

Скриптинг

Использование Скриптов

Скрипты позволяют:

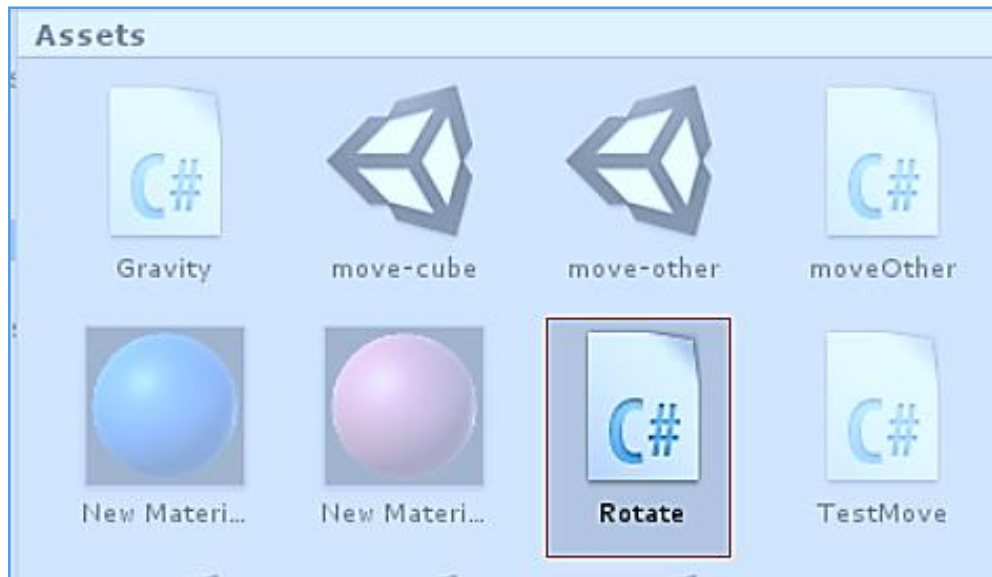
- изменять существующие компоненты;
- добавлять свои;
- активировать различные события;
- реагировать на ввод пользователя и т.д.

Unity изначально поддерживает **язык программирования C #**.

В дополнение к этому, с Unity можно использовать многие другие языки .NET, если они могут скомпилировать совместимую DLL

Создание Скриптов

- Меню **Create** в левом верхнем углу панели Project;
- главное меню **Assets > Create > C# Script** (или JavaScript скрипт);
- кнопка **Add Component** на панели Inspector;
- из контекстного меню панели **Assets**.



MonoBehaviour

MonoBehaviour - базовый класс, от которого наследуются все скрипты.

Содержит инструменты для

- работы с объектами;
- работы с компонентами;
- управления поведением объекта.

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

}
```

Public Methods

[BroadcastMessage](#)

[CompareTag](#)

[GetComponent](#)

[GetComponentInChildren](#)

[GetComponentInParent](#)

[GetComponents](#)

[GetComponentsInChildren](#)

[GetComponentsInParent](#)

[SendMessage](#)

[SendMessageUpwards](#)

[GetInstanceID](#)

[ToString](#)

Static Methods

[Destroy](#)

[DestroyImmediate](#)

[DontDestroyOnLoad](#)

[FindObjectOfType](#)

[FindObjectsOfType](#)

[Instantiate](#)

Структура файла скрипта

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }

}
```

Функция **Start** вызывается до первого вызова функции Update, и это идеальное место для выполнения инициализации.

Структура файла скрипта

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

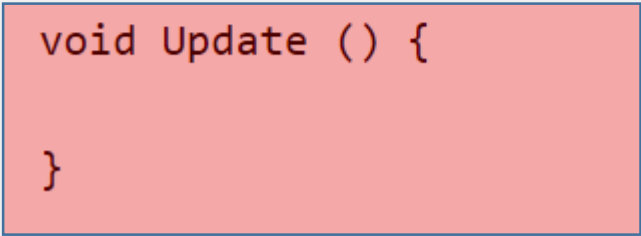
    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }

}
```



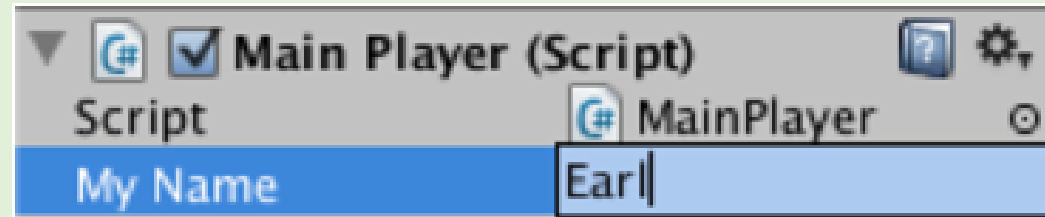
Функция **Update** вызывается при каждом обновлении кадра. Здесь описывается всё, что должно быть обработано с течением времени в игровом процессе.

Это может быть движение, срабатывание действий, ответная реакция на ввод пользователя.

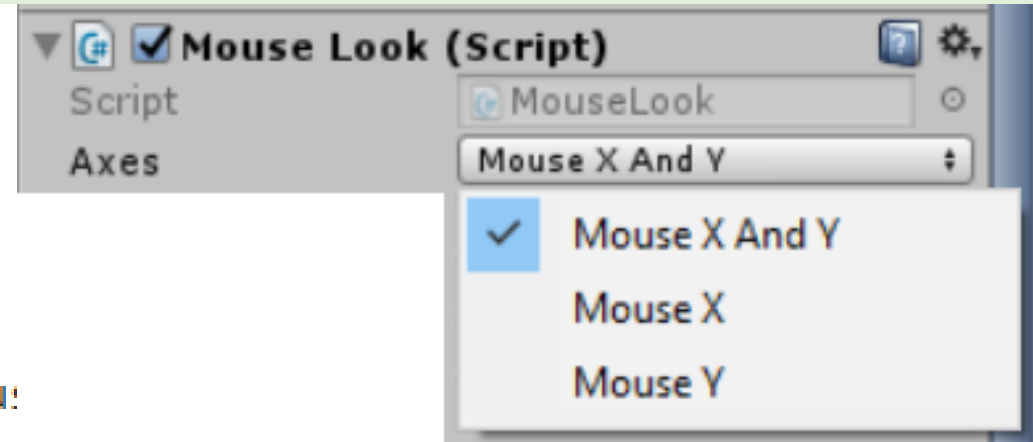
- **Awake ()** используется для инициализации любых переменных или состояния игры перед запуском игры. Awake () вызывается только один раз за время существования экземпляра скрипта. **Awake () всегда вызывается перед Start функциями объектов на сцене.** Это позволяет инициализировать скрипты. **Awake() вызывается, даже если сценарий является отключенным компонентом** активного GameObject, а Start() – при включенном компоненте скрипта.
- **FixedUpdate()** как правило вызывается чаще чем Update(). Вызывается в соответствии с таймером, не зависящим от частоты кадров, поэтому при расчётах передвижения внутри **FixedUpdate()**, не нужно умножать значения на Time.deltaTime. **FixedUpdate() позволяет получить более точные результаты для расчета физики.**
- **LateUpdate()** вызывается один раз за кадр, после завершения Update(). Часто **LateUpdate() используют для преследующей камеры.** Если перемещать персонаж в Update(), то вычисление перемещения камеры можно выполнить в **LateUpdate()**. Это обеспечит то, что персонаж изменит свое положение до того, как камера отследит его позицию.

Переменные на панели Inspector

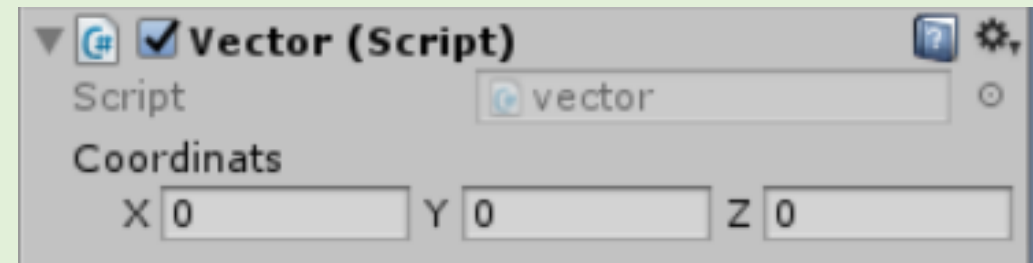
```
public string myName;
```



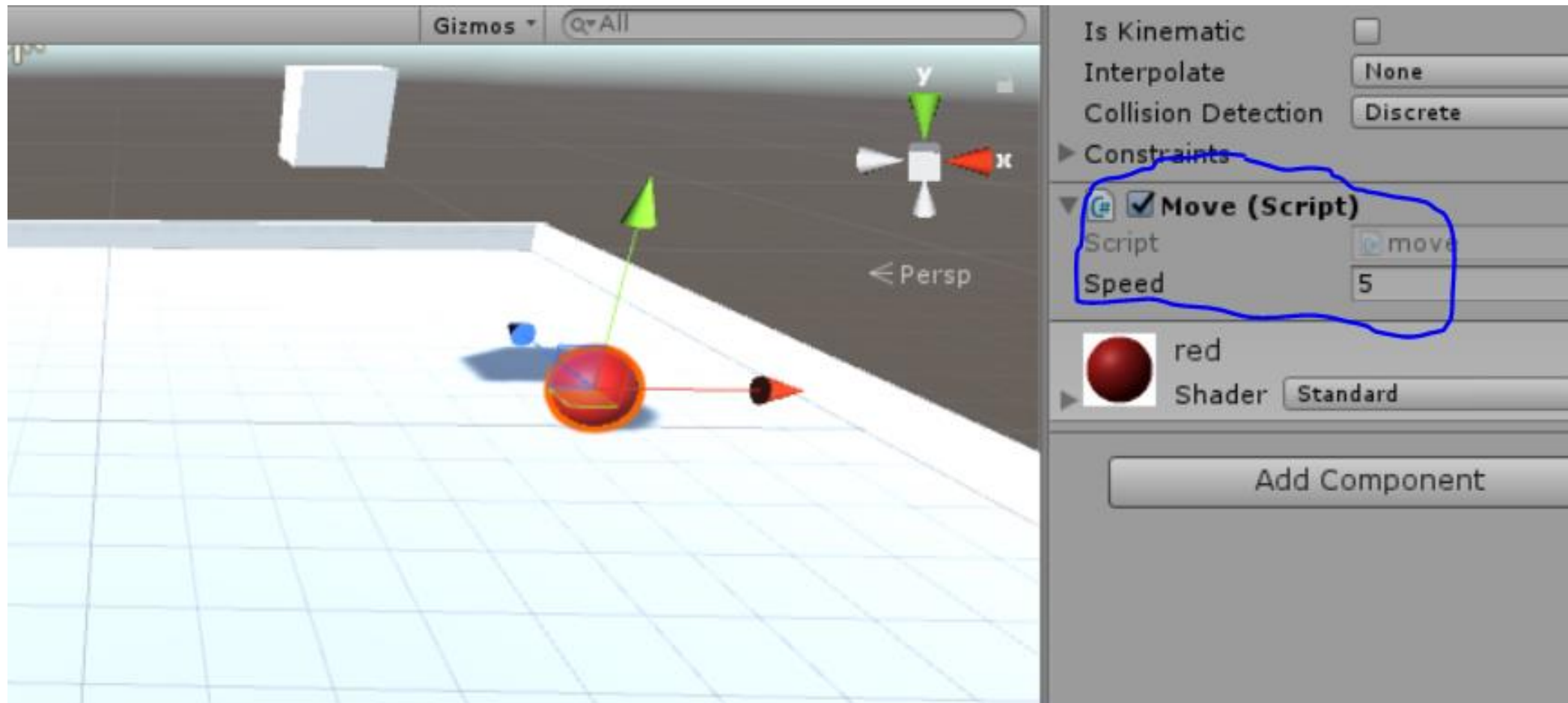
```
public enum RotationAxes
{
    MouseXAndY=0,
    MouseX=1,
    MouseY=2
}
public RotationAxes axes = RotationAxes.Mou:
```



```
public Vector3 coordinats;
```



Unity позволяет **изменять значение переменных скрипта в запущенной игре**. Это очень полезно чтобы увидеть эффекты от изменений сразу же, без остановки и перезапуска.



Обращение к компонентам

Для обращения к другим компонентам, присоединенным к тому же GameObject необходимо получить ссылку на экземпляр компонента. Это делается с помощью функции **GetComponent**.

```
void Start () {  
    Rigidbody rb = GetComponent<Rigidbody>();  
  
    // Change the mass of the object's Rigidbody.  
    rb.mass = 10f;  
}
```

Для часто используемых компонентов Unity предоставляет стандартные имена переменных, например

```
gameObject.GetComponent<Transform>().position= new Vector3(2,2,2)
```

можно упростить

```
transform.position= new Vector3(2,2,2)
```

Обращение к другим объектам

1. Связывание объектов через переменные

Самый простой способ найти нужный игровой объект - добавить в скрипт переменную типа `GameObject` с уровнем доступа `public`:

```
public class Enemy : MonoBehaviour {  
    public GameObject player;  
  
    void Start() {  
        // Start the enemy ten units behind the player character.  
        transform.position = player.transform.position - Vector3.forward * 10f;  
    }  
}
```

Обращение к другим объектам

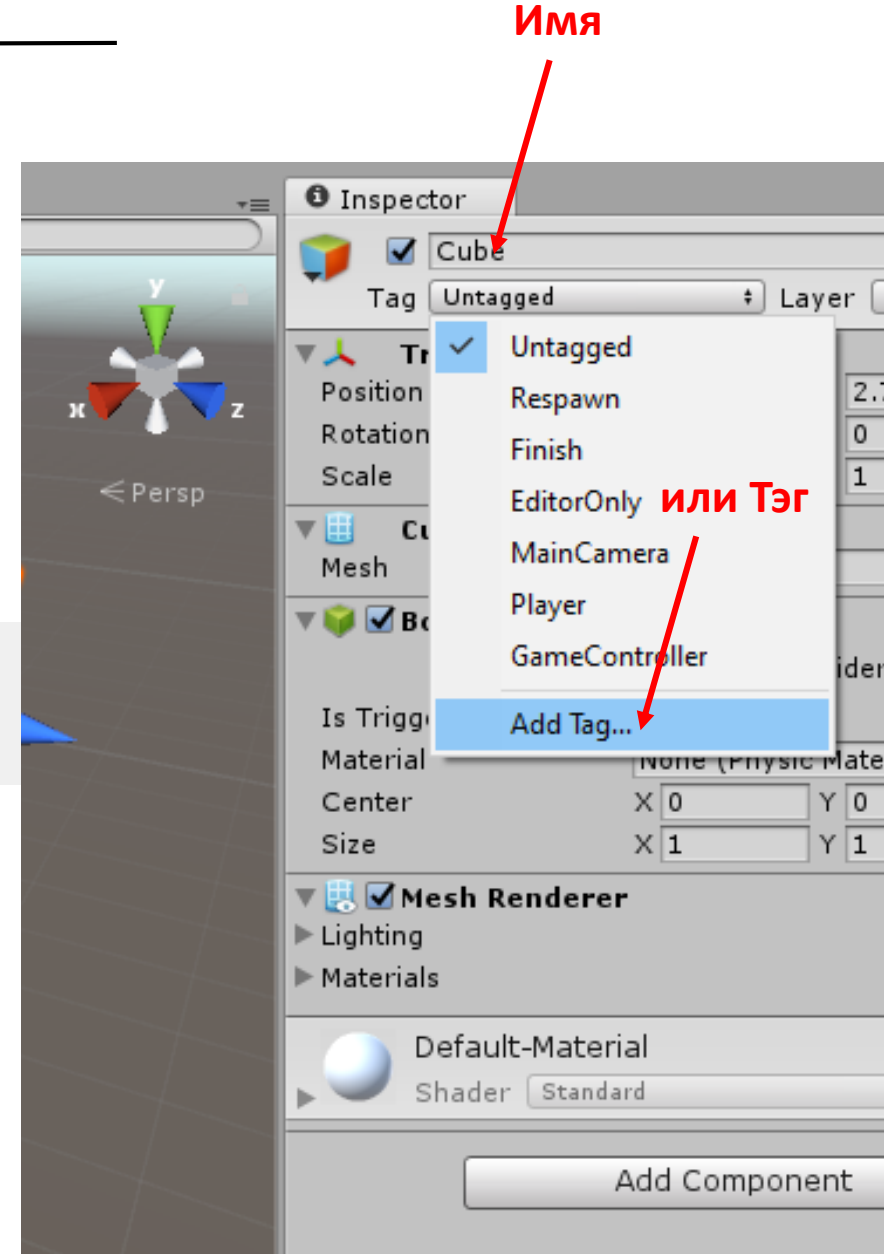
2. Нахождение объектов по имени или тегу

