

Лабораторная работа. Создание 2D-игры Snake.

Данный урок создан на основе следующего видео-урока: [How to make Snake in Unity \(Complete Tutorial\)](#)

Цель работы: Создание 2D-игры Snake с использованием игрового движка Unity и инструментов разработки, что позволяет получить практические навыки в программировании, проектировании игры и работе с графикой.

Задачи:

1. Разработка концепции игры:

- Определение основных механик и правил игры.
- Разработка схемы управления и взаимодействия пользователя.

2. Создание пользовательского интерфейса (UI):

- Разработка основного экрана игры.

3. Программирование игрового процесса:

- Реализация движения змейки.
- Обработка столкновений (с едой, границами игрового поля и самой змейкой).
- Реализация системы увеличения змейки при поглощении еды.
- Реализация случайного появления еды на игровом поле.
- Реализация системы очков

4. Тестирование и отладка:

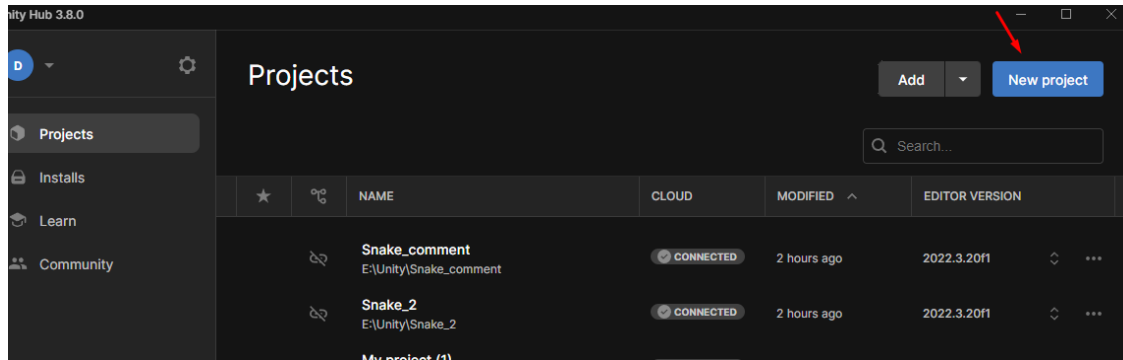
- Проведение тестирования для выявления багов и ошибок.
- Оптимизация игрового процесса для улучшения производительности.

5. Финальные настройки и сборка проекта:

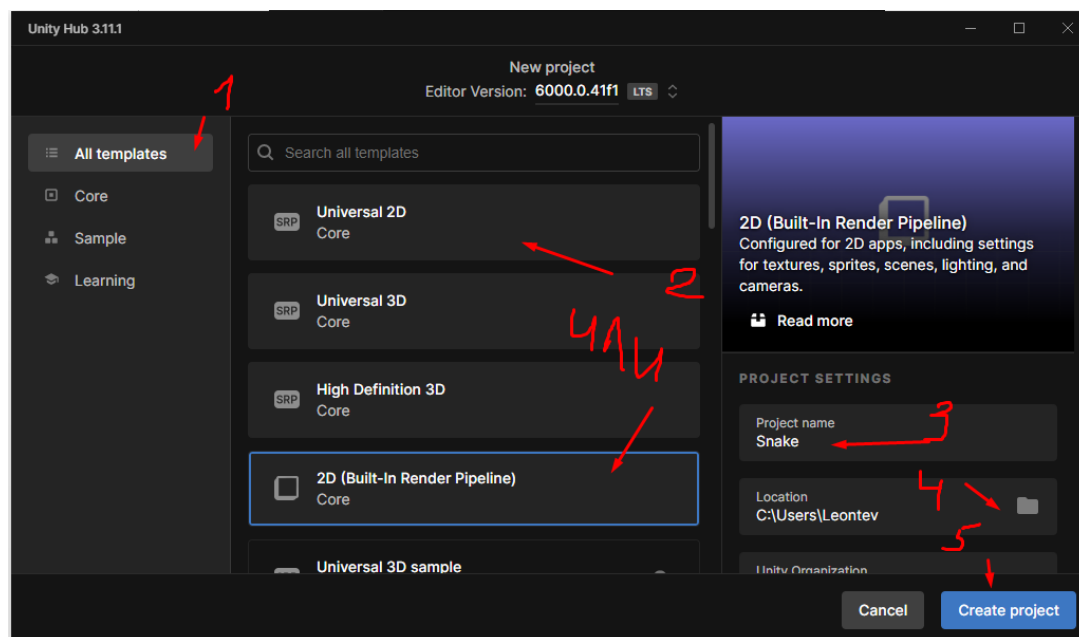
- Провести рефакторинг кода для улучшения читаемости и производительности.
- Скомпилировать и сохранить финальную версию игры.

Ход работы:

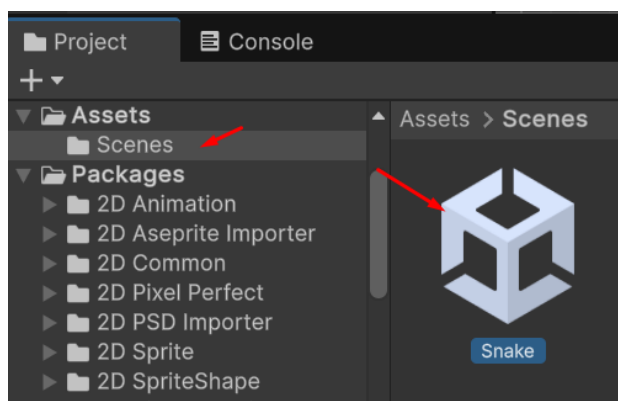
1. Запускаем **Unity Hub**. Создаём новый проект – **New project**:



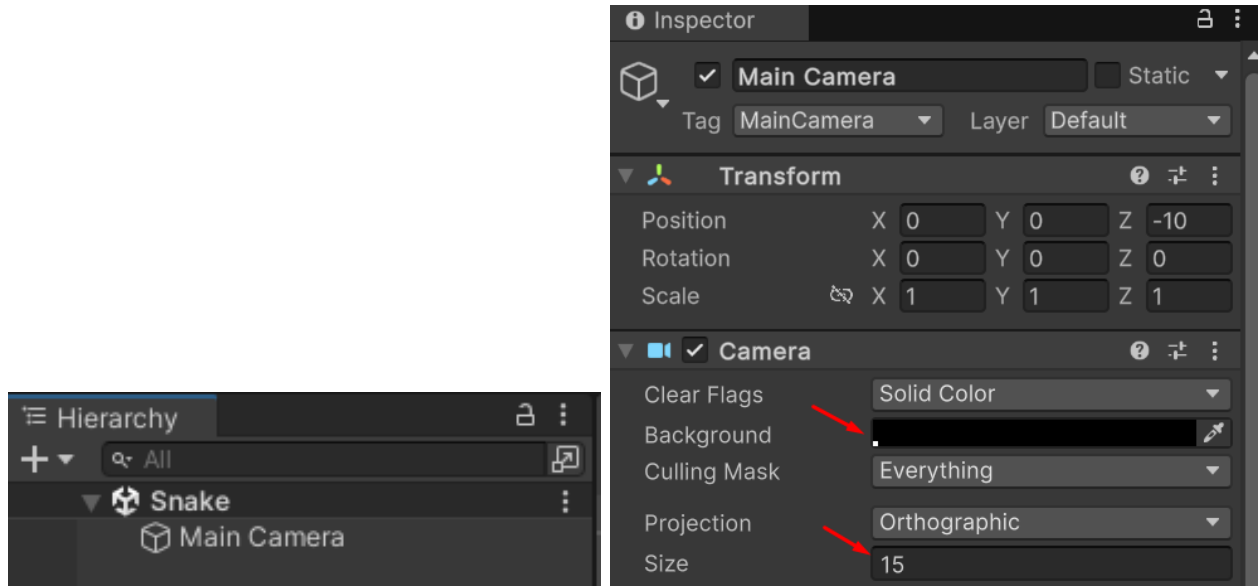
Далее мы можем выбрать из двух шаблонов – **Universal 2D** (используется в новых версиях Unity, имеет более продвинутую работу со светом, более удобную работу под мобильные игры и т.д.) или же **2D (Built-In Render Pipeline)** – более простой шаблон, но он меньше весит (порядка 500 Мб). В качестве примера я выберу второй вариант. Затем вводим **название проекта - Snake**, выбираем **место расположения** и нажимаем **Create project**:



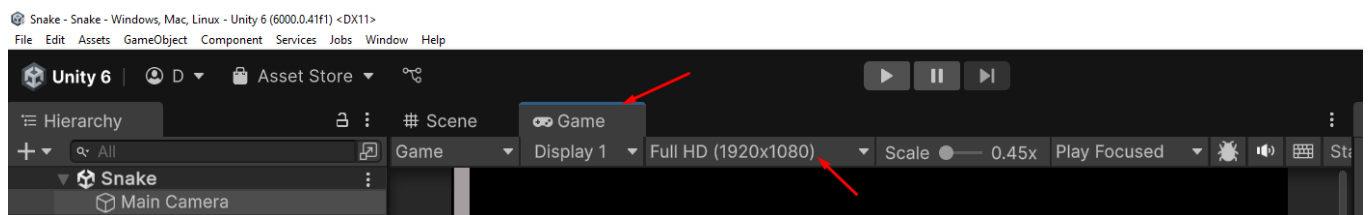
2. В папке **Scenes** меняем название сцены, на имя игры - **Snake**:



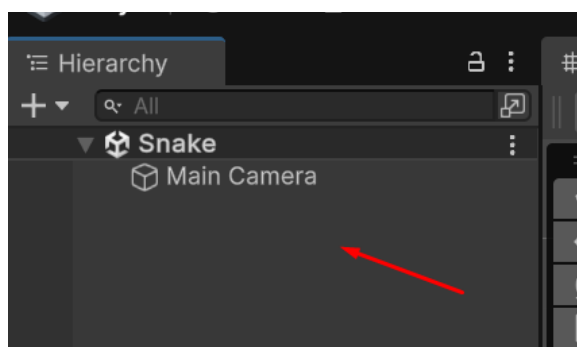
Меняем цвета объекта **Main Camera** (нажимаем на него в **Hierarchy**, и в **Inspector** появляются свойства объекта) на тот, который вам нравится (например, **черный**), и также меняем размер, в нашем случае возьмём **15** (указывает на область, которую видит наша камера, и мы отдадим наш zoom на кратное пяти число, иначе нам бы пришлось в будущем изменять размеры змейки и еды):



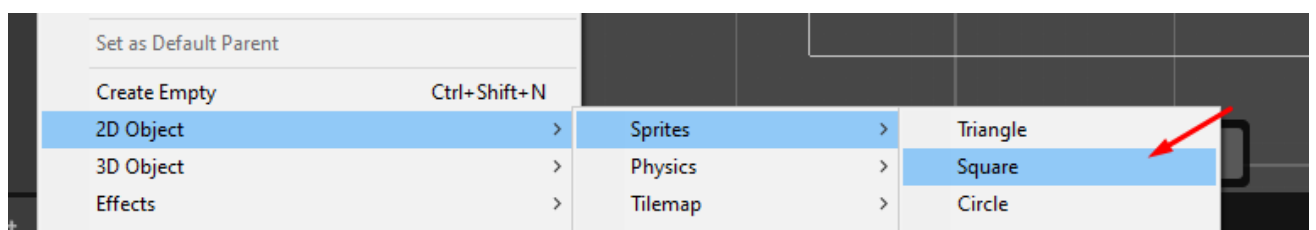
Во вкладке **Game** поменяйте разрешение на **1920x1080**:



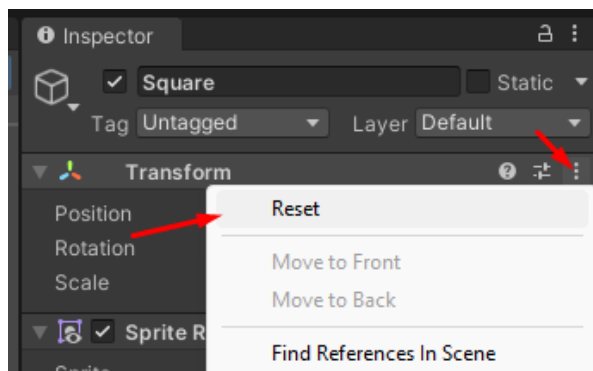
3. Перейдём к созданию игрока. Щёлкаем правой кнопкой мыши в **Hierarchy**:



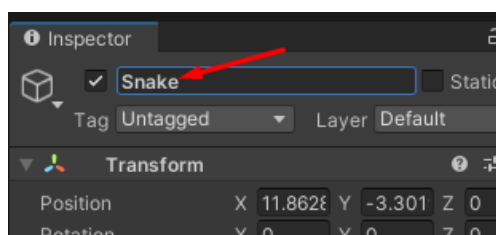
И выбираем **2D Object** → **Sprites** → **Square**:



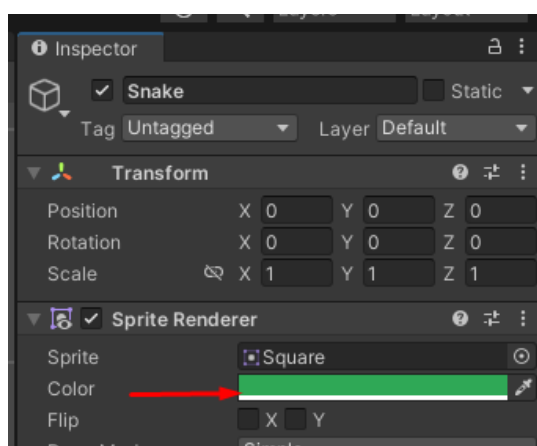
В **Inspector** сбрасываем для нашего объекта трансформацию (не забудьте, что объект должен быть выделен):



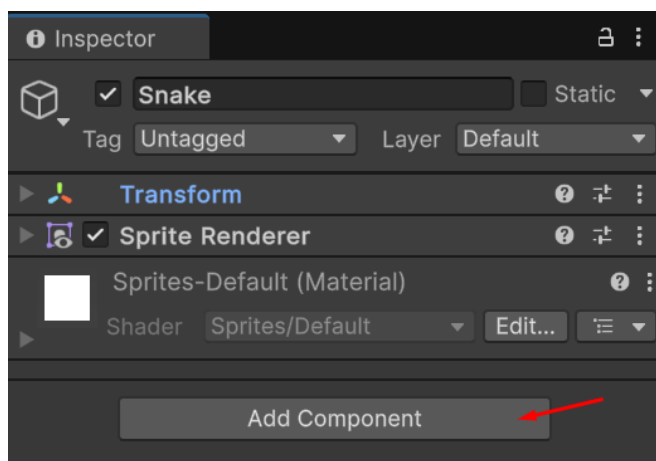
Меняем название на **Snake**:



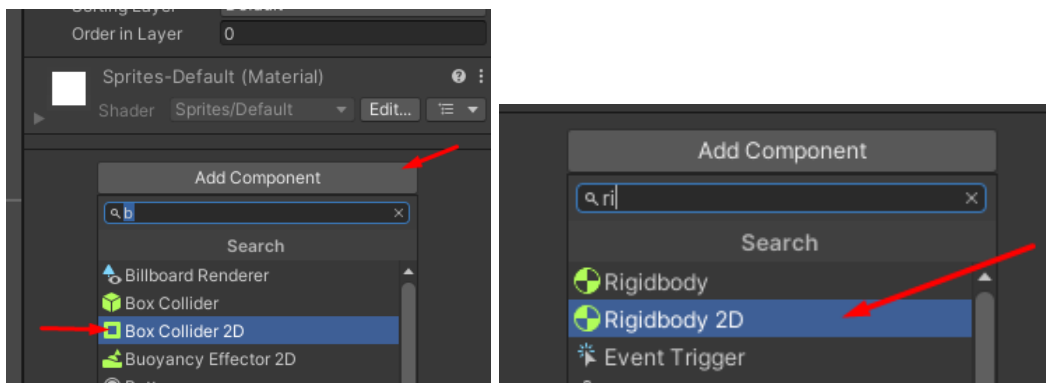
Меняем цвет змейки (например, на зеленый):



Далее добавляем два компонента, нажав на **Add Component**:



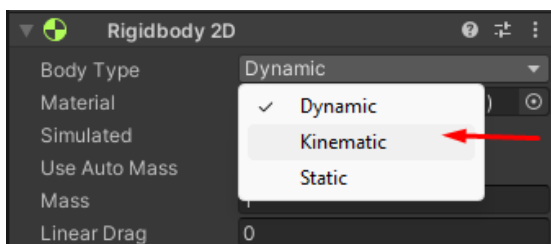
Box Collider 2D и **Rigidbody 2D**:



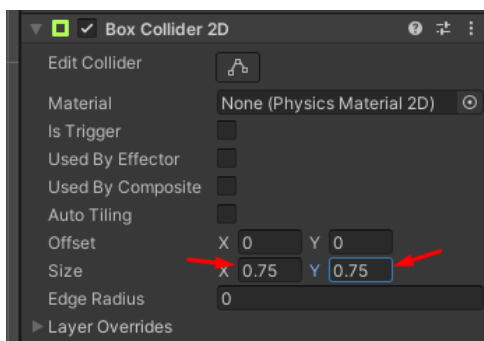
Rigidbody 2D – Добавление компонента (класса) Rigidbody2D к спрайту передает его под контроль физического движка. Само по себе это означает, что на спрайт будет воздействовать сила тяжести, и им можно управлять из скриптов с помощью сил. При добавлении соответствующего компонента collider спрайт также будет реагировать на столкновения с другими спрайтами.

Box Collider 2D – Коллайдер для 2D-физики, представляющий собой прямоугольник, выровненный по оси, влияет на столкновение объектов.

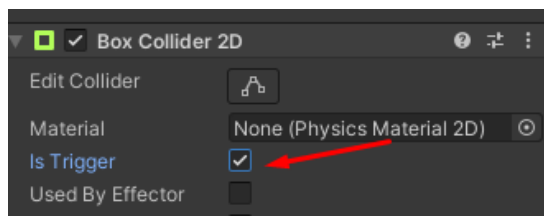
Так как игра у нас простая, на не нужны такие параметры как масса, гравитация и др., поэтому поменяем у компонента **Rigidbody 2D - Body Type** на **Kinematic**:



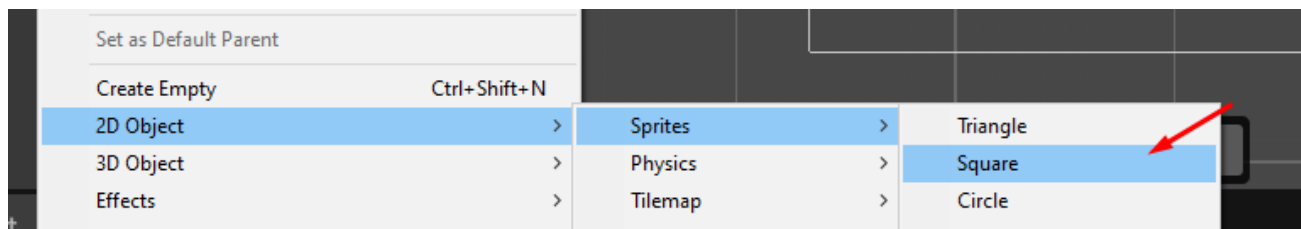
У **Box Collider 2D** изменим размер на **0.75** (чтобы при соприкосновении с собой игра не завершалась, или еда, которую мы позже добавим, не исчезала при касании боком змейки):



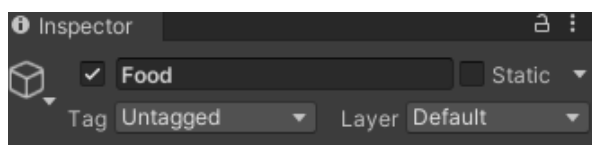
Ставим галочку **Is Trigger** (триггеры не вызывают физических столкновений, но используются для обнаружения определённых событий или реализации игровой механики):



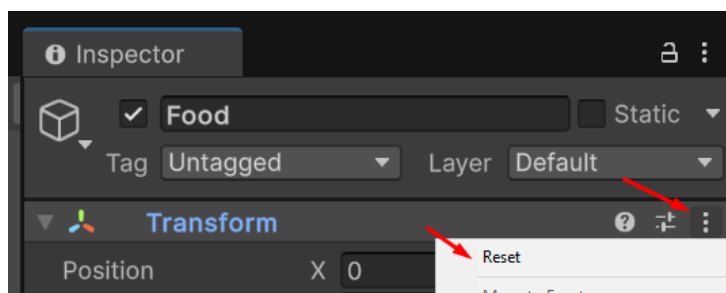
4. По аналогии создадим объект еду. Создаём новый 2D объект. Щёлкаем правой кнопкой мыши в **Hierarchy** и выбираем **2D Object** → **Sprites** → **Square**:



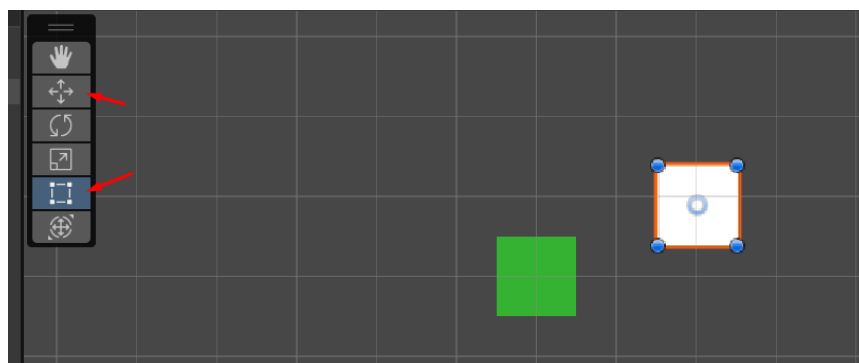
Называем его **Food**:



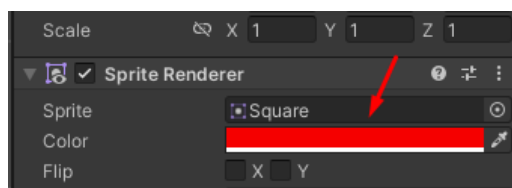
Сбрасываем трансформацию (нажав на три точки в углу):



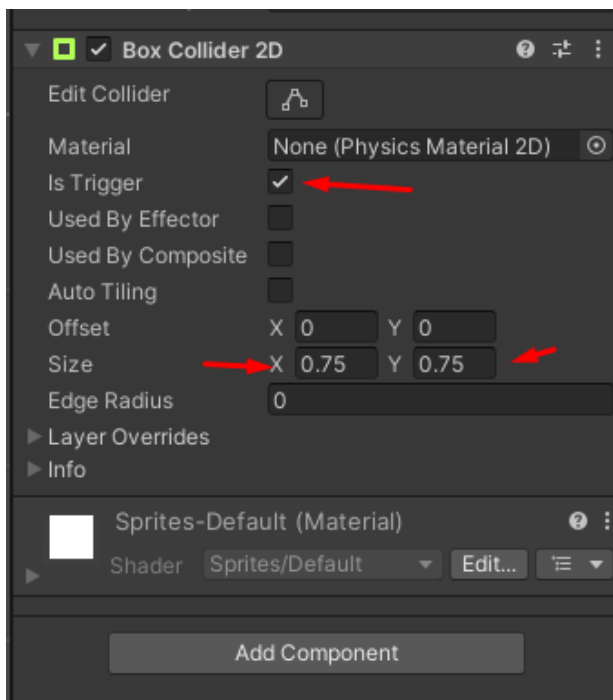
И перенесём немного в сторону от еды, воспользовавшись или инструментом **Move Tool** или инструментом **Rect Tool**:



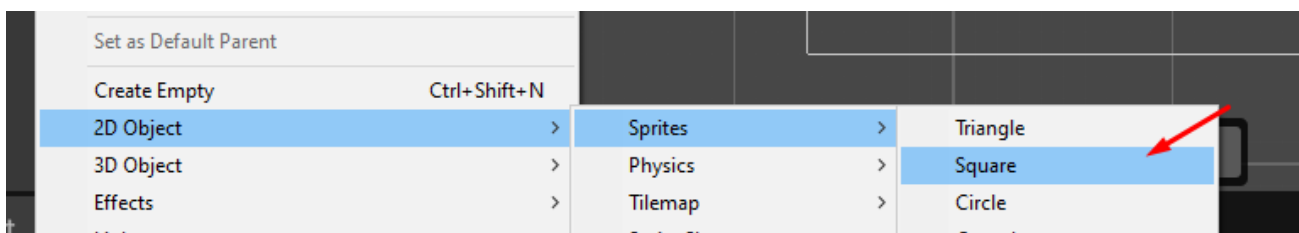
Меняем цвет (например, на красный):



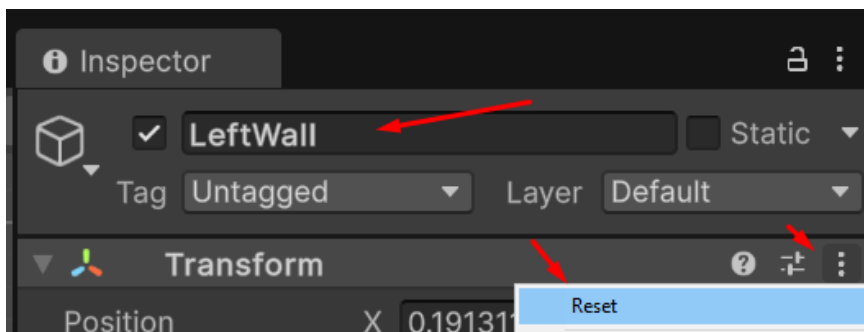
Добавляем **Box Collider 2D** (ставим галочку **Is Trigger** и меняем размер на **0.75**):



