**Введение в векторы и создание игровой логики**

Вычисление дистанции, направления, триггеры и коллизии

**Оглавление**

[Векторы](#_2s8eyo1)

[Перемещение объектов](#_17dp8vu)

[Quaternion и поворот объектов](#_3rdcrjn)

[Введение](#_95dluqdopemq)

[Знакомство с окном Navigation](#_2et92p0)

[Вкладка Object](#_cu2o1cpe16ym)

[Вкладка Bake](#_tqtshekvbjeo)

[Вкладка Areas](#_1t3h5sf)

[Добавление противника](#_4d34og8)

[Стандартный противник и его компоненты](#_al1g03qf43o3)

[Компонент NavMeshAgent](#_263fd7bc5iqv)

[Запекаем карту навигации](#_26in1rg)

[Другие компоненты, связанные с навигацией](#_lnxbz9)

[Компонент NavMeshObstacle](#_35nkun2)

[Компонент OffMeshLink](#_1ksv4uv)

[Практика](#_7u63nqmv7y68)

[Практическое задание](#_2xcytpi)

[Дополнительные материалы](#_1ci93xb)

[Используемая литература](#_3whwml4)

# 

# 

# 

## Векторы

При разработке игр часто приходится использовать такой математический объект, как вектор (направленный отрезок прямой). В Unity векторы применяются для позиции объекта, направления движения и различных расчетов.

**Определение длины вектора**

|  |
| --- |
| [SerializeField] private Vector2 \_vector2D;  [SerializeField] private Vector3 \_vector3D;  void Start()  {  print(\_vector2D.magnitude);  print(\_vector3D.magnitude);  } |

**Расстояние между двумя объектами** — позиция каждого объекта — это вектор. Если необходимо определить только отношение расстояний, то лучше использовать **sqrMagnitude**:

|  |
| --- |
| [SerializeField] private vector3 \_p1;  [SerializeField] private vector3 \_p2;  [SerializeField] private float \_distance;  void Start()  {  \_distance= Vector3.Distance(\_p1, \_p2);  print(\_distance);  \_distance= (\_p1 - \_p2).magnitude;  print(\_distance);  \_distance= Mathf.Sqrt((\_p1 - \_p2).sqrMagnitude);  print(\_distance);  } |

**Скалярное произведение** — скаляр (число) для векторов A и B, равное **(Ax\*Bx)+(Ay\*By)+(Az\*Bz)**. Если скалярное произведение равно нулю, то данные вектора перпендикулярны друг другу, если нормированное значение стремиться к единице, то вектора параллельны и сонаправленные, если нормированное значение стремится к минус единице, то вектора противоположно направленные:

|  |
| --- |
| [SerializeField] private vector3 \_p1;  [SerializeField] private vector3 \_p2;  void Start()  {  print(Vector3.Dot(p1,p2));  } |

**Векторное произведение** — при векторном произведении, получаем вектор, перпендикулярный двум исходным. Компоненты результирующего вектора C определяются следующим образом: Cx = Ay\*Bz — Az\*By, Cy = Az\*Bx — Ax\*Bz, Cz = Ax\*By — Ay\*Bx:

|  |
| --- |
| [SerializeField] private vector3 \_p1;  [SerializeField] private vector3 \_p2;  void Start()  {  print(Vector3.Cross(p1,p2));  } |

## Перемещение объектов

Так как мы фактически работаем в трехмерном пространстве, наши gameobject’ы имеют **transform.position**, который является структурой **vector3**. Рассмотрим варианты изменения позиции объекта (не связанные с физикой):

* **Используем иерархию объектов.** При перемещении объекта все те, для которых он является родителем, будут изменять свою позицию соответственно;
* **Изменяем позицию** **через компонент Transform** — простое, но далеко не оптимальное решение, так как операция **new** выполняется каждый кадр:

|  |
| --- |
| [SerializeField] private int \_speed = 1;  void Update ()  {  transform.position += new Vector3(Time.deltaTime \*\_speed , 0, 0);  } |

* **Применяем метод Translate**. В параметрах указывается вектор направления движения или координаты по отдельности. Последним параметром (необязательным) — относительно чего происходит движение (локальная или глобальная система координат, другой объект — например, камера). По умолчанию задана локальная система координат.