```
void Start() {  
    player = GameObject.Find("MainHeroCharacter");  
}
```

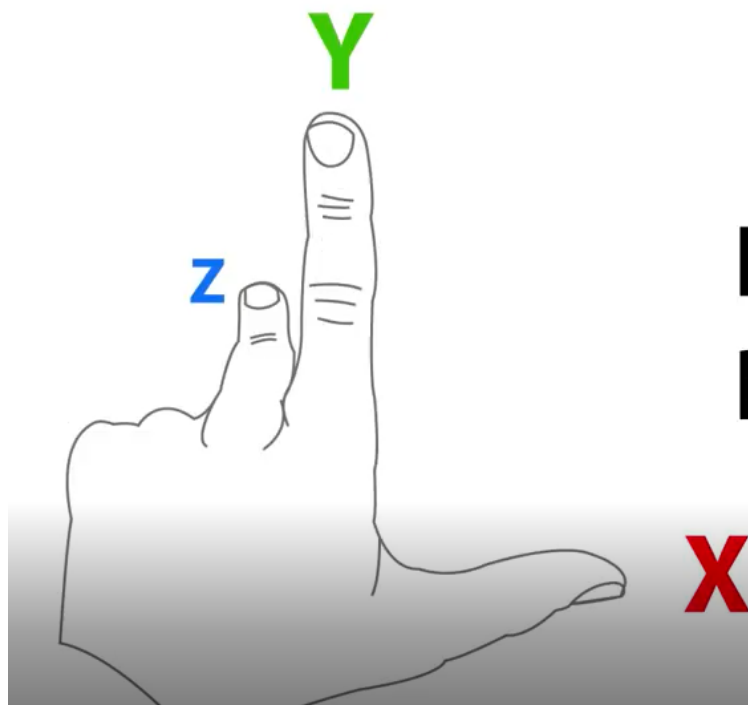
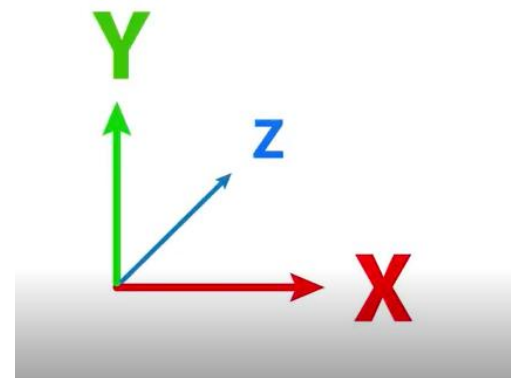
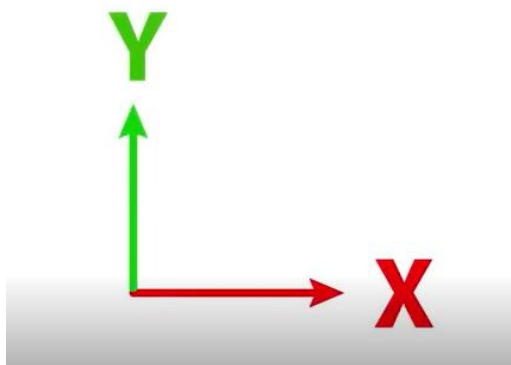
```
void Start() {  
    player = GameObject.FindWithTag("Player");  
}
```

Функция **GameObject.FindGameObjectWithTag** ищет и возвращает первый найденный объект с указанным тегом.

Функция **GameObject.FindObjectsWithTag** возвращает массив всех подходящих объектов.



Система координат

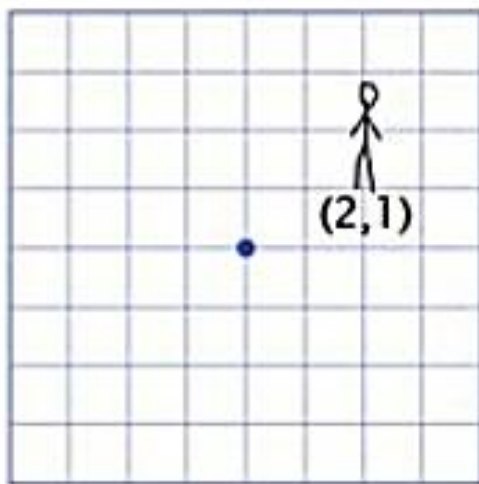


**Left Hand
Rule Coordinates**

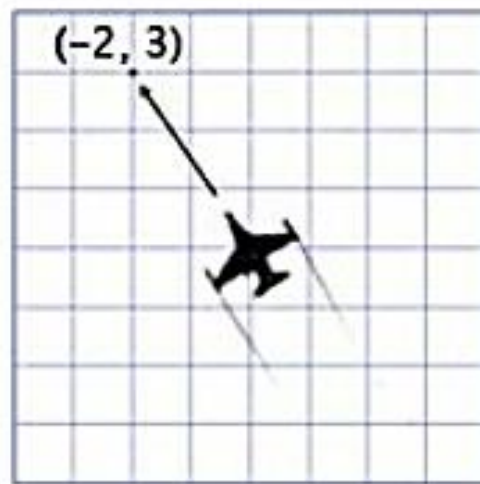
Что такое вектор?

Вектор сам по себе всего лишь набор цифр, который обретает тот или иной смысл в зависимости от контекста.

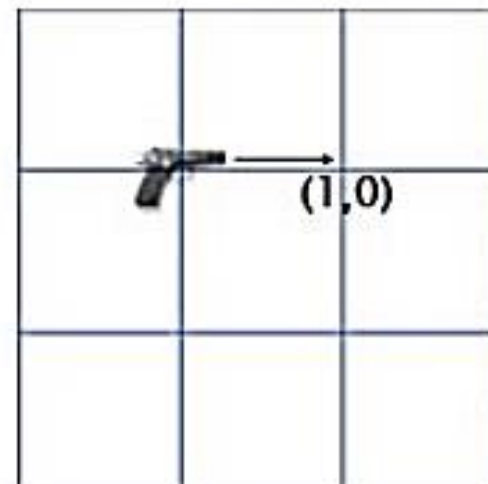
В играх вектора используются для хранения местоположений, направлений и скоростей. Ниже приведён пример двухмерного вектора:



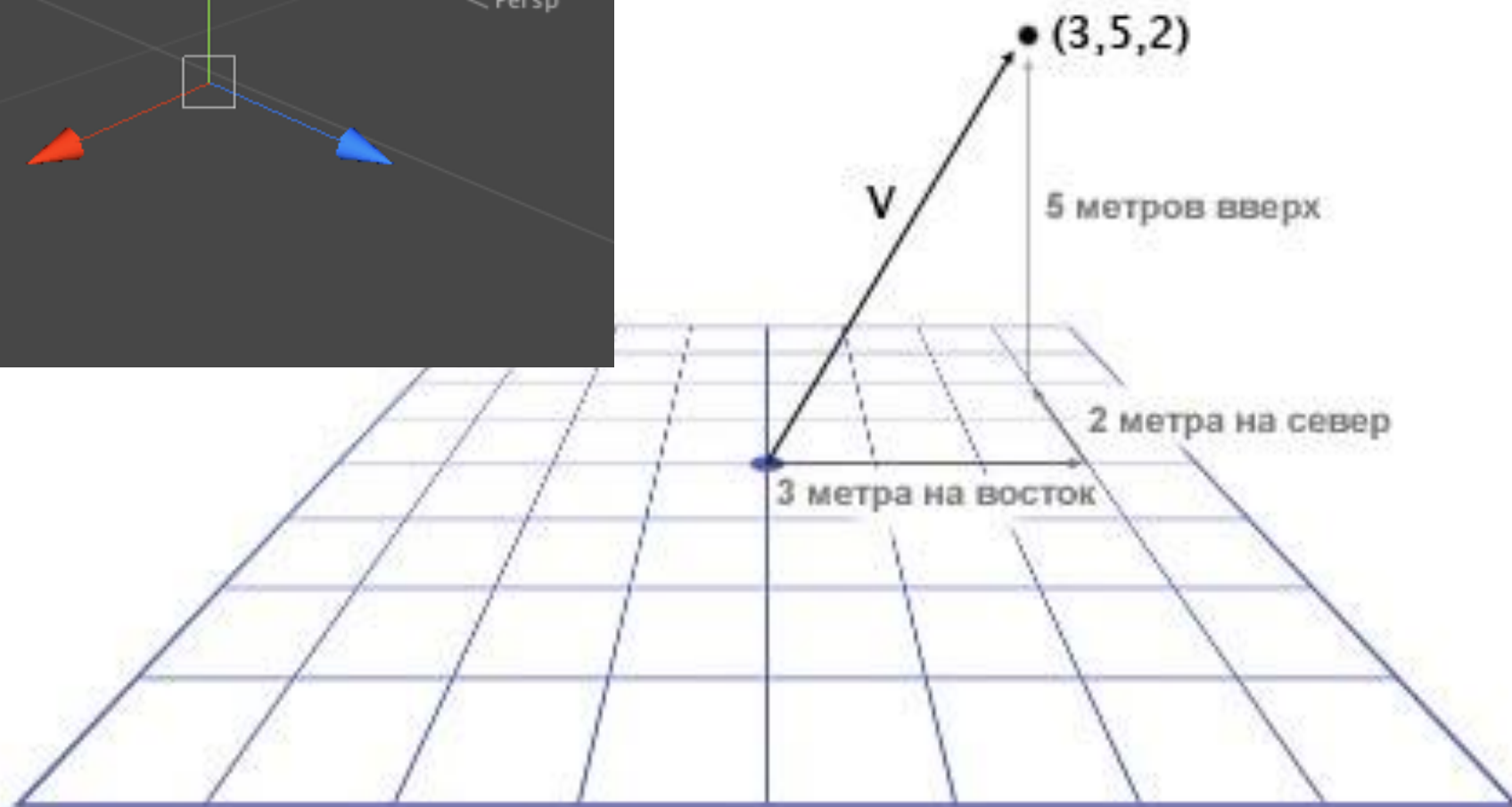
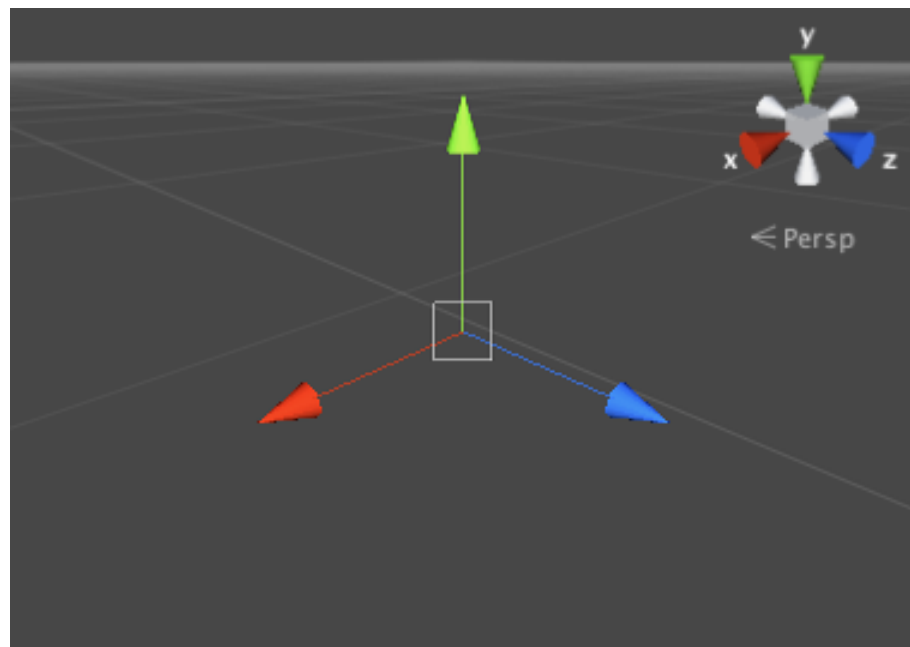
Местоположение



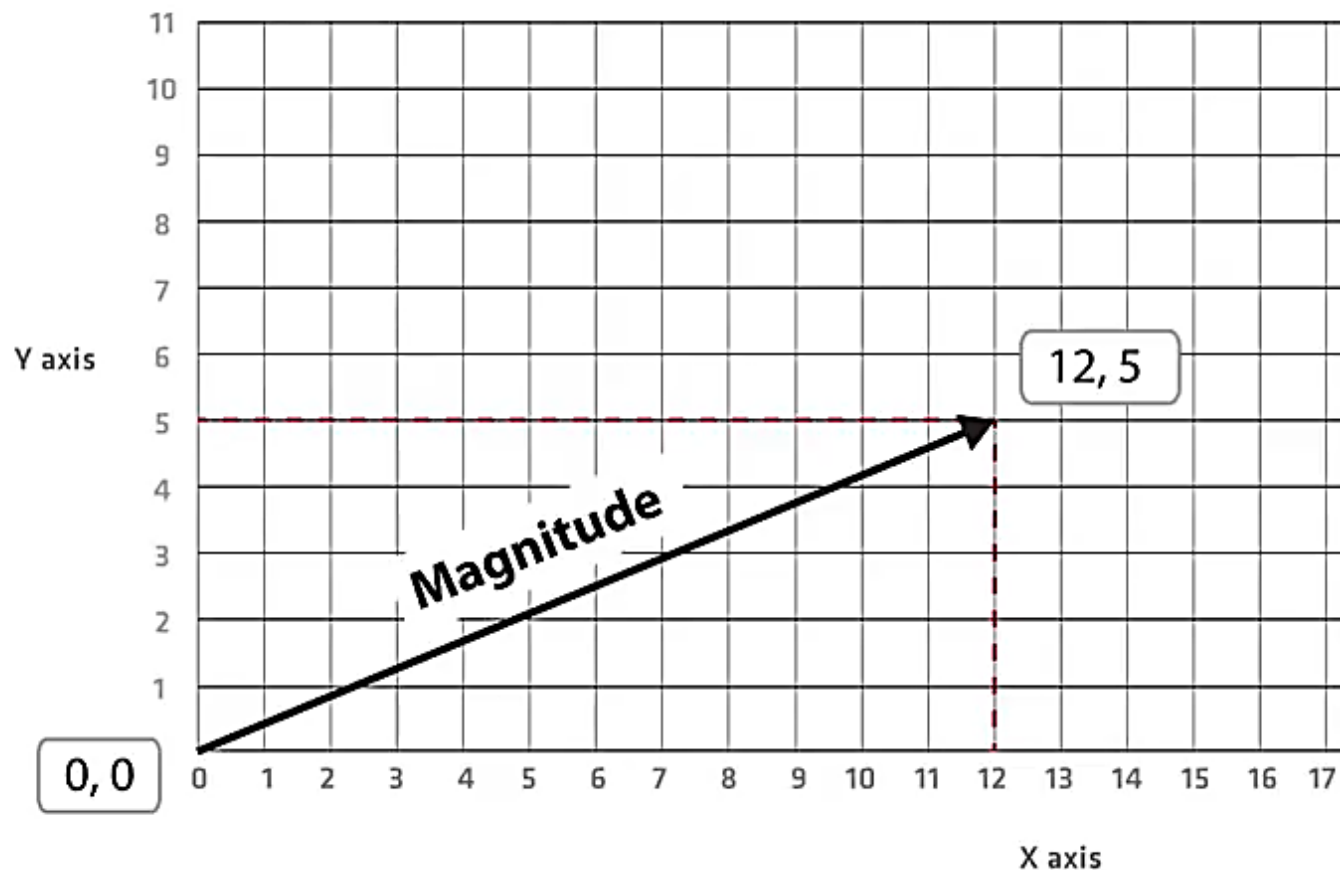
Скорость



Направление