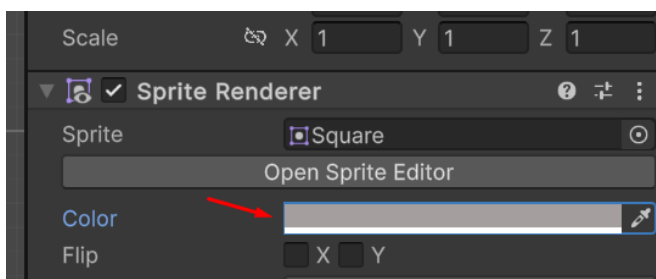
5. Создадим стены. Создаём новый 2D объект. Щёлкаем правой кнопкой мыши в **Hierarchy** и выбираем **2D Object** → **Sprites** → **Square**:



Меняем название на **LeftWall** и сбрасываем трансформацию:



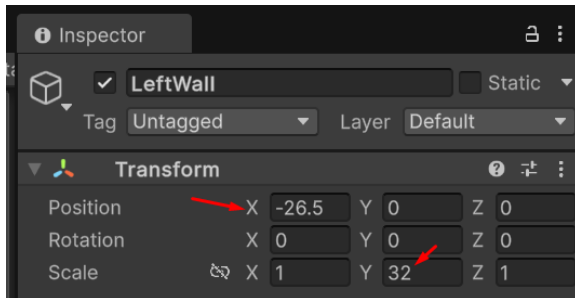
Меняем цвет:



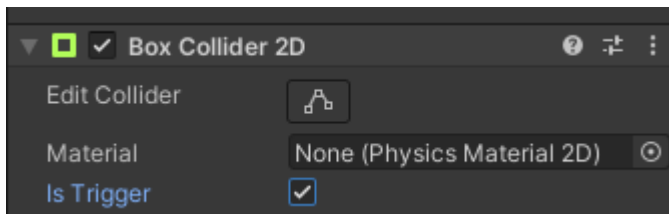
Далее нам нужно передвинуть её влево и изменить размер. Мы можем воспользоваться уже известными инструментами **Move Tool** или **Rect Tool**:



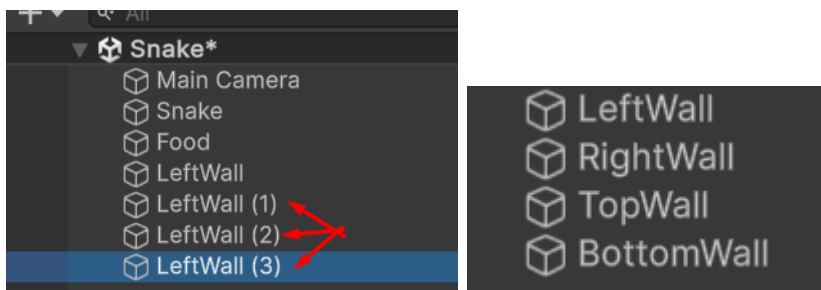
Или же ввести вручную (например **-26.5** позицию по **X** и **32** размер по **Y**):



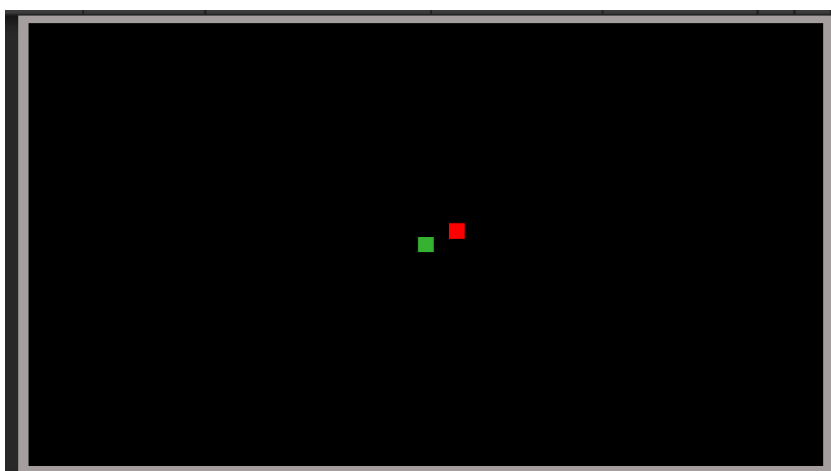
Добавляем **Box Collider 2D** (ставим галочку **Is Trigger**):



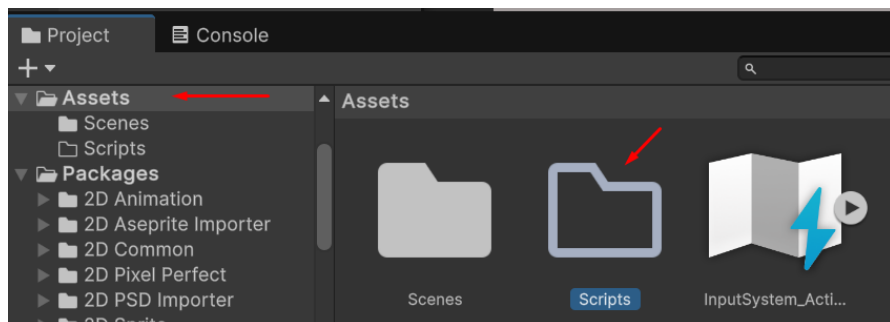
Затем создаём дубликаты наших стен (**Ctrl+D** выбрав объект в иерархии), с последующим изменением имени на **RightWall**, **TopWall**, **BottomWall**:



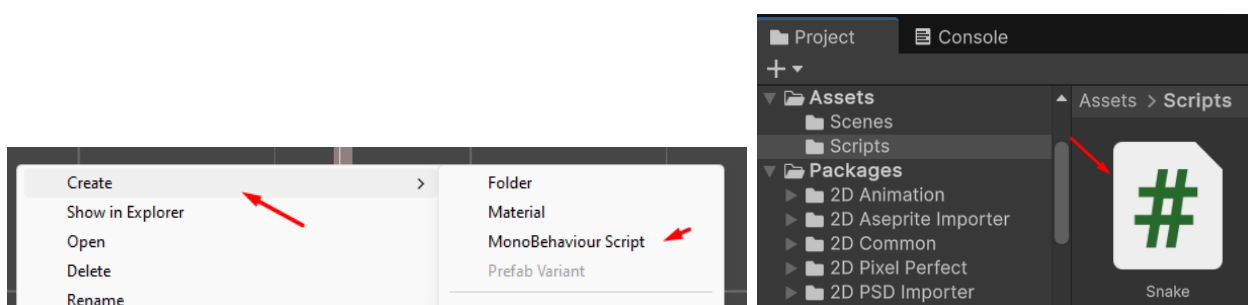
Далее меняем для них **позицию** и **размер**, чтобы получилось следующим образом:



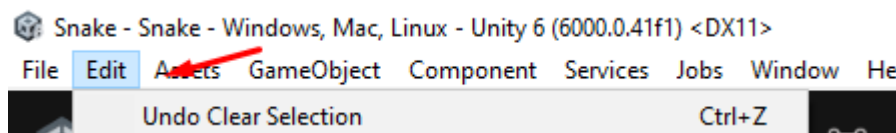
6. Перейдём к написанию логики движения нашей змейки. В папке **Assets** создаём папку **Scripts**:



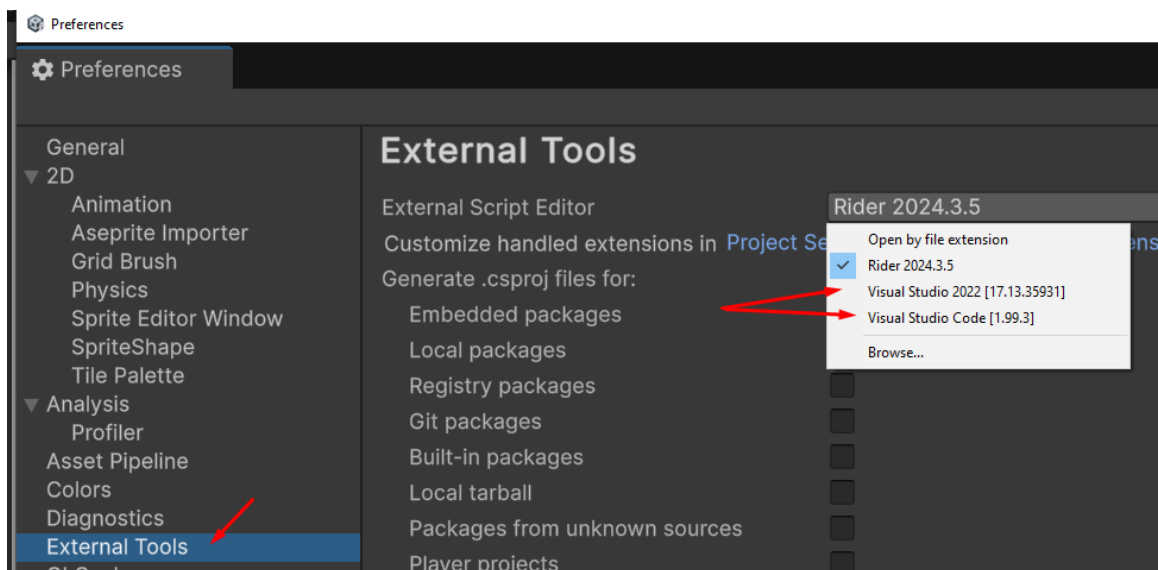
В ней создаём скрипт (щёлкаем правой кнопкой мыши), выбираем **Create – MonoBehaviour Script** и называем **Snake**:



Чтобы проверить какая программа открывает скрипты нужно перейти в **Edit → Preferences:**

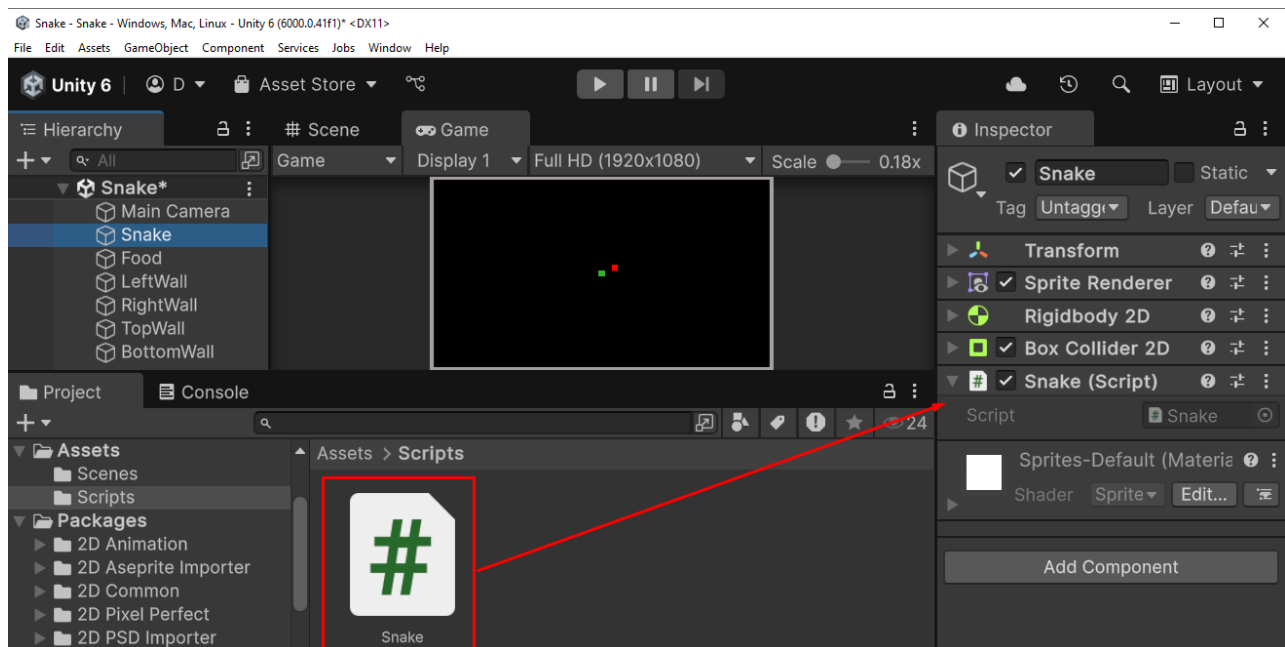


Далее **External Tools → External Script Editor:**

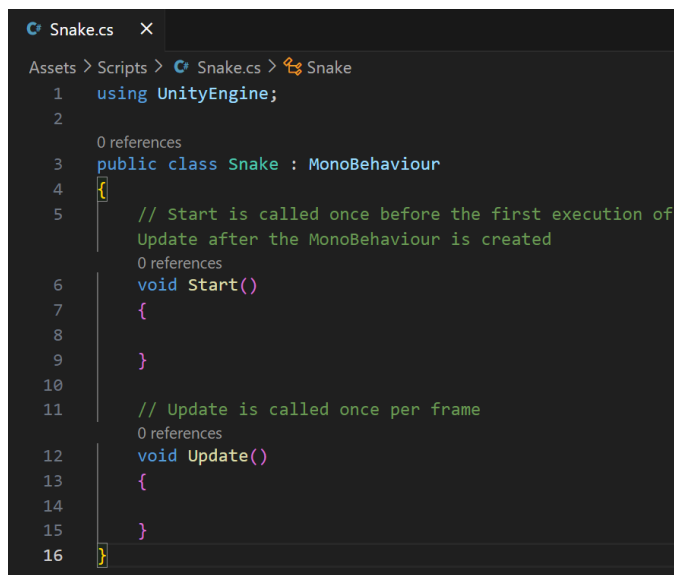


Выберете Visual Studio 2022 (или Visual Studio Code если вы любите его).