|  |
| --- |
| private void Update () { *// Движение объекта вперед со скоростью 1 единица в секунду*  transform.Translate(Vector3.forward\*Time.deltaTime);  } |

|  |
| --- |
| private void Update ()  { *// Движение объекта вперед в глобальной системе координат*  transform.Translate(Vector3.forward\*Time.deltaTime, Space.World);  } |

|  |
| --- |
| private void Update () { *// Движение объекта вдоль оси Z*  transform.Translate(0, 0, Time.deltaTime);  } |

|  |
| --- |
| private void Update () { *// Движение объекта вперед относительно камеры*  transform.Translate(Vector3.forward\*Time.deltaTime,  Camera.main.transform);  } |

* Применяем метод **MoveTowards** — движение объекта по прямой до указанной позиции с заданной скоростью.

|  |
| --- |
| [SerializeField] private Transform \_target; [SerializeField] private float \_speed = 1;  private void Update()  {  transform.position = Vector3.MoveTowards(transform.position, \_target.position, \_speed \* Time.deltaTime);  } |

* **Применяем метод Vector3.Lerp()** — линейная интерполяция между двумя векторами. Это метод, не относящийся напрямую к движению объекта, но часто использующийся для этой цели.

## Quaternion и поворот объектов

В Unity для представления поворота объекта используются кватернионы. Их сложно понять на интуитивном уровне, но просто использовать. В дополнительных материалах к уроку есть ссылки на статьи, где подробно рассматривается, что такое кватернионы и почему именно они, а не просто углы, используются в играх, робототехнике. При создании 2D-игры кватернионы будут встречаться реже, чем в 3D, но все-таки они тоже играют важную роль.

Большую часть задач, связанных с поворотом объекта, решают следующие методы:

* **Quaternion.Identity** — значение, которое соответствует отсутствию поворота.

|  |
| --- |
| private void Start() {  transform.rotation = Quaternion.identity; } |

* **Quaternion.LookRotation** — метод для поворота объекта в сторону позиции заданного объекта (цели).

|  |
| --- |
| [SerializeField] private Transform \_target;  private void Update() {  Vector3 relativePos = \_target.position - transform.position;  Quaternion rotation = Quaternion.LookRotation(relativePos);  transform.rotation = rotation; } |

* **Quaternion.Angle** — метод, возвращающий значение угла между двумя объектами.

|  |
| --- |
| [SerializeField] private Transform \_target;  [SerializeField] private void Update() {  float angle = Quaternion.Angle(transform.rotation, \_target.rotation);  print(angle);  } |

* **Quaternion.Euler** — метод, принимающий в качестве параметров вектор или три компонента вектора (x, y, z) и возвращающий значение в виде кватерниона.

|  |
| --- |
| private void Start() {  transform.rotation = Quaternion.Euler(0, 30, 0); } |

* **Quaternion.Slerp** — сферическая интерполяция между двумя значениями. Возвращаемое значение изменяется с заданной скоростью от одного значения (первый параметр) до другого (второй параметр).

|  |
| --- |
| [SerializeField] private Transform \_from; [SerializeField] private Transform \_to;  void Update() {  transform.rotation = Quaternion.Slerp(\_from.rotation, \_to.rotation,  Time.time); } |

* **Quaternion.FromToRotation** — создает поворот от вектора, заданного первым параметром, к вектору, заданному вторым.

|  |
| --- |
| private void Start() {  transform.rotation = Quaternion.FromToRotation(Vector3.up,  Vector3.right); } |

* **метод Rotate** — вращение объекта. В параметрах указывается вектор, вокруг которого происходит вращение, или его координаты по отдельности. Второй параметр (необязательный) — скорость поворота. Последний — в какой системе координат происходит поворот. По умолчанию задана локальная система координат.

|  |
| --- |
| private void Update() {  transform.Rotate(Vector3.up \* Time.deltaTime); *// Вращение объекта со скоростью 1 градус в секунду вокруг вектора, направленного вверх, для объекта в локальной системе координат* } |

|  |
| --- |
| private void Update() {  transform.Rotate(Vector3.up \* Time.deltaTime , Space.World);  *// Вращение объекта вокруг вектора, направленного вверх от объекта, в глобальной системе координат* } |

|  |
| --- |
| private void Update() {  transform.Rotate(Vector3.up, Time.deltaTime); *// Скорость поворота можно задать отдельным параметром* } |

|  |
| --- |
| private void Update() {  transform.Rotate(Time.deltaTime \* 10, 0, 0); *// Координаты вектора заданы по отдельности* } |

