

Рефлексия по материалам лабораторной работы

1. Функция **Update()** срабатывает с периодичностью 1 раз в кадр, т.е. зависит от **fps** (?), а fps в свою очередь зависит от производительности и загруженности сцены, что бы избежать зависимости скорости выполнения действий от производительности и загруженности сцены используют функцию **FixedUpdate()**.
2. Движение объекту можно задавать через изменение его положения по каким-либо осям

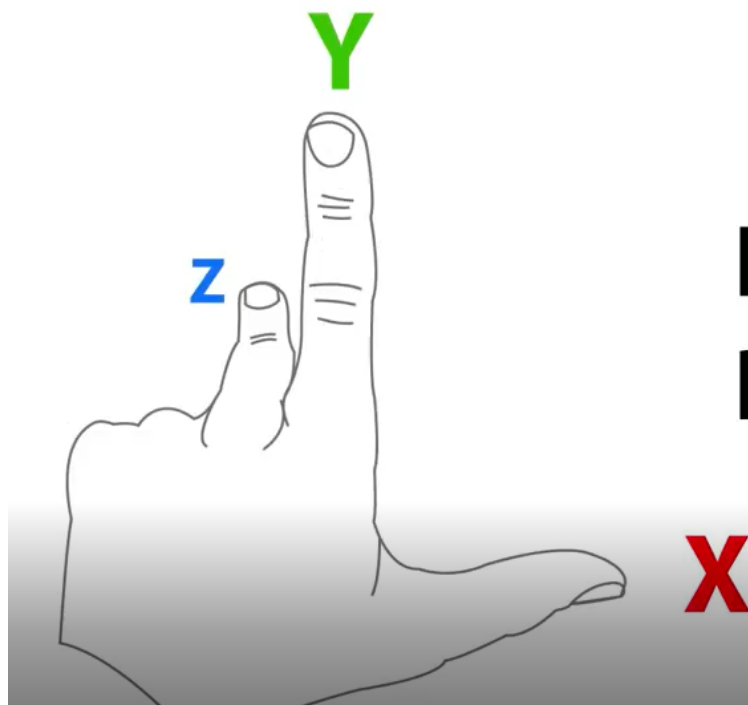
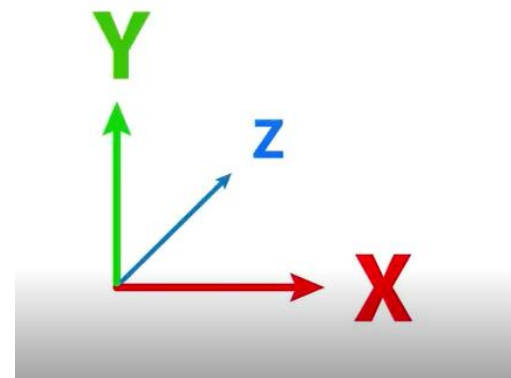
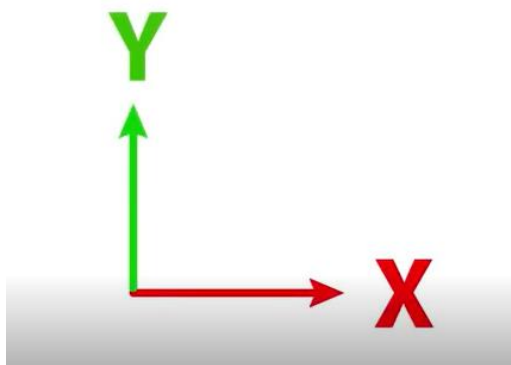
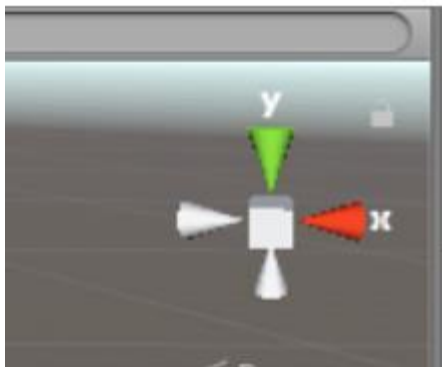
```
transform.position = new Vector3 (posX + speed, posY, posZ)
```

НО, для объектов с компонентом Rigidbody грамотно применять функцию **AddForce()**

3. Не стоит делать стены из объекта **Plane**, потому что стена получится прозрачной с одной из сторон. Для стен используют Box или импортированные стены.
4. Прикреплять камеру к объекту, делая ее дочерним элементом в проектах не стоит, мы просто попробовали, что бы увидеть эффект от запрета вращения сферы, **как правило камера закрепляется за объектом при помощи скрипта.**

Вектора в Unity

Система координат



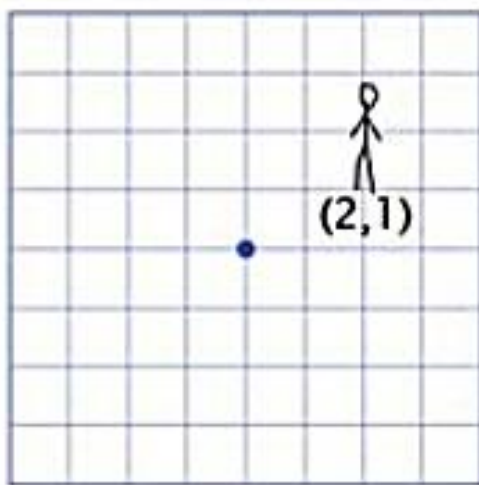
**Left Hand
Rule Coordinates**

Что такое вектор?

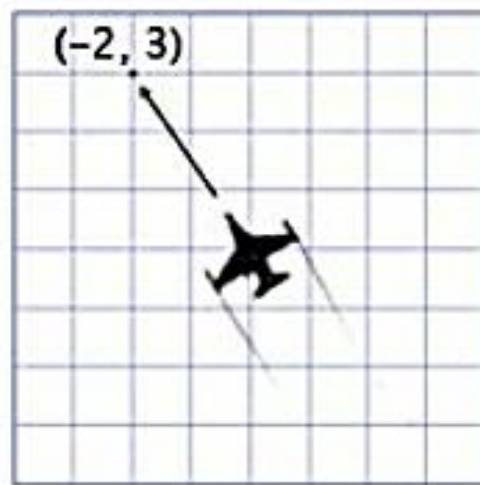
Вектор сам по себе всего лишь набор цифр, который обретает тот или иной смысл в зависимости от контекста.

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

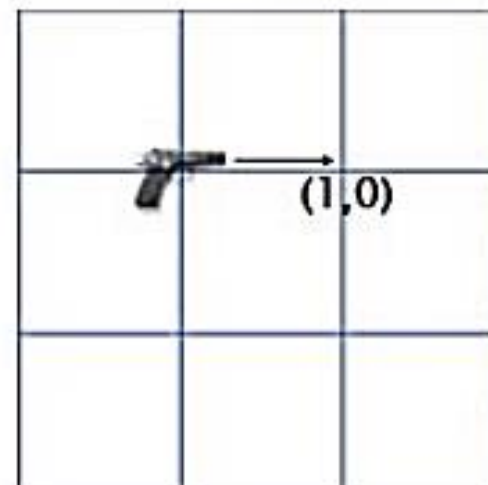
<https://habr.com/ru/post/131931/>



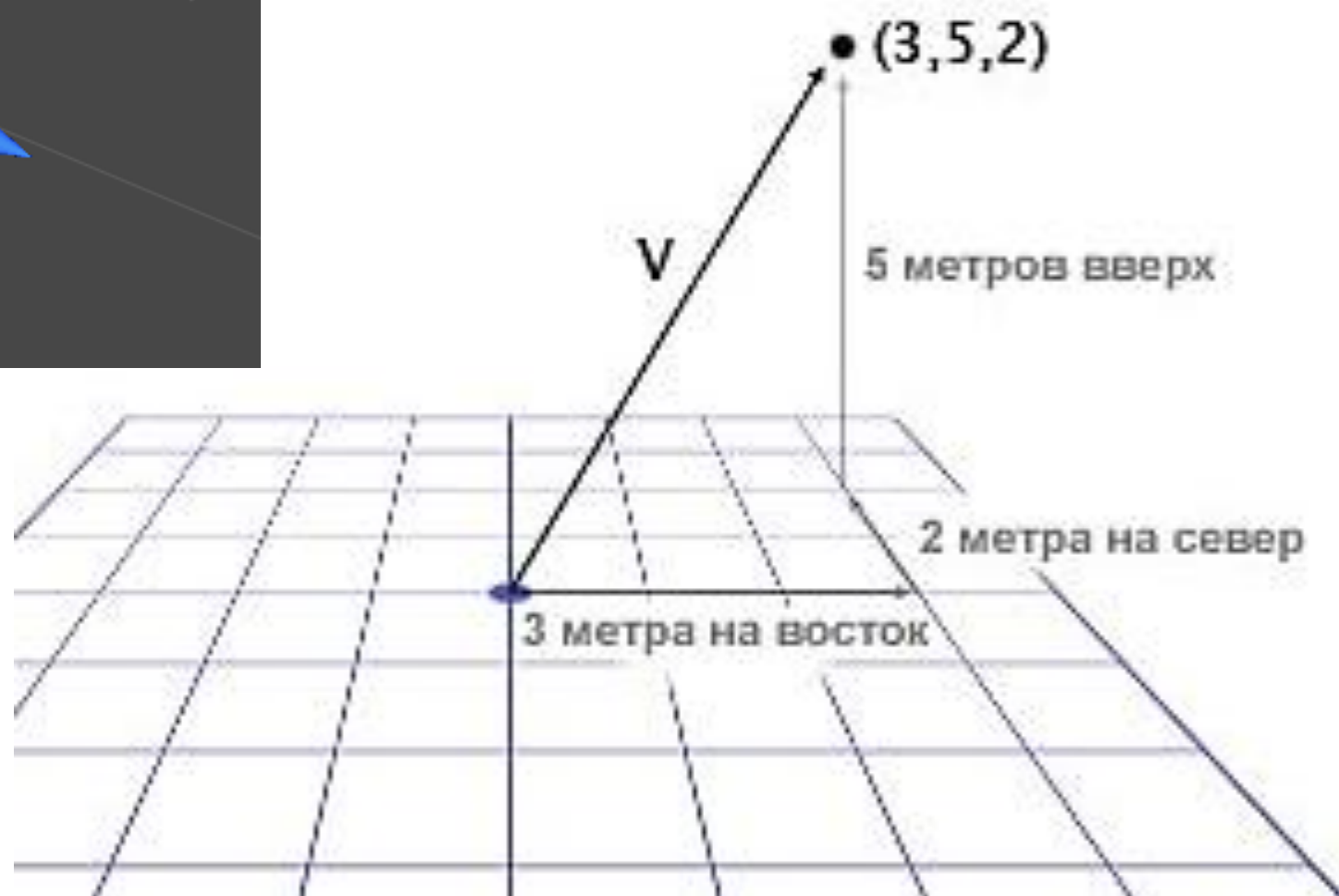
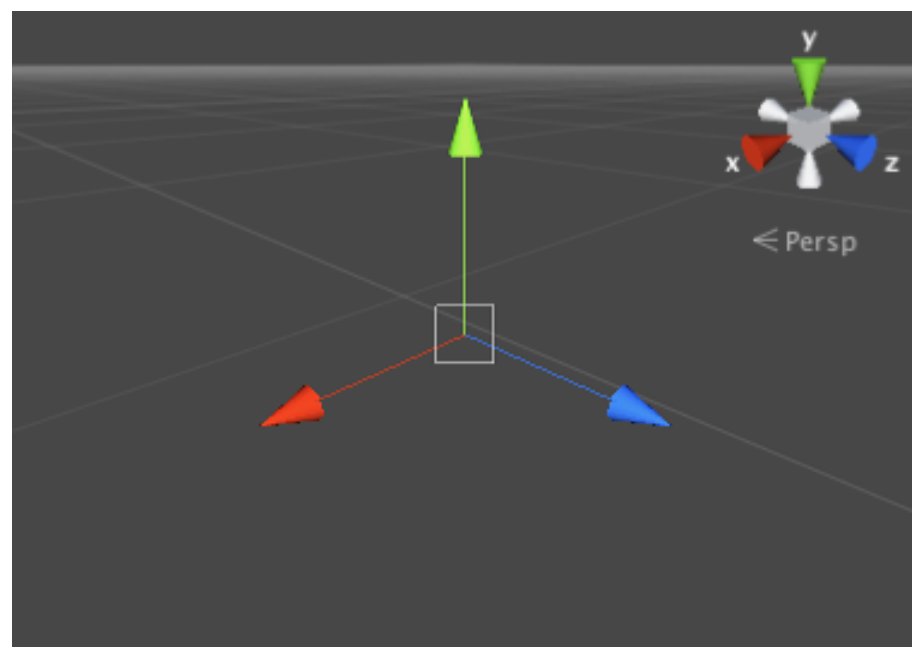
Местоположение



Скорость



Направление

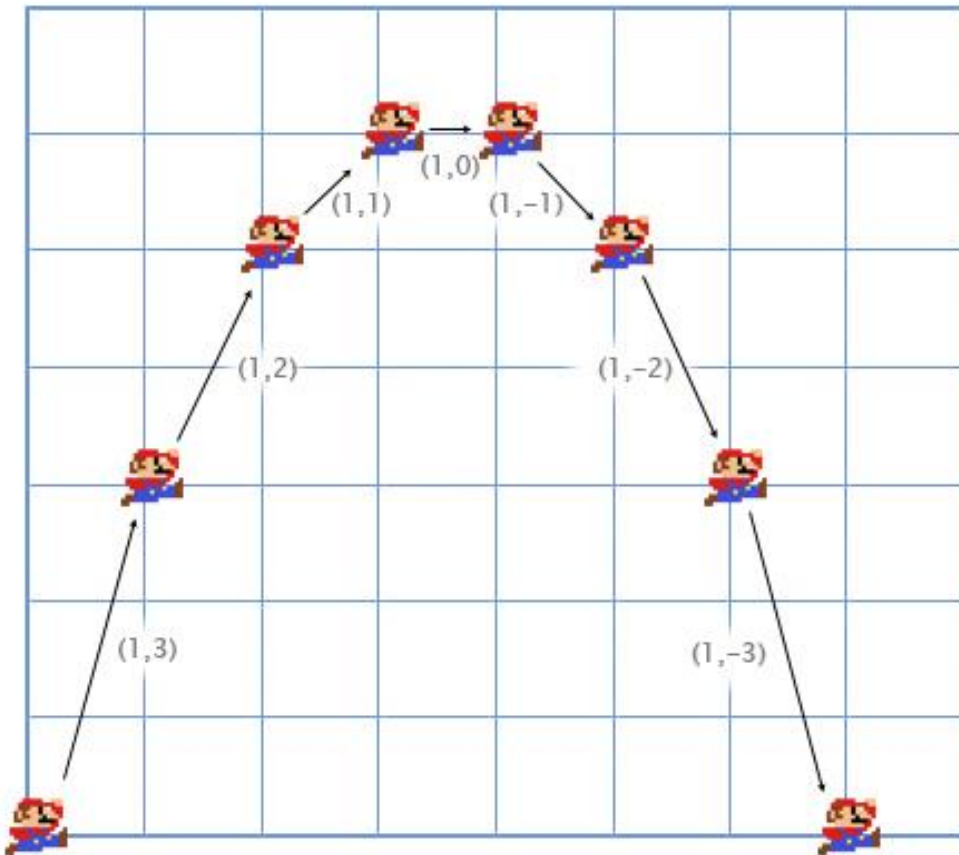


Сложение векторов

$$(0, 1, 4) + (3, -2, 5) = (0 + 3, 1 - 2, 4 + 5) = (3, -1, 9)$$

Зачем складывать вектора?

Любой физический объект будет иметь вектора для местоположения, скорости и ускорения. Для каждого кадра, мы должны интегрировать два вектора: добавить скорость к местоположению и ускорение к скорости.



