Geekbrains

**Создание 2D игры на Unity и C#**

IT-специалист:

Разработчик. Программист. Цифровые профессии

Александров И.В.

2024

**Содержание**

[**Введение** 3](#_Toc166671900)

[**1. Основная часть. Теория.** 5](#_Toc166671901)

[**1.1 Настройка и основы работы в Unity** 5](#_Toc166671902)

[**1.2 Иерархия, позиционирование, объекты, основные компоненты** 8](#_Toc166671903)

[**1.4 Применение C# в Unity** 25](#_Toc166671904)

[**1.5 Префабы** 29](#_Toc166671905)

[**1.6 GetComponent, GameObject и функции** 33](#_Toc166671906)

[**1.7 Корутины** 35](#_Toc166671907)

[**1.8 Массивы и списки** 38](#_Toc166671908)

[**2. Практическая часть. Создание игры.** 41](#_Toc166671909)

[**2.1 Перемещение и вращение игрока, Input** 41](#_Toc166671910)

[**2.2 Стрельба, создание пули и дополнительных функций** 44](#_Toc166671911)

[**2.3 Следование камеры за игроком** 49](#_Toc166671912)

[**2.4 Создание уровня, ассеты** 51](#_Toc166671913)

[**2.5 Создание анимации игрока** 54](#_Toc166671914)

[**2.6 Пользовательский интерфейс (UI)** 58](#_Toc166671915)

[**2.7 Создание врага, его передвижение и анимация** 60](#_Toc166671916)

[**2.8 Настройка врага с ближней атакой** 64](#_Toc166671917)

[**2.9 Система частиц, анимация получения урона** 67](#_Toc166671918)

[**2.10 Создание анимации выстрела игрока.** 69](#_Toc166671919)

[**2.11 Оставшиеся враги: стрелок и подрывник** 74](#_Toc166671920)

[**2.12 Рывок, полоса здоровья игрока, генератор волн врагов** 81](#_Toc166671921)

[**2.13 Неуязвимость игрока во время рывка, следы ног, экран проигрыша, монеты** 89](#_Toc166671922)

[**2.14 Внутриигровой магазин** 97](#_Toc166671923)

[**2.15 Post Processing (эффекты)** 106](#_Toc166671924)

[**2.16 Universal RP (свет)** 108](#_Toc166671925)

[**2.18 Прицел** 112](#_Toc166671926)

[**Заключение** 115](#_Toc166671927)

# **Введение**

Проект, который я представляю, представляет собой разработку полнофункциональной 2D видеоигры, выполненной с использованием игрового движка Unity и языка программирования C#. Основная цель проекта — продемонстрировать мои навыки в программировании и игровом дизайне, а также глубже изучить потенциал Unity и C# в создании интерактивных приложений. Это особенно актуально в контексте постоянно растущего спроса на видеоигры в свете динамичного развития игровой индустрии.

Цели и задачи проекта

Моя задача заключается не только в разработке игры, которая будет технически исполнима и визуально привлекательна, но и в создании интересного для пользователя продукта с уникальными игровыми механиками. Игра разрабатывается как шаблон, который в дальнейшем может быть расширен, улучшен и адаптирован под различные игровые потребности.

План работы

Работа над проектом включает следующие этапы:

* Установка и настройка инструментов: Настройка рабочей среды в Unity и Visual Studio Code.
* Проектирование: Разработка концепции игры, дизайна уровней и персонажей.
* Реализация: Написание кода для игровых механик, интерфейса и взаимодействия с пользователем.
* Тестирование и оптимизация: Поиск и устранение ошибок, улучшение производительности и пользовательского опыта.

Техническая реализация

В процессе разработки основное внимание уделялось программной реализации игровых механик, таких как управление персонажем, система боя и взаимодействие с игровым миром. Были исследованы и применены различные технологии Unity, включая системы частиц для создания визуальных эффектов, Post Processing для улучшения графики и Audio System для интеграции звуковых эффектов и музыкального сопровождения.

Обучение и развитие

Проект стал значимым шагом в моем обучении, позволив глубже понять принципы разработки игр и программирования на C#. Работа над игрой помогла мне освоить новые аспекты работы с Unity, улучшить навыки кодирования и дизайна, а также разработать стратегии для оптимизации и тестирования программного обеспечения.

