игрой в целом. Он отвечает за запуск и перезапуск сцены, сохраняет достижения и управляет загрузкой и т. д. В нашем примере мы создадим такой сценарий, он будет называться DragonPicker.cs. Подключать сценарии, управляющие

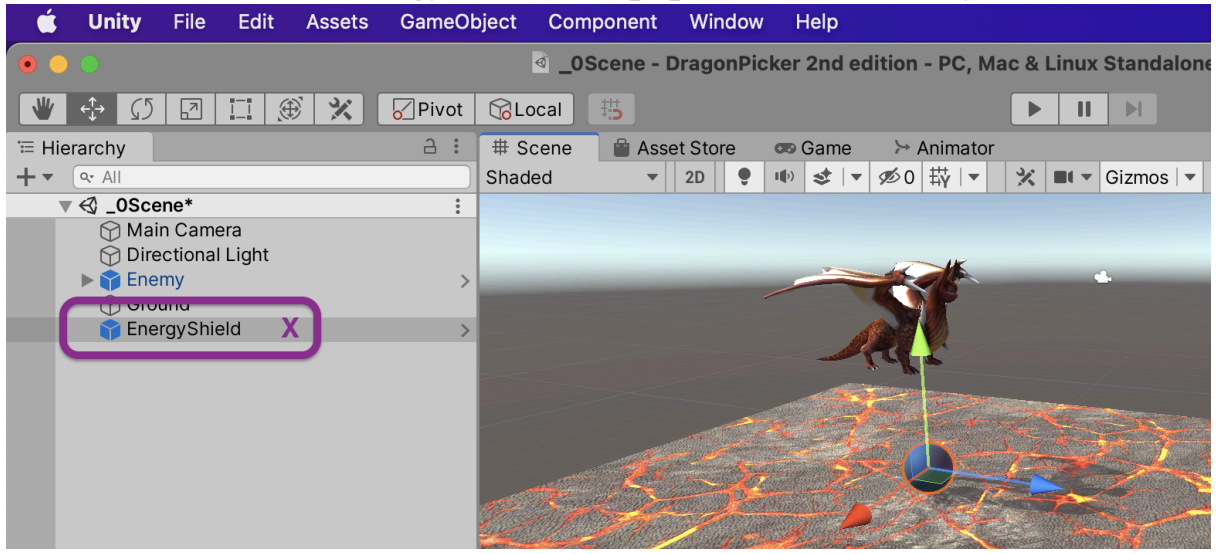
игрой, следует к тем объектам, которые гарантированно присутствуют в любой сцене, например к объекту Main Camera. Давайте это сделаем.

1. Создайте сценарий DragonPicker.cs и подключите его (можно так же, как и ранее - перетаскиванием) к элементу Main Camera в окне Hierarchy:



2. Помимо управления игрой этот сценарий будет создавать три экземпляра EnergyShield - объекта, который будет ловить летящие вниз яйца дракона. Условно игрок будет управлять энергетическими шарами EnergyShield, которые в то же время будут символизировать количество жизней. До этого момента объект EnergyShield статично висел на сцене. Создайте из него префаб, перетащив из окна Hierarchy в папку Assets/Scenes к другим префабам. Так как EnergyShield теперь будет генерироваться с

помощью скрипт-файла, то мы можем удалить статичный объект, который висел все это время статично на сцене. Найдите EnergyShield в иерархии объектов и удалите его со сцены:



3. Далее откройте недавно созданный скрипт-файл с именем DragonPicker. Напишите код, который будет создавать три экземпляра шаблона EnergyShield, располагая их на экране внутри друг друга.