Расстояние между объектами



$$x^2 + y^2 = m^2$$

$$\text{Magnitude} = \sqrt{x^2 + y^2}$$

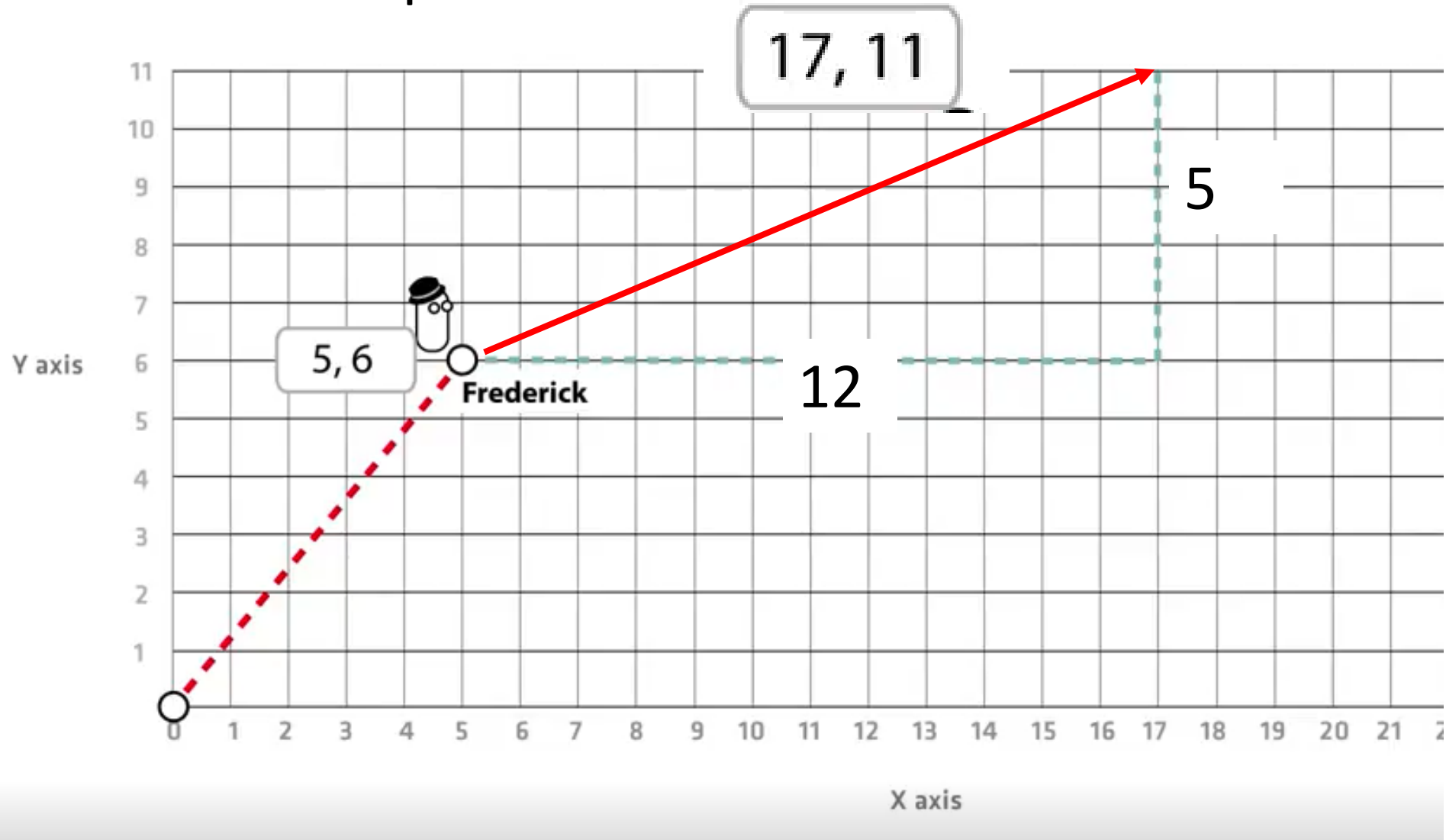
$$\text{Magnitude} = \sqrt{12^2 + 5^2}$$

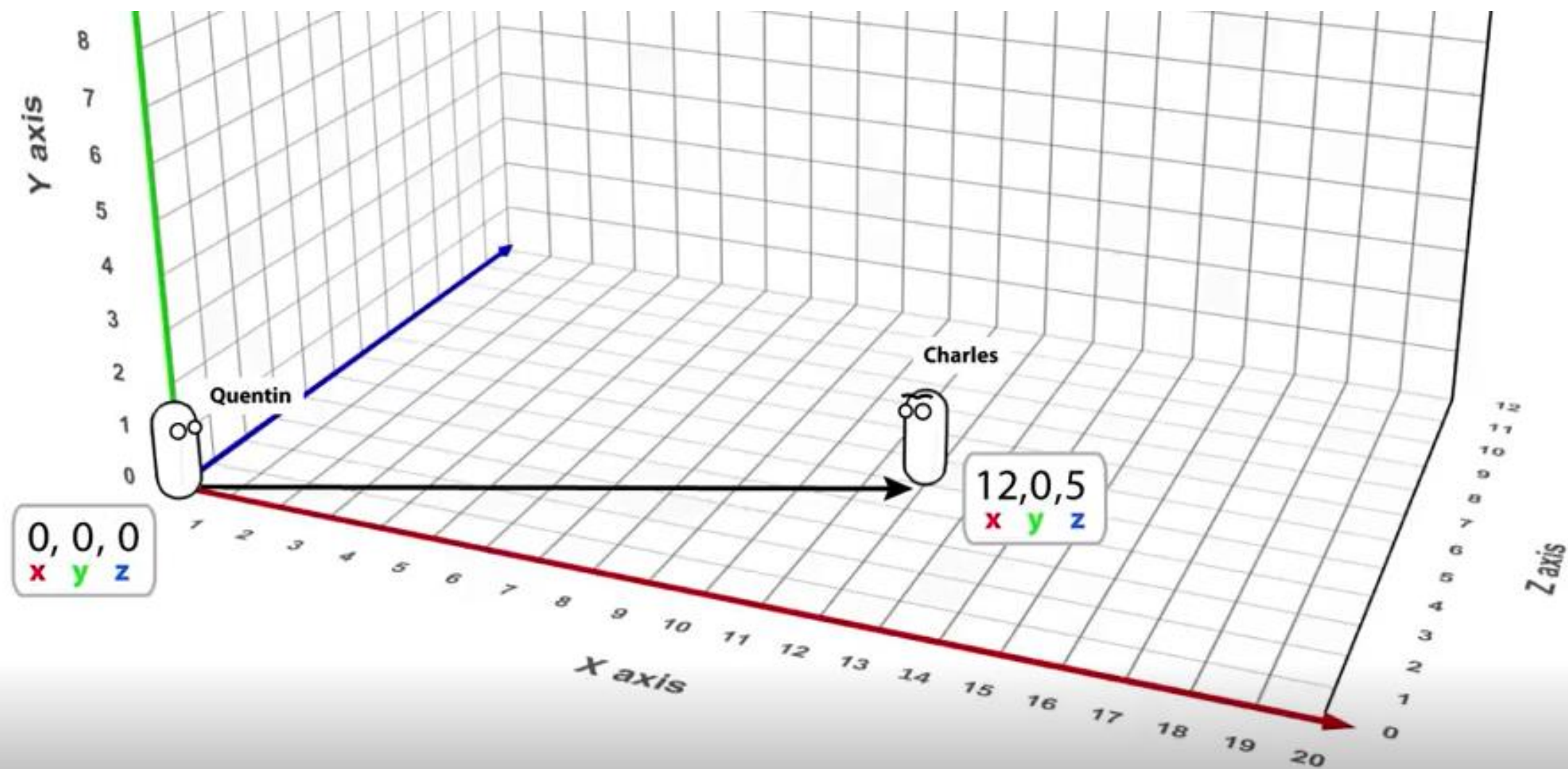
$$\text{Magnitude} = \sqrt{144 + 25}$$

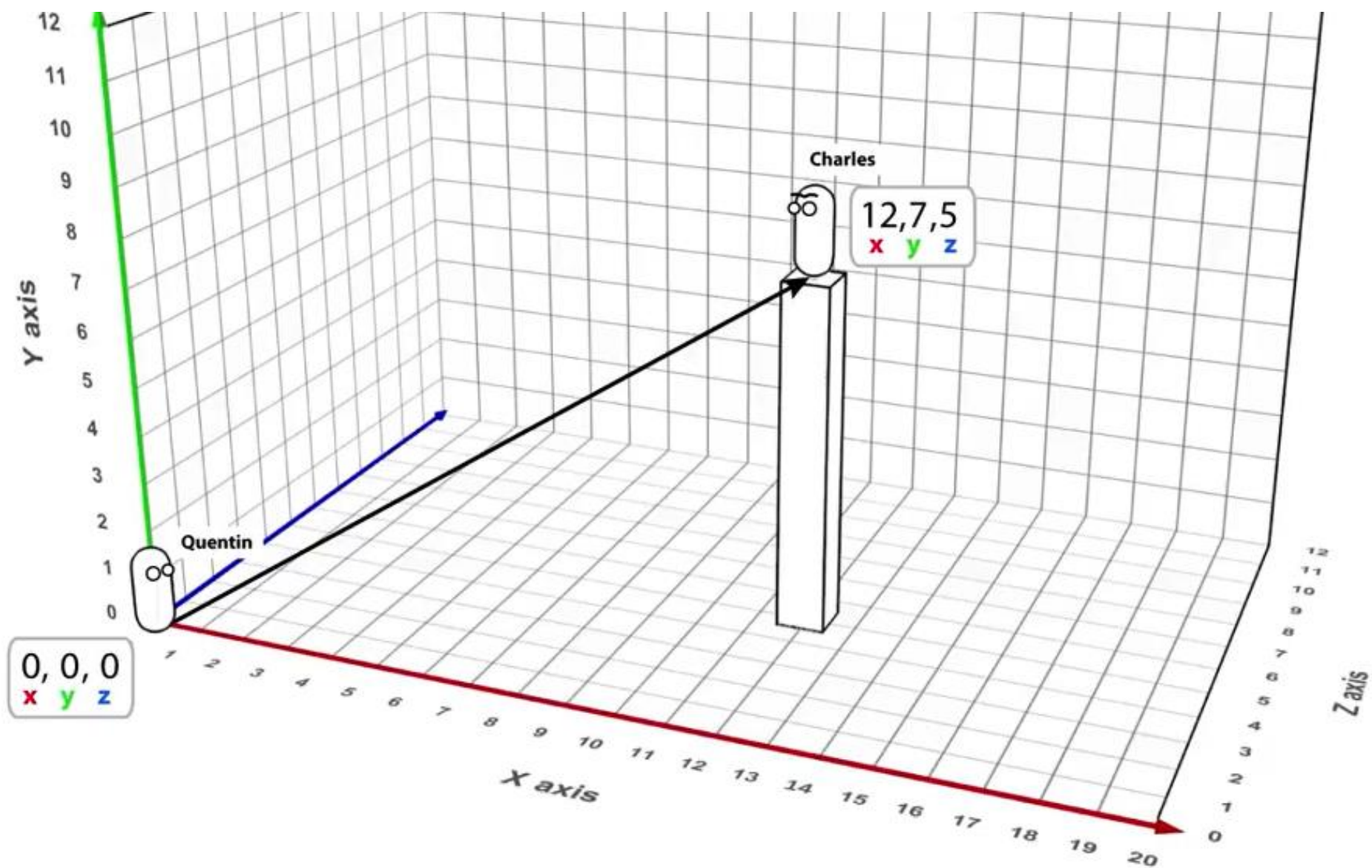
$$\text{Magnitude} = \sqrt{169}$$

$$\text{Magnitude} = 13$$

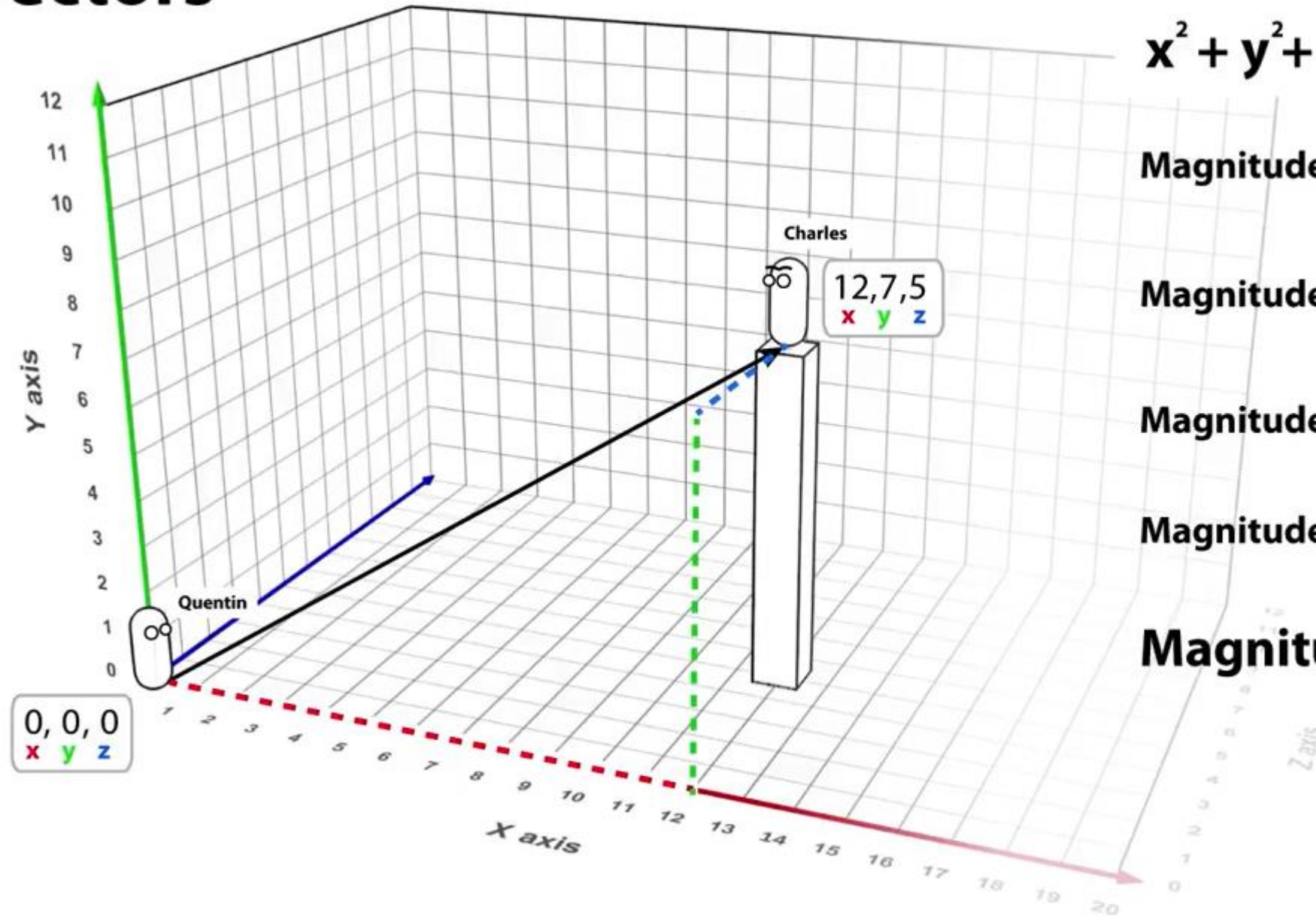
Скорость Фредерика (12,5) в час. Где он будет через час?
Сложение векторов







Vectors



$$x^2 + y^2 + z^2 = m^2$$

$$\text{Magnitude} = \sqrt{x^2 + y^2 + z^2}$$

$$\text{Magnitude} = \sqrt{12^2 + 7^2 + 5^2}$$

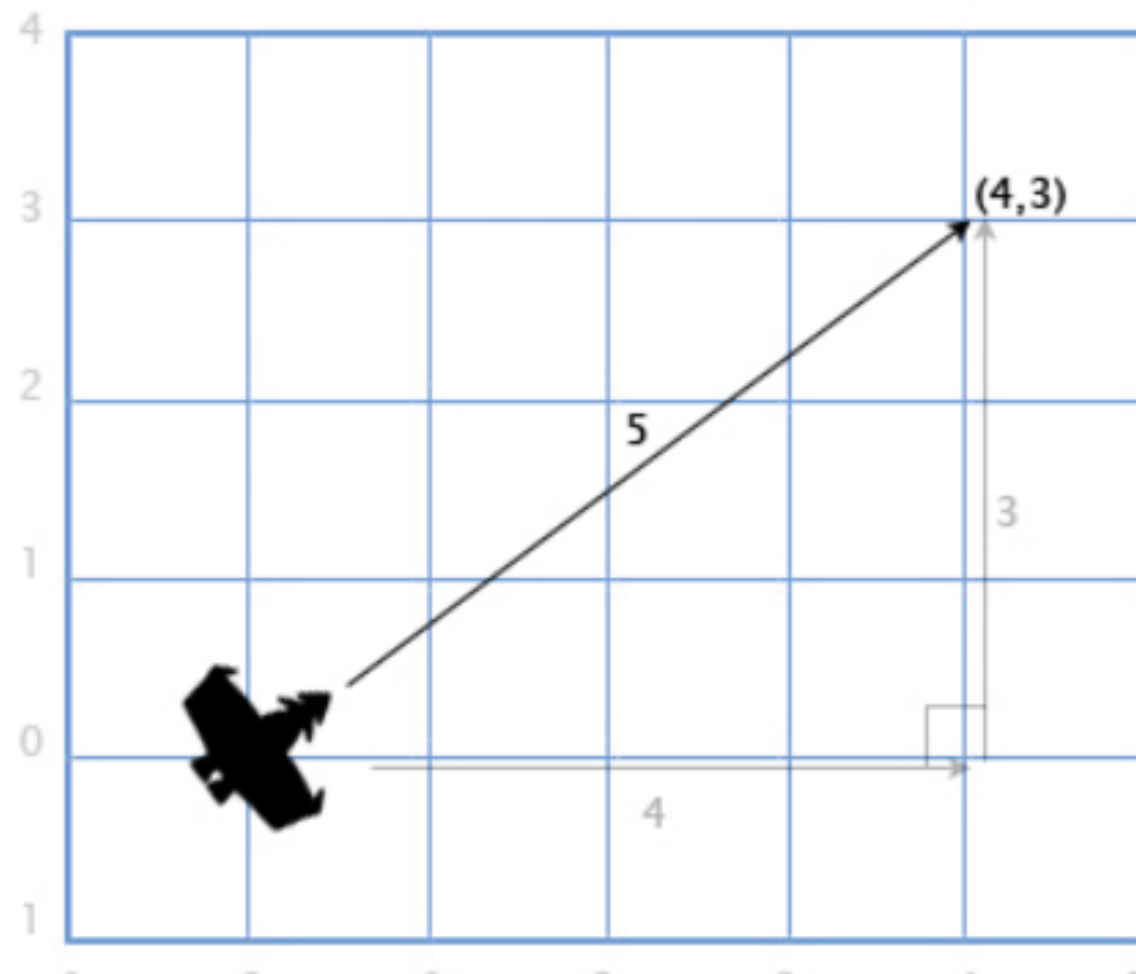
$$\text{Magnitude} = \sqrt{144 + 49 + 25}$$

$$\text{Magnitude} = \sqrt{218}$$

$$\text{Magnitude} = 14.76$$

Если у нас есть корабль с вектором скорости $V (4, 3)$, нужно узнать его скорость, чтобы посчитать потребность в экранном пространстве или сколько потребуется топлива.

Чтобы сделать это, нам понадобится найти длину (модуль) вектора V .



Расстояние между двумя объектами

1. Vector3.Distance(Vector3 a, Vector3 b)