Пример с прыжками Марио. Он начинает с позиции (0, 0). В момент начала прыжка его скорость (1, 3), он быстро движется вверх и вправо. Его ускорение равно (0, -1), так как гравитация тянет его вниз. На картинке показано, как выглядит его прыжок, разбитый на семь кадров и показана его скорость в каждом фрейме.

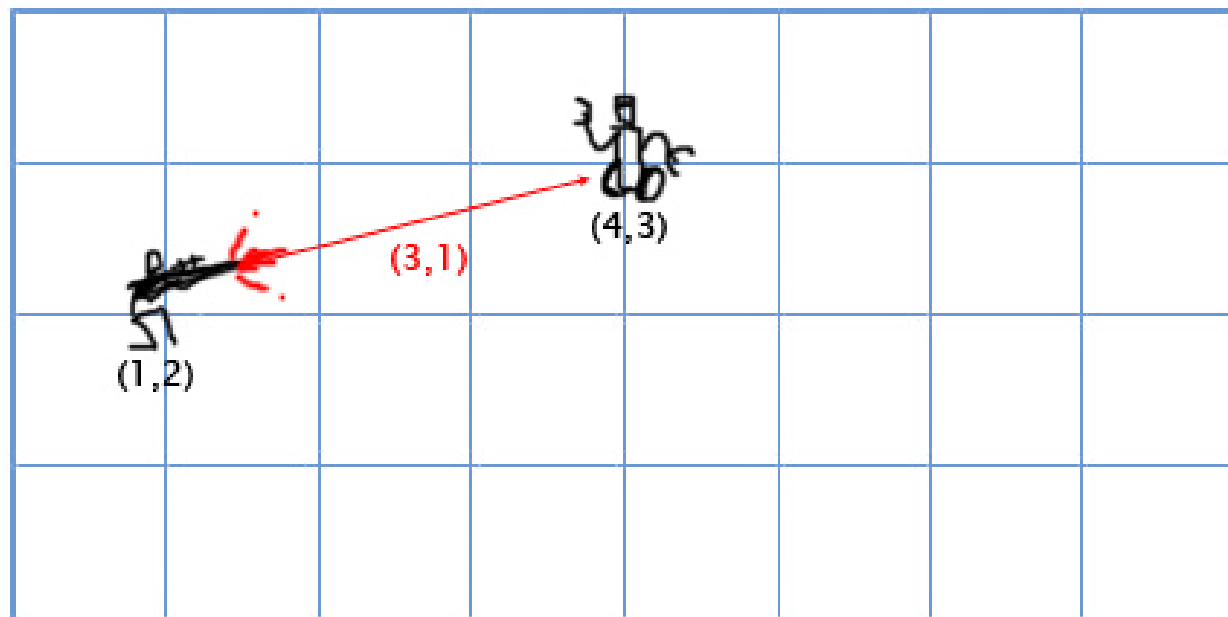
Для первого кадра добавляем скорость Марио (1, 3) к его местоположению (0, 0) и получаем его новые координаты (1, 3). Затем складываем ускорение (0, -1) с его скоростью (1, 3) и получаем новое значение скорости Марио (1, 2).

То-же самое для второго кадра. Добавляем скорость (1, 2) к местоположению (1, 3) и получаем координаты (2, 5). Затем добавляем ускорение (0, -1) к его скорости (1, 2) и получаем новую скорость (1, 1).

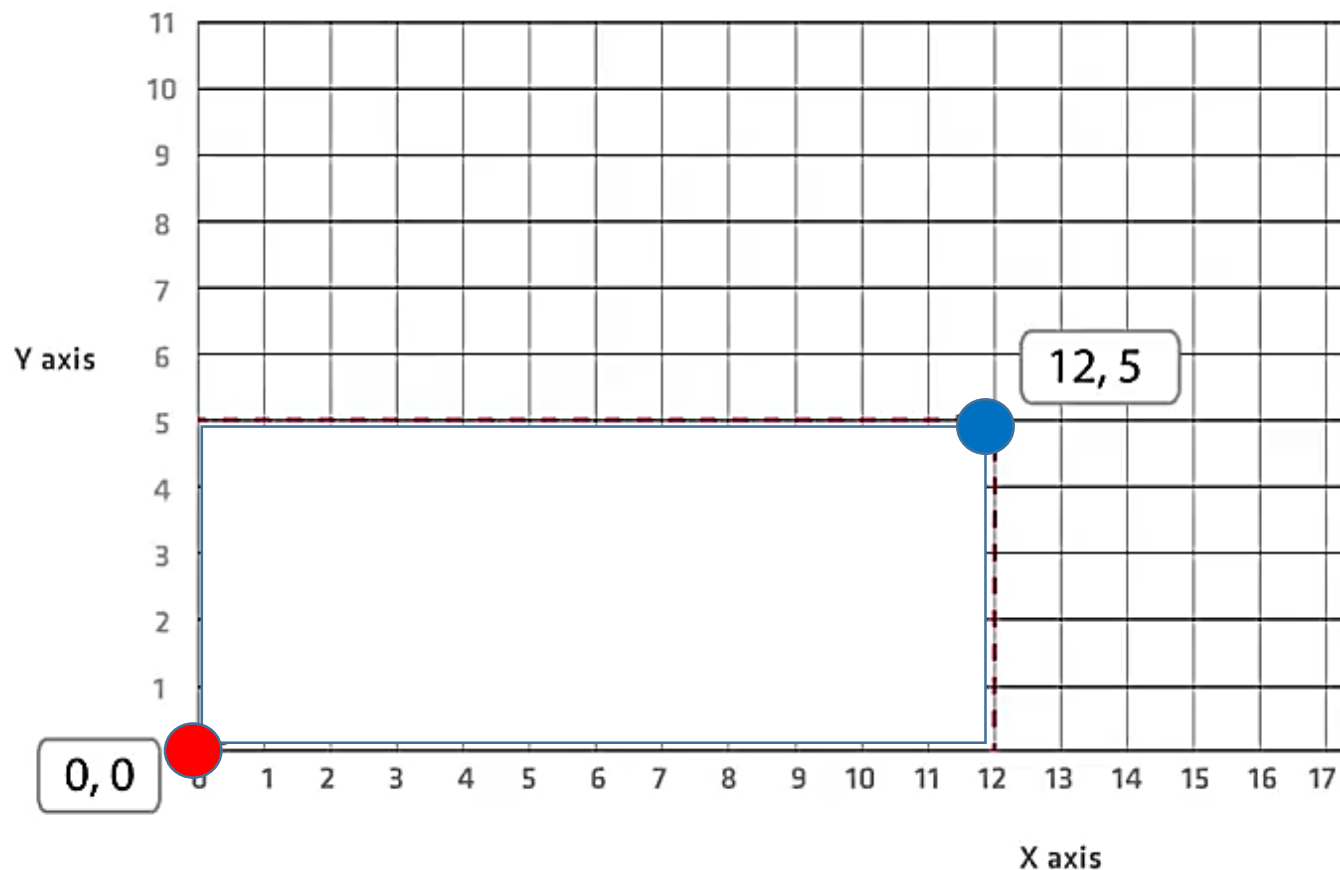
Вычитание векторов

Вычитание векторов удобно для получения вектора, который показывает из одного местоположения на другое. Например, пусть игрок находится по координатам (1, 2) с лазерным ружьём, а вражеский робот находится по координатам (4, 3). Чтобы определить вектор движения лазерного луча, который поразит робота, нам надо вычесть местоположение игрока из местоположения робота. Получаем:

$$(4, 3) - (1, 2) = (4-1, 3-2) = (3, 1)$$



Расстояние между объектами



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

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

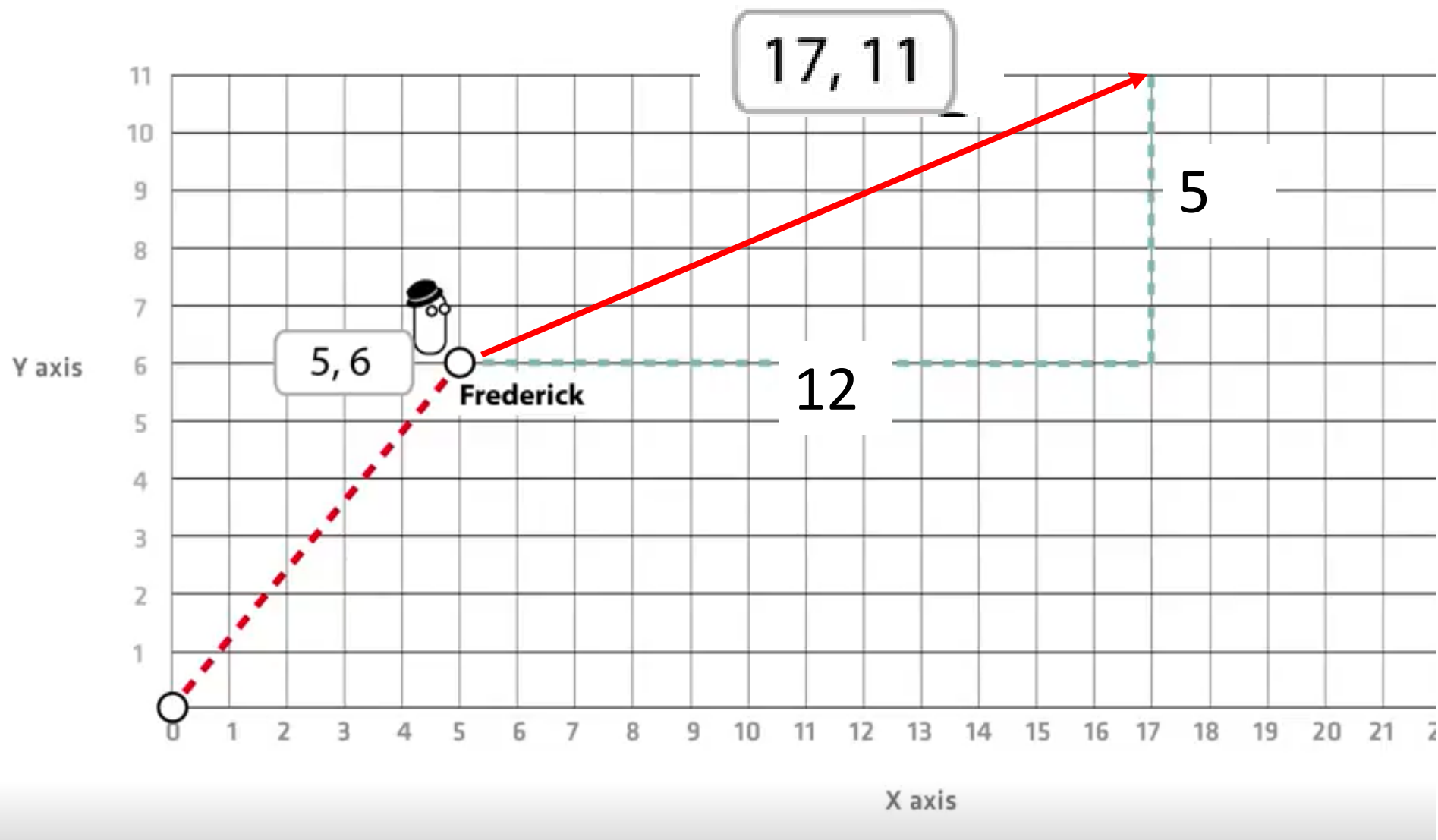
$$\text{Magnitude} = \sqrt{12^2 + 5^2}$$