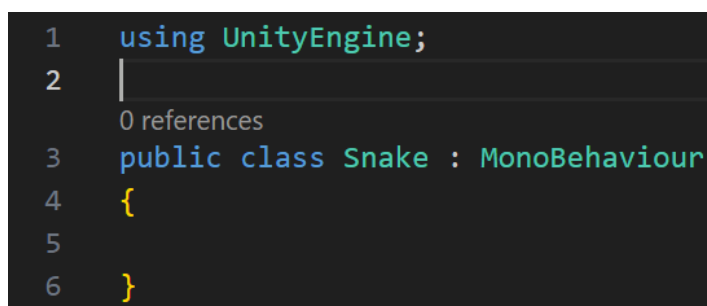
После нам надо перетащить **скрипт** на **объект Snake**:



Щёлкаем дважды на скрипт **Snake**, и он откроется в вашей IDE:



Можно удалить всё что находится внутри класса **Snake**, т.к. обычно код пишут с самостоятельно с нуля. В итоге скрипт будет выглядеть так:



using UnityEngine; - Подключает пространство имен **UnityEngine**, которое содержит основные классы и функции **Unity**.

public class Snake: MonoBehaviour - Объявляет публичный класс **Snake**, который наследуется от **MonoBehaviour**. Это позволяет использовать методы и свойства **Unity**, такие как **Start**, **Update**, и т.д.

➤ Теперь внутри класса объявляем **приватную поле направления**:

```
private Vector2 _direction;
```

Зачем это нужно: создаём приватное поле `_direction` типа `Vector2`, которая будет отвечать за направление движения змейки. Тип `Vector2` удобен, так как хранит координаты по X и Y.

➤ Пишем встроенный метод **Update()**, который вызывается в **Unity** каждый кадр:

```
private void Update()
{
    if (Input.GetKeyDown(KeyCode.W))
        _direction = Vector2.up;
    else if (Input.GetKeyDown(KeyCode.S))
        _direction = Vector2.down;
    else if (Input.GetKeyDown(KeyCode.A))
        _direction = Vector2.left;
    else if (Input.GetKeyDown(KeyCode.D))
        _direction = Vector2.right;
}
```

Зачем это нужно: Метод `Update()` вызывается каждый кадр. Здесь мы проверяем, нажал ли игрок одну из клавиш управления: W, A, S, D. Если нажата W — направление вверх (`Vector2.up`), S — вниз (`Vector2.down`), A — влево, D — вправо.

Таким образом, мы отслеживаем, куда игрок хочет повернуть змейку.

➤ Пишем встроенный метод **FixedUpdate()**, который вызывается в **Unity** с фиксированной частотой:

```
private void FixedUpdate()
{
    transform.position = new Vector2(
        Mathf.Round(transform.position.x) + _direction.x,
        Mathf.Round(transform.position.y) + _direction.y);
}
```

Зачем это нужно: Метод `FixedUpdate()` вызывается с фиксированным интервалом времени, и идеально подходит для работы с физикой и движением.

- `transform.position` — это текущая позиция объекта.
- `Mathf.Round()` округляет координаты до целых чисел, чтобы змейка двигалась по "сетке".
- К текущей позиции добавляется направление движения (`_direction`), которое мы задали ранее.

Таким образом, каждый фиксированный кадр змейка будет двигаться на один шаг в заданном направлении.

Итоговый код выглядит следующим образом:

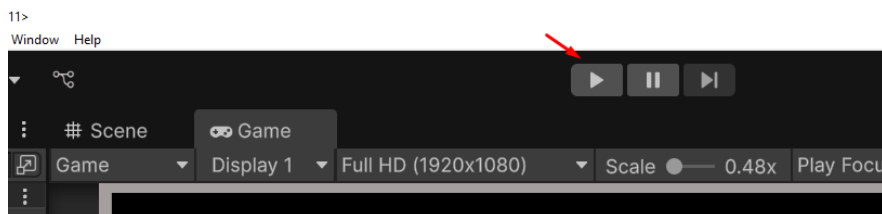
```
using UnityEngine;

0 references
public class Snake : MonoBehaviour
{
    6 references
    private Vector2 _direction;

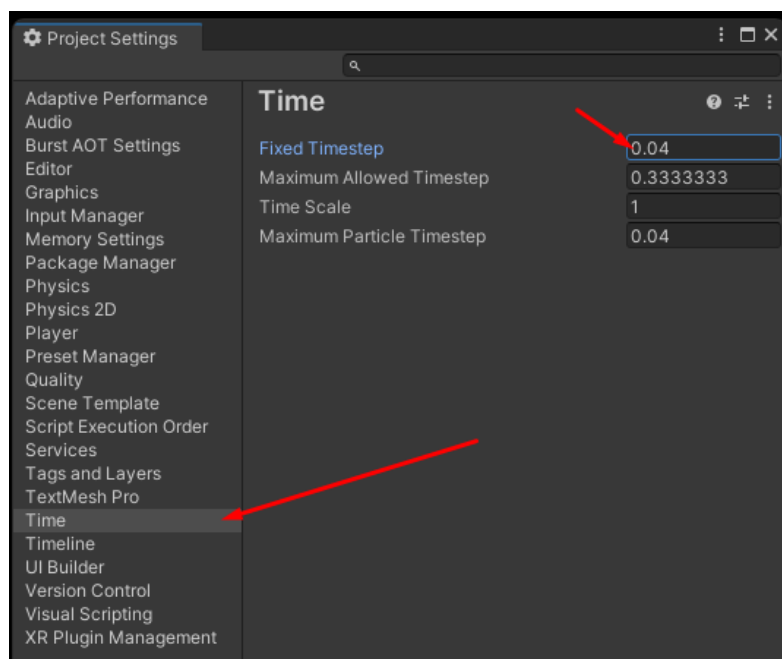
    0 references
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.W))
            _direction = Vector2.up;
        else if (Input.GetKeyDown(KeyCode.S))
            _direction = Vector2.down;
        else if (Input.GetKeyDown(KeyCode.A))
            _direction = Vector2.left;
        else if (Input.GetKeyDown(KeyCode.D))
            _direction = Vector2.right;
    }

    0 references
    private void FixedUpdate()
    {
        transform.position = new Vector2(
            Mathf.Round(transform.position.x) + _direction.x,
            Mathf.Round(transform.position.y) + _direction.y);
    }
}
```

Проверяем что всё работает в игре нажав на **Play** и двигаясь на **WASD**:

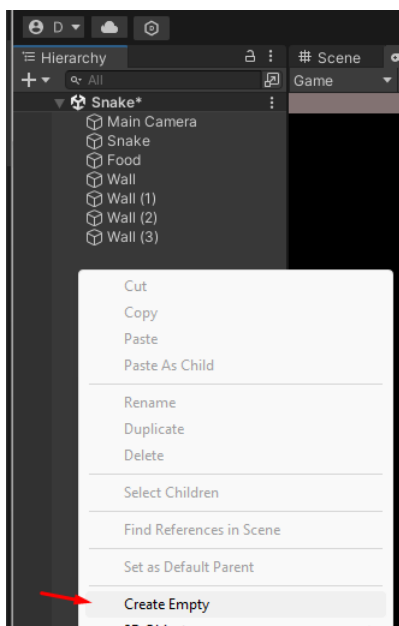


Давайте немного отрегулируем скорость движения змейки. Переходим в **Edit** → **Project Settings** → **Time** → **Fixed Timestep**. Поменяем на **0.04**:

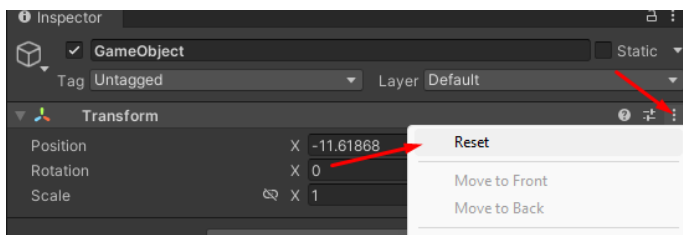


7. Теперь займёмся нашей едой. Мы хотим сделать так, чтобы еда появлялась в случайной позиции внутри определённой области. Нам нужно создать с вами новый объект, который не будет выходить за пределы стен.

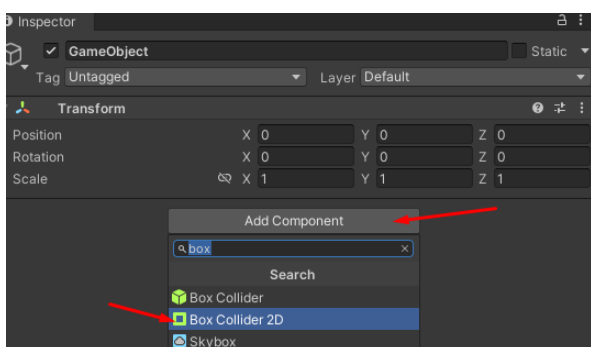
Нажимаем правой кнопкой мыши в нашей **Hierarchy** → **Create Empty**:



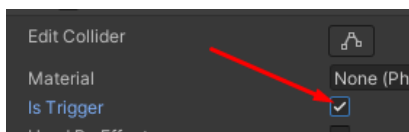
Reset:



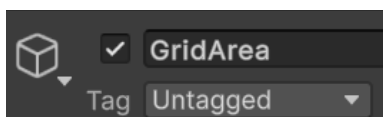
Добавляем **Box Collider 2D**:



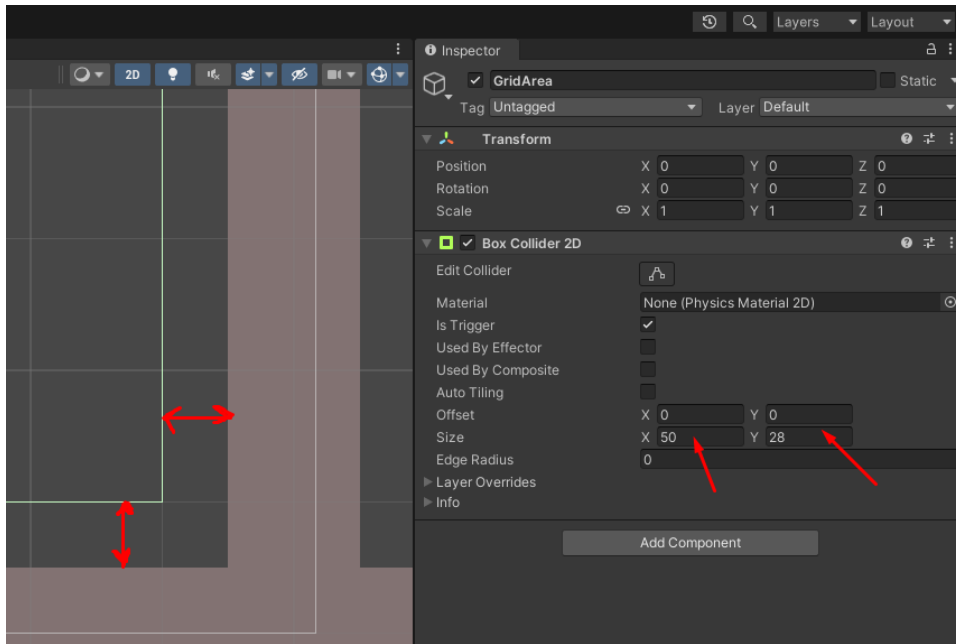
Поставим галочку **Is Trigger**:



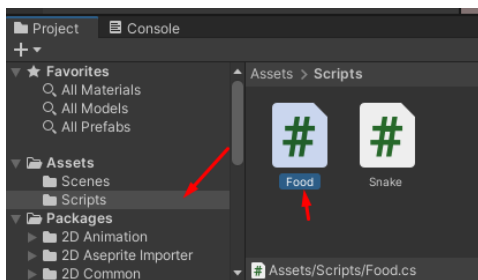
Называем объект - **GridArea**:



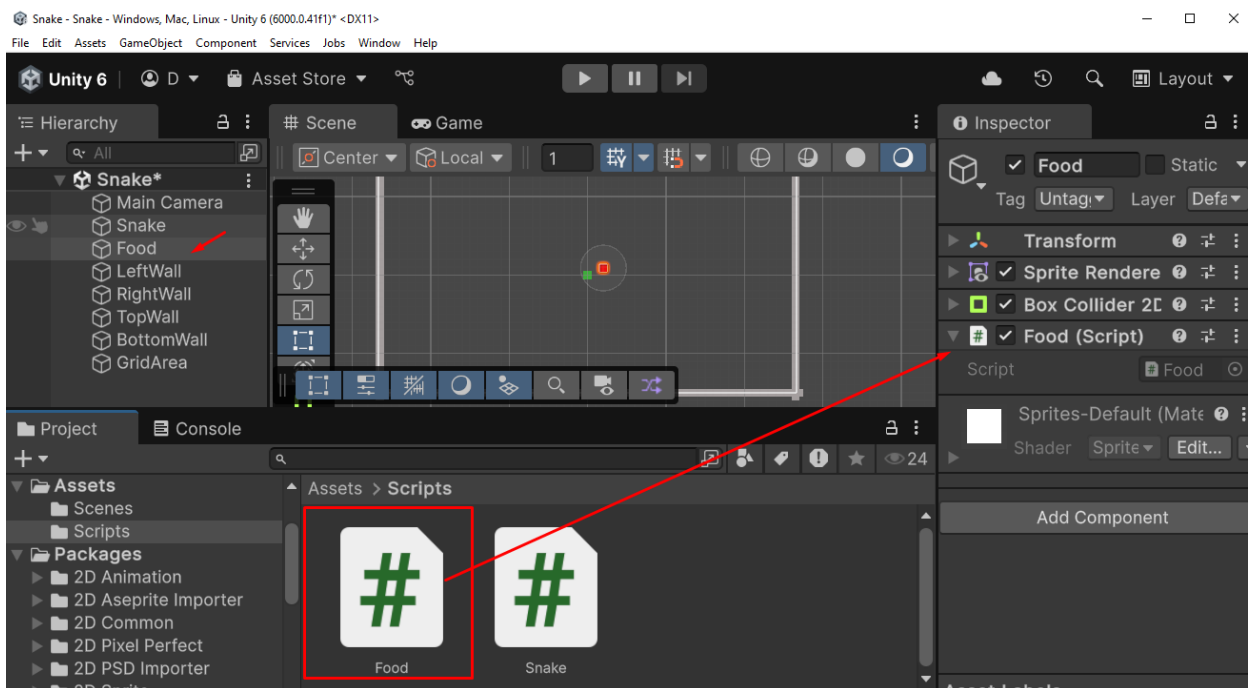
Настроим размер **Box Collider 2D** чтобы он не касался границ наших стен (в моём случае это $x = 50$ и $y = 28$):



В папке **Scripts** создаём скрипт с именем **Food**:



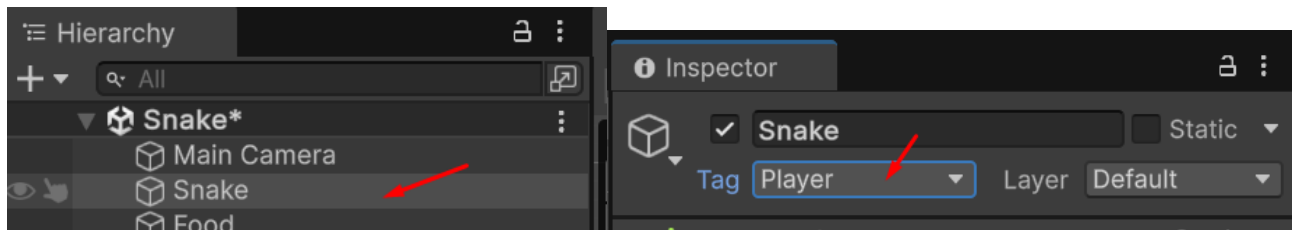
И добавляем его к еде (нажимаем на объект **Food** и переносим на него скрипт):



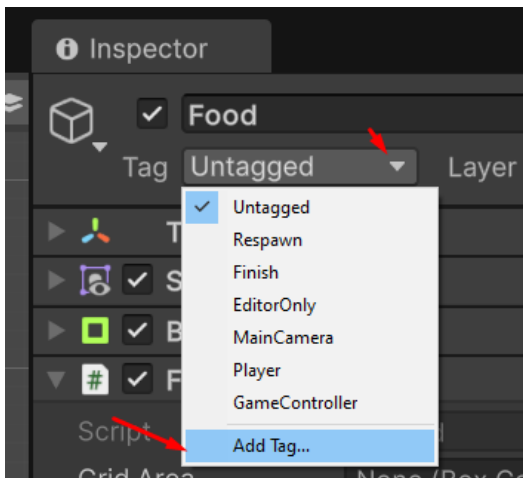
8. Перед тем как продолжить, нужно убедиться, что в проекте настроены теги для правильной работы скриптов.

Откройте каждый из соответствующих объектов в иерархии сцены и установите им следующие теги:

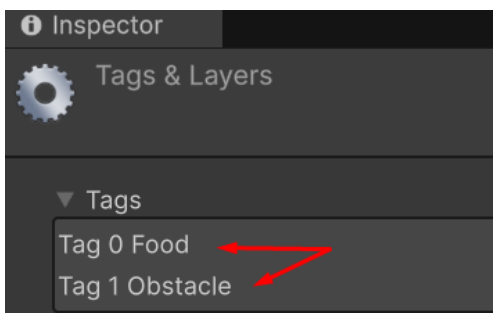
- Объект с игроком (змейкой) — **Player**:



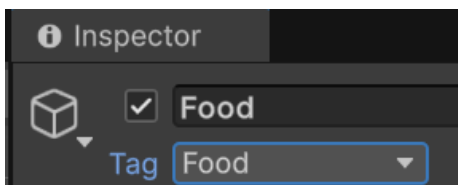
Новые теги добавляются через верхнюю часть инспектора объекта, в выпадающем меню "**Tag**". Нажмите "Add Tag...":



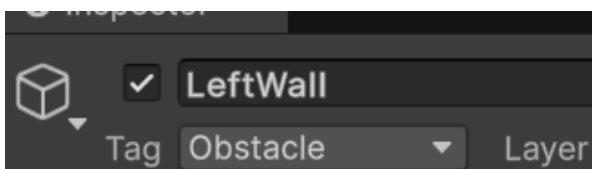
Добавьте два новых — для еды и препятствий - **Food** и **Obstacle**:



Выбираем тег **Food** для нашего объекта — еды:



И для всех **четырёх** стен — тег **Obstacle**:



9. Теперь открываем скрипт **Food**. Очищаем внутри класса:

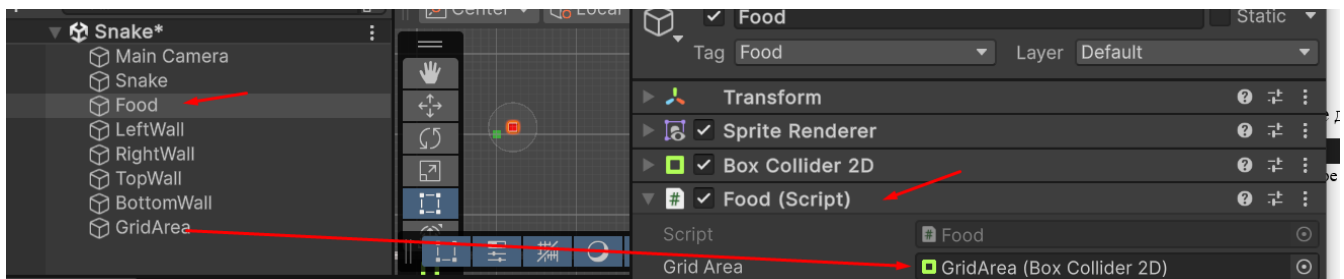
```
Snake.cs  Food.cs X
Assets > Scripts > Food.cs > Food
1  using UnityEngine;
2
   0 references
3  public class Food : MonoBehaviour
4  {
5
6  }
```

➤ Теперь внутри класса объявляем **публичное поле для области еды**:

```
public BoxCollider2D GridArea;
```

Зачем это нужно: создаём переменную, в которую в редакторе Unity можно будет перетащить объект, ограничивающий область появления еды.

Затем в самом **Unity** нужно перенести объект **GridArea** в поле скрипта **Food**:



➤ Создадим свой метод **RandomizePosition()**, который будет создавать объект в случайной позиции:

```
private void RandomizePosition()
{
    Bounds bounds = GridArea.bounds;
    float x = Random.Range(bounds.min.x, bounds.max.x);
    float y = Random.Range(bounds.min.y, bounds.max.y);
    transform.position = new Vector2(Mathf.Round(x),
    Mathf.Round(y));
}
```

Объяснение:

- bounds — границы объекта GridArea, внутри которых мы хотим разместить еду.
- Random.Range(...) — генерирует случайные координаты X и Y в этих границах.
- Mathf.Round(...) — округляем координаты, чтобы еда также попадала в "сетку", как и змейка.
- transform.position = ... — перемещаем еду в новую точку.

➤ При запуске сцены (метод **Start()**), добавим вызов нашего метода:

```
private void Start()
{
    RandomizePosition();
}
```


Start() вызывается при запуске сцены — мы сразу размещаем еду в случайной позиции с помощью метода **RandomizePosition()**.

➤ Затем вызовем встроенный метод **OnTriggerEnter2D()**:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player"))
        RandomizePosition();
}
```

Что делает этот метод:

- **OnTriggerEnter2D(...)** вызывается, когда другой объект входит в зону триггера еды.
- Если этот объект имеет тег **"Player"** (т.е. это змейка), то еда сразу меняет своё положение — имитируем, что её "съели", и она появилась в новом месте.

Итоговый код:

```
using UnityEngine;

0 references
public class Food : MonoBehaviour
{
    1 reference
    public BoxCollider2D GridArea;

    0 references
    private void Start()
    {
        RandomizePosition();
    }

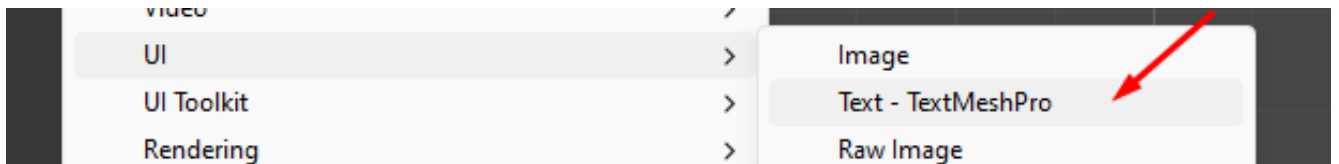
    2 references
    private void RandomizePosition()
    {
        Bounds bounds = GridArea.bounds;
        float x = Random.Range(bounds.min.x, bounds.max.x);
        float y = Random.Range(bounds.min.y, bounds.max.y);
        transform.position = new Vector2(Mathf.Round(x), Mathf.Round(y));
    }