```
public Transform box;  
  
private void Update()  
{  
    float dist = Vector3.Distance(box.position, transform.position);  
    Debug.Log(dist);  
}
```

2. Vector3.magnitude

```
float dist = (box.position - transform.position).magnitude;
```

Расстояние между двумя объектами. Оптимизация

Использование метода **Vector3.Distance** или свойства **Vector3.magnitude** имеет один недостаток.

Дело в том, что при расчете надо извлечь корень. Извлечение корня является достаточно затратной операцией и при частом вызове для большого числа объектов может привести к падению производительности.

В данном случае в качестве оптимизации можно использовать свойство **Vector3.sqrMagnitude**. Оно **вернет квадрат расстояния**, что вычисляется быстрее простого расстояния благодаря отсутствию операции извлечения корня.


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

[ExecuteInEditMode]

```
public class DistanceBetweenTwoObjects : MonoBehaviour
```

```
{
```

```
    public GameObject target1;
```

```
    public GameObject target2;
```

```
    public float distanceX;
```

```
    public float distanceY;
```

```
    public float distanceZ;
```

```
    public float distanceTotal;
```

```
    void Update()
```

```
    {
```

```
        Vector3 delta = target2.transform.position - target1.transform.position;
```

```
        distanceX = delta.x;
```

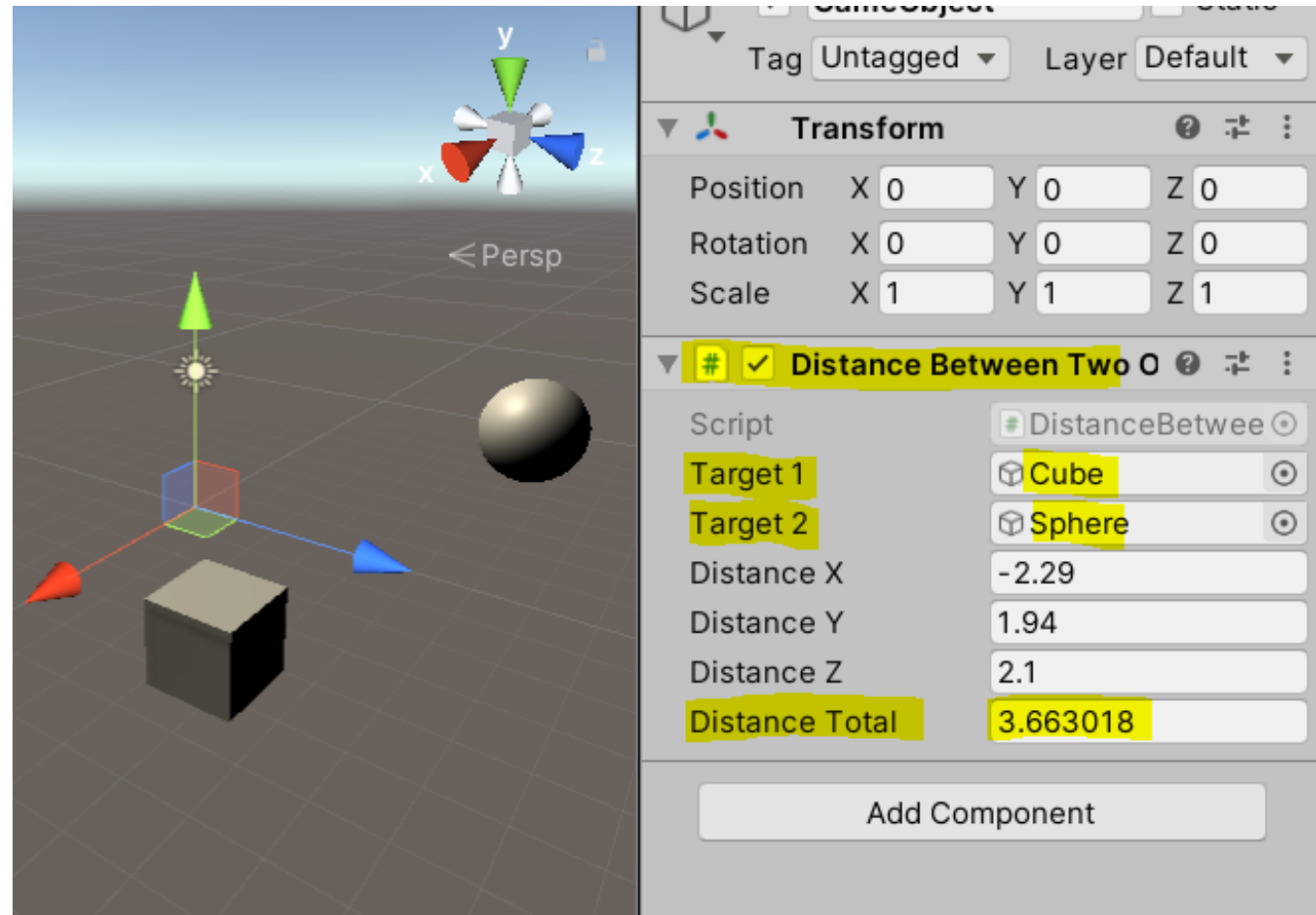
```
        distanceY = delta.y;
```

```
        distanceZ = delta.z;
```

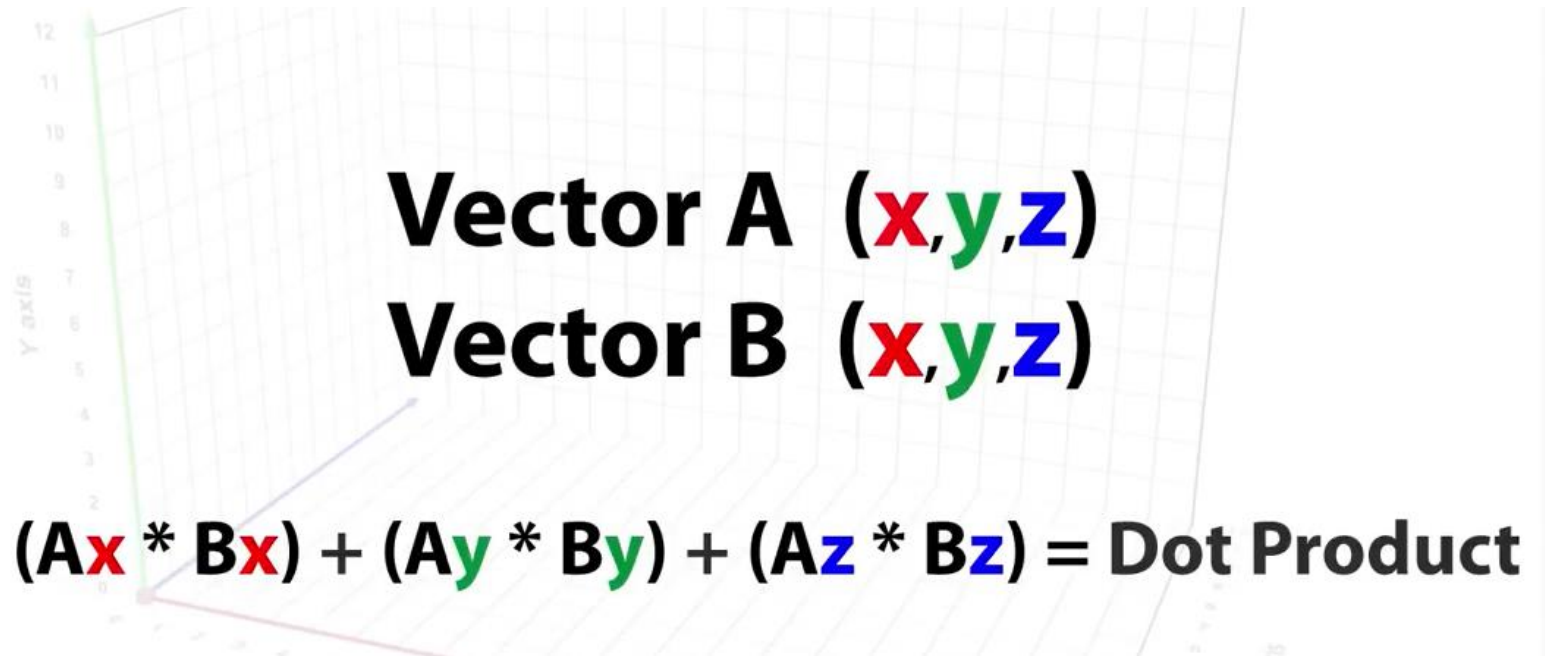
```
        distanceTotal = delta.magnitude;
```