* **Метод RotateTowards** — возвращает вектор, направленный в сторону объекта «цели». Первый параметр — текущее направление вектора, второй — необходимое направление, третий — скорость поворота, четвертый — максимальная длина вектора. Для наглядности используется метод **Debug.DrawRay()**, который отрисовывает результирующий вектор.

|  |
| --- |
| [SerializeField] private Transform \_target; [SerializeField] private float \_speed = 1;  private void Update() {  Vector3 targetDir = \_target.position - transform.position;   Vector3 newDir = Vector3.RotateTowards(transform.forward, targetDir,  \_speed \* Time.deltaTime, 0.0F);  Debug.DrawRay(transform.position, newDir, Color.red);  transform.rotation = Quaternion.LookRotation(newDir); } |

* **Метод RotateAround** — может изменять как позицию, так и поворот объекта. Первый параметр — точка, относительно которой происходит поворот, второй — ось вращения, третий — скорость.

|  |
| --- |
| [SerializeField] private Transform \_target;  private Update() {  transform.RotateAround(\_target.position, Vector3.right, 10 \*  Time.deltaTime); } |

* **метод LookAt** — поворот объекта в направлении «цели», заданной в качестве параметра. Также есть скрытый параметр — это вектор, указывающий направление оси вверх для текущего объекта. По умолчанию он соответствует глобальной оси Y. Данный метод часто используется для камеры.

|  |
| --- |
| [SerializeField] private Transform \_target;  private void Update() {  transform.LookAt(\_target); } |

# 

# Триггеры и коллизии

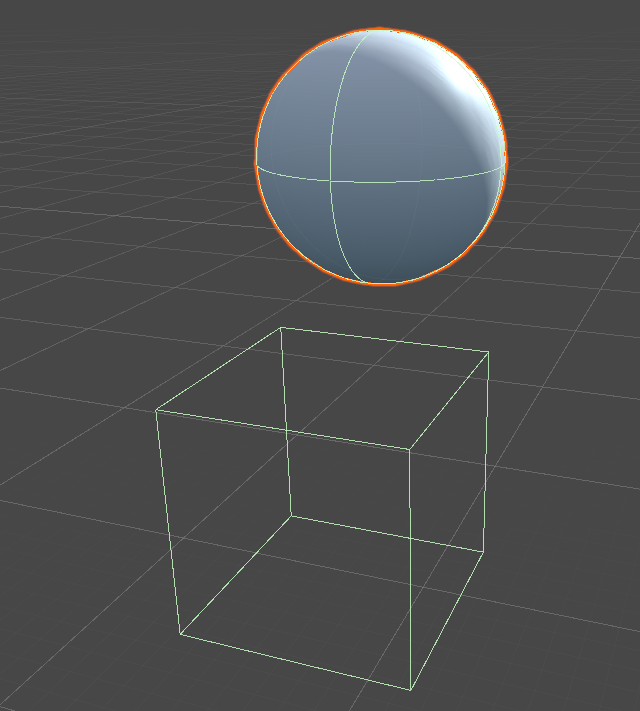
При взаимодействии объектов между собой в зависимости от них самих, их компонентов и характера взаимодействия возможно регистрировать и обрабатывать события.

## События, связанные с триггерами

Эти события происходят тогда, когда к объекту прикреплен коллайдер, который является триггером. Хотя бы у одного из объектов должен быть компонент **Rigidbody**:

* **OnTriggerEnter()** — происходит один раз в момент, когда объект касается объекта с триггером;
* **OnTriggerStay()** — происходит каждый кадр, пока объект находится внутри триггера другого объекта;
* **OnTriggerExit()** — происходит один раз в момент, когда объект выходит из триггера другого объекта.