    0 references
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
            RandomizePosition();
    }
}
```

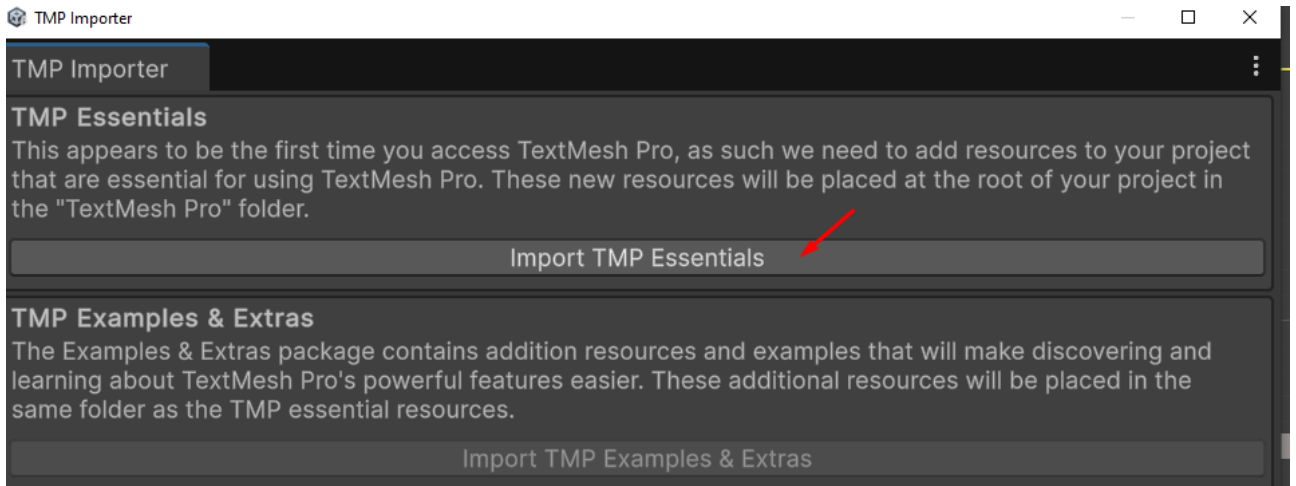
Запускаем и проверяем что при сборе еды она исчезает и появляется в новом случайном месте.

10. Теперь реализуем отображение счёта, который будет увеличиваться каждый раз, когда змейка съедает еду.

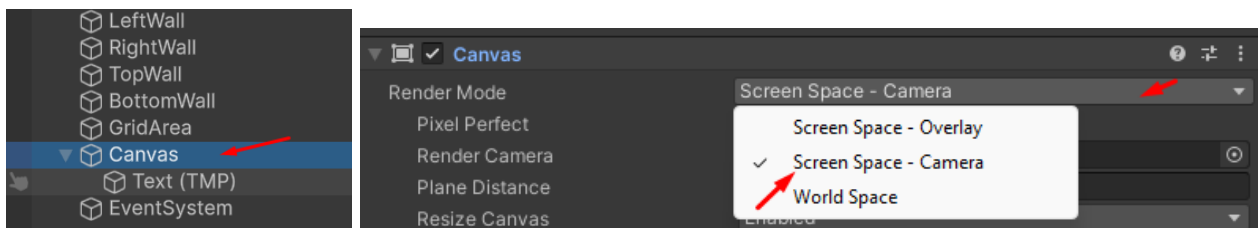
В **Hierarchy** нажимаем правой кнопкой мыши → **UI** → **Text - TextMeshPro**:



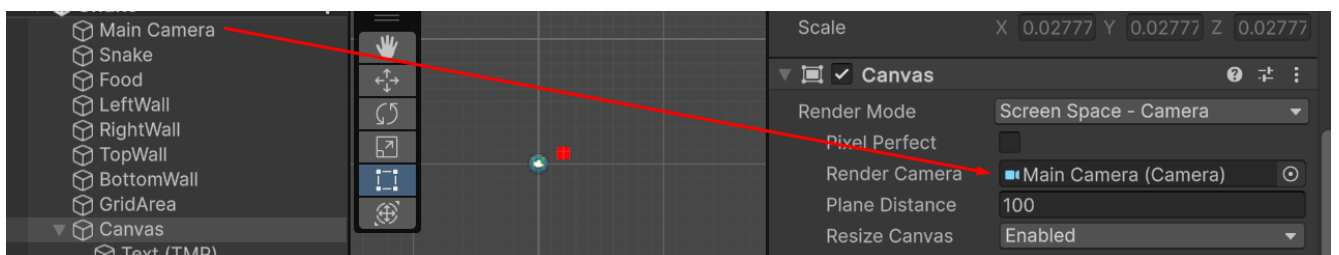
Появится окно с предложением импортировать ресурсы — нажимаем **Import**, а затем закрываем окно:



Выделите объект **Canvas** в иерархии. В **Inspector** у параметра **Render Mode** выбираем **Screen Space – Camera**:

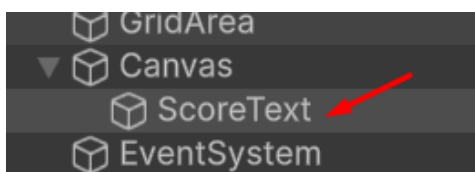


В появившееся поле **Render Camera** перетаскиваем туда объект **Main Camera**:



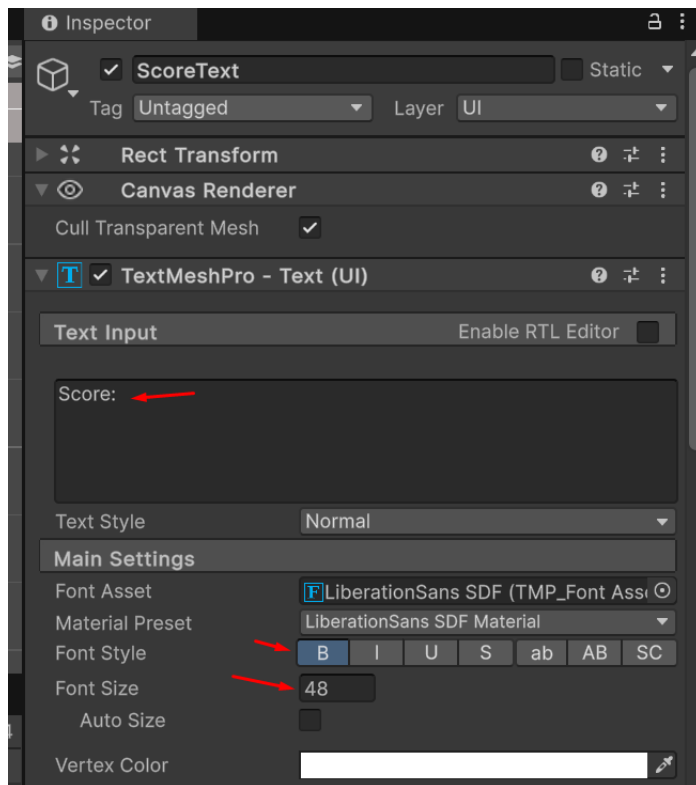
Это нужно, чтобы интерфейс корректно отображался в камере и всегда был на экране.

Переименуем объект с текстом на **ScoreText**:

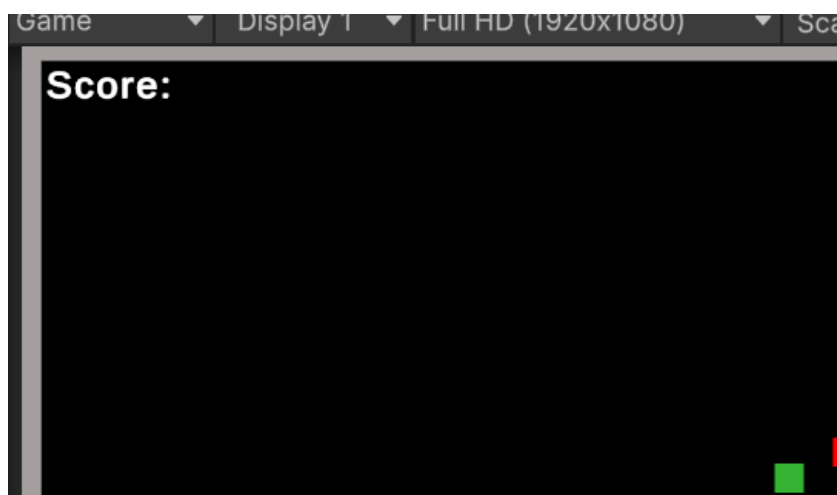


В **Inspector** измените:

- Текст на "Score:"
- **Размер шрифта, цвет и другие параметры по вкусу (или выберите следующие настройки):**



Разместите текст в левом верхнем углу экрана (можно использовать **RectTransform**):



Теперь добавим логику отображения очков в скрипт **Snake**.

- В начале скрипта подключаем **пространство имён TMPro для UI**:

```

1  using UnityEngine;
2  using TMPro;
3
4  0 references
5  public class Snake : MonoBehaviour
  
```

 A screenshot of a code editor showing the beginning of a C# script. Line 2, 'using TMPro;', is highlighted with a red arrow.

Это подключение позволяет использовать компонент **TextMeshProUGUI**, который мы применяем для отображения текста в интерфейсе.

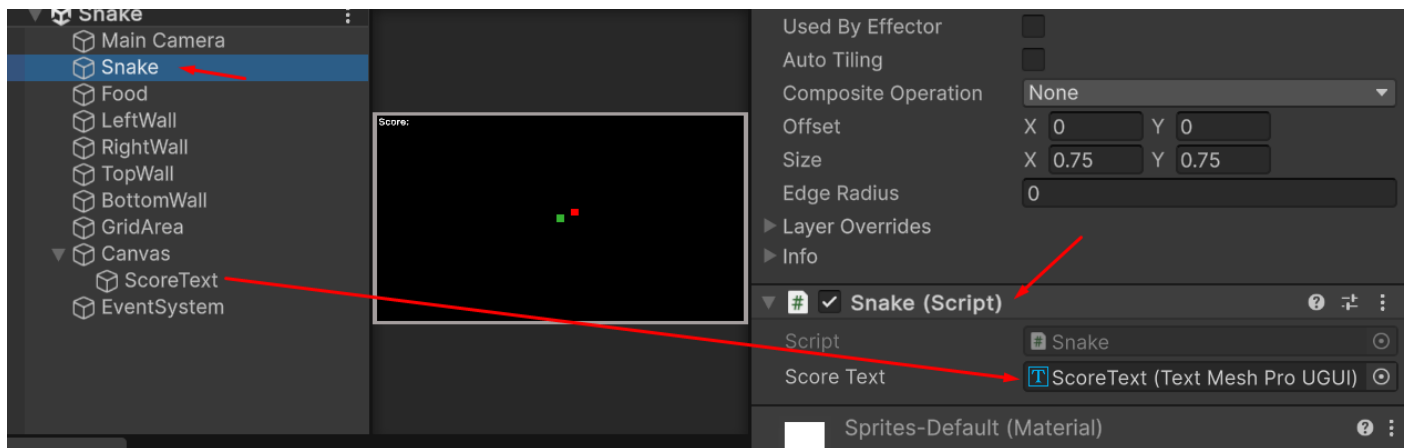
➤ Затем объявим **переменные**:

```
public TextMeshProUGUI scoreText;  
private int score = 0;
```

Зачем это нужно:

- **scoreText** — переменная-ссылка на UI-элемент, который мы будем менять.
- **score** — текущий счёт, который будем увеличивать при поедании еды.

После этого в **Unity** перетащите объект **ScoreText** из **Hierarchy** в соответствующее поле скрипта **Snake** в **Inspector**:



➤ Добавляем метод обновления текста **UpdateTextScore()**

```
private void UpdateTextScore()  
{  
    scoreText.text = $"Score: {score}";  
}
```

Этот метод обновляет текст на экране, отображая актуальное значение счёта.

➤ Вызываем обновление счёта при поедании еды. Создаём метод **OnTriggerEnter2D()**:

```
private void OnTriggerEnter2D(Collider2D other)  
{  
    if (other.CompareTag("Food"))  
    {  
        score++;  
        UpdateTextScore();  
    }  
}
```

Когда змейка касается еды (объект с тегом **Food**), мы:

- Увеличиваем значение переменной **score**.
- Обновляем текст на экране, вызвав **UpdateTextScore()**.

Запускаем и проверяем в Unity, что теперь при поедании еды счёт будет увеличиваться, и это сразу отобразится на экране.

11. Теперь напишем метод для сброса игры **ResetGame()**:

```
private void ResetGame()
{
    transform.position = Vector2.zero;
    score = 0;
    UpdateTextScore();
}
```

Мы сбрасываем координаты нашей змейки в центр экрана, обнуляем счёт, и запускаем метод подсчёта и вывода очков.

В методе **OnTriggerEnter2D()** добавим дополнительную проверку и запуск метода сброса игры, когда мы сталкиваемся со объектами где у нас есть тег **Obstacle** (наши стены):

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Food"))
    {
        score++;
        UpdateTextScore();
    }
    else if (other.CompareTag("Obstacle"))
    {
        ResetGame();
    }
}
```

12. Теперь мы добавим механику, при которой змейка будет удлиняться — после каждого съеденного кусочка еды будет появляться новый сегмент.

➤ Подключаем в начало скрипта **коллекции**:

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

Это пространство имён содержит коллекции, такие как **List**, которые мы будем использовать для хранения сегментов змейки.

➤ Объявляем список сегментов:

```
private List<Transform> _segments;
```

```
public class Snake : MonoBehaviour
{
    6 references
    private Vector2 _direction;
    1 reference
    public TextMeshProUGUI scoreText;
    3 references
    private int score = 0;
    0 references
    private List<Transform> _segments;
```

Создаём приватное поле **_segments**, которое будет содержать все части змейки — голову и хвостовые сегменты. Мы будем по этому списку управлять движением всех частей.

➤ Инициализируем список в методе **Start()**:

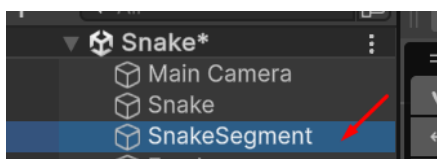
```
private void Start() // Метод Start вызывается один раз при
инициализации объекта.
{
    _segments = new List<Transform>(); // создаём пустой список
    _segments.Add(transform); // добавляем в список голову
змейки (текущий объект).
}
```

При запуске игры список должен быть пустым, кроме одного элемента — головы змейки. Мы добавляем **transform**, т.е. сам объект змейки, как первый элемент списка.

➤ Создаём **префаб** сегмента змейки

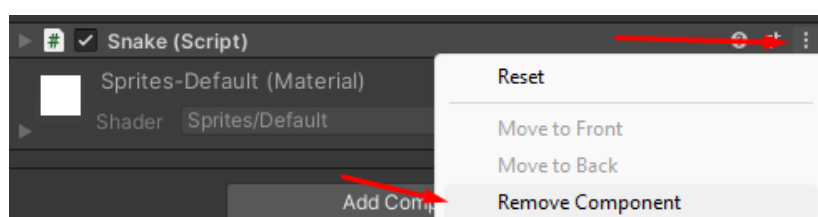
Prefab — это шаблон объекта, который можно копировать (создавать экземпляры) в любой момент игры.