В заключение, этот проект не только позволит мне продемонстрировать и углубить мои знания в области разработки игр, но и подчеркнул важность гибкости и адаптивности в процессе создания программных продуктов. Игра, разработанная в рамках данного проекта, не только будет предлагать пользователям увлекательный игровой процесс, но и открывать двери для дальнейших исследований и развития в области игрового дизайна.

# **1. Основная часть. Теория.**

## **1.1 Настройка и основы работы в Unity**

С самого начала нам необходимо установить все основные инструменты для выполнения нашего проекта. Для этого потребуется Unity Hub, последняя версия которого — 3.7.0, скачанный с официального сайта разработчика. После установки Unity Hub следует скачать версию Unity Editor. Проект выполнен на версии 2022.3.22f1 (Рис. 1).

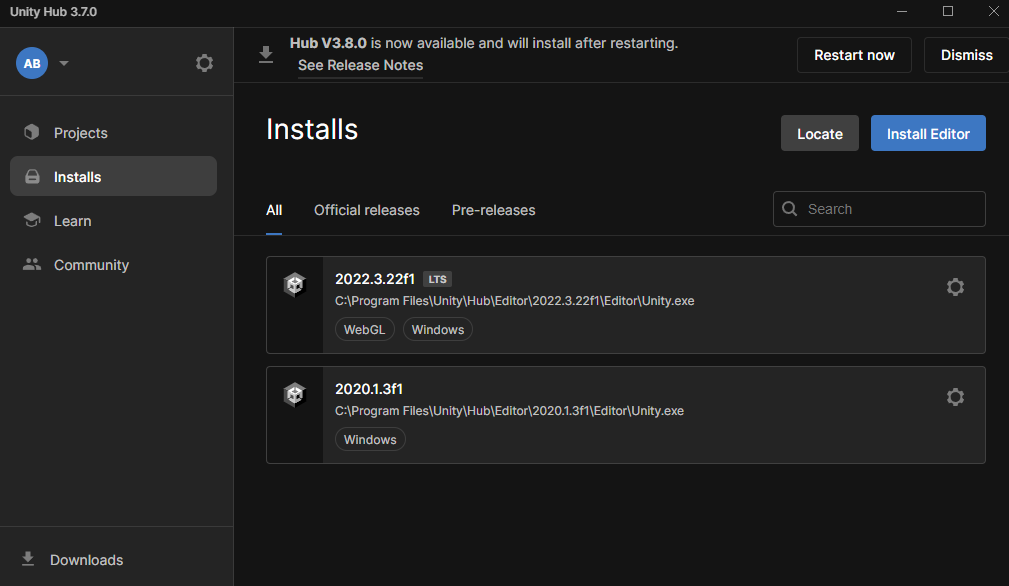


Рис. 1. Окно Unity Hub на установке Unity Editor.

Также для этого мы создаём новый проект во вкладке «Projects», нажимая на кнопку «New Project». В открывшемся окне выбираем версию редактора 2022.3.22f1 и параметр "2D", задаём имя проекту и директорию, где будут храниться все файлы игры (Рис. 2).