// Start Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class DragonPicker  : MonoBehaviour
{
   public GameObject energyShieldPrefab;
   public int numEnergyShield = 3;
   public float energyShieldButtomY = -6f;
```

```
    public float energyShieldRadius = 1.5f;
    void Start()
    {
        for (int i = 1; i <= numEnergyShield; i++)
        {
            GameObject tBasketGo = Instantiate<GameObject>(energyShieldPrefab);
            tBasketGo.transform.position = new Vector3(0, energyShieldButtomY, 0);
            tBasketGo.transform.localScale = new Vector3(1 * i, 1 * i, 1 * i);
        }
    }
    void Update()
    {
    }
}
```

// End Code

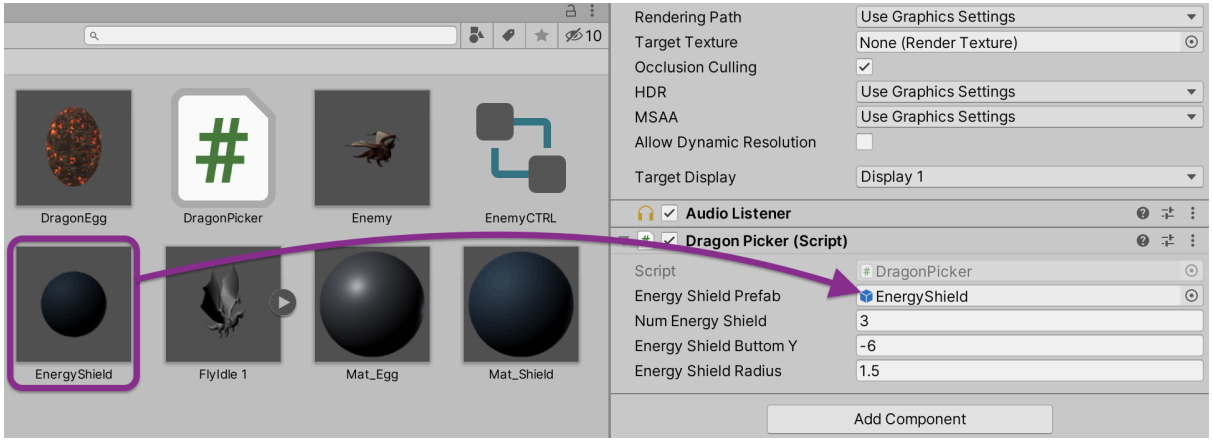
Скриншот листинга приведем ниже.

EnemyDragon.cs | DragonEgg.cs | DragonPicker.cs

No selection

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class DragonPicker : MonoBehaviour
6   {
7       public GameObject energyShieldPrefab;
8       public int numEnergyShield = 3;
9       public float energyShieldButtomY = -6f;
10      public float energyShieldRadius = 1.5f;
11
12      void Start()
13      {
14          for (int i = 1; i <= numEnergyShield; i++)
15          {
16              GameObject tBasketGo = Instantiate<GameObject>(energyShieldPrefab);
17              tBasketGo.transform.position = new Vector3(0, energyShieldButtomY, 0);
18              tBasketGo.transform.localScale = new Vector3(1 * i, 1 * i, 1 * i);
19          }
20      }
21
22      void Update()
23      {
24      }
25  }
```

4. Чтобы подключить EnergyShield к сценарию, выберите Main Camera, в Inspector для поля Energy Shield Prefab установите шаблон EnergyShield. Шаблон EnergyShield можно подключить, как и ранее перетаскиванием, либо выбрать из выпадающего списка (кликнув на «мишень»).



5. Нажмите кнопку Play и убедитесь, что написанный код создает три экземпляра EnergyShield в нижней части экрана. Корректно проверить генерацию этих объектов можно в окне иерархии объектов (в левой части среды разработки).

3.4 Скрипт-файл EnergyShield

1. В этом пункте будет написан код для перемещения игрового объекта EnergySheild вслед за указателем мыши. При желании вы можете изменить “интерфейс ввода”, адаптированный для клавиатуры, Touch Screen мобильных устройств (смартфонов, планшетов) и изменить тем самым механику игры.