```
    }
```

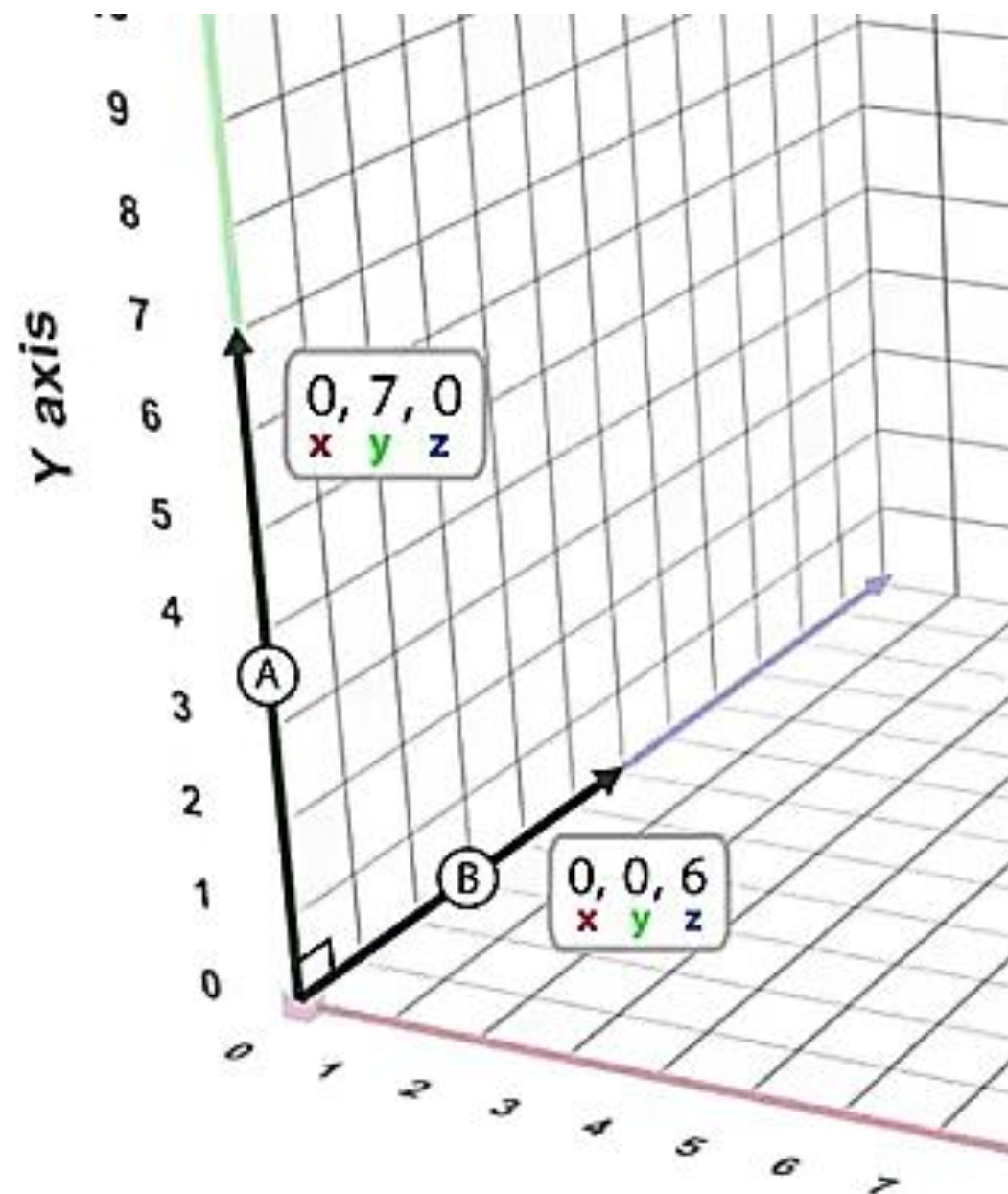
```
}
```



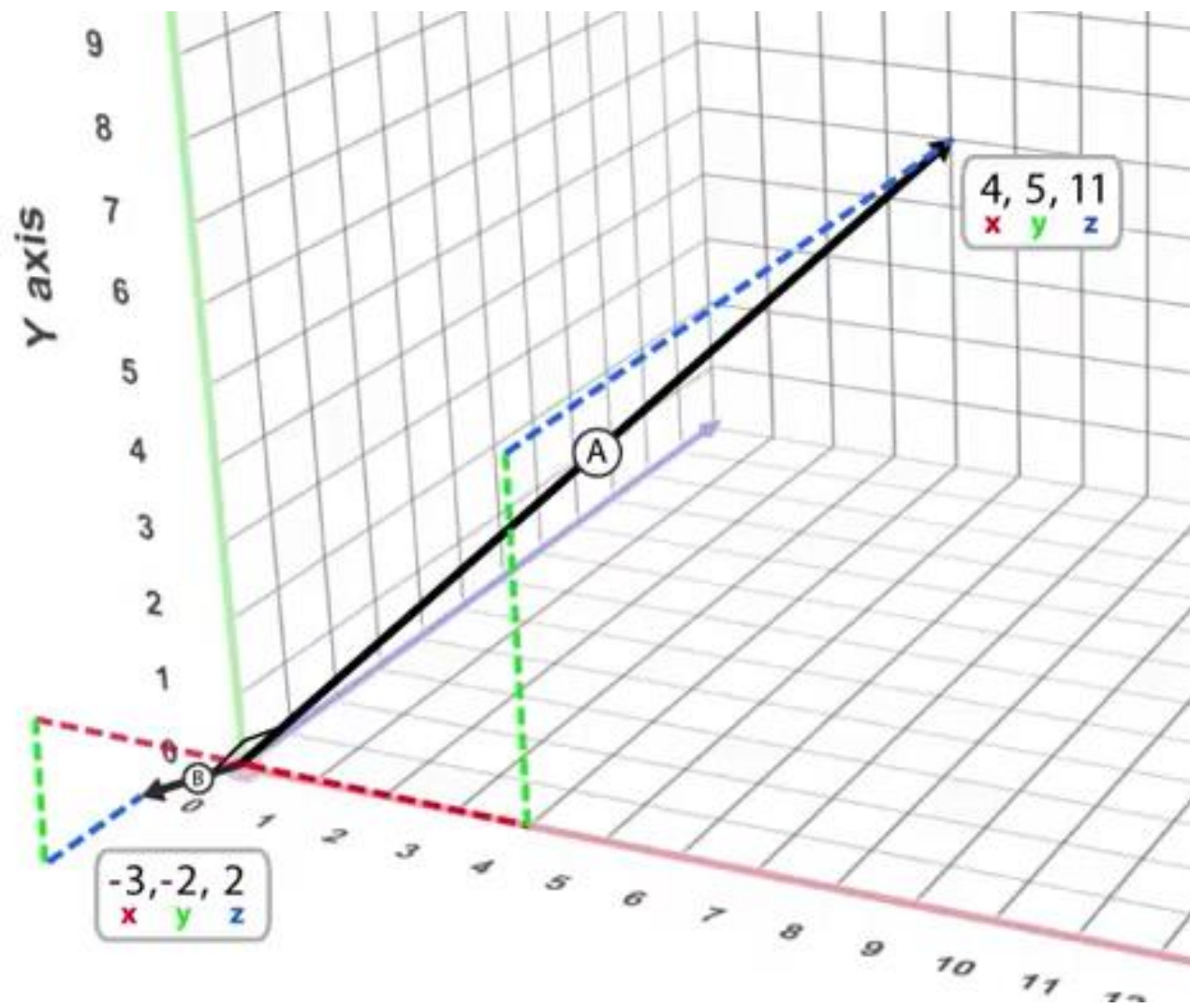
Скалярное произведение



Если скалярное произведение двух векторов равно 0, значит они перпендикулярны

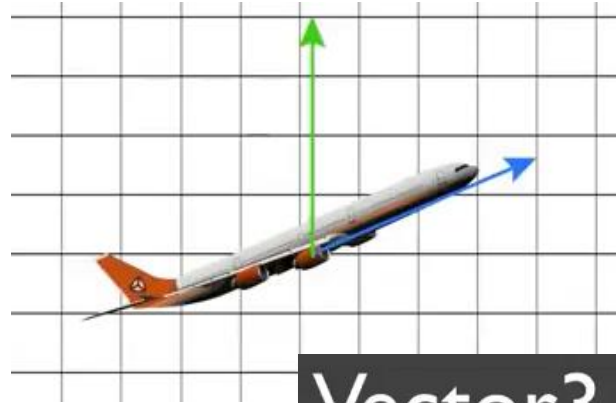
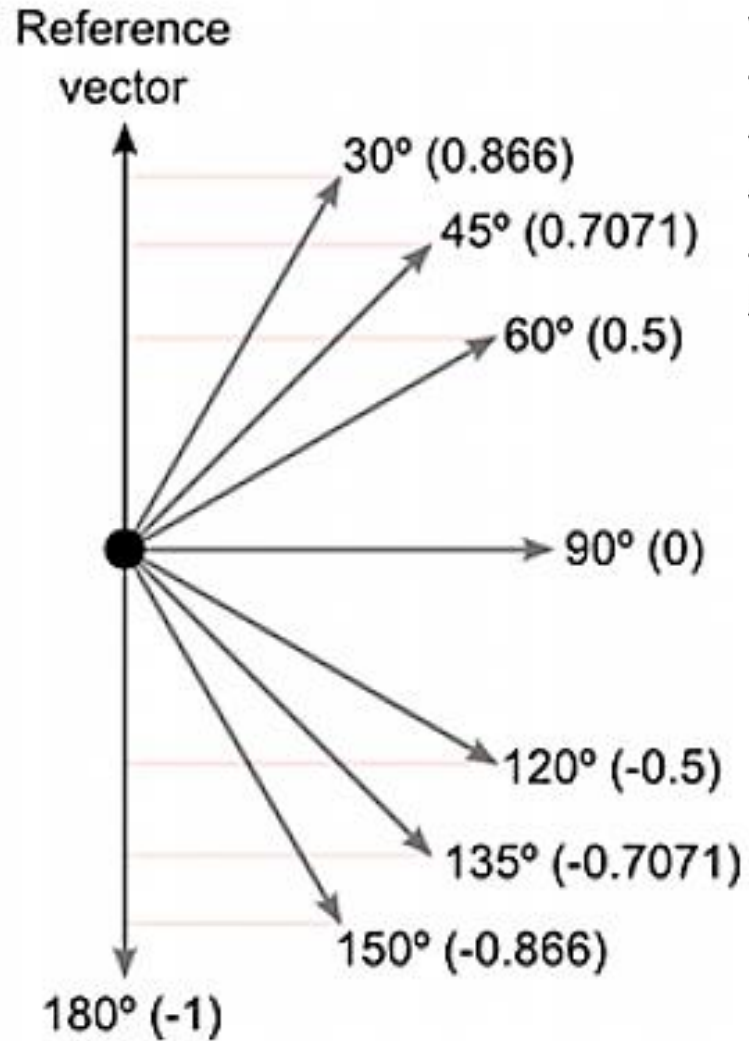


	A • B				
X	0	*	0	=	0
Y	7	*	0	=	0
Z	0	*	6	=	0
Total					0

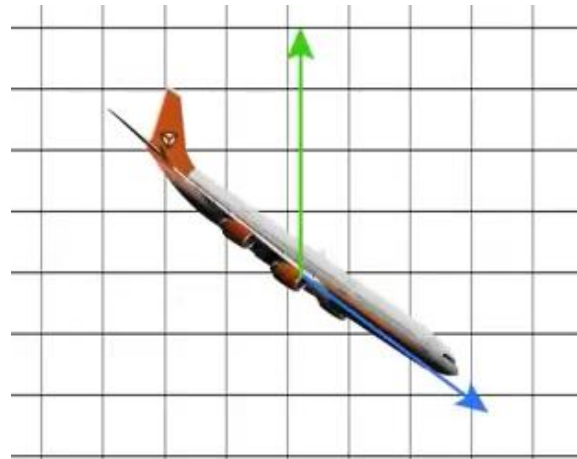


	A • B			
X	4	*	-3	= -12
Y	5	*	-2	= -10
Z	11	*	2	= 22
Total				0

Скалярное произведение равно произведению величин этих векторов, умноженному на косинус угла между ними.



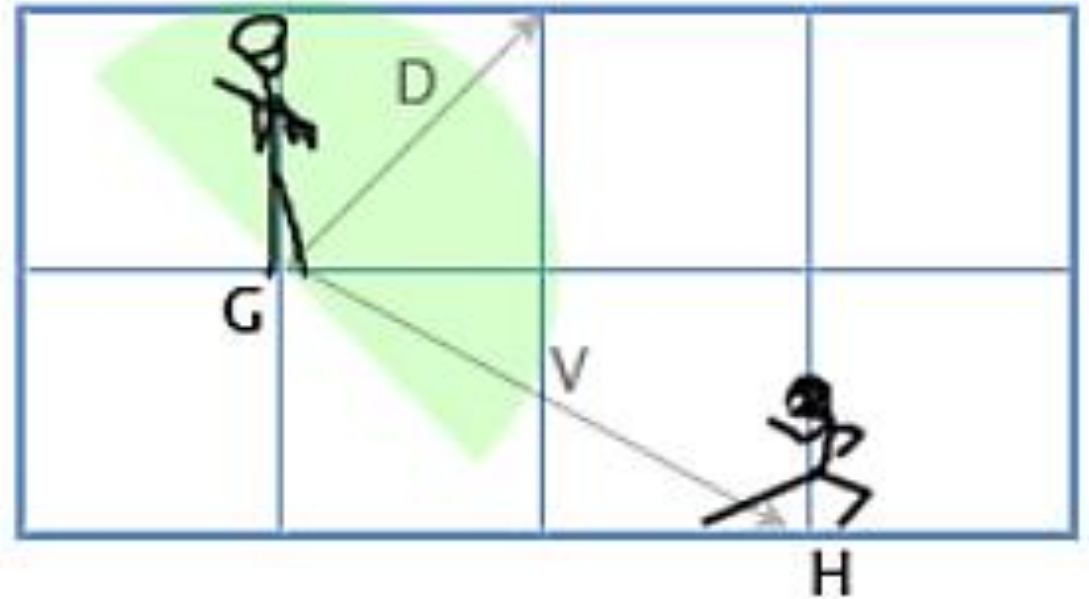
`Vector3.Dot(VectorA, VectorB)`



Если вектора указывают **в одном направлении**, то их скалярное произведение **больше нуля**. Когда они перпендикулярны друг другу, то скалярное произведение равно нулю. И когда они указывают **в противоположных направлениях**, их скалярное произведение меньше нуля.

Пример: Допустим у нас есть стражник, расположенный в $G(1, 3)$ смотрящий в направлении $D(1,1)$, с углом обзора 180 градусов. Главный герой игры подсматривает за ним с позиции $H(3, 2)$.

Как определить, находится-ли главный герой в поле зрения стражника или нет? Сделаем это путём скалярного произведения векторов D и V (вектора, направленного от стражника к главному герою). Мы получим следующее:

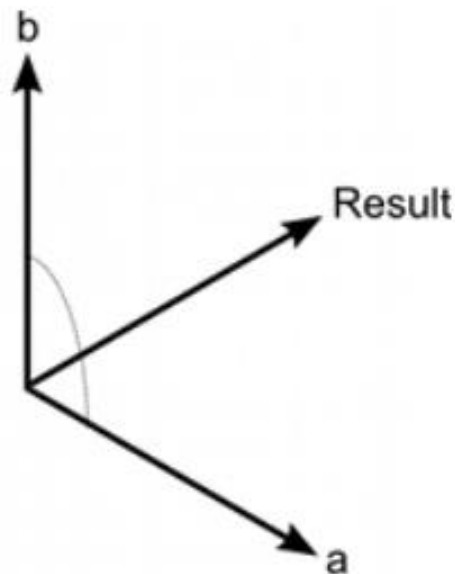


$$V = H - G = (3, 2) - (1, 3) = (3-1, 2-3) = (2, -1)$$

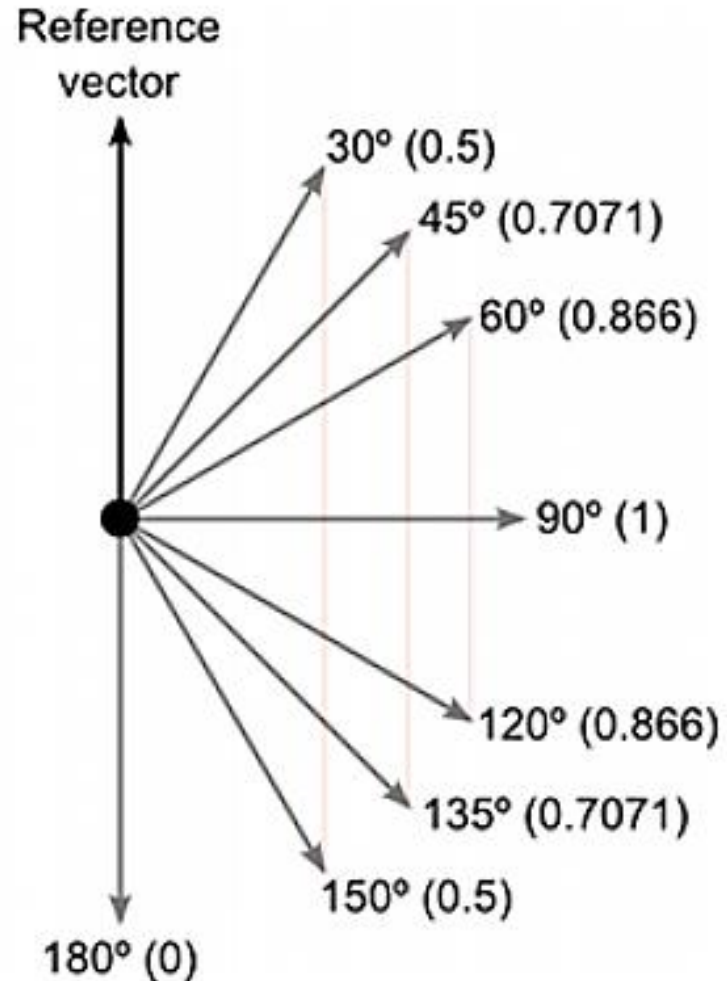
$$D \cdot V = (1, 1) \cdot (2, -1) = 1 \cdot 2 + 1 \cdot (-1) = 2 - 1 = 1$$

Векторное произведение

Итоговый вектор перпендикулярен двум исходным векторам. Можно использовать “правило левой руки”, чтобы запомнить направление выходного вектора относительно исходных векторов. Если первый параметр совпадает с большим пальцем руки, а второй параметр с указательным пальцем, то результат будет указывать в направлении среднего пальца.



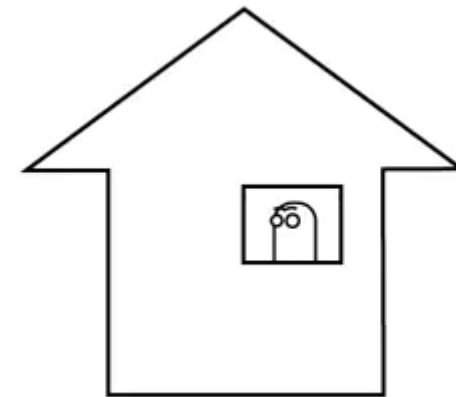
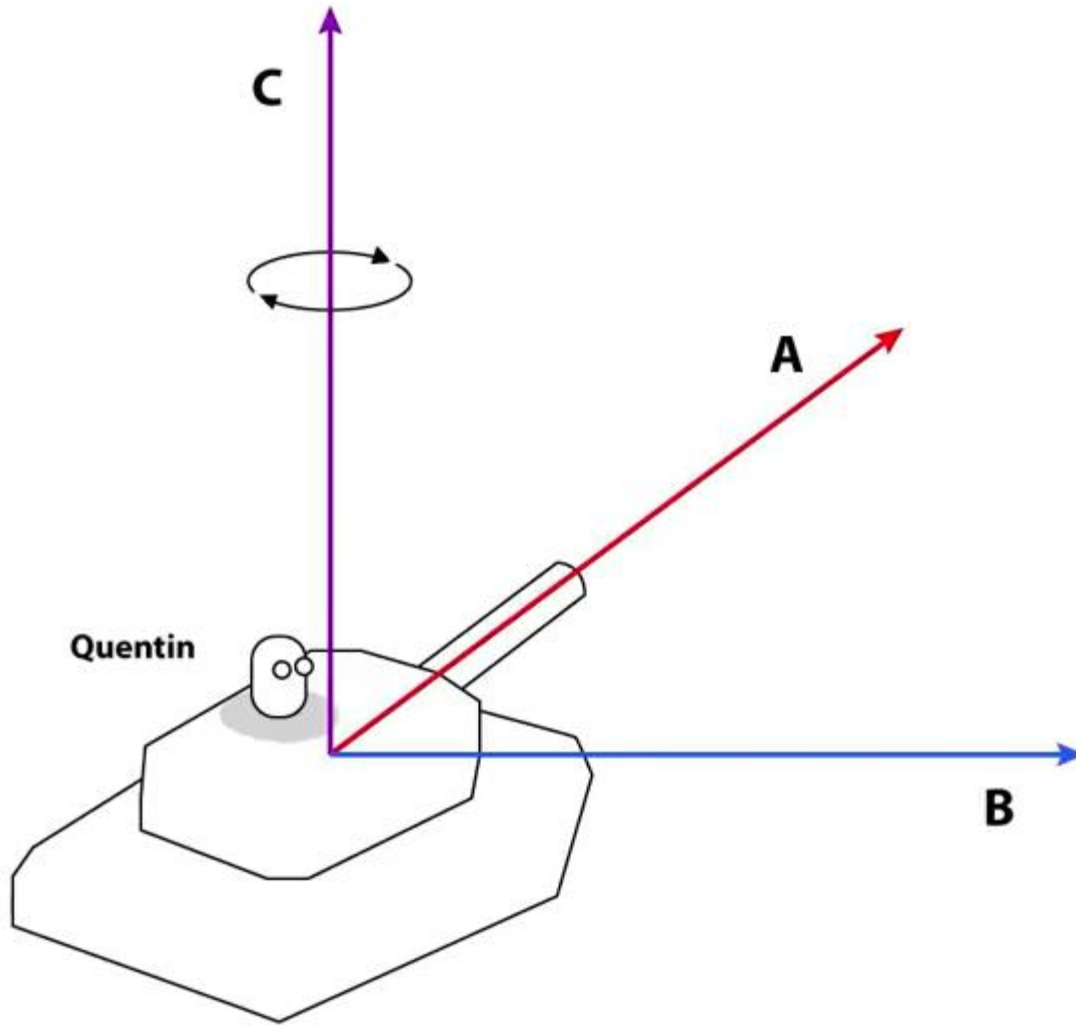
Векторное произведение



Величина результата равна произведению величин исходных векторов, умноженному на синус угла между ними.