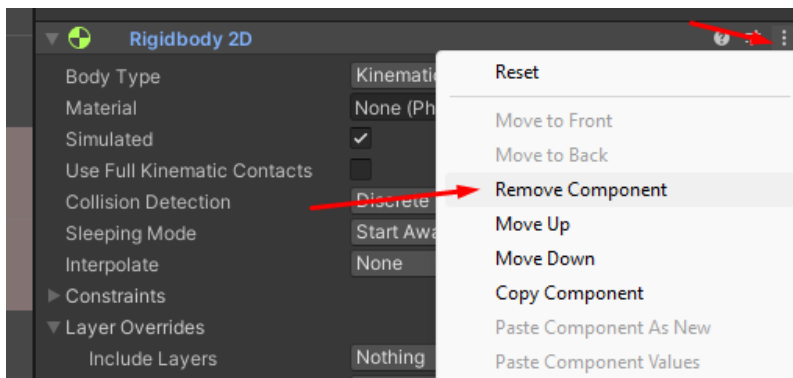
В **Hierarchy** выделите объект **Snake**, нажмите **Ctrl+D** для создания копии и переименуйте копию в **SnakeSegment**:



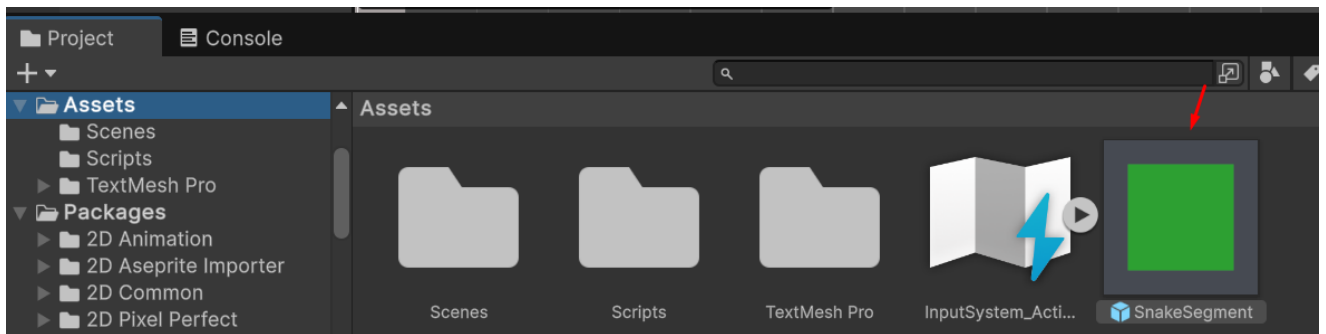
Удалите с **SnakeSegment** компоненты, которые не будут использоваться:

- **Rigidbody2D**
- Скрипт **Snake**

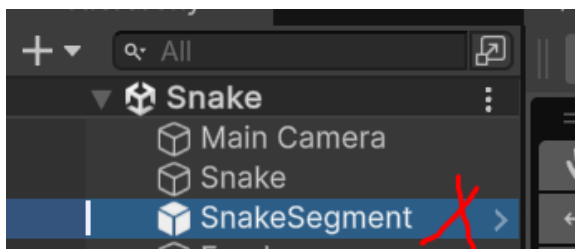




Перетащите **SnakeSegment** в папку **Assets** — он превратится в **префаб (.prefab)**:



После этого можно удалить **SnakeSegment** из сцены (**Hierarchy**):



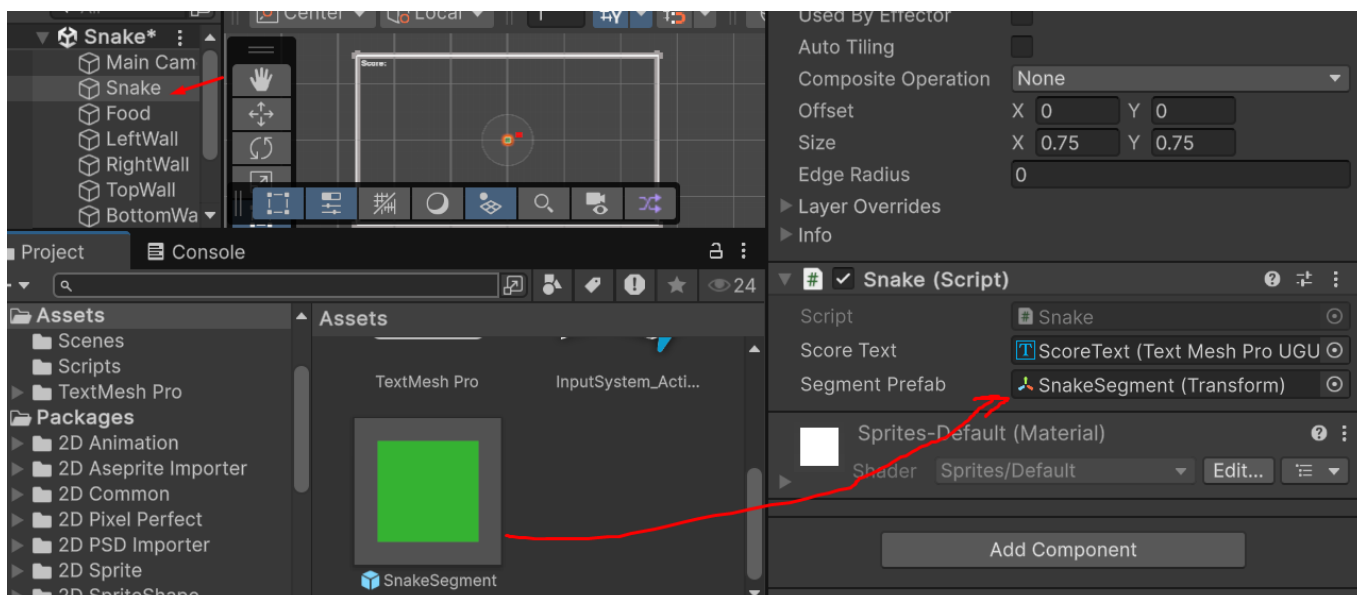
➤ Добавляем ссылку на **префаб** в скрипт

Возвращаемся в наш скрипт **Snake**. Создадим публичное поле для нашего префаба:

```
public class Snake : MonoBehaviour
{
    6 references
    private Vector2 _direction;
    1 reference
    public TextMeshProUGUI scoreText;
    3 references
    private int score = 0;
    2 references
    private List<Transform> _segments;
    0 references
    public Transform segmentPrefab;
}
```

Это поле позволит задать в редакторе, какой префаб использовать при создании новых сегментов змейки.

Затем перетащите **префаб SnakeSegment** в это поле в инспекторе **объекта Snake**:



Далее напишем метод для увеличения змейки **Grow()**:

```
private void Grow()
{
    Transform segmentNew = Instantiate(segmentPrefab); // Создаём
    // новый экземпляр префаба
    segmentNew.position = _segments[_segments.Count - 1].position;
    // Помещаем его туда, где находится последний сегмент
    _segments.Add(segmentNew); // Добавляем новый сегмент в список
}
```

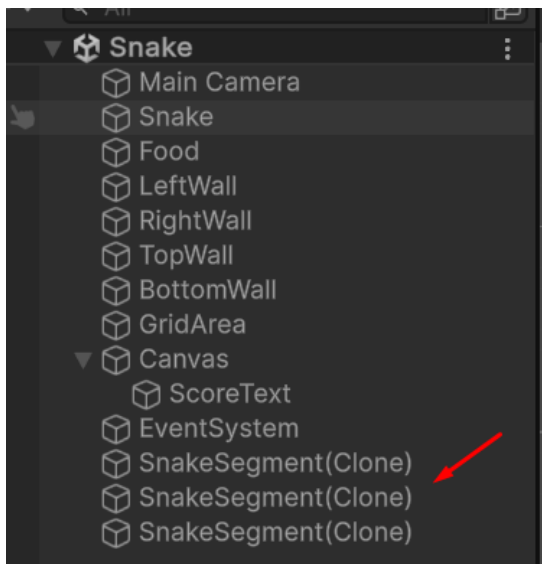
Когда змейка ест еду, создаём новый сегмент и добавляем его в конец змейки. Изначально он будет стоять на месте последнего сегмента — позже мы "сдвинем" его с помощью кода движения.

- Добавляем вызов метода **Grow()** при поедании еды. Изменим метод **OnTriggerEnter2D** так:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Food"))
    {
        score++;
        UpdateTextScore();
        Grow(); // Увеличиваем змейку
    }
    else if (other.CompareTag("Obstacle"))
    {
        ResetGame();
    }
}
```

Каждый раз, когда змейка съедает еду, вызываем **Grow()** и увеличиваем её длину.

Если запустить игру, то может показаться странным, что сегмент добавился один и находится в одном месте. Также в иерархии мы видим, что **новые префабы** появляются:



Решим эту проблему, для этого добавим в **FixedUpdate** код, который будет "проталкивать" каждый сегмент на позицию предыдущего:

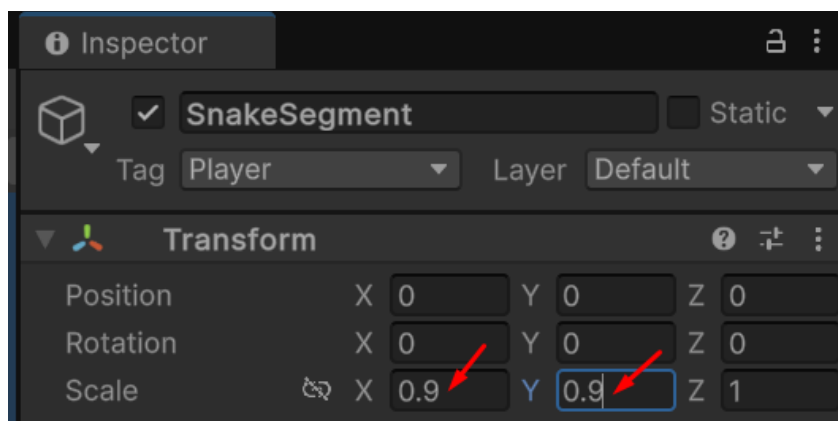
```
private void FixedUpdate()
{
    for (int i = _segments.Count - 1; i > 0; i--)
        _segments[i].position = _segments[i - 1].position;

    transform.position = new Vector2(
        Mathf.Round(transform.position.x) + _direction.x,
        Mathf.Round(transform.position.y) + _direction.y);
}
```

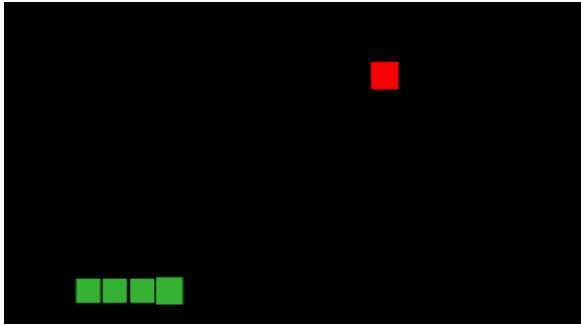
Это простой способ заставить сегменты "следовать" друг за другом: каждый следующий становится на место предыдущего, начиная с конца. После этого цикл, как обычно, продолжается с движением головы.

Чтобы хвост выглядел чуть меньше, можно выделить префаб SnakeSegment и в **Inspector** уменьшить масштаб:

- Установите **Scale** на 0.9 по всем осям (X, Y):



Запускайте, и проверьте что теперь корректно увеличиваются сегменты и змейки:



Исправление багов

При тестировании игры можно столкнуться с несколькими багами. Давайте поочерёдно разберём их и решим.

Баг 1. После смерти змейки остаются её сегменты

Когда змейка сталкивается с собой или стеной, игра перезапускается, но все хвостовые сегменты продолжают оставаться на сцене.

Решение:

Добавим очистку сегментов в метод **ResetGame()**:

```
private void ResetGame()
{
    // Цикл, который уничтожает все сегменты змейки, кроме головы.
    for (int i = 1; i < _segments.Count; i++)
        Destroy(_segments[i].gameObject);

    _segments.Clear(); // Очищает список _segments.
    _segments.Add(transform); // Добавляет голову змейки в список _segments.

    transform.position = Vector2.zero;
    score = 0;
    UpdateTextScore();
}
```

Мы вручную удаляем каждый сегмент (начиная с индекса 1, т.к. 0 — это голова), чтобы они не оставались на сцене. Затем обнуляем список и добавляем только голову, чтобы начать игру заново.

Баг 2. Змейка может двигаться в противоположном направлении (сама в себя)

Это может привести к мгновенному "самоубийству" змейки, например, если быстро нажать D, а затем A.

Решение:

Обновим метод **Update()** следующим образом:

```
private void Update()
{
    if (Input.GetKeyDown(KeyCode.W) && _direction != Vector2.down)
        _direction = Vector2.up;
    else if (Input.GetKeyDown(KeyCode.S) && _direction != Vector2.up)
        _direction = Vector2.down;
    else if (Input.GetKeyDown(KeyCode.A) && _direction != Vector2.right)
        _direction = Vector2.left;
    else if (Input.GetKeyDown(KeyCode.D) && _direction != Vector2.left)
        _direction = Vector2.right;
}
```

Мы добавили проверки, чтобы змейка не могла резко развернуться на 180°.

Баг 3. Змейка не умирает при столкновении с самой собой

На данный момент при столкновении с сегментами змейки ничего не происходит.

Решение:

Добавим в **FixedUpdate()** проверку на самопересечение:

```
private void FixedUpdate()
{
    for (int i = _segments.Count - 1; i > 0; i--)
        _segments[i].position = _segments[i - 1].position;

    transform.position = new Vector2(
        Mathf.Round(transform.position.x) + _direction.x,
        Mathf.Round(transform.position.y) + _direction.y);