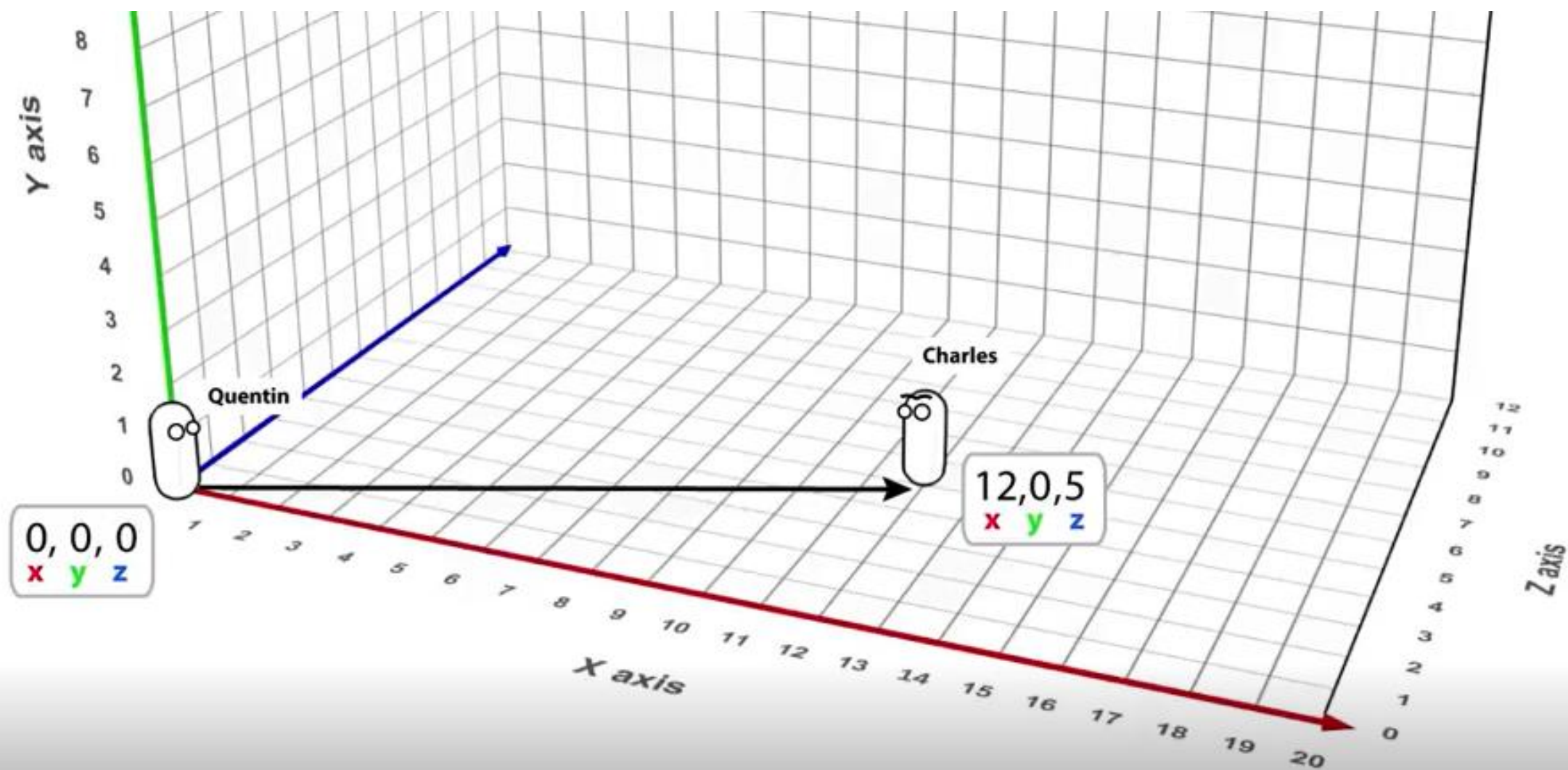
$$\text{Magnitude} = \sqrt{144 + 25}$$

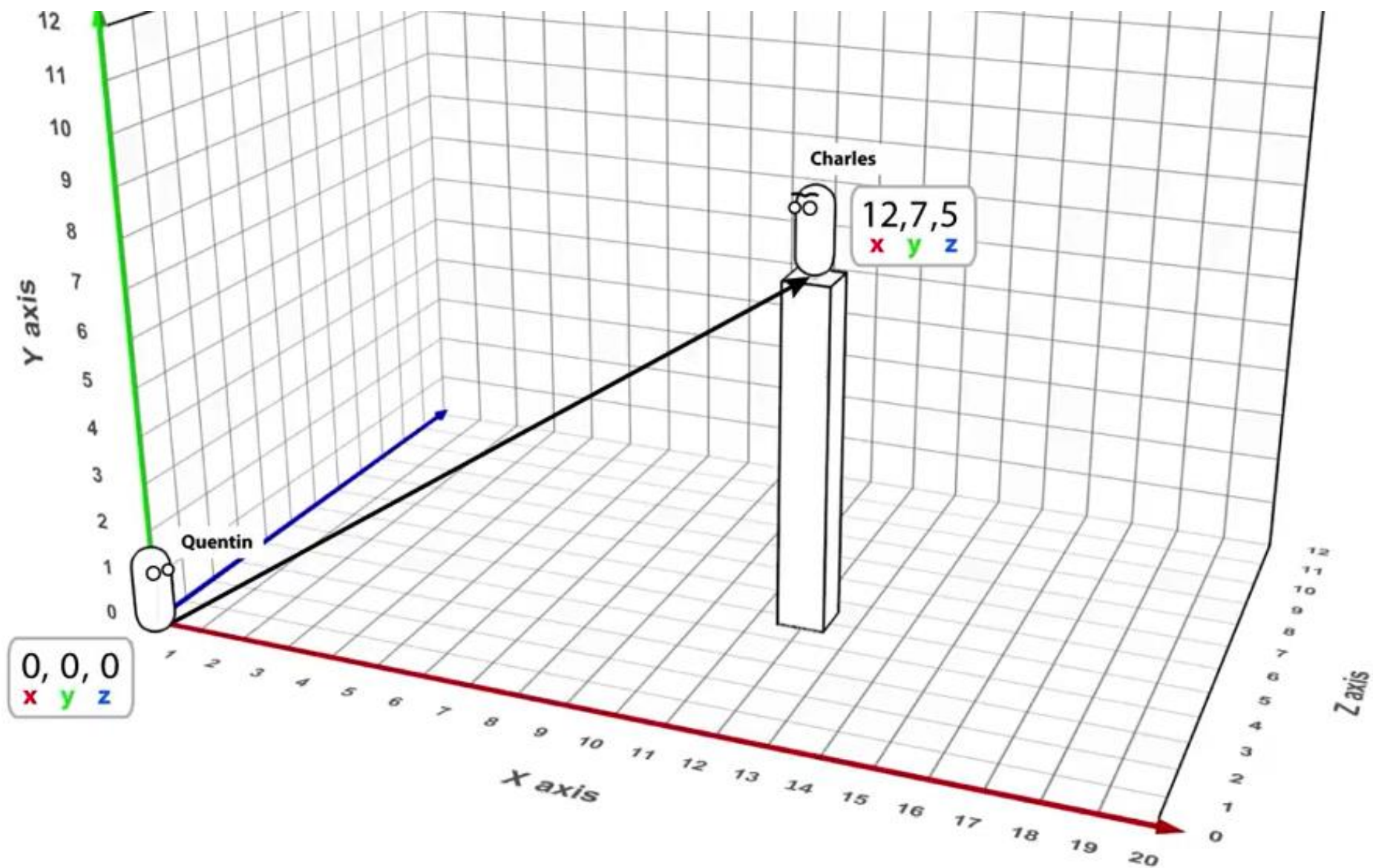
$$\text{Magnitude} = \sqrt{169}$$

$$\text{Magnitude} = 13$$

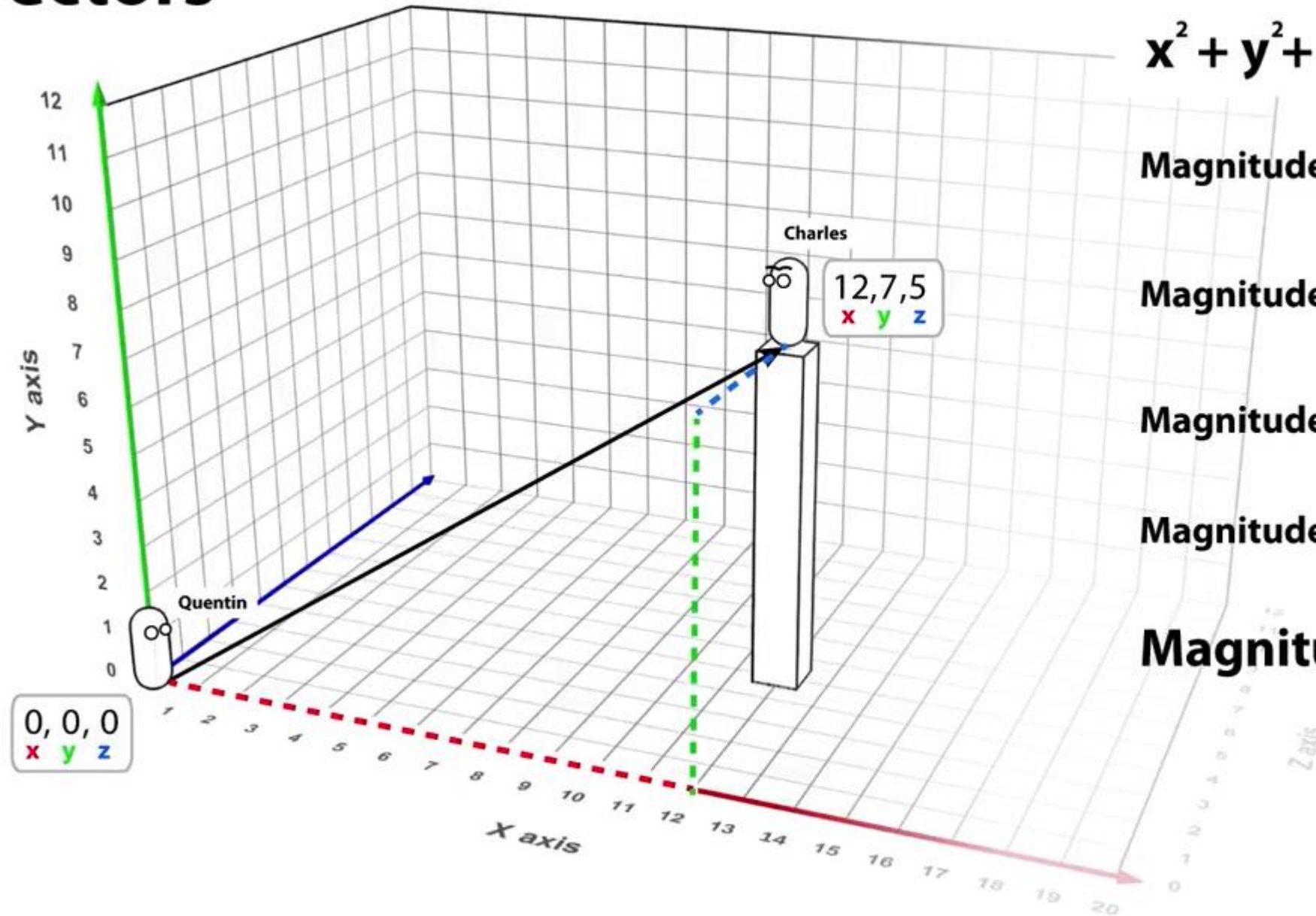
Скорость Фредерика (12,5) в час. Где он будет через час?
Сложение векторов







Vectors



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

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

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

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

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

$$\text{Magnitude} = 14.76$$

Структура **Vector3**

Эта структура используется в Unity для передачи трехмерных позиций и направлений. Она также содержит функции для выполнения обычных векторных операций.

Статические свойства:

Vector3.back	Vector3(0, 0, -1)
Vector3.down	Vector3(0, -1, 0)
Vector3.forward	Vector3(0, 0, 1)
Vector3.left	Vector3(-1, 0, 0)
Vector3.right	Vector3(1, 0, 0)
Vector3.up	Vector3(0, 1, 0)
Vector3.zero	Vector3(0, 0, 0)
Vector3.one	Vector3(1, 1, 1)

Методы:

magnitude	Returns the length of this vector
normalized	Returns this vector with a magnitude of 1
sqrMagnitude	Returns the squared length of this vector

Статические методы

Angle	Угол между векторами.
Distance	Расстояние между двумя векторами.
Dot	Скалярное произведение двух векторов.
Cross	Векторное произведение двух векторов..

Расстояние между двумя объектами в Unity

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

Расстояние также можно подсчитать, вычислив разность между координатами объектов, получив в результате объект типа Vector3, а затем обратиться к его свойству magnitude:

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

Расстояние между двумя объектами. Оптимизация

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

```
public Transform box;
private const int cMaxDistance = 10;

private void Update()
{
    float dist = (box.position - transform.position).sqrMagnitude;
    light.enabled = dist <= cMaxDistance * cMaxDistance; //квадрат
    Debug.Log(dist);
}
```

```
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;
```

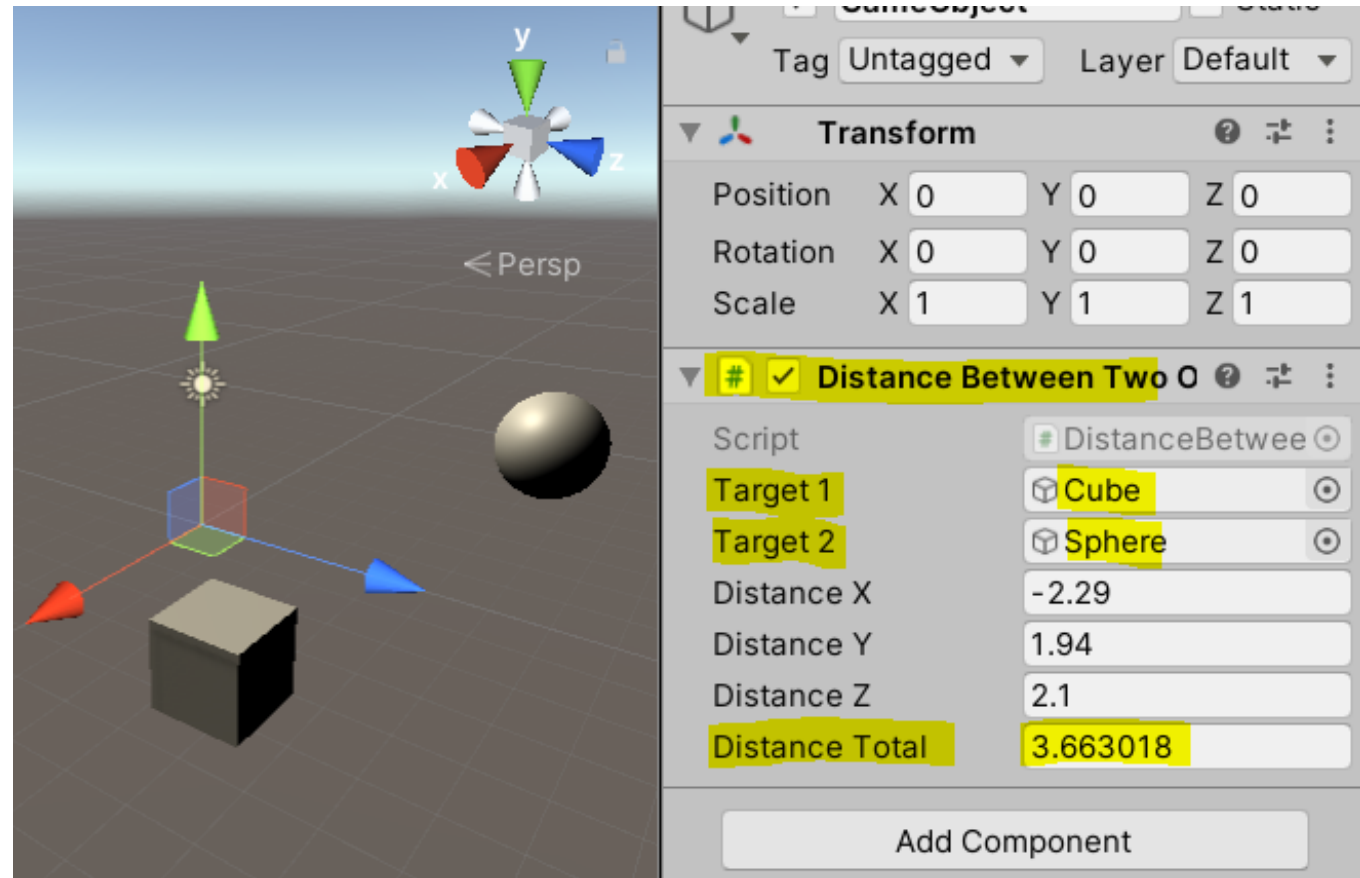
```
        distanceY = delta.y;
```

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

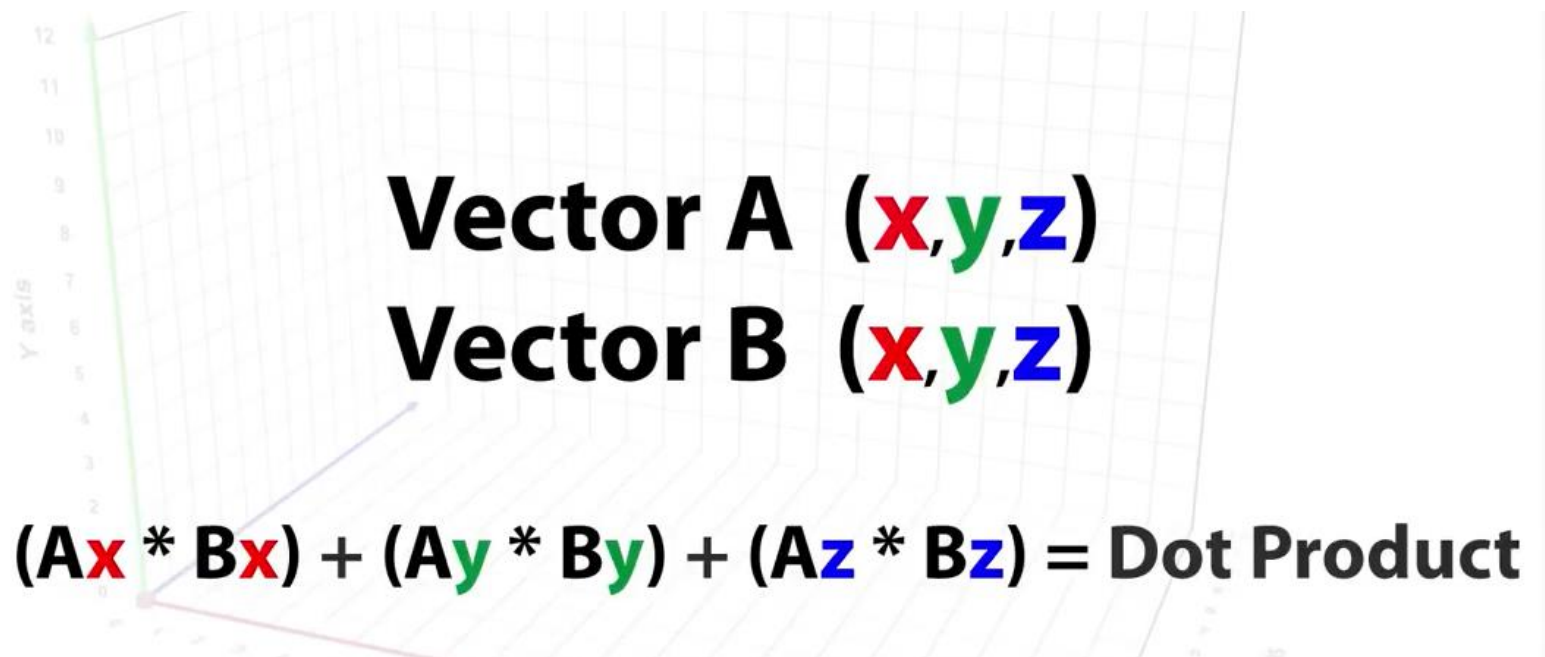
```
        distanceTotal = delta.magnitude;
```

```
    }
```