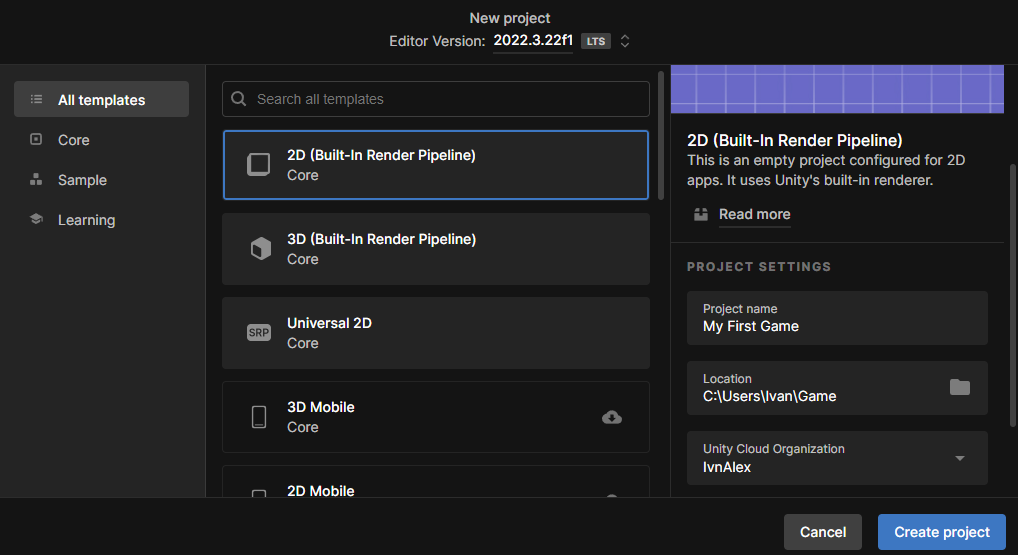


Рис. 1. Окно Unity Hub на создании проекта.

Также для этого мы попадаем в среду для полной разработки игры, где видим основные окна для взаимодействия с проектом (Рис. 3):

* Иерархия (Hierarchy): окно необходимо для отслеживания структуры проекта и просмотра моделей объектов. Для добавления объекта достаточно кликнуть правой кнопкой мыши и выбрать интересующий объект.
* Сцена (Scene): одно из основных окон, в котором будут добавляться объекты, с которыми будет происходить основная работа. После добавления на сцене появится объект, с которым мы можем работать.
* Игра (Game): окно показывает, что будет видеть игрок, поэтому важно иметь его под рукой, чтобы понимать, что происходит в самом проекте.
* Проект (Project): предоставляет доступ ко всем файлам проекта.
* Консоль (Console): выводит информацию для отладки.

Инспектор (Inspector): здесь отображаются данные и свойства выбранного объекта, что позволяет нам настраивать его параметры и поведение в игре.