В примере ниже на сцене находятся два объекта: первый — пустой объект с компонентом **BoxCollider**, который является триггером; второй — сфера с добавленным компонентом **Rigidbody**. Скрипт прикреплен к первому объекту.



|  |
| --- |
| private void OnTriggerEnter() {  print("Enter"); }  private void OnTriggerStay() {  print("Stay"); }  private void OnTriggerExit() {  print("Exit"); } |

Также из событий можно получить информацию об объекте, с которым произошло взаимодействие.

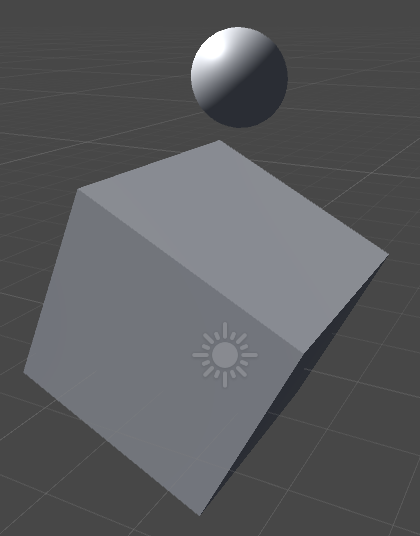
|  |
| --- |
| private void OnTriggerEnter(Collider coll) {  print($"Enter {coll.name}"); } |

## События, связанные с коллизиями

Эти события происходят, когда оба объекта имеют коллайдеры. И хотя бы у одного из объектов должен быть компонент **Rigidbody**:

* **OnCollisionEnter()** — происходит один раз, в момент касания объектов;
* **OnCollisionStay()** — происходит каждый кадр, пока объекты касаются друг друга;
* **OnCollisionExit()** — происходит один раз в момент, когда касание объектов заканчивается.

В примере ниже на сцене находятся два объекта: первый — куб со скриптом и остальными компонентами по умолчанию; второй — сфера с добавленным компонентом **Rigidbody**.



|  |
| --- |
| private void OnCollisionEnter(Collision collision)  {  print("Collision enter");  }  private void OnCollisionStay(Collision collision)  {  print("Collision stay");  }  private void OnCollisionExit(Collision collision)  {  print("Collision exit");  } |

Доступ к информации об объекте, с которым произошло взаимодействие, осуществляется аналогично.

# 

# Практика

На практике, мы создадим ,базовую игровую логику в виде перемещения персонажа расчета вращения в направлении движения, создания стреляющих и убивающих снарядов, а также создание мин.

# Практическое задание

1. Сделать бомбу, если противник наступает, всё взрывается.
2. Добавить загадки и открывающиеся двери.
3. Сделать турель (противника). Если игрок в радиусе поражения турели, она поворачивается в его сторону и начинает стрелять.
4. Реализовать спаунер противников.
5. Реализовать разворот персонажа в направлении движения.
6. Добавить поднимаемые объекты (жизни, патроны).
7. \* Реализовать стрельбу или “бросок предмета”.
8. \* Реализовать своего передвигающегося противника с полем зрения.
9. \* Добавить загадки и открывающиеся/исчезающие двери.
10. \* Добавить звуки для объектов (например двери) и взрывов.
11. \* Сделать прыжок.
12. \*\* Сделайте счётчик убитых, как только Джон победит всех, он выиграл.

# Дополнительные материалы

1. [Порядок выполнения функций.](http://docs.unity3d.com/ru/current/Manual/ExecutionOrder.html)
2. [Класс **Time**](https://docs.unity3d.com/ScriptReference/Time.html).
3. [**Quaternion**](https://habrahabr.ru/post/183908/).
4. [**Gimbal lock**](https://habrahabr.ru/post/183116/).
5. [**Документация к NavMesh V2**](https://docs.unity3d.com/Manual/NavMesh-BuildingComponents.html).
6. [**Ссылка на скачивание NavMesh V2**](https://github.com/Unity-Technologies/NavMeshComponents).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальный сайт Unity](https://unity3d.com/ru).
2. [Векторы](https://unity3d.com/ru/learn/tutorials/topics/scripting/vector-maths?playlist=17117).
3. [Метод **Translate**](https://docs.unity3d.com/ScriptReference/Transform.Translate.html).
4. [Метод **Rotate**](https://docs.unity3d.com/ScriptReference/Transform.Rotate.html).
5. [**NavMesh**](https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html).