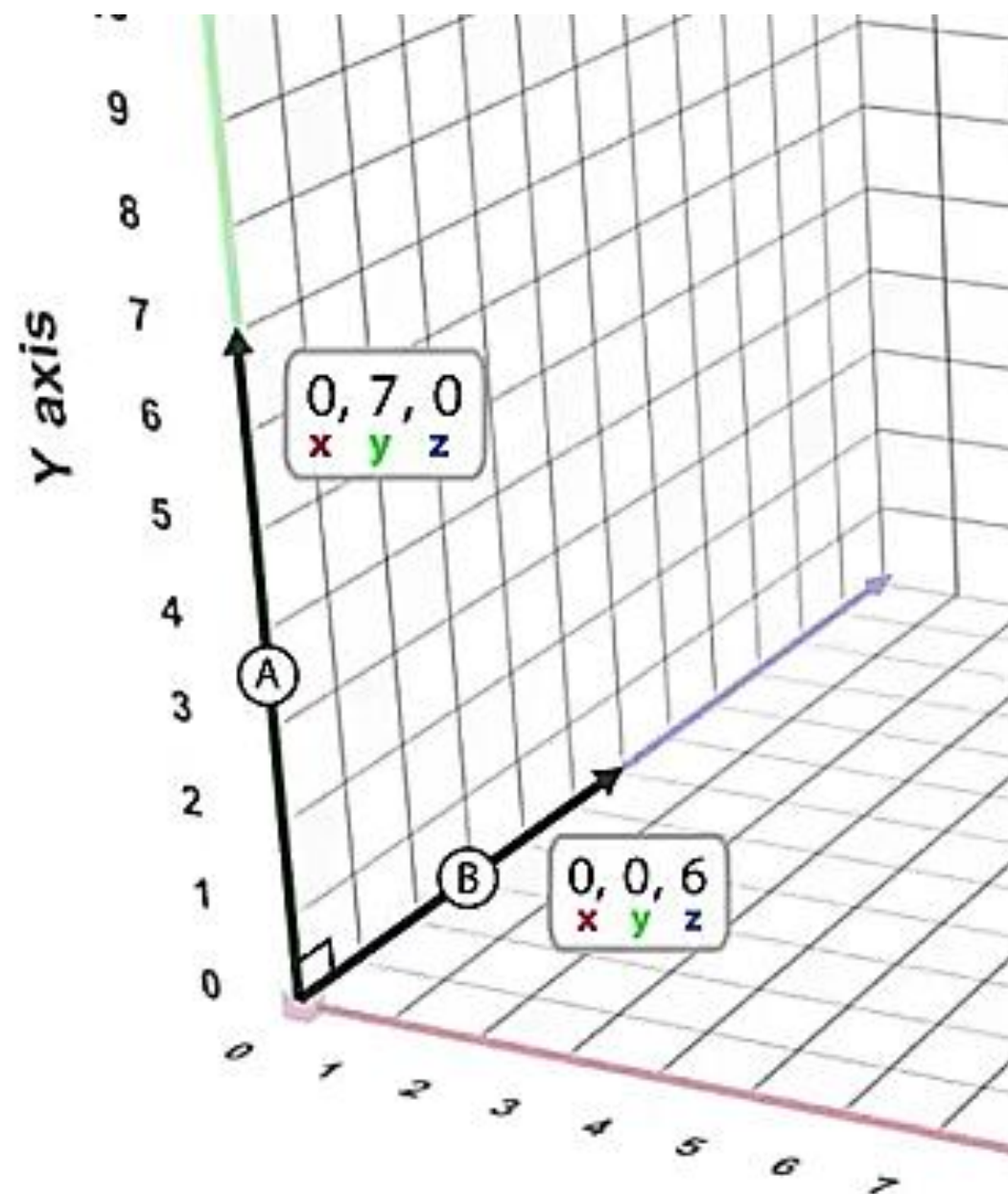
```
}
```



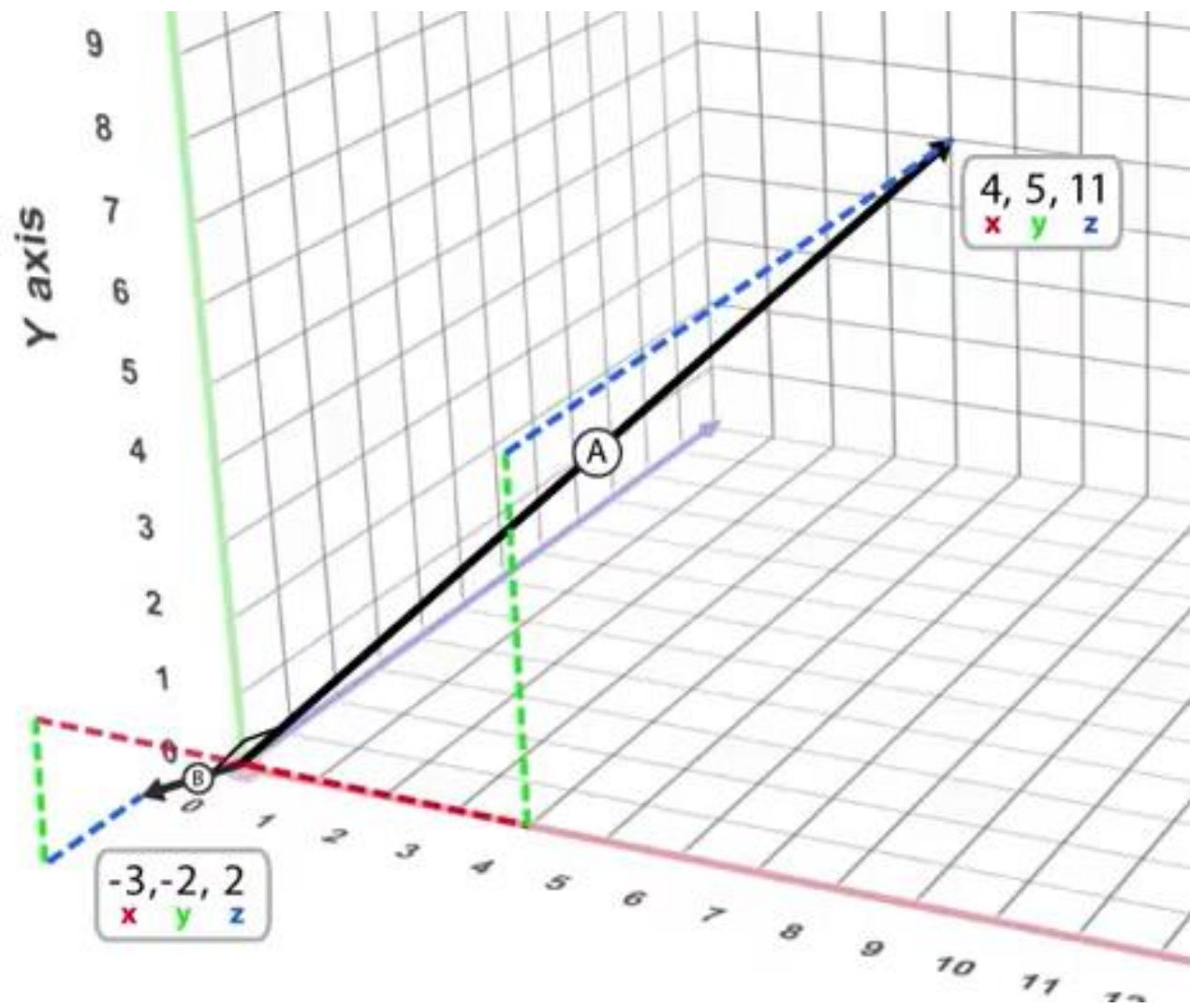
Скалярное произведение



Если скалярное произведение двух векторов равно 0, значит они перпендикулярны

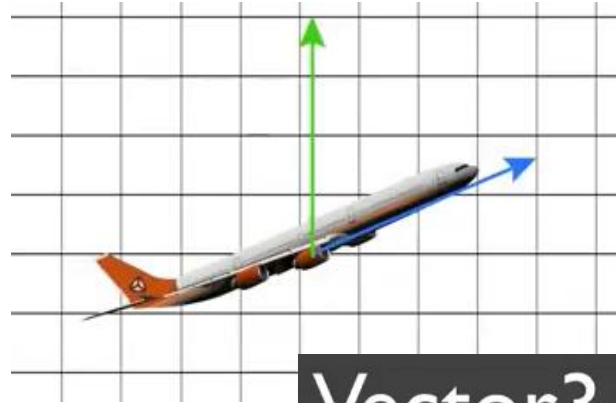
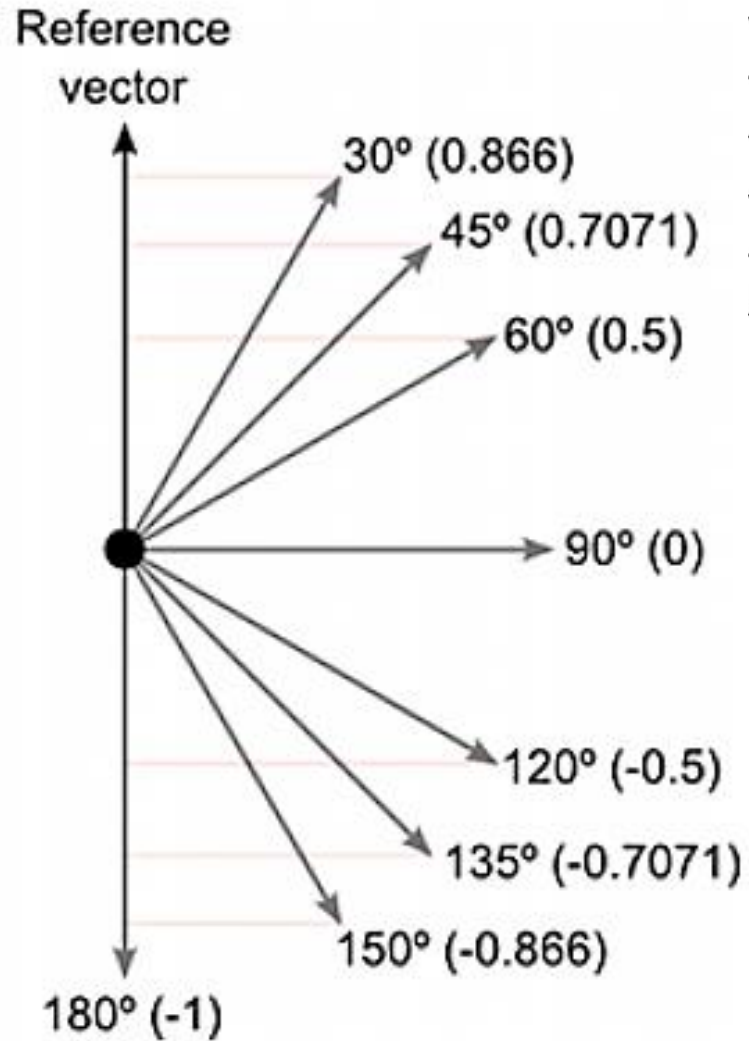


	A • B				
X	0	*	0	=	0
Y	7	*	0	=	0
Z	0	*	6	=	0
Total					<u>0</u>

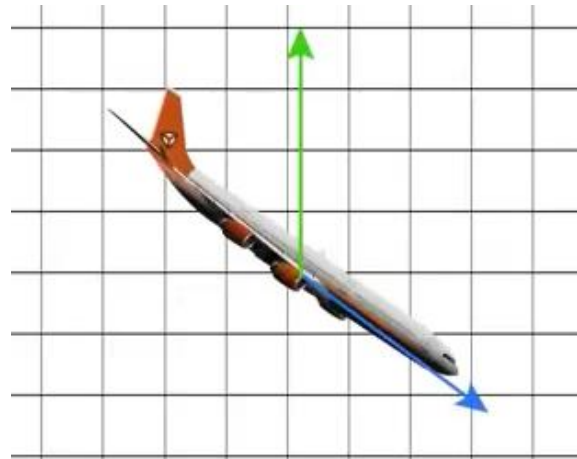


	A • B			
X	4	*	-3	= -12
Y	5	*	-2	= -10
Z	11	*	2	= 22
Total				0

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



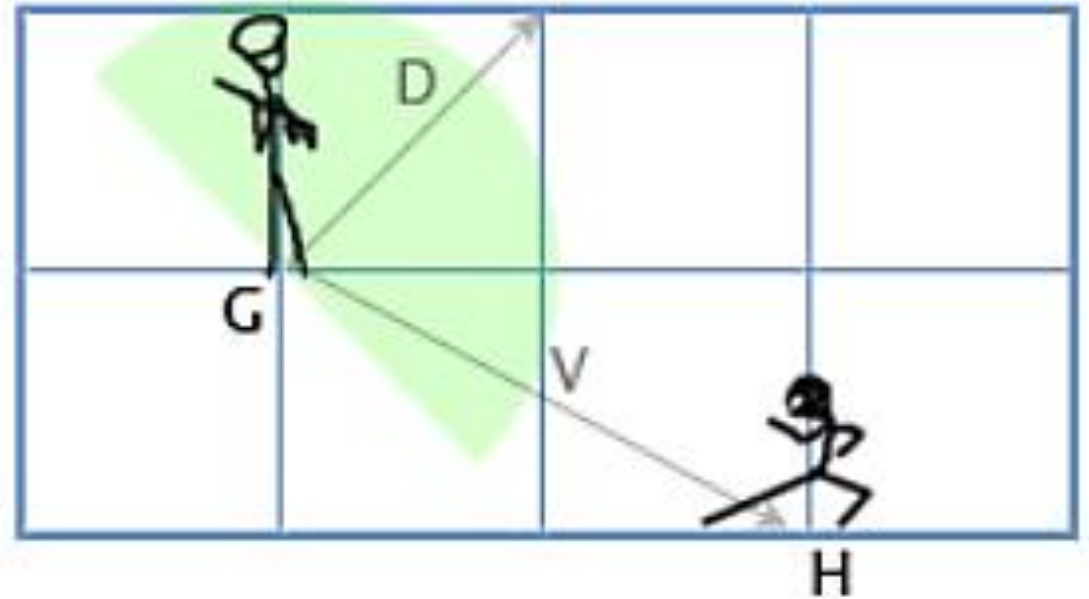
`Vector3.Dot(VectorA, VectorB)`



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:

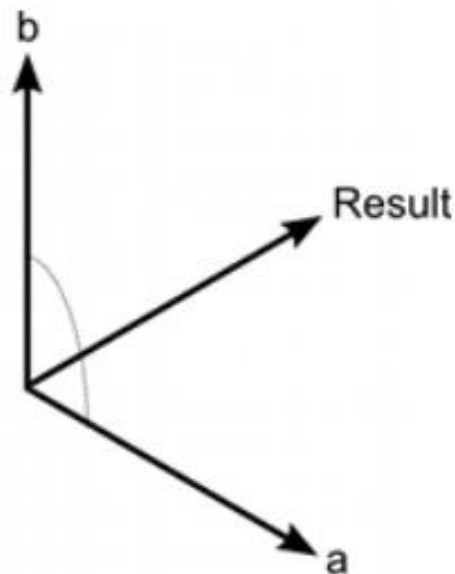


$$V = H - G = (3, 2) - (1, 3) = (3-1, 2-3) = (2, -1)$$

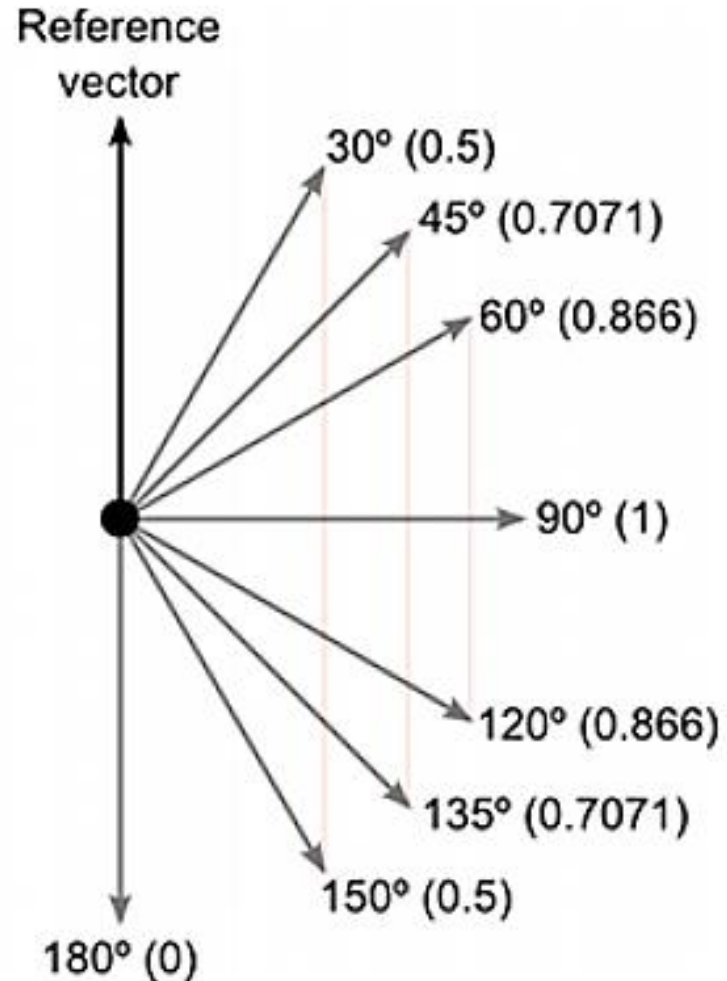
$$D \cdot V = (1, 1) \cdot (2, -1) = 1 \cdot 2 + 1 \cdot (-1) = 2 - 1 = 1$$