    // Проверка на столкновение с сегментами змейки
    for (int i = 1; i < _segments.Count; i++)
    {
        if (transform.position == _segments[i].position)
        {
            ResetGame();
            break;
        }
    }
}
```

Мы проходимся по всем сегментам змейки, кроме головы (начиная с $i = 1$), и сравниваем их позицию с позицией головы. Если совпадают — вызываем `ResetGame()`.

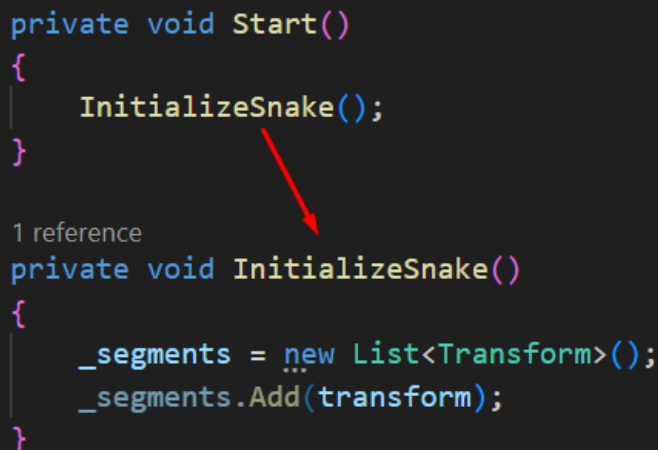
Рефакторинг

На текущий момент у нас получился **работающий, но громоздкий и неструктурированный код**. Весь функционал — управление, счёт, генерация

сегментов, логика столкновений и перезапуска — находится в одном скрипте **Snake.cs**. Это затрудняет чтение и поддержку кода, особенно если проект будет расти.

В будущих лабораторных мы будем применять более правильное написание скриптов. А сейчас произведём **рефакторинг** — это процесс улучшения внутренней структуры кода без изменения его поведения.

➤ Вынесем код из метода **Start()** в новый **InitializeSnake()**:



```
private void Start()
{
    InitializeSnake();
}

1 reference
private void InitializeSnake()
{
    _segments = new List<Transform>();
    _segments.Add(transform);
}
```

Разделим в методе **FixedUpdate** код на 3 метода:

```
private void FixedUpdate()
{
    MoveSegments();
    MovePlayer();
    CheckCollision();
}
```

Сами методы.

Перемещение сегментов:

```
private void MoveSegments()
{
    for (int i = _segments.Count - 1; i > 0; i--)
        _segments[i].position = _segments[i - 1].position;
}
```

Движение головы:

```
private void MovePlayer()
{
    transform.position = new Vector2(
        Mathf.Round(transform.position.x) + _direction.x,
        Mathf.Round(transform.position.y) + _direction.y);
}
```

Проверка на столкновение с телом:

```
private void CheckCollision()
{
    for (int i = 1; i < _segments.Count; i++)
    {
        if (transform.position == _segments[i].position)
        {
            ResetGame();
            break;
        }
    }
}
```

Считывание клавиш в **Update()** выведем в метод **HandleInput()**:

```
private void Update()
{
    HandleInput();
}
```

```
private void HandleInput()
{
    if (Input.GetKeyDown(KeyCode.W) && _direction != Vector2.down)
        _direction = Vector2.up;
    else if (Input.GetKeyDown(KeyCode.S) && _direction != Vector2.up)
        _direction = Vector2.down;
    else if (Input.GetKeyDown(KeyCode.A) && _direction != Vector2.right)
        _direction = Vector2.left;
    else if (Input.GetKeyDown(KeyCode.D) && _direction != Vector2.left)
        _direction = Vector2.right;
}
```

Итоговый код **Snake** со всеми правками:

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

public class Snake : MonoBehaviour
{
    private Vector2 _direction;
    public TextMeshProUGUI scoreText;
    private int score = 0;
    private List<Transform> _segments;
    public Transform segmentPrefab;
    private void Start()
    {
        InitializeSnake();
    }
    private void Update()
    {
        HandleInput();
    }
}
```

```

private void FixedUpdate()
{
    MoveSegments();
    MovePlayer();
    CheckCollision();
}
private void InitializeSnake()
{
    _segments = new List<Transform>();
    _segments.Add(transform);
}
private void HandleInput()
{
    if (Input.GetKeyDown(KeyCode.W) && _direction != Vector2.down)
        _direction = Vector2.up;
    else if (Input.GetKeyDown(KeyCode.S) && _direction != Vector2.up)
        _direction = Vector2.down;
    else if (Input.GetKeyDown(KeyCode.A) && _direction != Vector2.right)
        _direction = Vector2.left;
    else if (Input.GetKeyDown(KeyCode.D) && _direction != Vector2.left)
        _direction = Vector2.right;
}
private void MoveSegments()
{
    for (int i = _segments.Count - 1; i > 0; i--)
        _segments[i].position = _segments[i - 1].position;
}
private void MovePlayer()
{
    transform.position = new Vector2(
        Mathf.Round(transform.position.x) + _direction.x,
        Mathf.Round(transform.position.y) + _direction.y);
}
private void CheckCollision()
{
    for (int i = 1; i < _segments.Count; i++)
    {
        if (transform.position == _segments[i].position)
        {
            ResetGame();
            break;
        }
    }
}
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Food"))
    {
        score++;
        UpdateTextScore();
        Grow();
    }
    else if (other.CompareTag("Obstacle"))
        ResetGame();
}

```

```

}
private void UpdateTextScore()
{
    scoreText.text = $"Score: {score}";
}
private void Grow()
{
    Transform segmentNew = Instantiate(segmentPrefab);
    segmentNew.position = _segments[_segments.Count - 1].position;
    _segments.Add(segmentNew);
}
private void ResetGame()
{
    for (int i = 1; i < _segments.Count; i++)
        Destroy(_segments[i].gameObject);
    _segments.Clear();
    _segments.Add(transform);
    transform.position = Vector2.zero;
    score = 0;
    UpdateTextScore();
}
}

```

Добавляем фичи:

13. Выход из игры по нажатию клавиши ESC

Добавим возможность выхода из игры по нажатию клавиши **Escape**. Это будет работать только в собранной (сбилденной) версии игры, а не в редакторе:

```

private void HandleInput()
{
    if (Input.GetKeyDown(KeyCode.W) && _direction != Vector2.down)
        _direction = Vector2.up;
    else if (Input.GetKeyDown(KeyCode.S) && _direction != Vector2.up)
        _direction = Vector2.down;
    else if (Input.GetKeyDown(KeyCode.A) && _direction != Vector2.right)
        _direction = Vector2.left;
    else if (Input.GetKeyDown(KeyCode.D) && _direction != Vector2.left)
        _direction = Vector2.right;

    else if (Input.GetKeyDown(KeyCode.Escape))
        Application.Quit();
}

```

14. Увеличение скорости змейки

Сделаем интересную механику — повышение скорости игры по мере набора очков. Для этого будем управлять параметром **Time.fixedDeltaTime**. Он отвечает за частоту вызова метода **FixedUpdate**.

❖ Шаг 1: Объявим переменные в начале скрипта:

```
public class Snake : MonoBehaviour
{
    0 references
    private float initialFixedDeltaTime; // Сохраняем начальную скорость
    0 references
    public float speedIncrease = 0.005f; // Насколько будем ускоряться
    10 references
    private Vector2 _direction;
```

❖ Шаг 2: Сохраняем начальное значение **Time.fixedDeltaTime** в методе **Start()** чтобы можно было сбросить его при перезапуске игры:

```
private void Start()
{
    InitializeSnake();
    initialFixedDeltaTime = Time.fixedDeltaTime; // Сохраняем начальное значение
}
```

❖ Шаг 3: Увеличиваем скорость в **OnTriggerEnter2D** при сборе еды:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Food"))
    {
        score++;
        UpdateTextScore();
        Grow();
        Time.fixedDeltaTime -= speedIncrease; //
        Увеличиваем скорость при сборе еды
    }
    else if (other.CompareTag("Obstacle"))
        ResetGame();
}
```

❖ Шаг 4: Сбрасываем скорость при смерти

Сбрасываем скорость до начального значения при перезапуске игры в **ResetGame**:


```
private void ResetGame()
{
    for (int i = 1; i < _segments.Count; i++)
        Destroy(_segments[i].gameObject);

    _segments.Clear();
    _segments.Add(transform);

    transform.position = Vector2.zero;
    score = 0;
    UpdateTextScore();
    //Сбрасываем скорость при смерти
    Time.fixedDeltaTime = initialFixedDeltaTime;
}
```

15. Переход змейки на другую сторону экрана

Реализуем эффект телепортации, когда змейка выходит за границы — она появляется с другой стороны экрана.

Напишем метод **WrapAroundScreen**, он будет проверять, выходит ли змейка за границы экрана, и перемещает её на противоположную сторону, если это происходит:

```
private void WrapAroundScreen()
{
    Vector3 position = transform.position;
    float screenWidth = Camera.main.orthographicSize *
Camera.main.aspect;
    float screenHeight = Camera.main.orthographicSize;

    if (position.x > screenWidth)
        position.x = -screenWidth;
    else if (position.x < -screenWidth)
        position.x = screenWidth;

    if (position.y > screenHeight)
        position.y = -screenHeight;
    else if (position.y < -screenHeight)
        position.y = screenHeight;

    transform.position = position;
}
```

Vector3 position = transform.position;

✦ Сохраняем текущую позицию змейки (её головы).

```
float screenWidth = Camera.main.orthographicSize * Camera.main.aspect;
```

```
float screenHeight = Camera.main.orthographicSize;
```

✎ Определяем видимые границы игрового экрана по ширине и высоте:

Camera.main.orthographicSize — это половина высоты камеры.

aspect — это соотношение сторон (ширина / высота).

Умножая их, получаем половину ширины экрана.

🔗 Проверка выхода за границы

```
if (position.x > screenWidth)
```

```
    position.x = -screenWidth;
```

```
else if (position.x < -screenWidth)
```

```
    position.x = screenWidth;
```

✦ Если позиция по X выходит за правую границу — переносим влево, и наоборот.

✦ То же самое делаем с координатой Y — проверяем выход за верхнюю/нижнюю границу и

«перекидываем» позицию.

🔗 Итог

```
transform.position = position;
```

✦ Обновляем позицию объекта змейки на рассчитанную — теперь она будет появляться с противоположной стороны при выходе за экран.

Добавим его вызов в **FixedUpdate**:

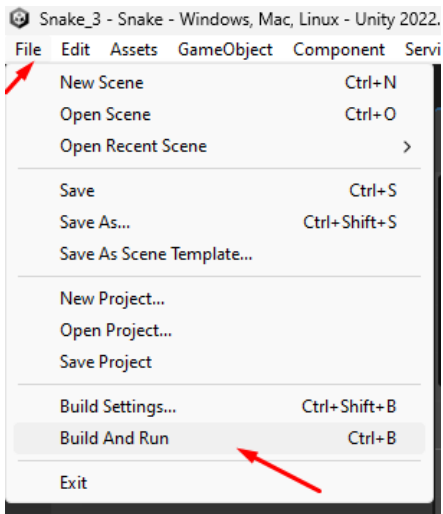
```
private void FixedUpdate()
{
    MoveSegments();
    MovePlayer();
    CheckSelfCollision();
    WrapAroundScreen();
}
```

Удалим проверку на столкновение со стенами в методе **OnTriggerEnter2D()**:

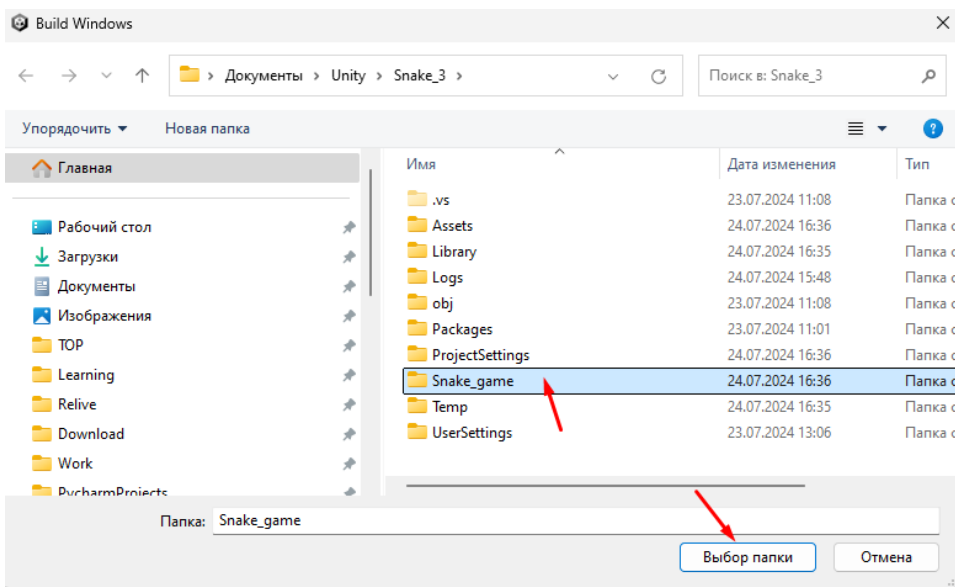
```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Food"))
    {
        score++;
        UpdateTextScore();
        Grow();
        Time.fixedDeltaTime -= speedIncrease;
    }
    else if (other.CompareTag("Obstacle"))
    {
        ResetGame();
    }
}
```

Итоговый билд

16. Осталось только скомпилировать нашу игру. Переходим в **File – Build And Run**:



Выбираете любую папку куда хотите сохранить игру, или создаёте новую папку:



После можете запустить игру через .exe:

