**ОБРАБОТКА ВЗАИМОДЕЙСТВИЯ ИЕРАРХИЧЕСКОЙ МОДЕЛИ**

**И**

**ОБЪЕКТОВ СЦЕНЫ**

**Создание скрипта для prefab снаряда**

Для организации взаимодействия снаряда танка с целями на сцене наиболее оптимальным образом будет генерация снарядов из их **префаба** (шаблона), предварительно размещенного в **Assets**. С этой целью необходимо создать **3D-объект** для снаряда, добавить для него компоненту **Rigidbody**, затем создать **prefab** снаряда и, перетянув его со сцены в папку Assets, удалить его, наконец, со сцены.

Создаваемый на основе объекта сцены префаб, кроме физических характеристик исходного объекта, может обладать и программным кодом, добавленным к этому объекту, и нести этот код во всех экземплярах генерируемых на сцену из префаба, поскольку сам код интерпретируется в Unity просто как одна из прочих компонент объекта.

Можно добавлять скрипт к объекту сразу перед его последующим переводом в префаб, а можно и после перевода объекта в префаб добавить его как новую компоненту.

В новый скрипт, добавленный на префаб снаряда, прежде всего, в метод **Update()** необходимо записать постоянное перемещение снаряда вперед:

**transform.position += transform.TransformDirection(Vector3.up \* coreSpeed);**

где: **TransformDirection** – задает перемещение в глобальных координатах сцены; **coreSpeed** – публичная переменная скорости, которую нужно объявить выше в данном скрипте.

После выстрелов можно заметить, что снаряды постоянно находятся на сцене и не удаляются, даже если отлетели уже на большое расстояние и по-прежнему остаются в памяти, что уменьшает производительность компьютера. Чтобы исправить данный факт, в код снаряда в метод **Start()** нужнодобавить метод **Destroy()**,который будет уничтожать снаряд, например, через 5 секунд:

**Destroy(gameObject, 5f);**

где **gameObject** – удаляемый объект, в данном случае снаряд (игровой объект – это любой объект, к которому добавлен скрипт). **Создание скрипта для ствола танка**

Добавить в код скрипта для ствола объявление публичной переменной о снаряде **core** и связать ее с **prefab** снаряда

**public GameObject core;**

В функцию **Update()** скрипта для ствола вставить код обработки нажатия на клавишу пробела:

**if (Input.GetKeyDown(KeyCode.Space))  
 {**

**Vector3 forwardofstvol = transform.position +**

**transform.TransformDirection(Vector3.up \* 4f);**

**GameObject newcore = Instantiate(core,forwardofstvol,transform.rotation);  
 }**

Здесь: метод **TransformDirection ()** задает сдвиг в глобальных координатах сцены, **Vector3.up**задает направление вдоль ствола (вектор единичной длины), а **4f** – заданная, исходя из размеров импортированной модели танка длина его ствола, т. е. вектор **forwardofstvol** задает точку в глобальном пространстве сцены на кончике ствола.

При выборе направления вдоль ствола **Vector3.up** необходимо учитывать конкретное расположение осей координат относительно точки привязки **Pivot**, полученное вами при создании модели танка в **3ds MAX** при требуемом развороте осей координат, таким образом, чтобы вертикально вверх была направлена ось **Y**, как принято в среде **Unity**, а не ось **Z**, что принято в среде **3ds MAX.** Но поскольку в среде Unity система координат левая, а в среде **3ds MAX** – правая то все оси повернуть как нужно не получается и необходимо подправлять нужное направление для импортированных из среды **3ds MAX** объектов искусственным подбором направления, которое в данном случае в Unity, вообще говоря, определяется вектором направления **Vector3.forward.** Возможно при повороте осей координат с точкой **Pivot** дляствола вам потребуются и другие направления: **Vector3.right** или **Vector3.left** – это будет видно по полету снаряда, он должен всегда лететь вдоль ствола и вперед.

**Обработка попадания в цель снаряда**

Для обработки столкновений снаряда с объектами-целями на сценеобычноиспользуется встроенная функция **OnCollisionEnter().** Но, если обработка всех целей будет одинаковой, то оптимально задать для всех объектов-целей один и тот же маркер-тег. Для этого в **Инспекторе** под именем каждого объекта-цели в строке **Tag** нужно вести из предлагаемого списка общее имя тега, например **goal** (вначале необходимо добавить в список это имя командой **Add Tag …**). При этом все объекты будут обнаруживаться по имени тега, а не по их именам **name**, которые у всех объектов обязаны быть разными. Это позволит все такие объекты обрабатывать одним и тем же программным кодом.

Теперь в методе **OnCollisionEnter** проверяемое условие столкновения с объектами можно задавать для определённого тега:

**private void OnCollisionEnter(Collision col)  
    {  
        if (col.gameObject.tag == "goal")**  
        {

Далее при выполнения условия можно записать любые действия для снаряда и цели. Например, для исчезновения изображения снаряда со сцены сразу после попадания в цель можно записать строку кода:

**GetComponent<Renderer>().enabled = false;**

Для изменения цвета цели после попадания в него снаряда на красный:

**col.gameObject.GetComponent<Renderer>().material.color= new Color(1, 0, 0);**

**Добавление эффекта взрыва**

Для придания естественности событию попадания снаряда в цель можно использовать имеющиеся в стандартных **Assets** среды Unity эффектах с частицами, которые хранятся в особых разработанных префабах по адресу **Standard Assets/ParticleSystems/Prefabs,** из которых для генерации эффекта взрыва можно выбрать префаб **Explosion.**

В скрипт для префаба снаряда необходимо добавить публичную объектную переменную, например, **explosion1** и связать ее с префабом **Explosion** в списке **Assets** для скрипта префаба снаряда в **Инспекторе:**

**public GameObject explosion1;**

Затем добавить в функции **OnCollisionEnter()** в условие проверки попадания в цель строку с генерацией префаба взрыва

**Instantiate(explosion1, gameObject.transform);**