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

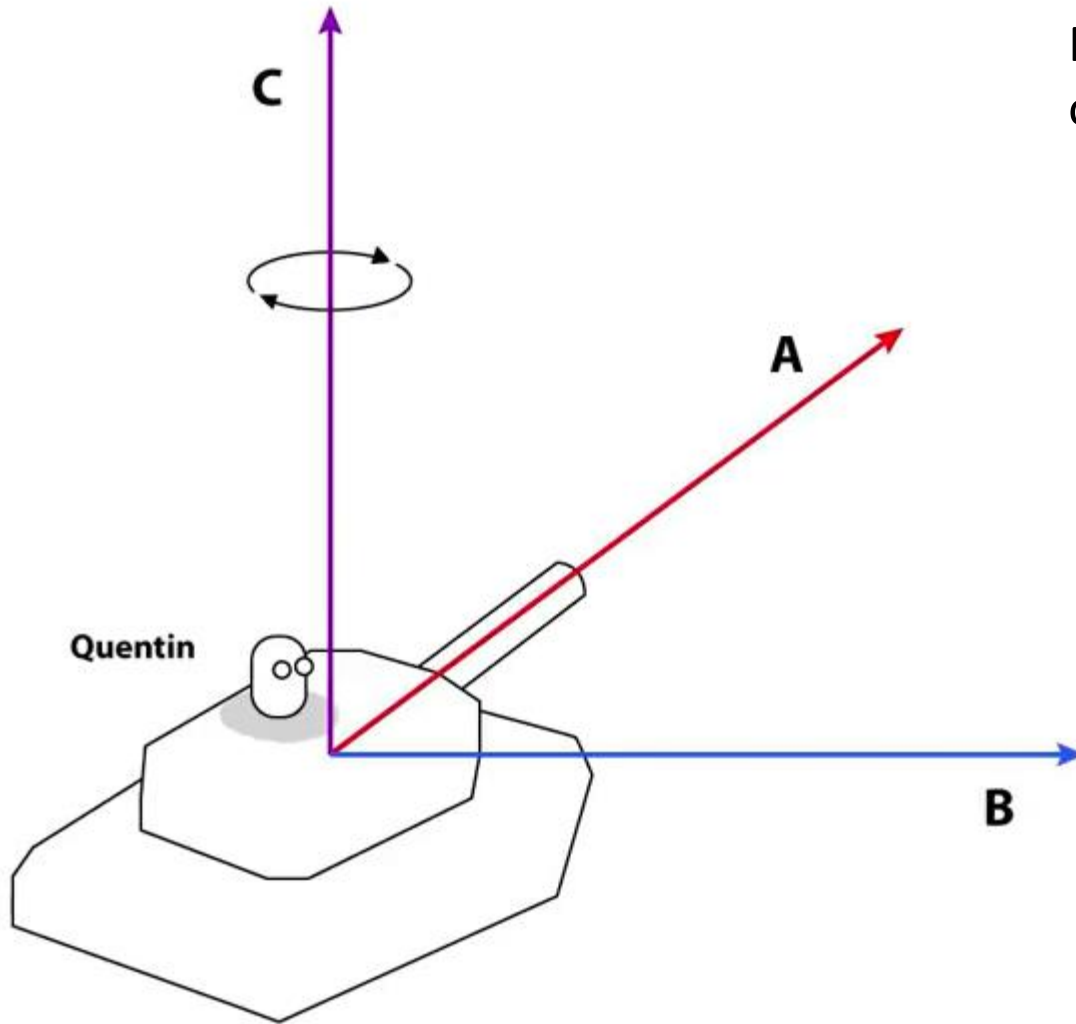


Величина результата равна произведению величин исходных векторов, умноженному на синус угла между ними.

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


Векторное произведение. Применение

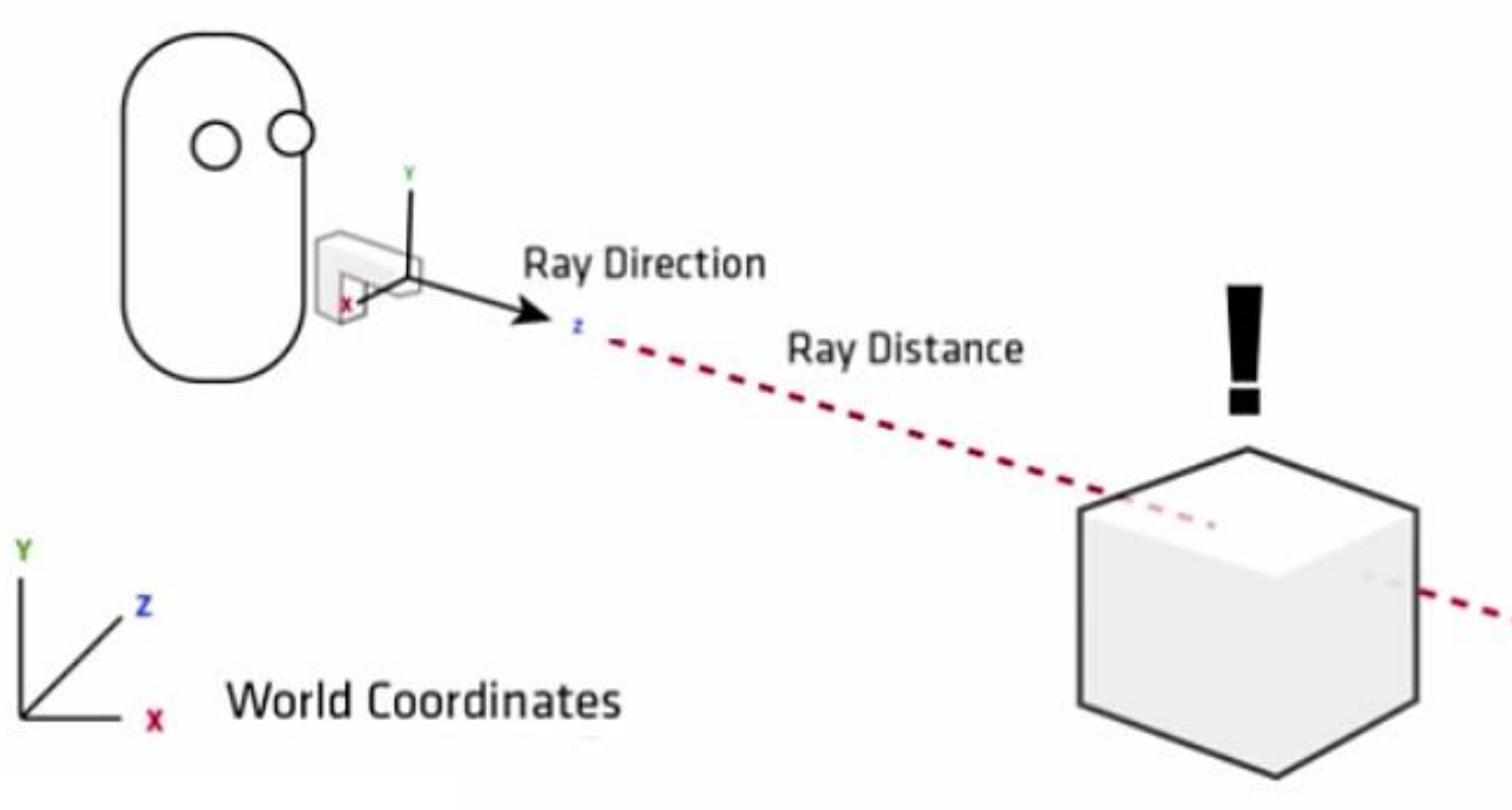
Позволяет вычислить нормаль **C** к поверхности, образованной двумя векторами **A** и **B**



Лучи. Рэйкастинг.

Raycast

Physics.Raycast

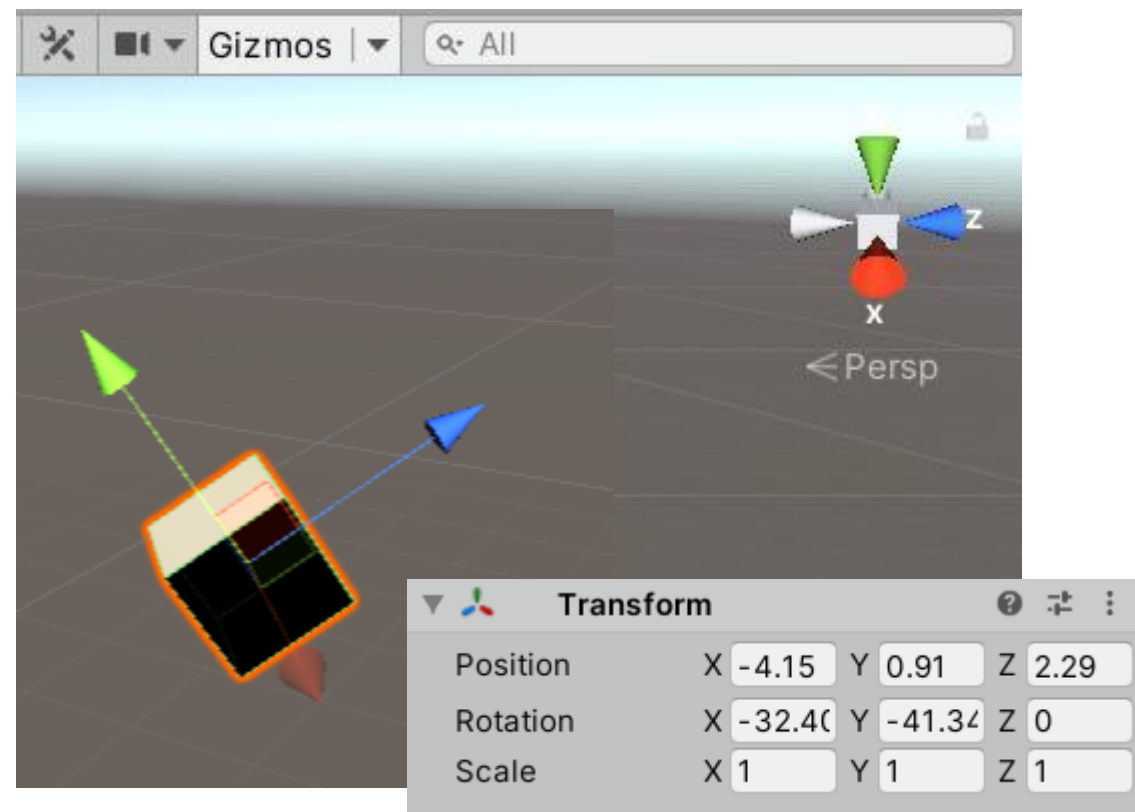
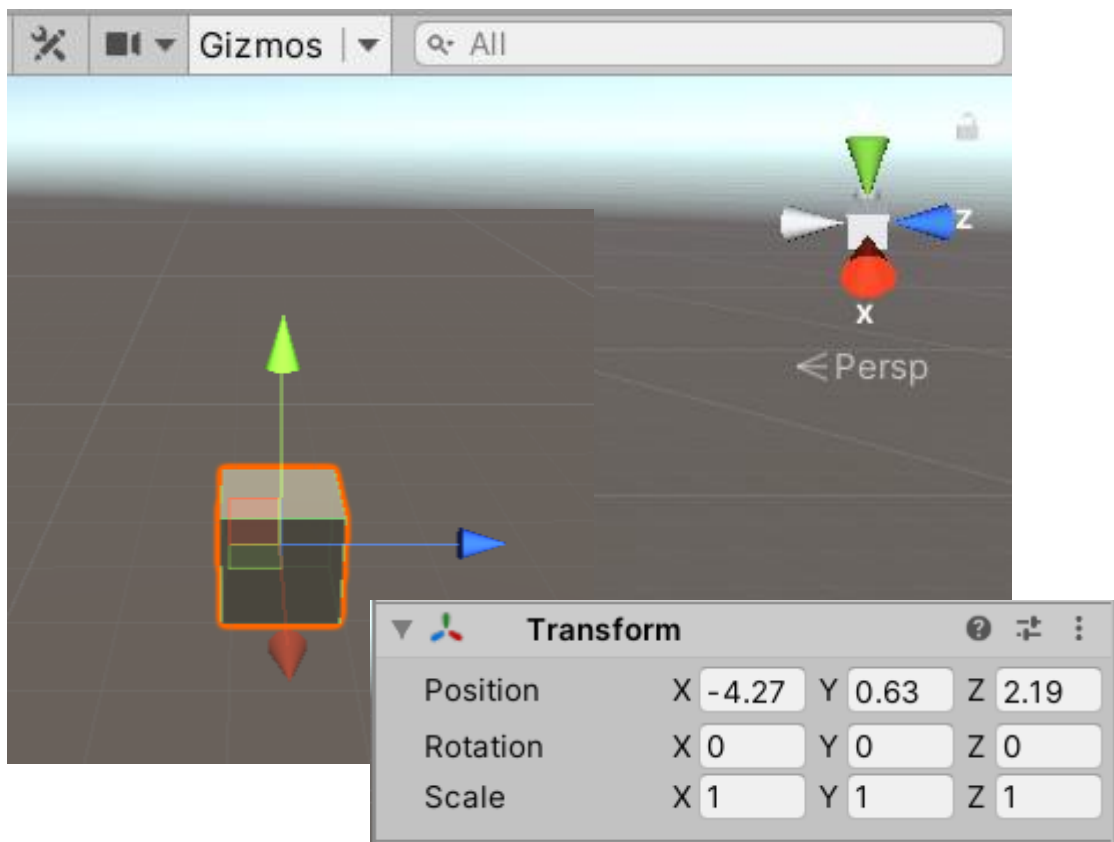


```
public static bool Raycast (  
    Vector3 origin,  
    Vector3 direction,  
    out RaycastHit hitInfo,  
    float maxDistance,  
    int layerMask,  
    QueryTriggerInteraction queryTriggerInteraction  
)
```

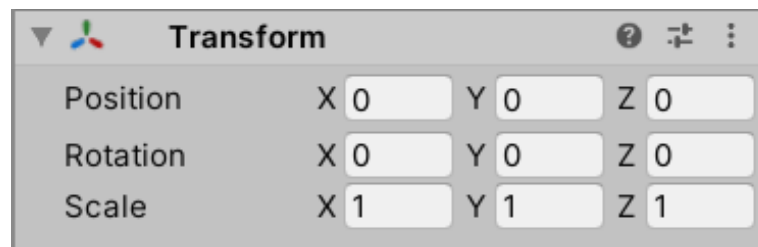
Raycast

origin	Начальная точка луча в мировых координатах.
direction	Направление луча.
maxDistance	Максимальное расстояние, которое луч должен проверять на наличие столкновений.
layerMask	Маска слоя, которая используется для выборочного игнорирования коллайдеров при бросании луча.
queryTriggerInteraction	Указывает, должен ли этот запрос удалять триггеры.