```
Vector3.Cross(VectorA, VectorB)
```


Векторное произведение. Применение



Прямолинейное движение

```
transform.position = new Vector3 (posX + speed, posY, posZ);
```

Метод **Translate()** двигает объект вдоль вектора на расстояние равное его длине (параллельный перенос).

```
transform.Translate(new Vector3(0.0f, 0.0f, 1.0f))
```



Что произойдет в результате выполнения этого кода?

- Не подходит для Rigidbody;
- Is Kinematic (кинематическое движение).

Прямолинейное движение

Объекту с компонентом **Rigidbody** нужно задавать движение с помощью приложения силы (передав таким образом все расчеты движения физическому движку игры). В противном случае вы можете начать конфликтовать с физикой объекта.

AddForce() добавляет Rigidbody силу вдоль направления вектора силы. **ForceMode** позволяет изменить тип силы на ускорение, импульс или изменение скорости.

```
void FixedUpdate()
{
    rb.AddForce(0,0, thrust, ForceMode.Impulse);
}
```

Вращение

Вращения в 3D-приложениях обычно представлены одним из двух способов: **кватернионами или углами Эйлера**. У каждого есть свои достоинства и недостатки.

углы Эйлера

- ✓ **Преимущество** : углы Эйлера имеют интуитивно понятный формат, состоящий из трех углов.
- ✓ **Ограничение** : углы Эйлера страдают от [Gimbal Lock](#) . При применении трех вращений по очереди, первое или второе вращение может привести к тому, что третья ось будет указывать в том же направлении, что и одна из предыдущих осей. Это означает, что «степень свободы» была потеряна, потому что третье значение вращения не может быть применено вокруг уникальной оси.

кватернионы

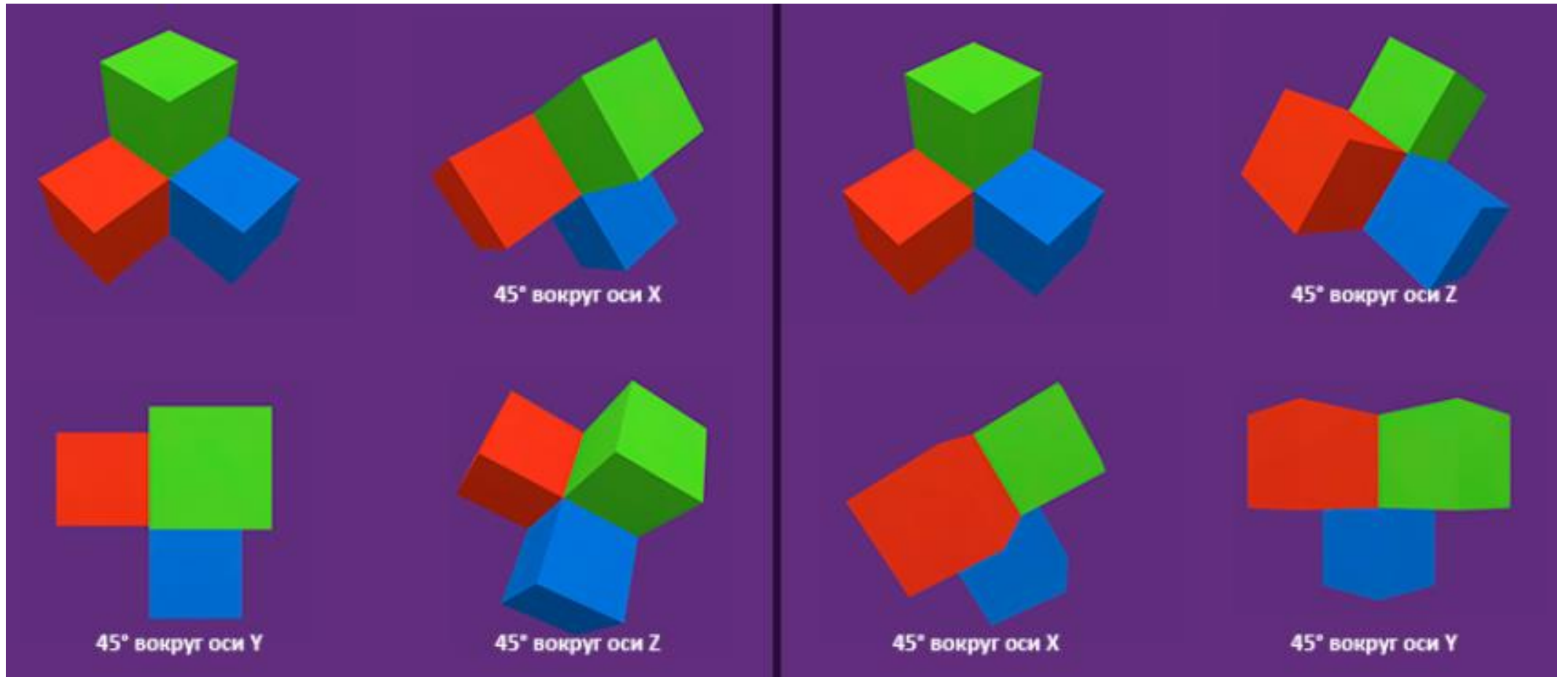
- ✓ **Преимущество** : вращения кватернионов не страдают от блокировки кардана.
- ✓ **Ограничение** : числовое представление Quaternion интуитивно непонятно.

В Unity все вращения игровых объектов хранятся внутри как кватернионы, потому что преимущества перевешивают ограничения.

Однако в Инспекторе преобразований мы отображаем вращение с использованием углов Эйлера, потому что это легче понять и отредактировать. Новые значения, введенные в инспектор для поворота игрового объекта, преобразуются «под капотом» в новое значение поворота Quaternion для объекта.

Углы Эйлера

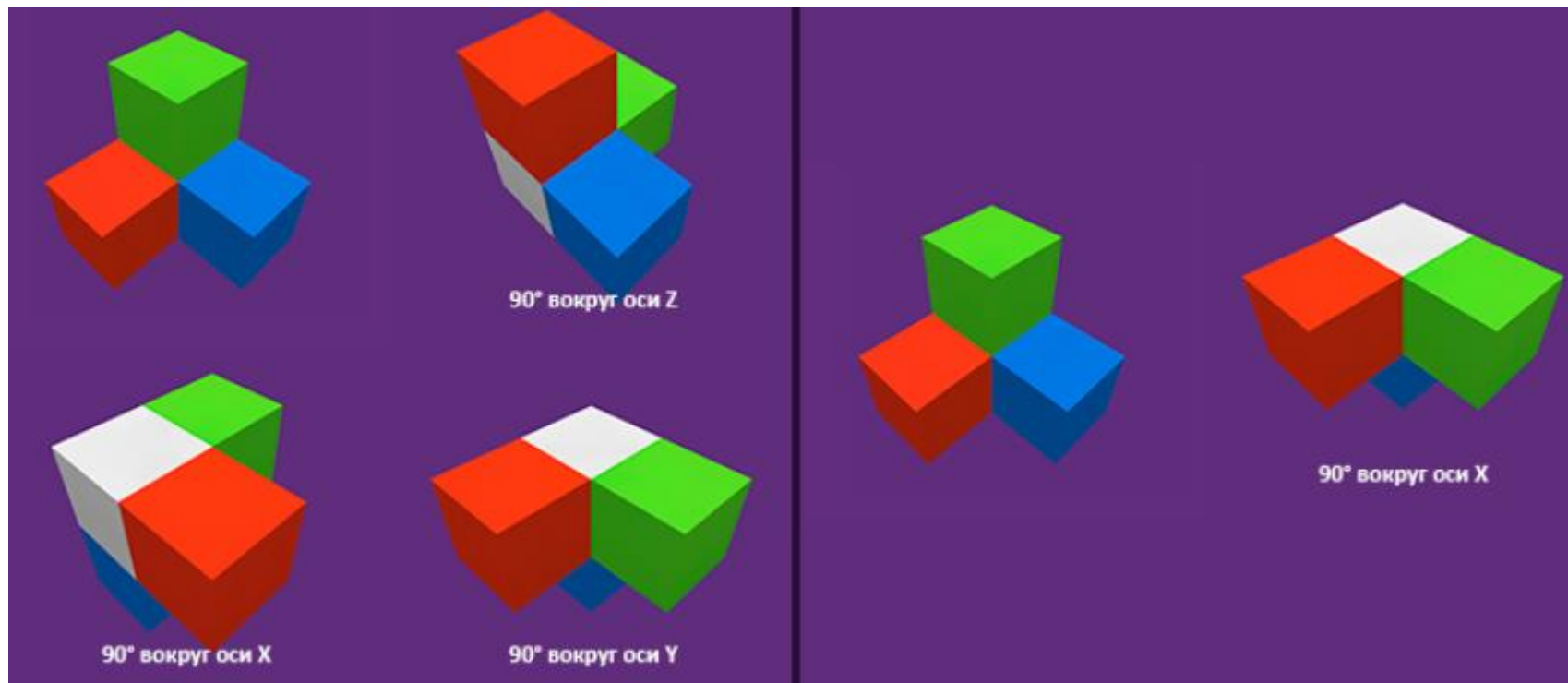
1. Результат поворота зависит от порядка поворотов по осям



Углы Эйлера

2. Шарнирный замок

Если вращение по вокруг оси X будет равно 90° или -90° , то вращения вокруг Z и Y компенсируют друг друга. Например $(90^\circ, 90^\circ, 90^\circ)$ превратится в $(90^\circ, 0^\circ, 0^\circ)$.



$(90^\circ, 130^\circ, 140^\circ)$, или $(90^\circ, 130^\circ, 140^\circ)$, или $(90^\circ, 30^\circ, 40^\circ) = (90^\circ, 0^\circ, 10^\circ)$

[демо](#)

Кватернионы

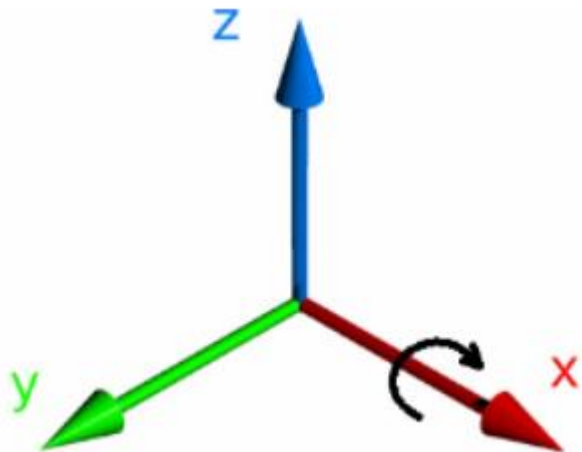
Кватернионы — система гиперкомплексных чисел, образующая векторное пространство размерностью четыре над полем вещественных чисел.

Для разработчика — это прежде всего инструмент, описывающий действие — поворот вокруг оси на заданный угол:

(w, vx, vy, vz) ,

где v — ось, выраженная вектором;

w — компонента, описывающая поворот (косинус половины угла).



$(0.7071, 0.7071, 0, 0)$

[демо](#)

Имея дело с обработкой поворотов в скриптах, рекомендуется использовать класс **Quaternion** и его функции для создания и изменения значений вращения.

Есть некоторые ситуации, когда допустимо использовать углы Эйлера, но лучше использовать функции класса Quaternion, которые имеют дело с углами Эйлера. Получение, изменение и повторное применение значений Эйлера из вращения может вызвать непреднамеренные побочные эффекты.

Вот пример **ошибки** при использовании гипотетического примера попытки повернуть объект вокруг оси X со скоростью 10 градусов в секунду. Вот чего следует *избегать* :

```
void Update () {  
    var rot = transform.rotation;  
    rot.x += Time.deltaTime * 10;  
    transform.rotation = rot;  
}
```

А вот пример правильного использования углов Эйлера в скрипте :

```
float x;  
void Update () {  
    x += Time.deltaTime * 10;  
    transform.rotation = Quaternion.Euler(x,0,0);  
}
```


Создание и уничтожение игровых объектов

```
public GameObject enemy;  
  
void Start() {  
    for (int i = 0; i < 5; i++) {  
        Instantiate(enemy);  
    }  
}
```

Instantiate() делает копию существующего объекта

Destroy() уничтожает объект

```
void OnCollisionEnter(Collision otherObj) {  
    if (otherObj.gameObject.tag == "Missile") {  
        Destroy(gameObject, .5f);  
    }  
}
```

Destroy(this);

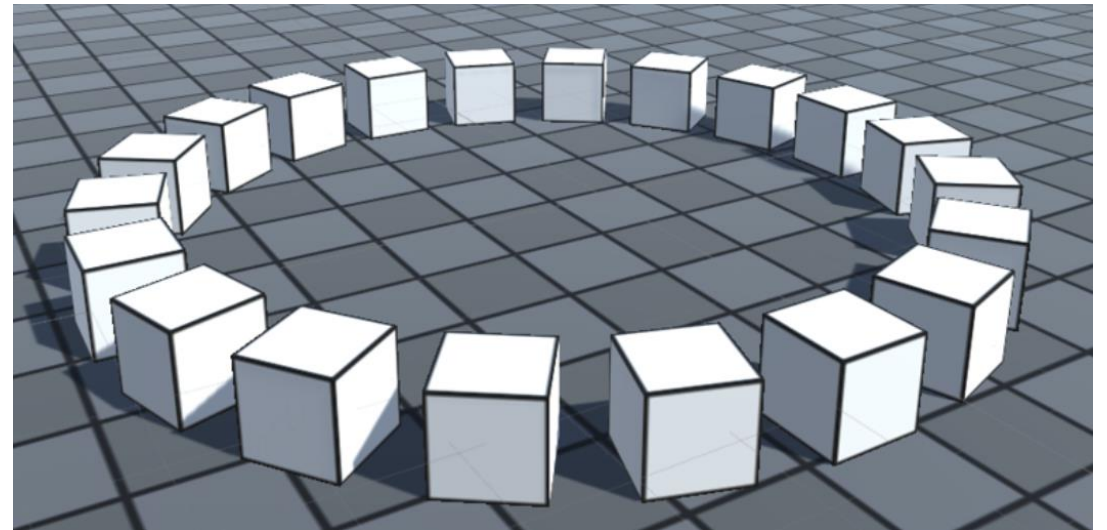
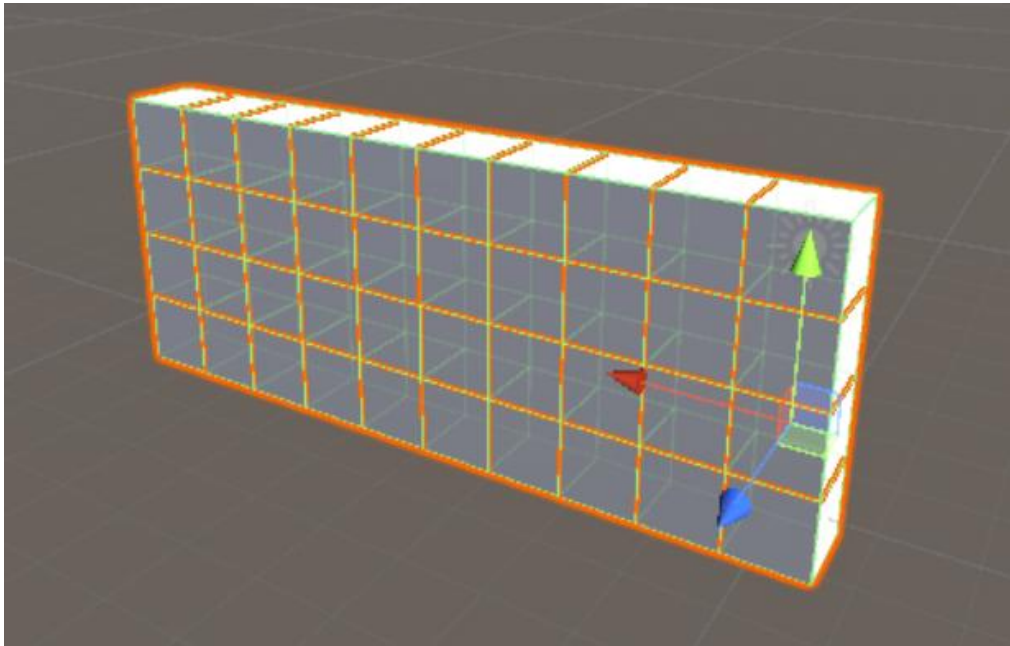
Процедурная генерация префабов