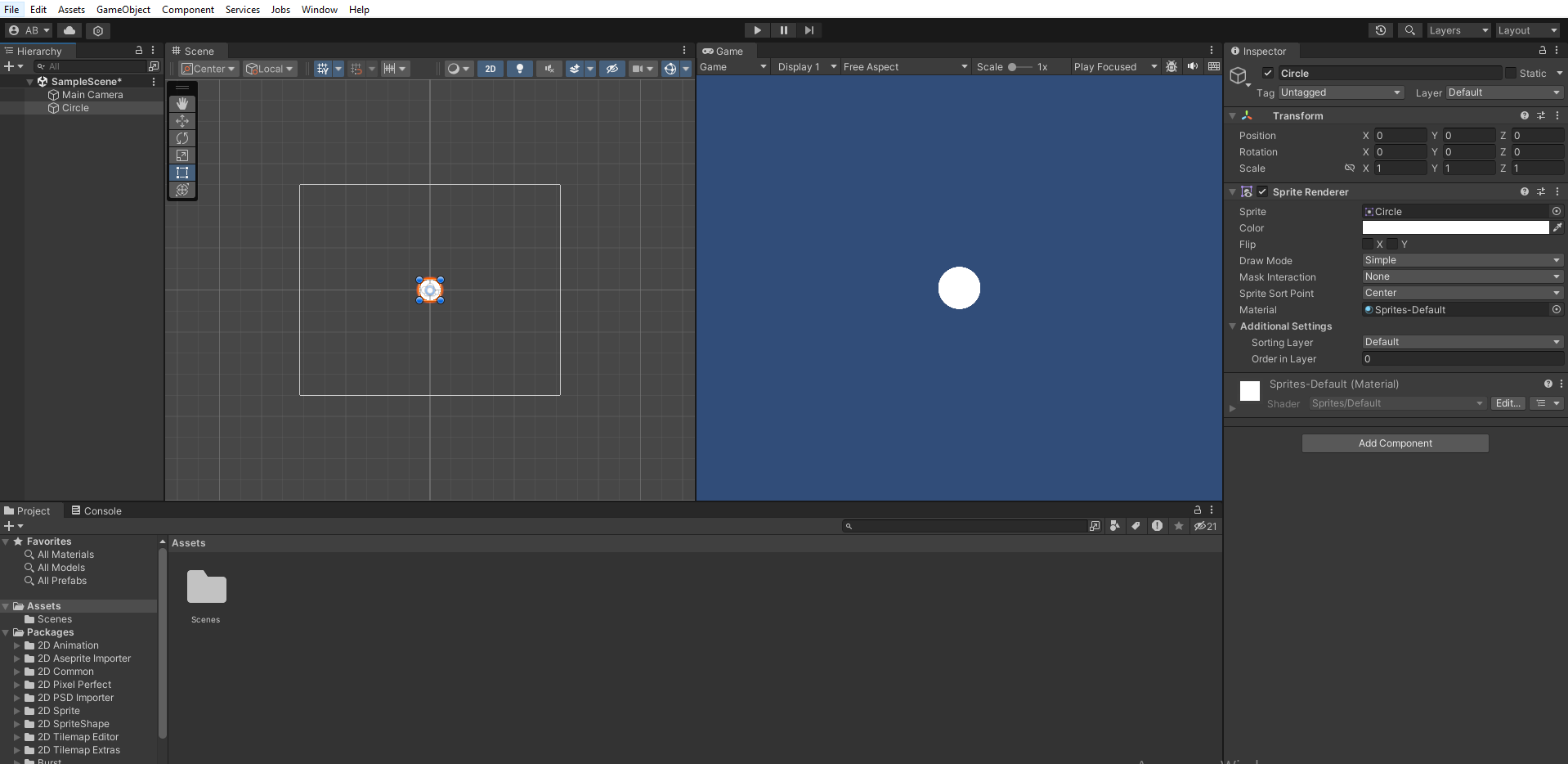


Рис. 3. Среда Unity с основными окнами.

Также для этого стоит просмотреть папку проекта. В ней можно заметить файлы с расширением .meta, которые отвечают за данные и свойства в окне Инспектор. Как можно заметить, кроме папки Assets в директориях файлов проекта существуют и другие важные папки (Рис. 4):

* «Library» – хранит библиотеки, необходимые для работы Unity.
* «Logs» – содержит логи проекта.
* «Packages» – предназначена для импортирования дополнительных пакетов.
* «ProjectSettings» – включает глобальные настройки проекта.
* «Temp» – временная папка, которая удаляется при закрытии проекта.
* «UserSettings» – содержит пользовательские настройки.

Чтобы кто-нибудь другой мог попробовать поиграть в вашу игру, необходимо скопировать только папки «Assets», «Packages» и «ProjectSettings». Остальные папки Unity сгенерирует автоматически.

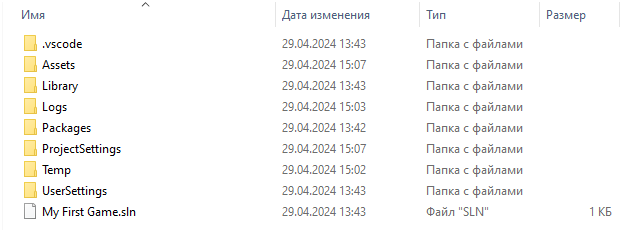


Рис. 4. Структура проекта.

## **1.2 Иерархия, позиционирование, объекты, основные компоненты**

Важно отметить, что все свойства объектов в Unity состоят из компонентов, каждый из которых играет свою роль. Например, компонент «Transform» отвечает за положение объекта в пространстве, то есть его позицию, вращение и масштаб (Рис. 5).

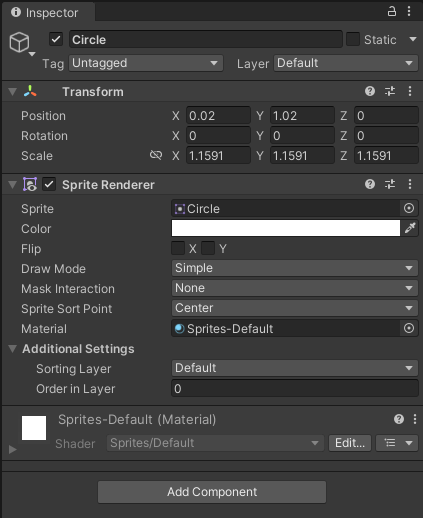


Рис. 5. Компоненты объектов.

Для всех этих настроек существуют специальные инструменты: «Move Tool», «Rotate Tool», «Scale Tool», «Rect Tool» и «Transform Tool» (Рис. 6).