Глобальная и локальная система координат



После команды **Reset** в контекстном меню
координаты обнуляются



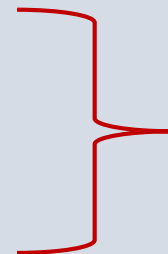
Как можно задавать направление?

Три основных направления совпадают с направлением осей:

up-y forward-z right-x

transform.up
transform.forward
transform.right

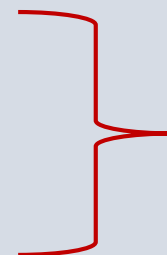
transform.-up
transform.-forward
transform.-right



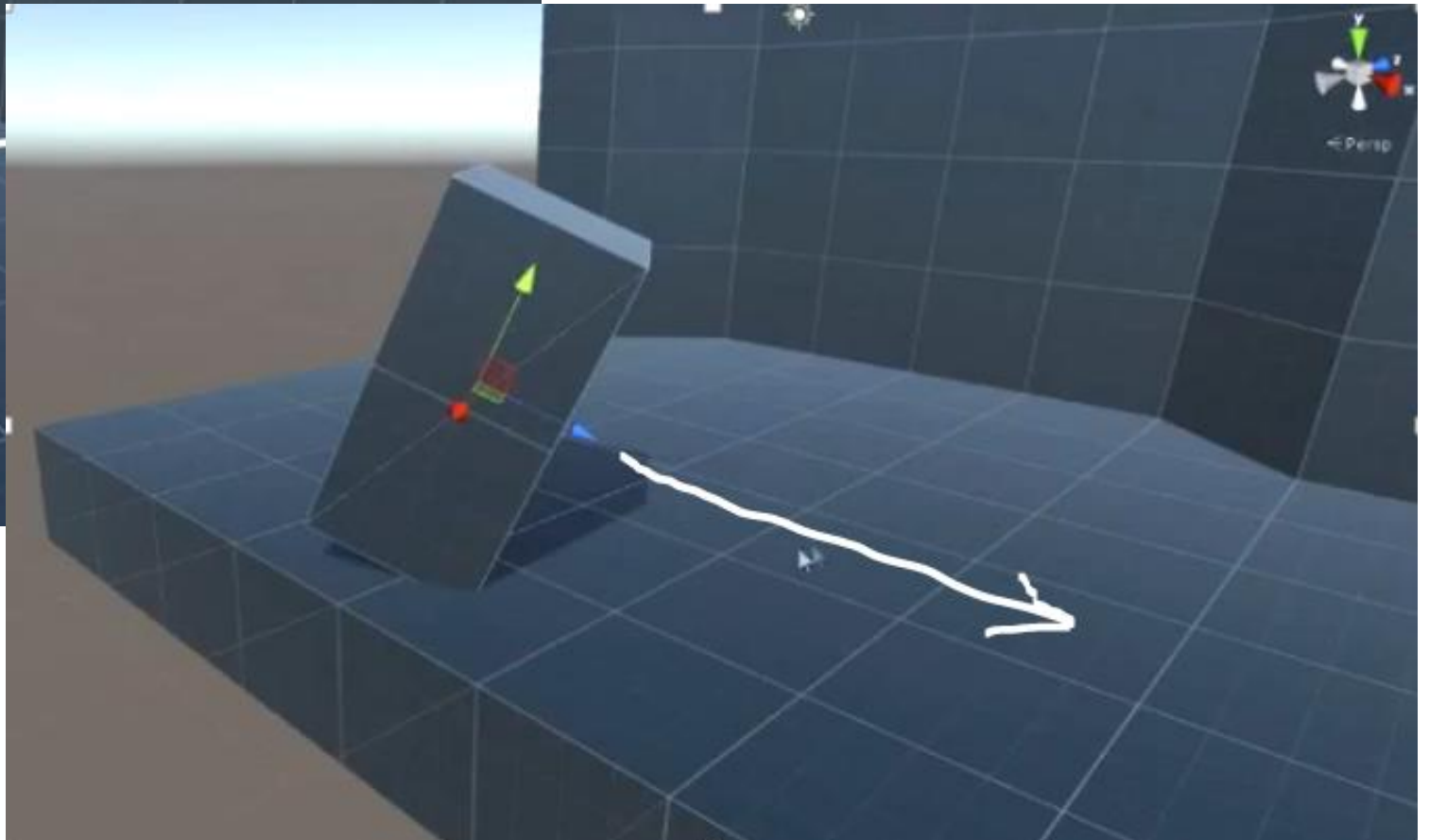
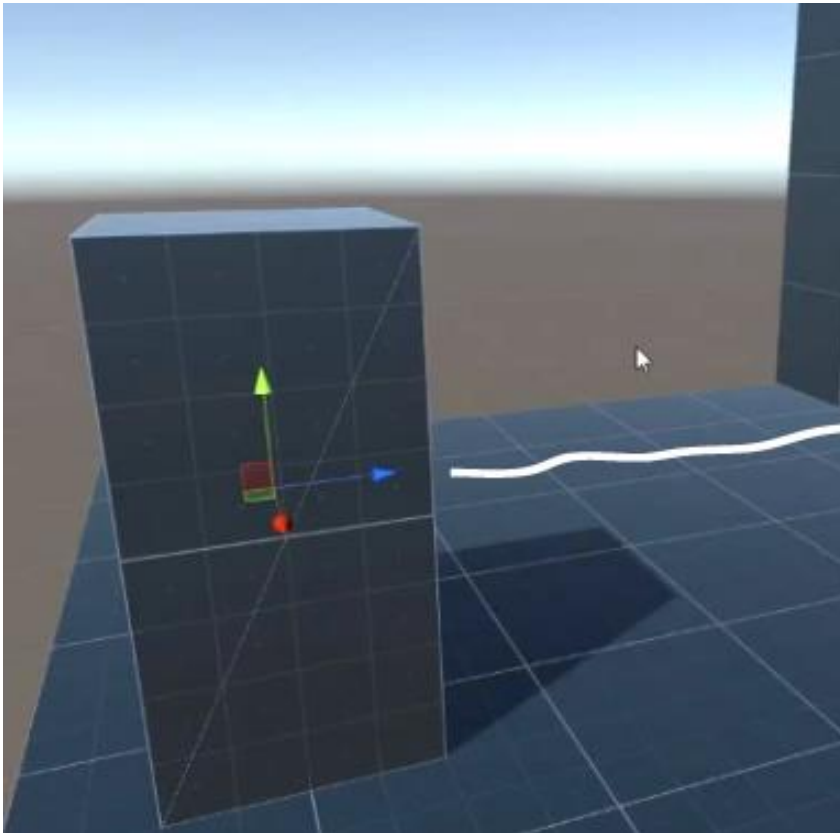
Локальные
координаты

Vector3.up
Vector3.forward
Vector3.right

Vector3.-up
Vector3.-forward
Vector3.-right



Глобальные
координаты



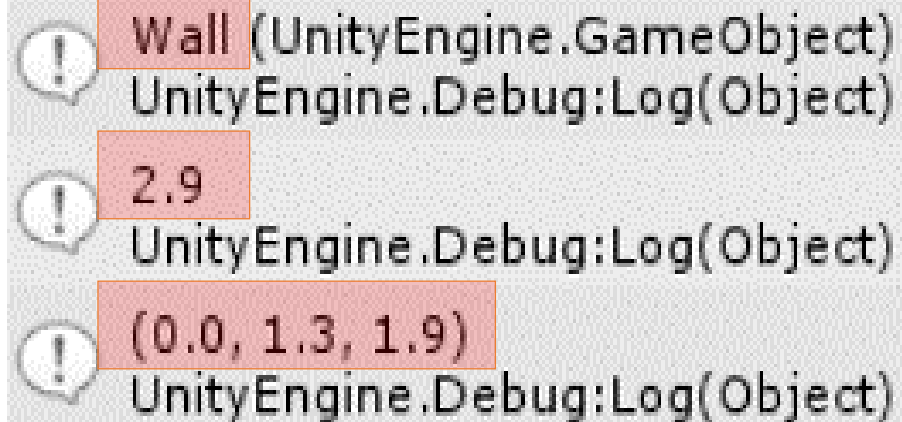
transform.up*7.0f



transform. - forward *1.0f

Raycast

```
RaycastHit info;  
if (Physics.Raycast (transform.position,transform.forward*4f, out info,4f))  
{  
    Debug.Log(info.collider.gameObject);  
    Debug.Log(info.distance);  
    Debug.Log(info.point);  
}
```



The screenshot shows three log entries in the Unity console, each preceded by a speech bubble icon containing an exclamation mark. The first entry is 'Wall (UnityEngine.GameObject)' with 'UnityEngine.Debug:Log(Object)' below it. The second entry is '2.9' with 'UnityEngine.Debug:Log(Object)' below it. The third entry is '(0.0, 1.3, 1.9)' with 'UnityEngine.Debug:Log(Object)' below it. The text 'Wall', '2.9', and '(0.0, 1.3, 1.9)' are highlighted with red boxes.

Wall (UnityEngine.GameObject)
UnityEngine.Debug:Log(Object)
2.9
UnityEngine.Debug:Log(Object)
(0.0, 1.3, 1.9)
UnityEngine.Debug:Log(Object)

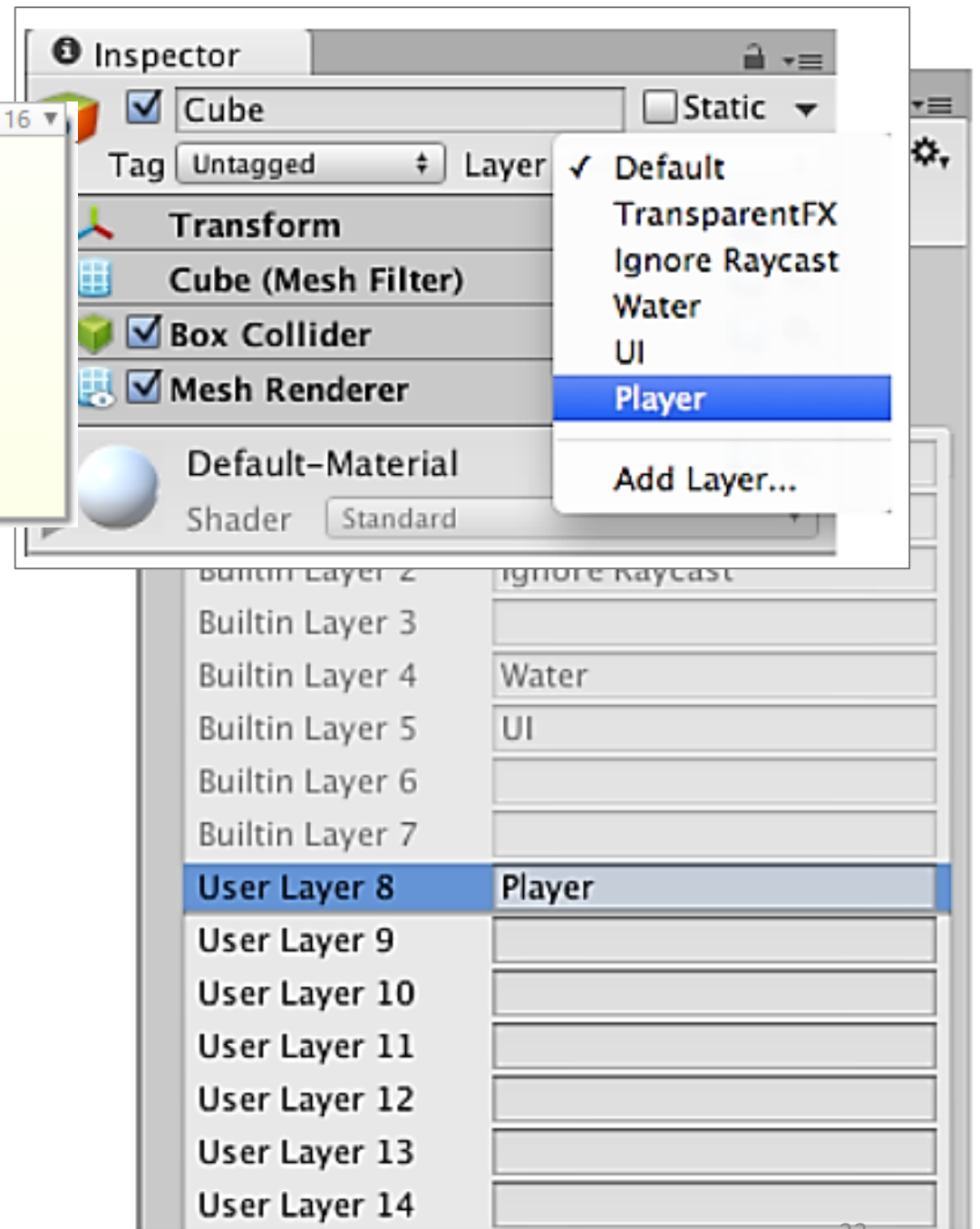
info.rigidbody

.....

info.transform

.....

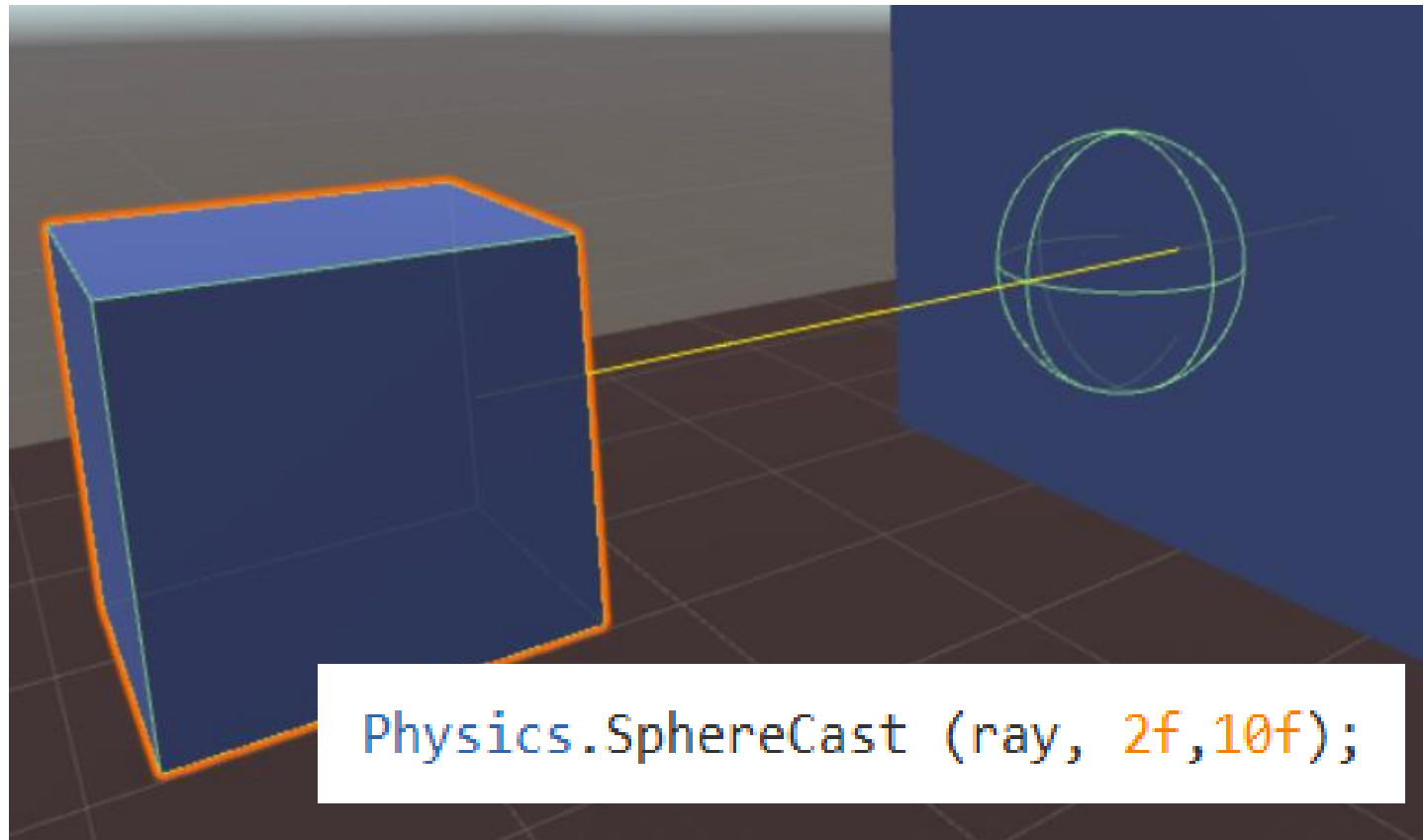

```
public static bool Raycast (
    Vector3 origin,
    Vector3 direction,
    out RaycastHit hitInfo,
    float maxDistance,
    int layerMask,
    QueryTriggerInteraction queryTriggerInteraction
)
```



`int layerMask = DefaultRaycastLayers`

Слои чаще всего используются Камерами для визуализации только части сцены и Светами для освещения только частей сцены. Но их также можно использовать с помощью **raycasting**, чтобы выборочно игнорировать коллайдеры или создавать столкновения.