```
public class InstantiationExample : MonoBehaviour
{
    public GameObject myPrefab;

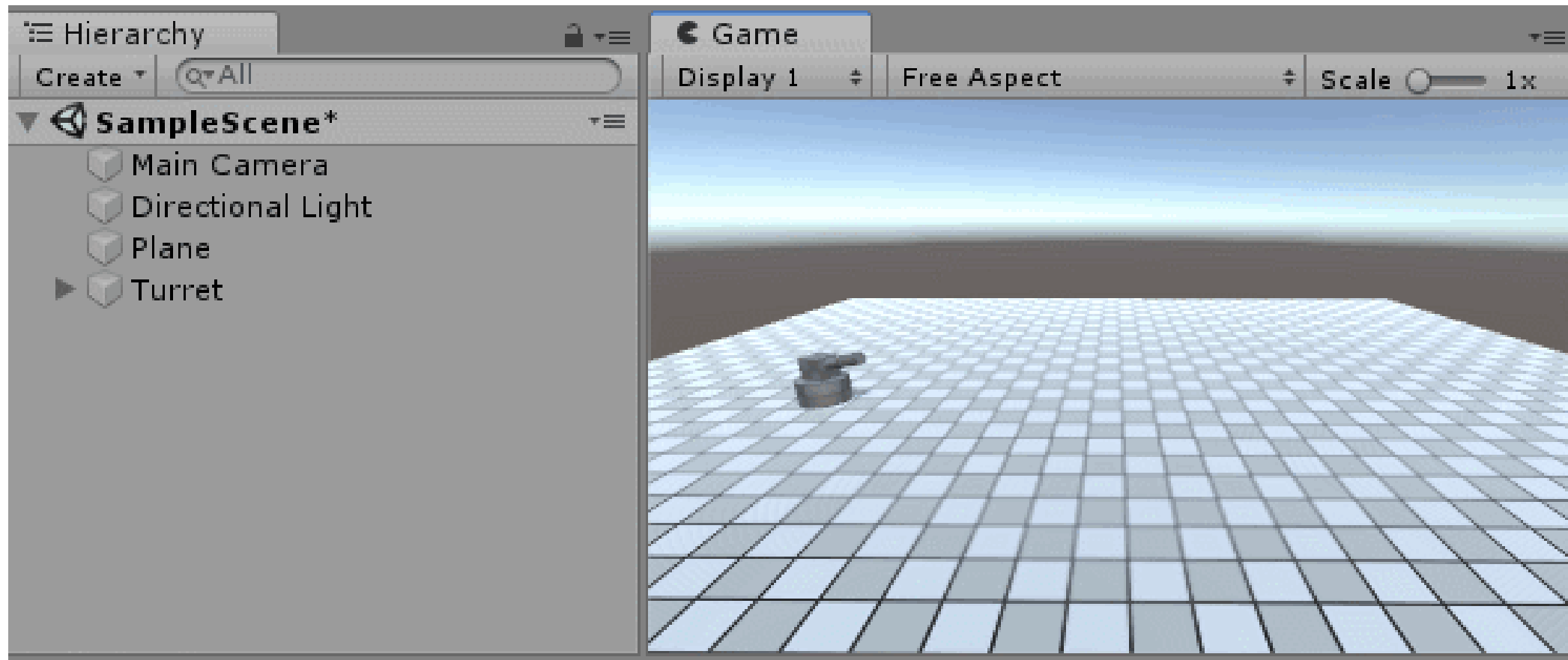
    // This script will simply instantiate the Prefab when the game starts.
    Ссылка: 0
    void Start()
    {
        // Instantiate at position (0, 0, 0) and zero rotation.
        Instantiate(myPrefab, new Vector3(0, 0, 0), Quaternion.identity);
    }
}
```

Процедурная генерация префабов

```
for (int y = 0; y < height; ++y)
{
    for (int x = 0; x < width; ++x)
    {
        Instantiate(myPrefab, new Vector3(x, y, 0), Quaternion.identity);
    }
}
```



Процедурная генерация префабов (снаряд)



Процедурная генерация префабов (снаряд)

```
public class FireProjectile : MonoBehaviour
{
    public Rigidbody projectile;
    public float speed = 4;
    void Update()
    {
        if (Input.GetButtonDown("Fire1"))
        {
            Rigidbody p = Instantiate(projectile, transform.position, transform.rotation);
            p.velocity = transform.forward * speed;
        }
    }
}
```

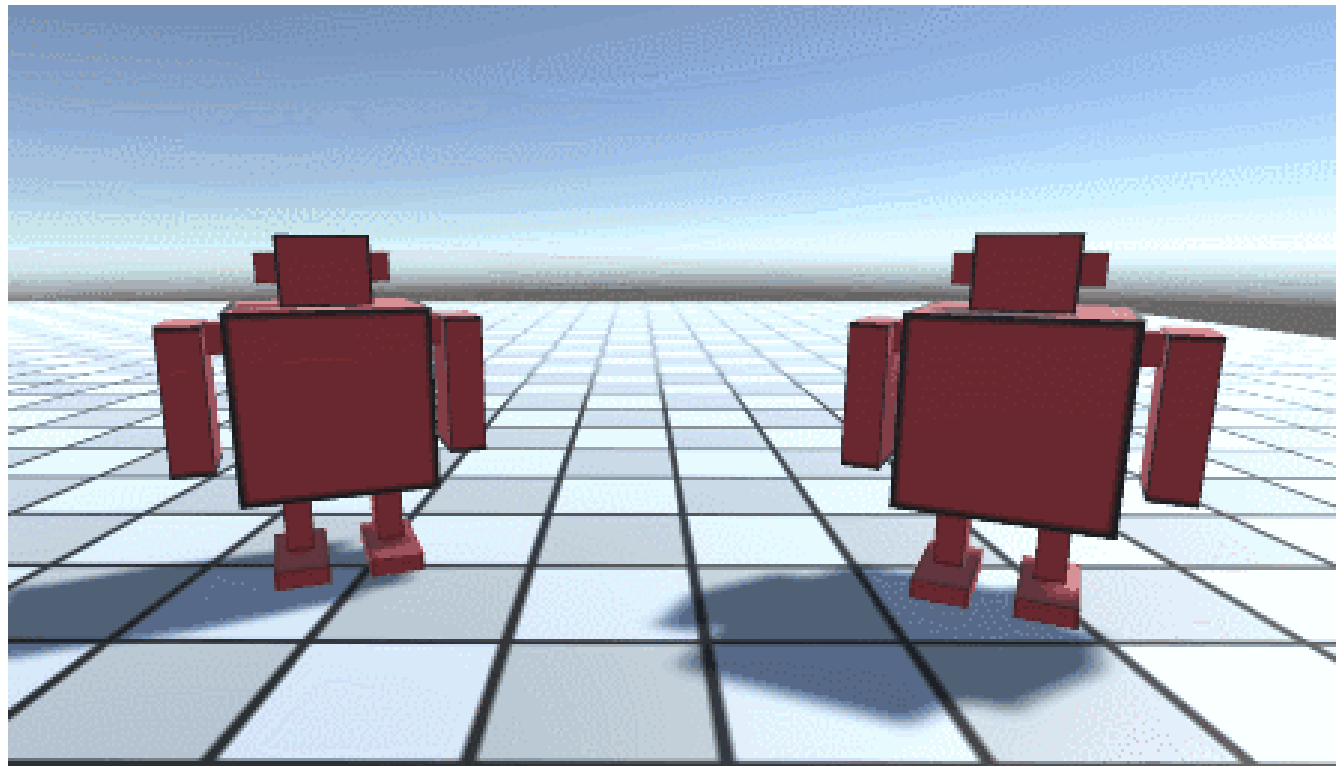
скрипт помещается
в пустой объект

скрипт помещается
в префаб снаряда