2. Создайте сценарий EnergyShield.cs и подключите его к шаблону EnergyShield.prefab:

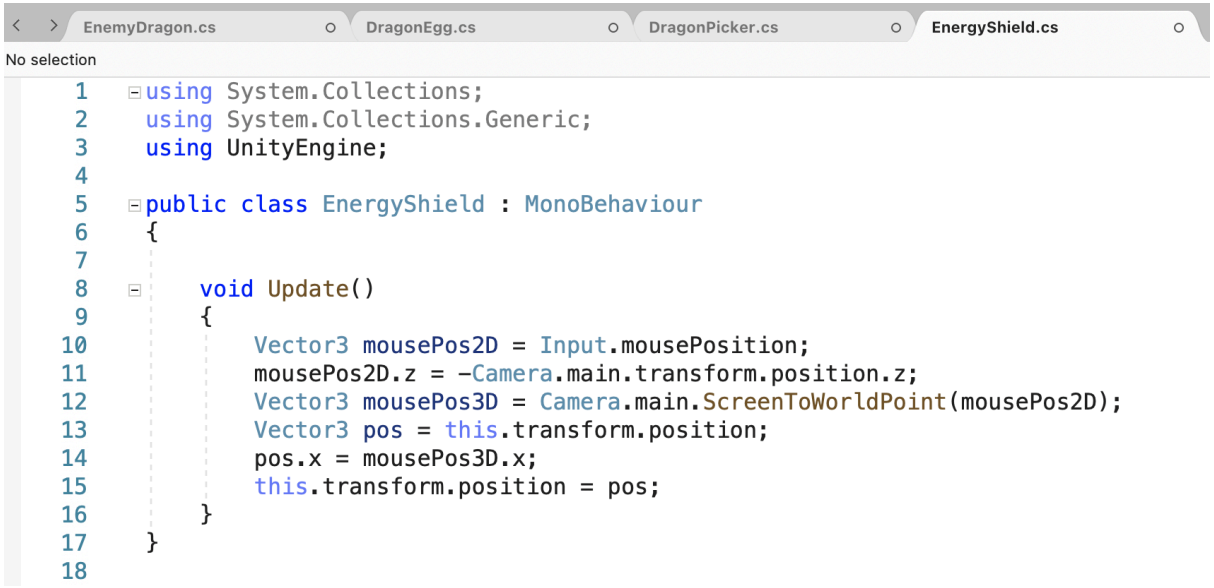
3. Напишите в созданный скрипт-файл код, приведенный ниже. Он будет отвечать за перемещение энергетических щитов Energy Shield вслед за курсором мыши:

```
// Start Code
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;
public class EnergyShield  : MonoBehaviour
{
   void Update()
   {
      Vector3 mousePos2D = Input.mousePosition;
      mousePos2D.z = -Camera.main.transform.position.z;
      Vector3 mousePos3D = Camera.main.ScreenToWorldPoint(mousePos2D);
      Vector3 pos = this.transform.position;
      pos.x = mousePos3D.x;
      this.transform.position = pos;
   }
}
```

// End Code

Скриншот листинга показан ниже.



4. Проверьте, что при нажатии на кнопку Play, энергетические шары перемещаются вслед за указателем мыши.

5. Реализовать ловлю объектов DragonEgg можно с помощью метода OnCollisionEnter. В тот же скрипт-файл EnergyShield.cs добавьте новый метод OnCollisionEnter ниже метода Update():

// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class *EnergyShield : MonoBehaviour*
{
  void *Update()*
  {

```
        Vector3 mousePos2D = Input.mousePosition;
        mousePos2D.z = -Camera.main.transform.position.z;
        Vector3 mousePos3D = Camera.main.ScreenToWorldPoint(mousePos2D);
        Vector3 pos = this.transform.position;
        pos.x = mousePos3D.x;
        this.transform.position = pos;
    }
    private void OnCollisionEnter(Collision coll)
    {
        GameObject Collided = coll.gameObject;
        if (Collided.tag == "DragonEgg")
        {
            Destroy(Collided);
        }
    }
}
// End Code
```

Скриншот листинга приведен ниже.

EnemyDragon.cs | DragonEgg.cs | DragonPicker.cs | **EnergyShield.cs**

No selection

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class EnergyShield : MonoBehaviour
6   {
7       void Update()
8       {
9           Vector3 mousePos2D = Input.mousePosition;
10          mousePos2D.z = -Camera.main.transform.position.z;
11          Vector3 mousePos3D = Camera.main.ScreenToWorldPoint(mousePos2D);
12          Vector3 pos = this.transform.position;
13          pos.x = mousePos3D.x;
14          this.transform.position = pos;
15      }
16
17      private void OnCollisionEnter(Collision coll)
18      {
19          GameObject Collided = coll.gameObject;
20          if (Collided.tag == "DragonEgg")
21          {
22              Destroy(Collided);
23          }
24      }
25  }
```

- метод OnCollisionEnter срабатывает (вызывается) каждый раз, когда с коллайдером EnergyShield что-то сталкивается. В локальную переменную Collided записывается ссылка на игровой объект (coll.gameObject), столкнувшийся с энергетическим шаром. Далее следует проверка, если этот объект имеет тег DragonEgg (см. раздел 2.4, п.8), - то он уничтожается - Destroy(Collided);

6. Сохраните сценарий, запустите игру (нажмите Play) и проверьте что драконьи яйца DragonEgg ловятся (точнее уничтожаются) при взаимодействии с энергетическими шарами EnergyShield.

7. Сохраните сцену _0Scene (ПКМ по имени сцены в окне Hierarchy и из выпадающего меню выберите Save Scene).

Самостоятельное задание

После выполнения пунктов в разделе 3, игра стала похожа на классическую и вполне работоспособную игру с ловлей объектов. На данном этапе этот функционал позволяет при желании модифицировать некоторые правила и аспекты игры, подобрать оптимальный уровень сложности и т. д.

Существует большое количество игр с похожей на Dragon Picker механикой, возможно вы уже отметили схожесть с тысячами подобных игр. Даже традиционный Flappy Bird можно получить из нашего Dragon Picker'а, перевернув сцену на 90 градусов, создав эффект полета движением заднего фона сцены, и изменив немного условие, при котором уничтожаться при пересечении с объектом будет "корзина", а не падающий объект.

В качестве дополнительного задания можете самостоятельно подумать над тем, чтобы расширить функционал и правила игры Dragon Picker. Например:

- вы можете добавить второй вид падающих объектов, которые будут либо наносить урон при ловле,
- либо наоборот - добавлять жизни.

Придуманные правила и функционал должны делать игру более интересной. Добавляйте только те элементы, которые вы бы смогли реализовать с незначительными временными затратами. Помните, что внесение даже самых маленьких дополнений в игру может кардинально изменить ее восприятие. Но в то же время может привести к неработоспособности или некорректной работе игры в самых непредвиденных местах.

Приложение А.

Для удобства в «Приложении А» приводится содержание всех скрипт-файлов, которые мы использовали при создании игры.

DragonEgg.cs

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class DragonEgg : MonoBehaviour
{
    public static float bottomY = -30f;
    public AudioSource audioSource;
    private void OnTriggerEnter(Collider other)
    {
        ParticleSystem ps = GetComponent<ParticleSystem>();
        var em = ps.emission;
        em.enabled = true;
        Renderer rend;
        rend = GetComponent<Renderer>();
        rend.enabled = false;
        audioSource = GetComponent<AudioSource>();
        audioSource.Play();
    }
    void Update()
    {
        if (transform.position.y < bottomY)
        {
            Destroy(this.gameObject);
            DragonPicker apScript = Camera.main.GetComponent<DragonPicker>();
            apScript.DragonEggDestroyed();
```

```
    }
  }
}
// End Code
```

DragonPicker.cs

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class DragonPicker  : MonoBehaviour
{
  public GameObject energyShieldPrefab;
  public int numEnergyShield = 3;
  public float energyShieldButtomY = -6f;
  public float energyShieldRadius = 1.5f;
  public List<GameObject> basketList;
  void Start()
  {
    basketList = new List<GameObject>();
    for (int i = 1; i <= numEnergyShield; i++)
    {
      GameObject tBasketGo = Instantiate<GameObject>(energyShieldPrefab);
      tBasketGo.transform.position = new Vector3(0, energyShieldButtomY, 0);
      tBasketGo.transform.localScale =
      new Vector3(1 * i, 1 * i, 1 * i);
```