Physics.SphereCast

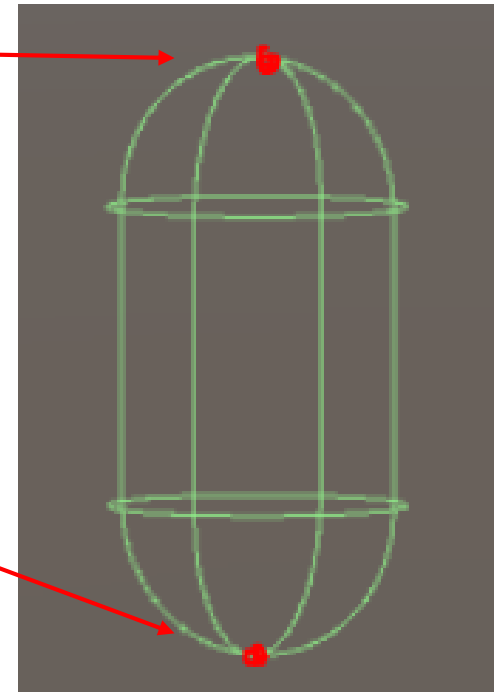


Physics.CapsuleCast

```
public static bool CapsuleCast (  
    Vector3 point1,  
    Vector3 point2,  
    float radius,  
    Vector3 direction  
)
```

```
public static bool CapsuleCast (  
    Vector3 point1,  
    Vector3 point2,  
    float radius,  
    Vector3 direction,  
    out RaycastHit hitInfo,  
    float maxDistance,  
    int layerMask,  
    QueryTriggerInteraction queryTriggerInteraction  
)
```

Physics.CapsuleCast



```
public float speed = 5.0f;  
public float obstacleRange = 0.5f;
```



Демо

ССЫЛОК: 0

```
void Update()  
{  
    transform.Translate(0, 0, speed * Time.deltaTime);  
    Ray ray = new Ray(transform.position, transform.forward);  
    RaycastHit hit;  
    if (Physics.SphereCast(ray, 1.0f, out hit))  
    {  
        if (hit.distance < obstacleRange)  
        {  
            float angle = Random.Range(-110, 110);  
            transform.Rotate(0, angle, 0);  
        }  
    }  
}
```

Задача:

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.

Выход:


- InvokeRepeating
- Coroutine

MonoBehaviour.InvokeRepeating

public void **InvokeRepeating**(string **methodName**, float **time**, float **repeatRate**);

*Вызывает метод **methodName** по истечении времени **time** секунд, затем повторяет вызов каждые **repeatRate** секунд.*

```
7 public class Repeat : MonoBehaviour
8 {
9     public Rigidbody ob;
10
11     // Запуск через 2 секунды.
12     // снаряд будет запускаться каждые 0,3 секунды
13     // Ссылка: 0
14     void Start()
15     {
16         InvokeRepeating("LaunchProjectile", 2.0f, 0.3f);
17     }
18     // Ссылка: 0
19     void LaunchProjectile()
20     {
21         Rigidbody instance = Instantiate(ob);
22         instance.velocity = Random.insideUnitSphere * 5;
23     }
24 }
```



Корутины

Корутины (сопрограммы)

Когда вызывается функция, 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;
    }
}
```


Корутины (сопрограммы)

✓ Объявление

```
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;  
    }  
}
```

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

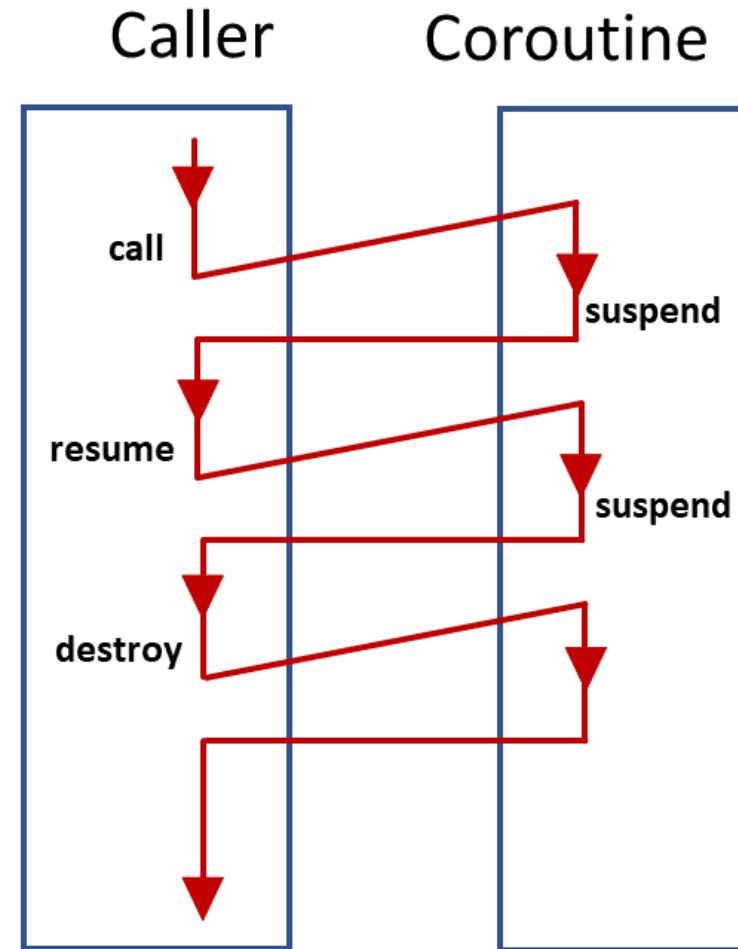
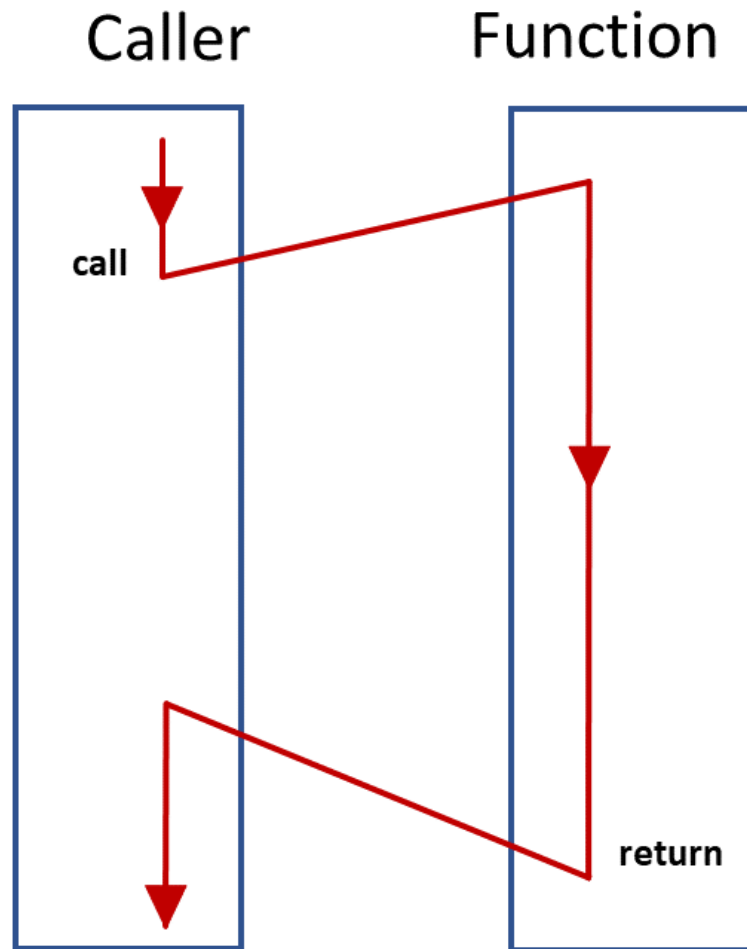
Можно ввести временную задержку
`yield return new WaitForSeconds(.1f);`

✓ Вызов

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

Корутины представляют собой простые C# итераторы, возвращающие **IEnumerator** и использующие ключевое слово **yield**. В Unity корутины регистрируются и выполняются до первого **yield** с помощью метода **StartCoroutine**. Далее Unity опрашивает зарегистрированные корутины после каждого вызова **Update** и перед вызовом **LateUpdate**, определяя по возвращаемому в **yield** значению, когда нужно переходить к следующему блоку кода.

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



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
```

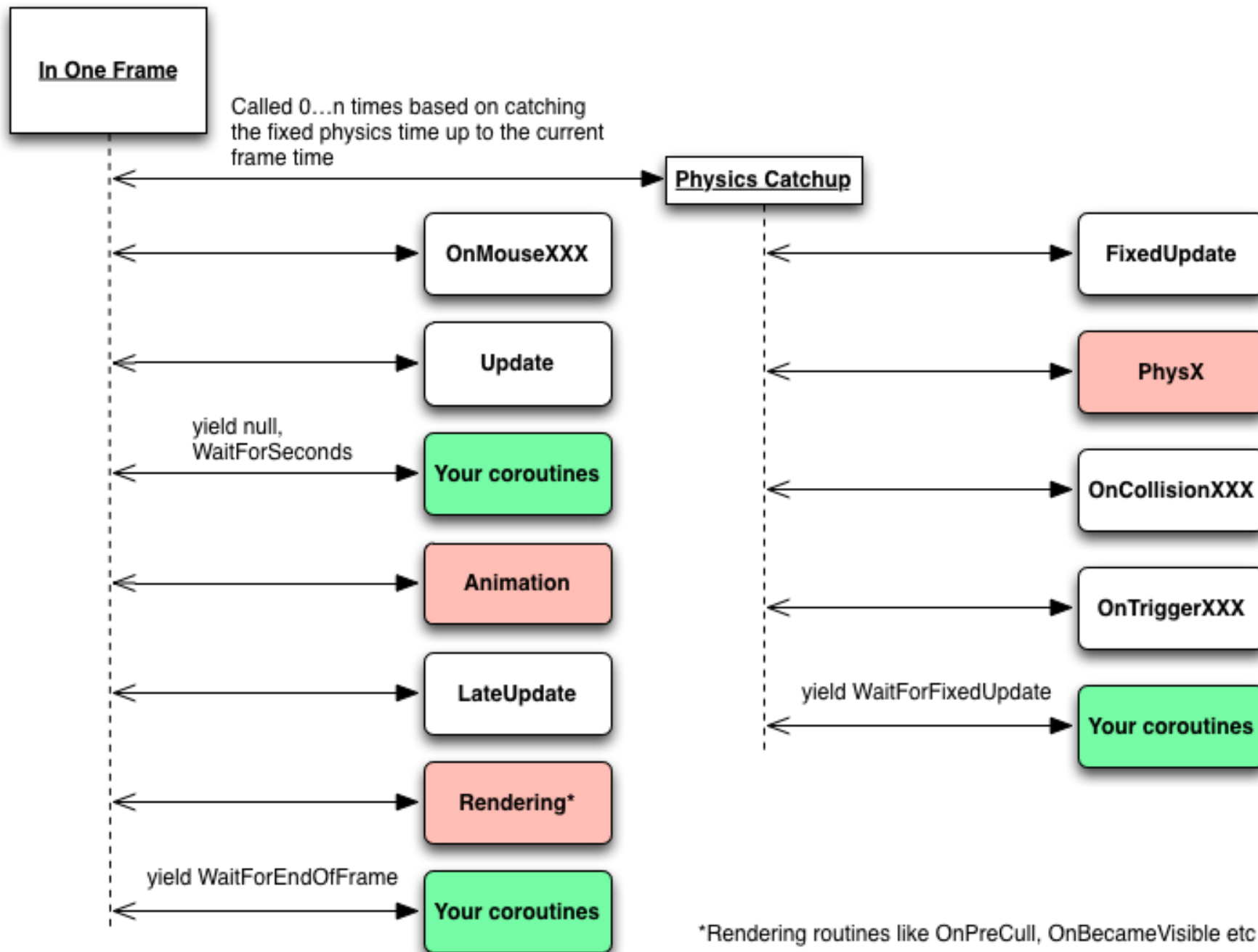
Продолжить по завершению другого корутина:

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

Продолжить после загрузки удаленного ресурса:

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

<https://habr.com/post/216185/>



Корутины не являются асинхронными, они выполняются в основном потоке приложения, в том же, что и отрисовка кадров, инстансирование объектов и т.д., если заблокировать поток в корутине, то остановится все приложение, корутины с асинхронностью использовали бы "IEnumerator", который Unity не поддерживает. Корутина позволяет растянуть выполнение на несколько кадров, что бы не нагружать 1 кадр большими вычислениями.

*Rendering routines like OnPreCull, OnBecameVisible etc are called here

Корутину можно применять для длительных операций, которые можно "размазать" по кадрам.