```
public class Projectile : MonoBehaviour
{
    public GameObject explosion;
    void OnCollisionEnter()
    {
        Instantiate(explosion, transform.position, transform.rotation);
        Destroy(gameObject);
    }
}
```

Процедурная генерация префабов

Замена персонажа обломками



Функции событий

Руководство

[API скриптов](#)

MonoBehaviour

[MonoBehaviour.OnPlayerDisconnected](#)

Called on the server whenever a player disconnected from the server.

[MonoBehaviour.OnPostRender](#)

OnPostRender is called after a camera finished rendering the scene.

[MonoBehaviour.OnPreCull](#)

OnPreCull вызывается до того, как камера отсечёт сцену.

[MonoBehaviour.OnPreRender](#)

OnPreRender вызывается перед тем, как камера начнёт рендерить сцену.

[MonoBehaviour.OnRenderImage](#)

OnRenderImage вызывается после того как весь рендеринг для отрисовки изображений

Функции событий

- **Update события** (Update, FixedUpdate, LateUpdate) ;
- **события инициализации** (Start, Awake);
- **события GUI** (элементов управления);
- **события мыши** (OnMouseOver, OnMouseDown);
- **события физики** (OnCollisionEnter, OnTriggerEnter)

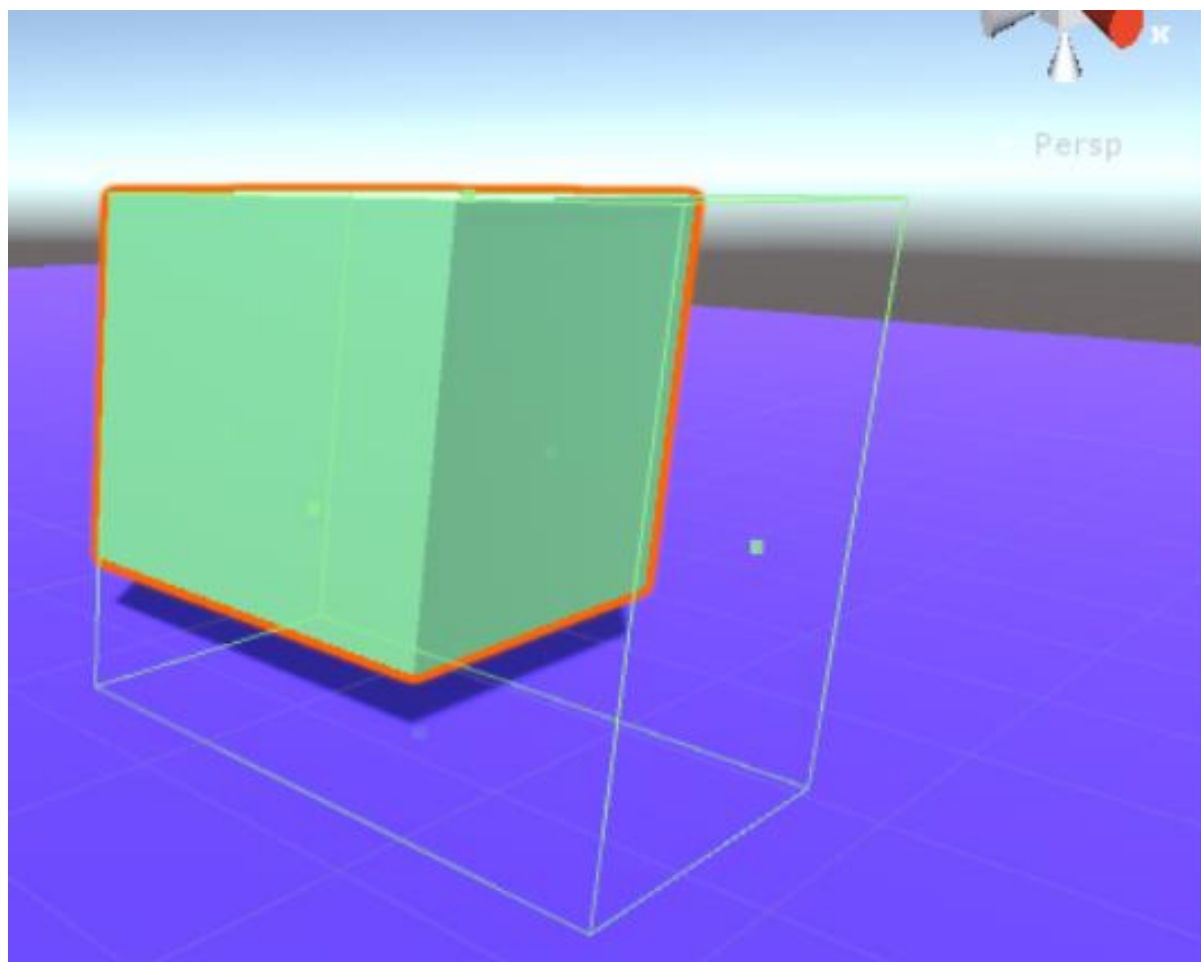
Обнаружение столкновений

Информация о столкновении передаётся в события **Collider.OnCollisionEnter**, **Collider.OnCollisionStay** и **Collider.OnCollisionExit**.

OnCollisionEnter(Collision)

Класс Collision содержит информацию о точках соприкосновения, скорости воздействия, “личность” входящего объекта и т.д.

<u>collider</u>	Collider, с которым мы столкнулись (Read O
<u>contacts</u>	Точки соприкосновения сгенерированные ф
<u>gameObject</u>	GameObject, с чьим коллайдером мы столкн
<u>impulse</u>	The total impulse applied to this contact pair to
<u>relativeVelocity</u>	Относительная линейная скорость двух сто
<u>rigidbody</u>	Rigidbody, с которым мы столкнулись (Read мы столкнулись это коллайдер , к которому
<u>transform</u>	Transform объекта, с которым мы столкнули



Position	X	1.59	Y	1.36	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	2	Y	2	Z	2

▼ **Cube (Mesh Filter)**

Mesh

Cube

▼ ☒ **Box Collider**

Edit Collider

Is Trigger ☐

Material

None (Physic Material)

Center
X

0.2112761

 Y

-0.1919779

 Z

0

Size
X

1.422552

 Y

1.383956

 Z

1

▼ ☒ **Mesh Renderer**

► Lighting

▼ Materials
Size

1

Element 0

New Material 1

Коллайдеры

- Статические коллайдеры
- Динамические коллайдеры

Статические коллайдеры – это коллайдеры **без Rigidbody**.

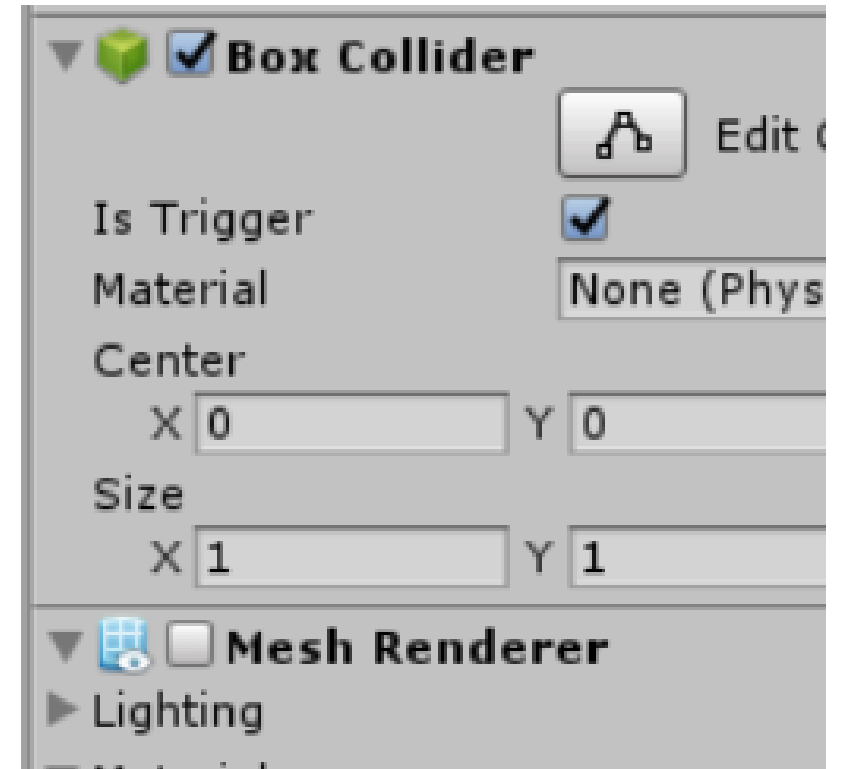
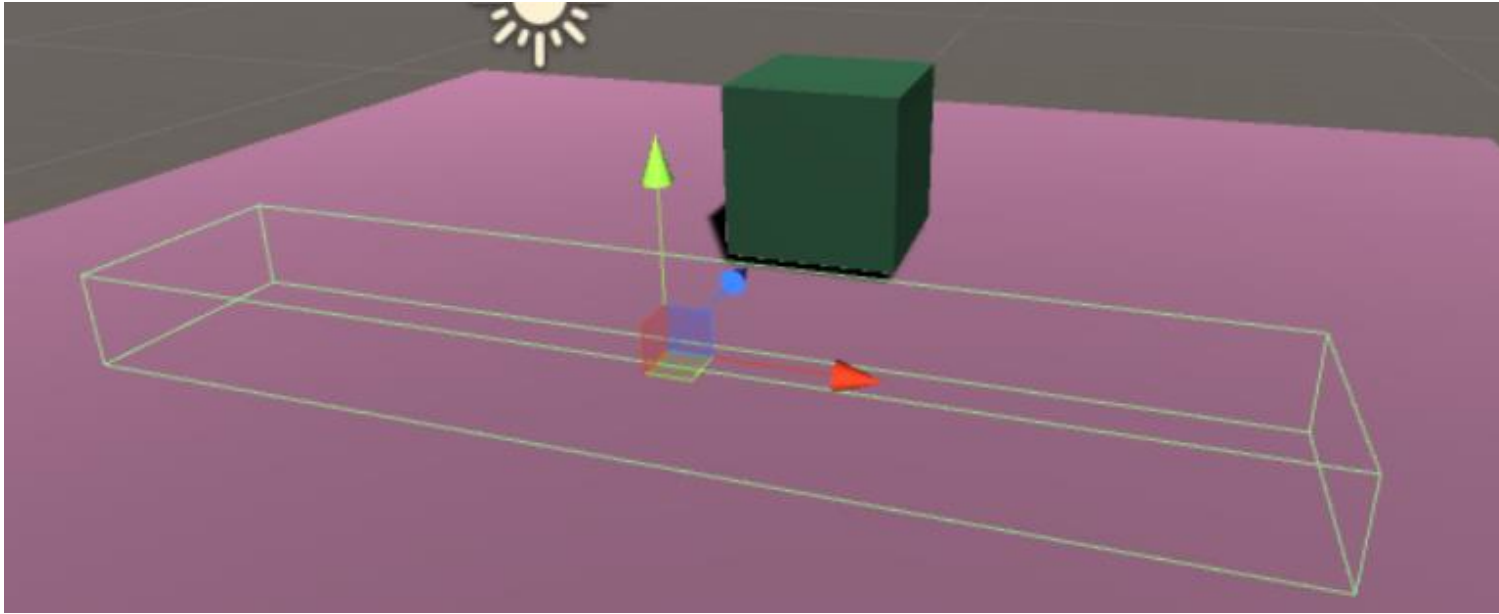
Не зависят от столкновений.

Нельзя изменять позицию через Transform, так как это будет сильно влиять на производительность.

Динамические коллайдеры – это коллайдеры **с Rigidbody**.

Зависят от столкновений.

Триггер-?



При взаимодействии с триггером вызываются функции OnTriggerEnter, OnTriggerStay и OnTriggerExit.

Триггерные события отправляются только в том случае, если один из коллайдеров имеет Rigidbody.

OnTriggerEnter(Collider)

Матрица взаимодействий

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		collision			trigger	trigger
Rigidbody Collider	collision	collision	collision	trigger	trigger	trigger
Kinematic Rigidbody Collider		collision		trigger	trigger	trigger
Static Trigger Collider		trigger	trigger		trigger	trigger
Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger
Kinematic Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger

Input

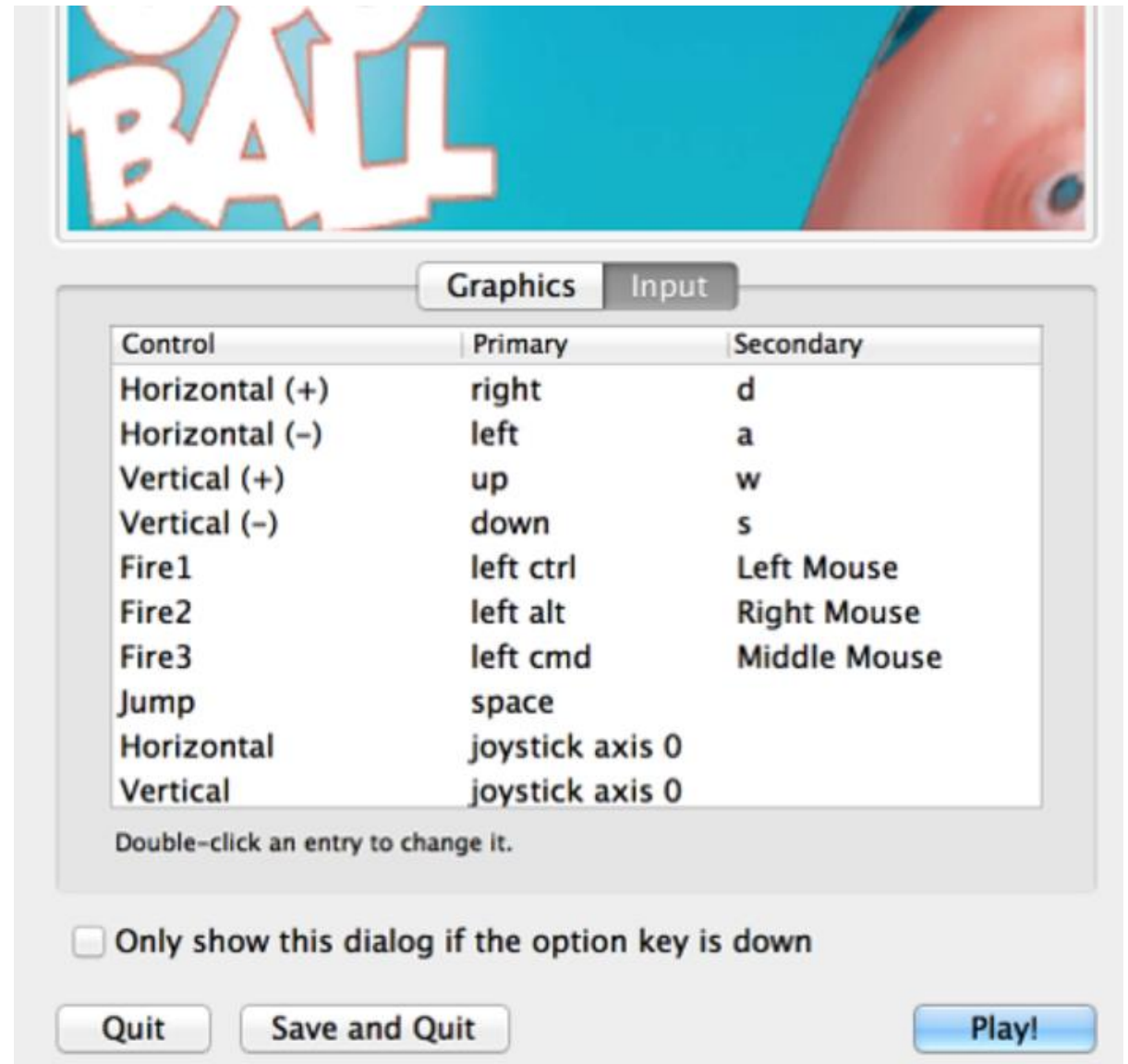
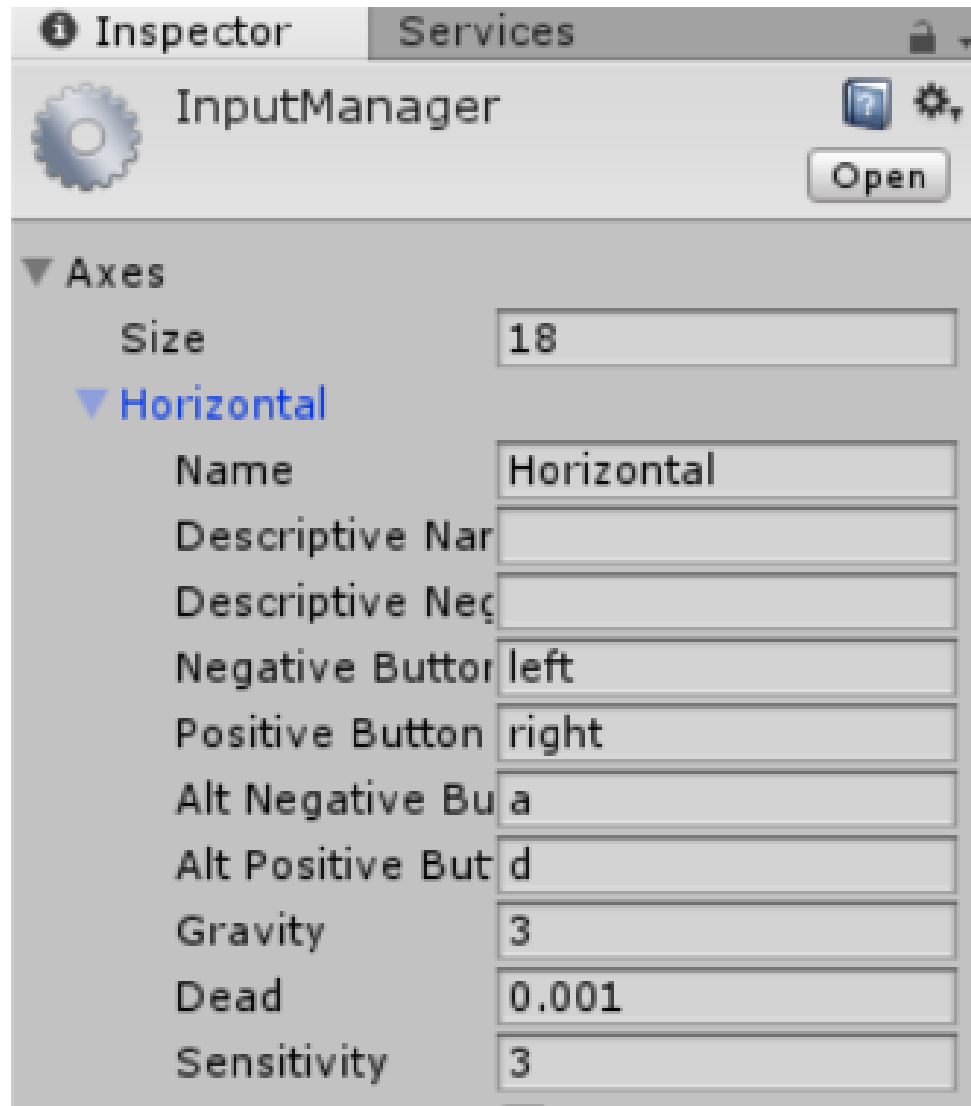
Этот класс используется для чтения информации с осей координат, установленных в Менеджере Ввода, а также для доступа к данным акселерометра/мультиач на мобильных устройствах.

Для обработки ввода, отвечающего за контроль постоянного движения, используйте `GetAxis`

```
void Update() {  
    float h = horizontalSpeed * Input.GetAxis("Mouse X");  
    float v = verticalSpeed * Input.GetAxis("Mouse Y");  
    transform.Rotate(v, h, 0);  
}
```

Менеджер Ввода

Edit->Project Settings->Input



Сопрограммы (корутины)

Когда вы вызываете функцию, она выполняется до завершения перед возвратом. Это фактически означает, что любое действие, происходящее в функции, должно происходить в рамках обновления одного кадра, таким образом вызов функции не может содержать последовательность событий во времени.

В качестве примера рассмотрим задачу постепенного уменьшения значения альфа (непрозрачности) объекта до тех пор, пока он не станет полностью невидимым.

```
void Fade()  
{  
    for (float ft = 1f; ft >= 0; ft -= 0.1f)  
    {  
        Color c = renderer.material.color;  
        c.a = ft;  
        renderer.material.color = c;  
    }  
}
```


Сопрограммы (корутины)

```
IEnumerator Fade()  
{  
    for (float ft = 1f; ft >= 0; ft -= 0.1f)  
    {  
        Color c = renderer.material.color;  
        c.a = ft;  
        renderer.material.color = c;  
        yield return null;  
    }  
}
```

✓ Объявление сопрограммы

Сопрограмма похожа на функцию, которая может приостанавливать выполнение и возвращать управление Unity, но затем продолжать с того места, где она была остановлена, в следующем кадре.

*Нулевая строка **yield return** - это точка, в которой выполнение будет приостановлено и возобновится в следующем кадре.*

```
void Update()  
{  
    if (Input.GetKeyDown("f"))  
    {  
        StartCoroutine ("Fade");  
    }  
}
```

*Можно ввести временную задержку
yield return new WaitForSeconds(.1f);*

✓ Вызов сопрограмы

Пример: нужно создать 10.000 объектов. Порциями по 10-100 или просто в цикле. Если сделать это в методе Update, то пока цикл не отработает обновления экрана не будет, приложение "висит" все это время.

Корутину можно применять для длительных операций, которые можно "размазать" по кадрам. Для данной задачи можно вызывать примерно следующую последовательность действий:

```
// счетчик цикла
Debug.Log("Инстанцируем объекты и складываем их в массив");
yield return new WaitForSeconds(1);
Debug.Log("делаем доп работу с этим массивом");
yield return new WaitForSeconds(1);
Debug.Log("Еще какая-то работа");
yield return new WaitForSeconds(1);
```

```
void Start () {  
    StartCoroutine(Test(StartAction, FinalAction));  
}  
  
IEnumerator Test(Action actBefore, Action actAfter) {  
    actBefore();  
  
    for (int i = 0; i < 5; i++) {  
        Debug.Log("Test" + i);  
        yield return new WaitForSeconds(1.5f);  
        Debug.Log("Test" + i + i);  
    }  
  
    actAfter();  
}  
  
void StartAction() {  
    Debug.Log("I'm a start action");  
}  
  
void FinalAction() {  
    Debug.Log("I'm a final action");  
}
```

Например, необходимо чтобы до прогона действий и после что-то происходило.

другой пример с пулями:

```
void Start () {
    StartCoroutine("FireThriceAndWait");
}

IEnumerator FireThriceAndWait () {
    while (true) {
        fire();
        yield return new WaitForSeconds(0.5f);
        fire();
        yield return new WaitForSeconds(0.5f);
        fire();
        yield return new WaitForSeconds(5f);
    }
}

void fire(){
    Instantiate(enemy_bullet,this.transform.position, Quaternion.LookRotation(target.transfo
}
```

Мигание спрайта (уменьшить прозрачность, увеличить) с интервалом 0.5 сек.

```
IEnumerator Test() {  
    while (true) {  
        var color = obj.GetComponent<Renderer>().material.color;  
        for (float i = 1; i >= 0; i-=0.1f) {  
            color.a = i;  
            obj.GetComponent<Renderer>().material.color = color;  
            yield return null;  
        }  
  
        yield return new WaitForSeconds(0.5f);  
  
        for (float i = 0; i < 1; i += 0.1f) {  
            color.a = i;  
            obj.GetComponent<Renderer>().material.color = color;  
            yield return null;  
        }  
        yield return new WaitForSeconds(0.5f);  
    }  
}
```

Существует несколько вариантов для возвращаемых в **yield** значений:

Продолжить после следующего FixedUpdate:

```
yield return new WaitForFixedUpdate();
```

Продолжить после следующего LateUpdate и рендеринга сцены:

```
yield return new WaitForEndOfFrame();
```

Продолжить через некоторое время:

```
yield return new WaitForSeconds(0.1f); // продолжить примерно через 100ms
```

Продолжить по завершению другого корутина:

```
yield return StartCoroutine(AnotherCoroutine());
```

Продолжить после загрузки удаленного ресурса:

```
yield return new WWW(someLink);
```