```
            basketList.Add(tBasketGo);
        }
    }
    void Update()
    {
    }
    public void DragonEggDestroyed()
    {
        GameObject[] tDragonEggArray =
        GameObject.FindGameObjectsWithTag("DragonEgg");
        foreach (GameObject tGO in tDragonEggArray)
        {
            Destroy(tGO);
        }
        int basketIndex = basketList.Count - 1;
        GameObject tBasketGo = basketList[basketIndex];
        basketList.RemoveAt(basketIndex);
        Destroy(tBasketGo);
        if (basketList.Count == 0)
        {
            SceneManager.LoadScene("_0Scene");
        }
    }
}
```

// End Code

EnemyDragon.cs

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class EnemyDragon : MonoBehaviour
{
   [Header("Set in Inspector")]
   public GameObject dragonEggPrefab;
   public float speed = 1f;
   public float timeBetweenEggDrops = 1f;
   public float leftRightDistance = 10f;
   public float chanceDirections = 0.1f;
   void Start()
   {
      Invoke("DropEgg", 2f); // 1
   }
   void DropEgg() // 2
   {
      Vector3 myVector = new
      Vector3(0.0f, 5.0f, 0.0f);
      GameObject egg =
      Instantiate<GameObject>(dragonEggPrefab);
      egg.transform.position =
      transform.position + myVector;
      Invoke("DropEgg", timeBetweenEggDrops);
   }
```

```
    void Update()
    {
        Vector3 pos = transform.position;
        pos.x += speed * Time.deltaTime;
        transform.position = pos;
        if (pos.x < -leftRightDistance)
        {
            speed = Mathf.Abs(speed);
        }
        else if (pos.x > leftRightDistance)
        {
            speed = -Mathf.Abs(speed);
        }
    }
    private void FixedUpdate()
    {
        if (Random.value < chanceDirections)
        {
              speed *= -1;
        }
    }
}
```

// End Code

EnergyShield.cs

// Start Code

```
using System.Collections;
```

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class EnergyShield  : MonoBehaviour
{
   public Text scoreGT;
   public AudioSource audioSource;
   void Start()
   {
      GameObject scoreGO = GameObject.Find("Score");
      scoreGT = scoreGO.GetComponent<Text>();
      scoreGT.text = "0";
   }
   void Update()
   {
      Vector3 mousePos2D = Input.mousePosition;
      mousePos2D.z = -Camera.main.transform.position.z;
      Vector3 mousePos3D = Camera.main.ScreenToWorldPoint(mousePos2D);
      Vector3 pos = this.transform.position;
      pos.x = mousePos3D.x;
      this.transform.position = pos;
   }
   private void OnCollisionEnter(Collision coll)
   {
      GameObject Collided = coll.gameObject;
      if (Collided.tag == "DragonEgg")
      {
```

```
        Destroy(Collided);
      }
      int score = int.Parse(scoreGT.text);
      score += 1;
      scoreGT.text = score.ToString();
      audioSource = GetComponent<AudioSource>();
      audioSource.Play();
   }
}
```

// End Code

MainMenu.cs

// Start Code

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class MainMenu : MonoBehaviour
{
   public void PlayGame()
   {
   SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
   }
   public void QuitGame()
   {
      Application.Quit();
   }
```

```
}
// End Code
```

Pause.cs

```
// Start Code
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class Pause : MonoBehaviour
{
    private bool paused = false;
    public GameObject panel;
        void Update()
        {
            if (Input.GetKeyDown(KeyCode.Space))
            {
                if (!paused)
                {
                    Time.timeScale = 0;
                    paused = true;
                    panel.SetActive(true);
                }
                else
                {
                    Time.timeScale = 1;
                    paused = false;
```

```
                panel.SetActive(false);
            }
        }
        if (Input.GetKeyDown(KeyCode.Escape))
        {
    SceneManager.LoadScene(SceneManager.
    GetActiveScene().buildIndex - 1);
        }
    }
}
```

// End Code