Пример с генерацией пуль:

```
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
}
```

```
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");
}
```

Мы хотим, чтобы до прогона действий и после что-то происходило, например логирование сообщений сделано что-то или нет. Как это делать в InvokeRepeating или Update? Вешать всякий флаги было сделано что-то или нет, зашел в метод или нет? Зачем, если можно сделать её в корутине

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);  
    }  
}
```

Углы Эйлера и Кватернионы

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

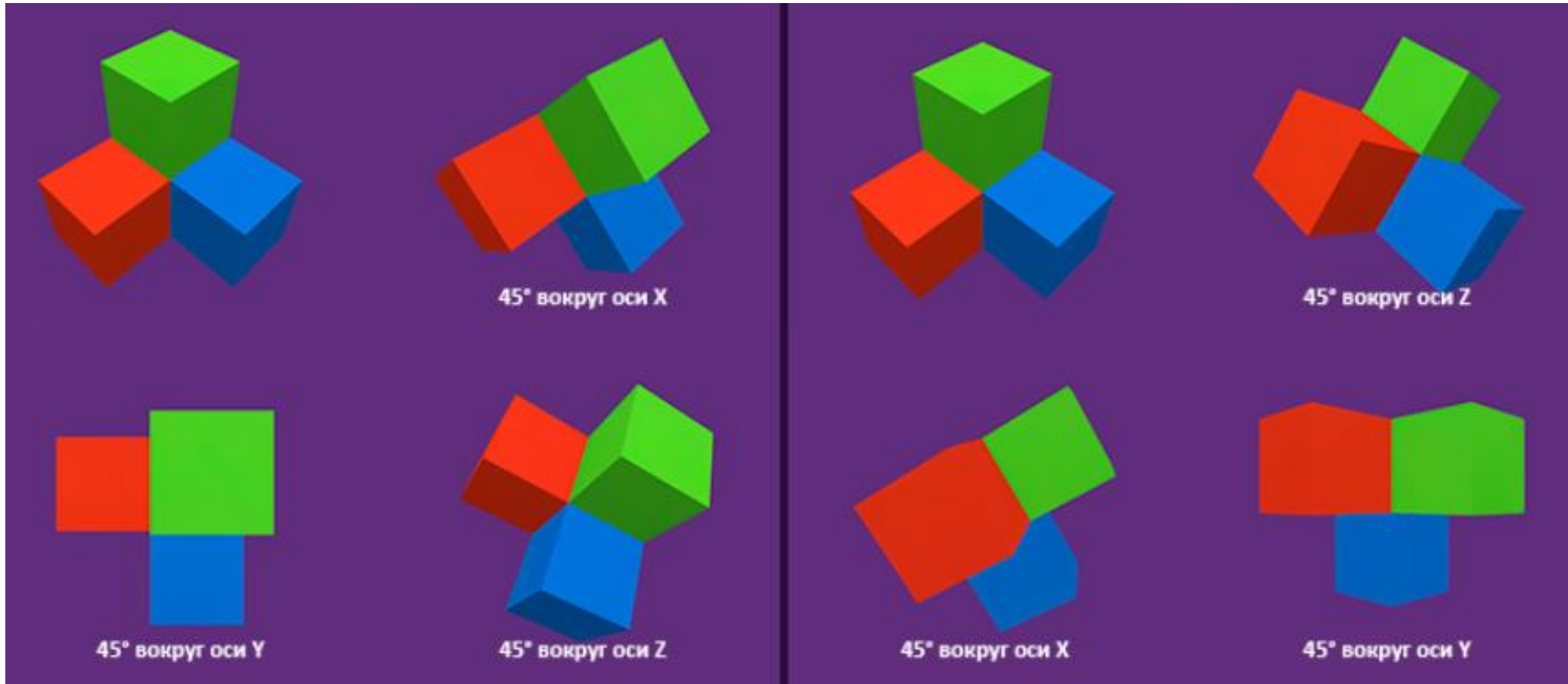
- ✓ **Преимущество** : вращения кватернионов не страдают от блокировки кардана.
- ✓ **Ограничение** : числовое представление Quaternion интуитивно непонятно.

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

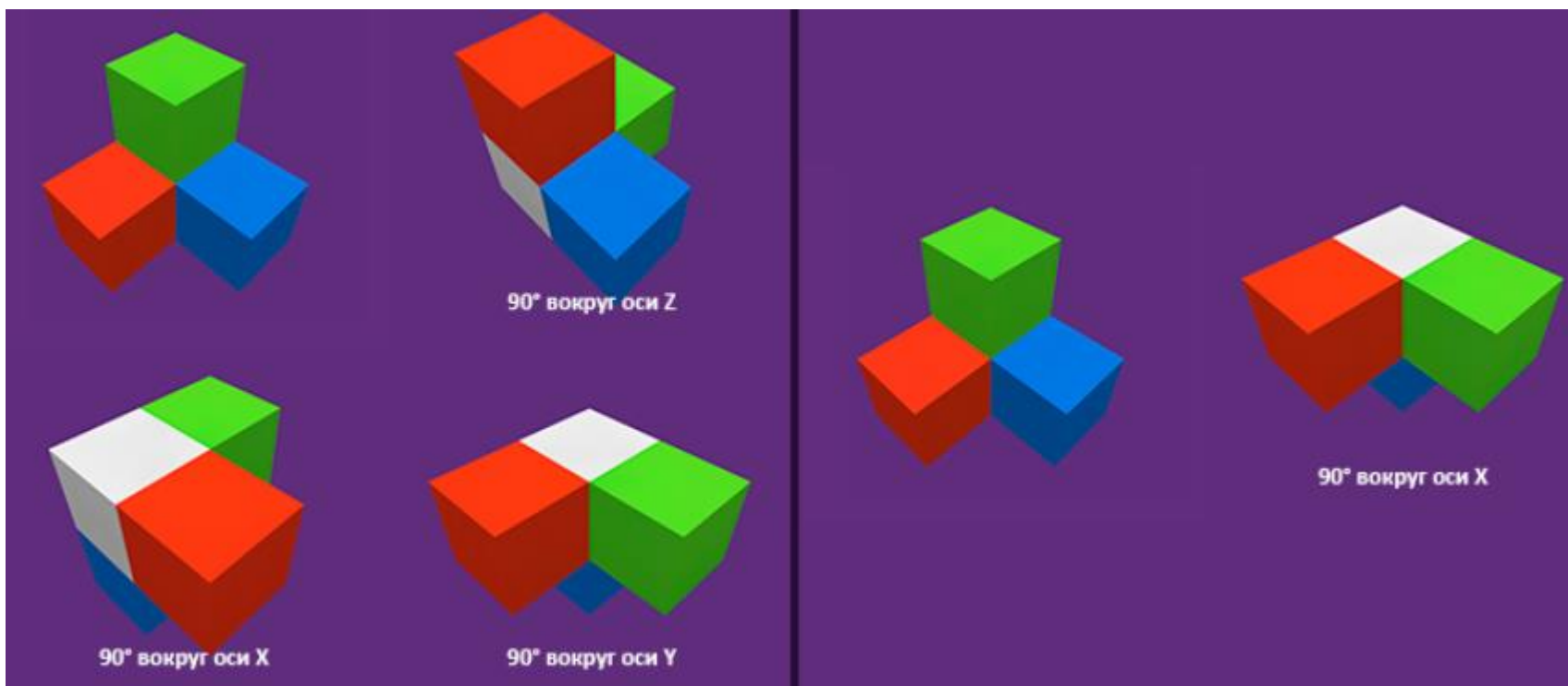
1. Результат поворота зависит от порядка поворотов по осям



Углы Эйлера

2. Шарнирный замок

Если вращение 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

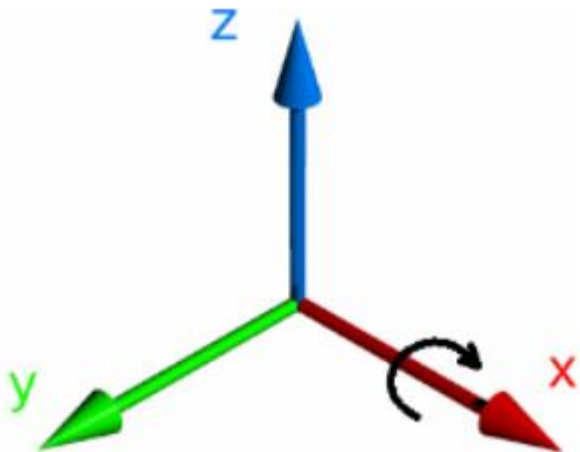
Кватернионы — система гиперкомплексных чисел, образующая векторное пространство размерностью четыре над полем вещественных чисел.

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);  
}
```

Кватернионы используются для представления вращений.

В Unity все вращения представлены в виде кватернионов. Их использование решает проблему "шарнирного замка" (gimbal lock).

Для начала разберемся как получить кватернион из привычных векторов и углов.

Первый способ (углы указываются в градусах):

```
//Изменяем вращение объекта  
transform.rotation.eulerAngles = new Vector3(0f, 90f, 0f);
```

Второй способ (значения вращения по каждой оси в градусах):

```
Quaternion.Euler(float x, float y, float z);  
//или  
Quaternion.Euler(Vector3 euler);
```

Третий способ (через ось и угол вращения вокруг неё в градусах):

```
//По оси и углу поворота  
Quaternion.AngleAxis(Vector3 axis, float angle);
```

Получить углы Эйлера из кватерниона можно через свойство eulerAngles:

```
//Выводим вращение объекта в привычных значениях  
Debug.Log(transform.rotation.eulerAngles);
```

Или можно получить ось и угол вращения вокруг нее:

```
float angle = 0f;  
Vector3 axis;  
//Получение оси и угла поворота  
transform.rotation.ToAngleAxis(out angle, out axis);  
Debug.Log("Angle: " + angle + " Axis: " + axis);
```

Теперь разберемся, как с их помощью вращать объекты.

Для этого нужно кватернион, из которого мы хотим повернуть (текущее положение), умножить на кватернион, на который хотим повернуть.

Пример:

```
//Поворачиваем объект на 90 градусов  
transform.rotation *= Quaternion.Euler(0f, 90f, 0f);
```

Но кватернионы - это тот случай, когда от перемены мест множителей произведение меняется. Поэтому такой код даст иной результат:

```
transform.rotation = Quaternion.Euler(0f, 90f, 0f) * transform.rotation;
```


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);

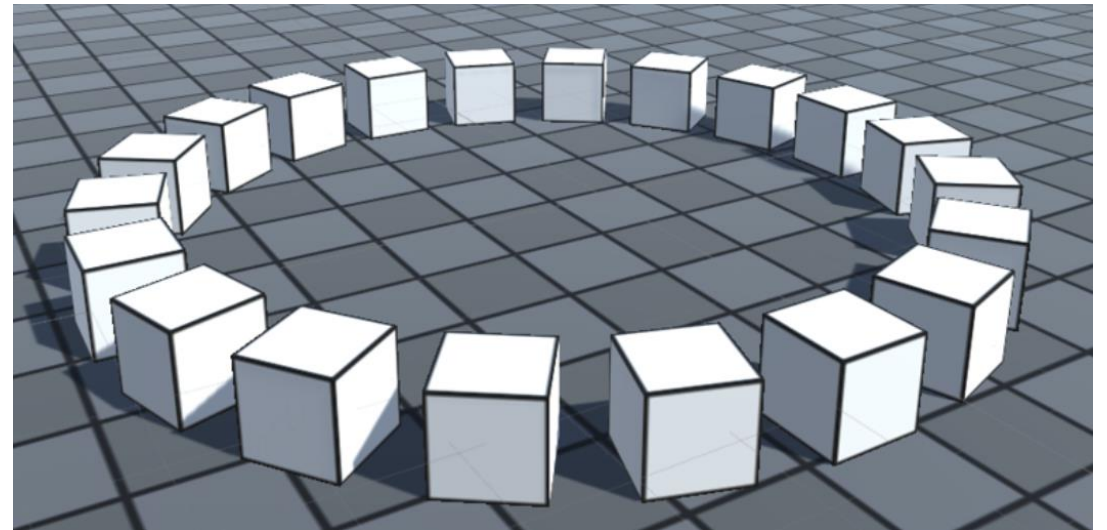
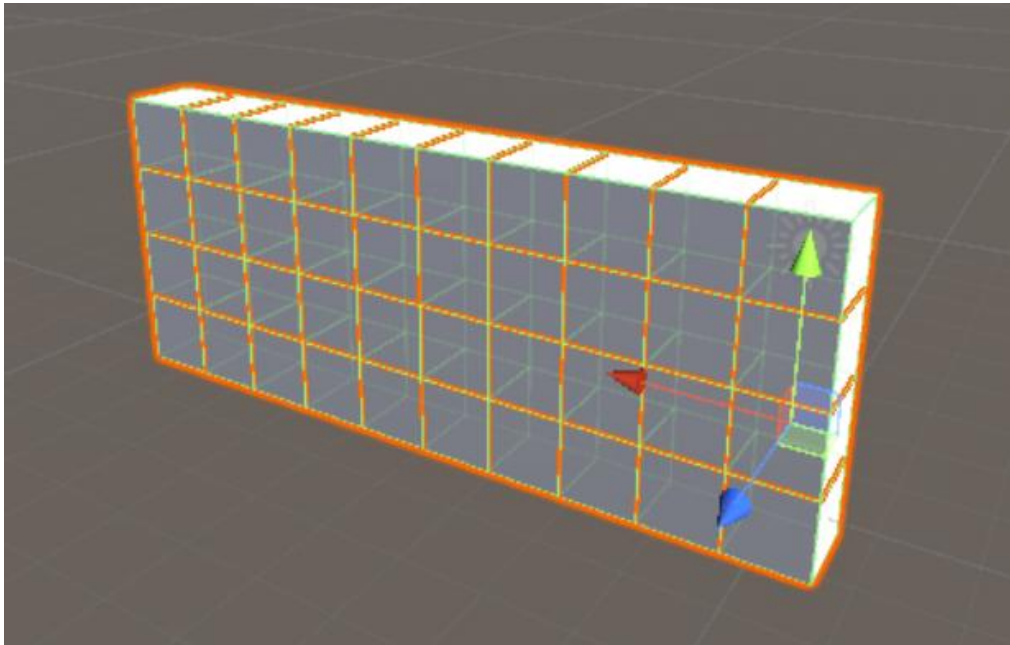
Процедурная генерация префабов

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

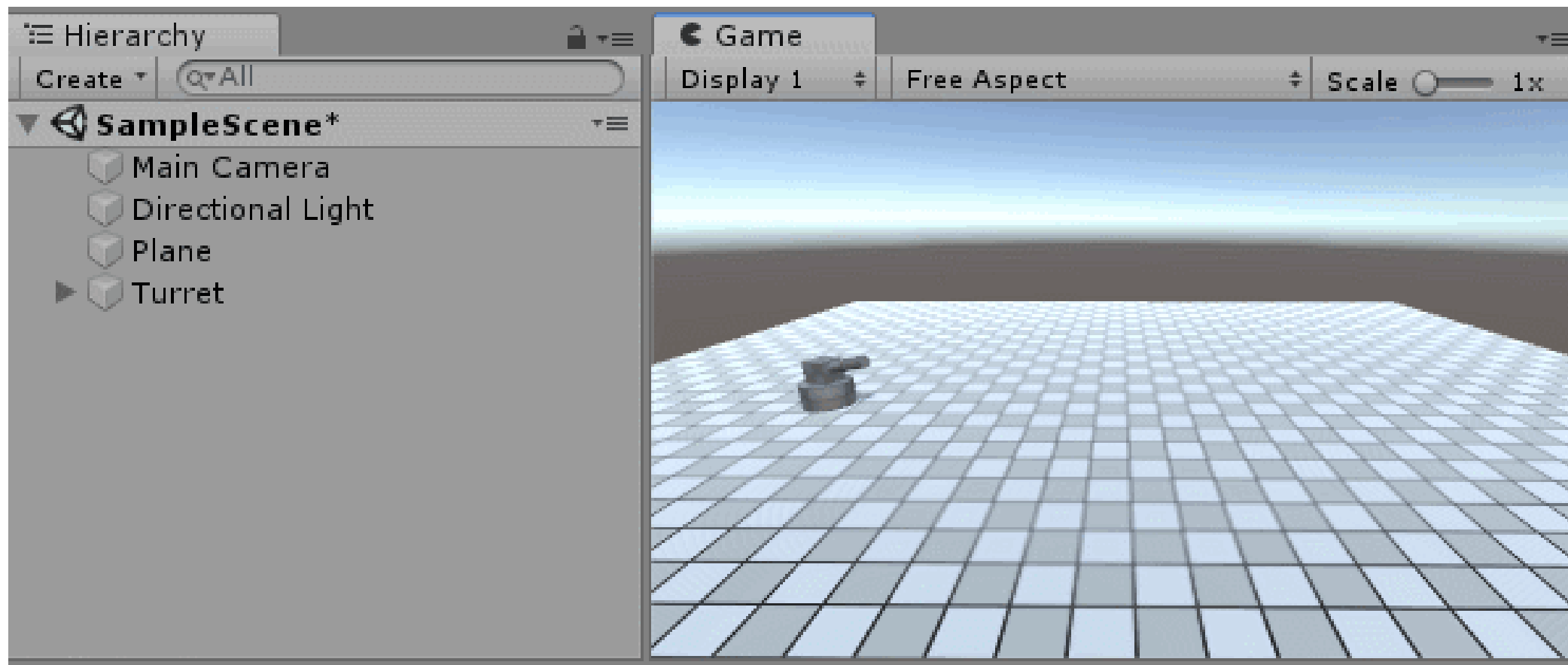
    // This script will simply instantiate the Prefab when the game starts.
    Ссылка: 0
    void Start()
    {
        // Instantiate at position (0, 0, 0) and zero rotation.
        Instantiate(myPrefab, new Vector3(0, 0, 0), Quaternion.identity);
    }
}
```

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