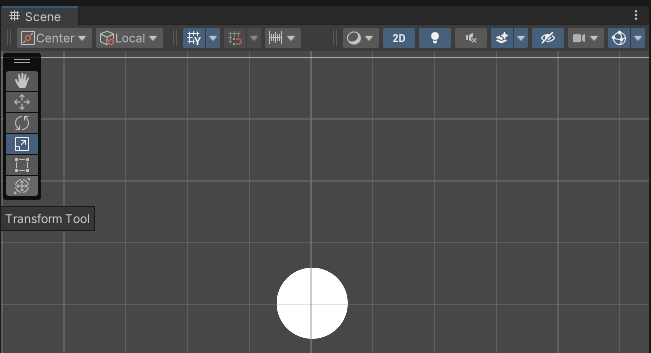


Рис. 6. Инструменты для компонента «Transform».

Рассмотрим иерархию объектов в Unity и концепцию их локального и глобального положения.

Иерархия объектов

В Unity иерархия объектов представляет собой структуру, которая показывает, как объекты (часто называемые «игровыми объектами» или «GameObjects») организованы в проекте. Эта структура влияет на то, как объекты взаимодействуют друг с другом и как они представлены в игровом мире. Иерархия также определяет, какие объекты являются "родителями" и какие "детьми" (Рис. 7).

Родительский объект (Parent Object): Контролирует определенные аспекты его дочерних объектов. Например, перемещение родителя автоматически перемещает все его дочерние объекты.

Дочерний объект (Child Object): Подчиняется некоторым параметрам своего родителя. Например, если родительский объект будет перемещен, повернут или масштабирован, то же самое произойдет с дочерними объектами.

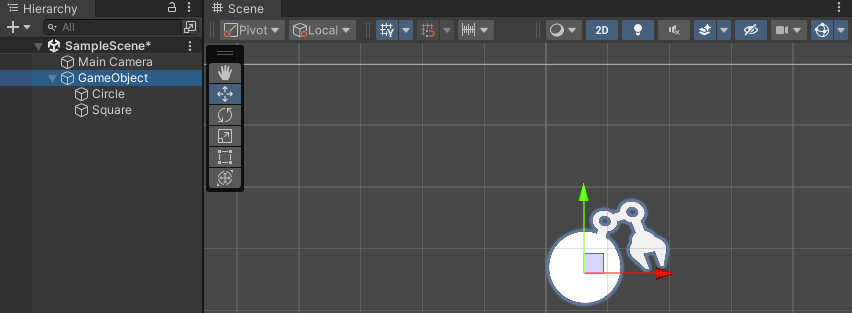


Рис. 7. Родительский и дочерние объекты.

Локальное и глобальное положение

Каждый объект в Unity имеет свойства «Transform». Эти свойства могут быть заданы в двух разных контекстах: локальном и глобальном (или мировом) (Рис. 8).

Локальное положение (Local Position): Это положение объекта относительно его родительского объекта. Если объект является дочерним, его локальное положение будет отсчитываться от позиции родителя, что означает, что если родительский объект переместится, локальные координаты дочернего объекта останутся теми же, но его глобальное положение изменится.

Глобальное положение (Global or World Position): Это абсолютное положение объекта в игровом мире. Оно не зависит от того, имеет ли объект родительский объект или нет.

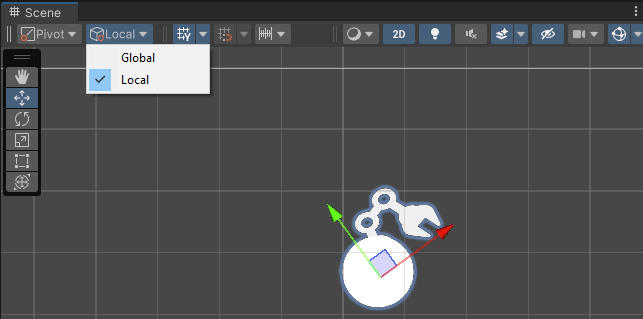


Рис. 8. Локальное и глобальное положение объекта.

Понимание различия между локальными и глобальными координатами важно для правильного позиционирования и анимации объектов в Unity, особенно когда дело доходит до сложных иерархий объектов, где дочерние объекты имеют множество уровней родительских объектов. Это знание помогает избежать ошибок в расчетах перемещений, вращений и масштабирования.

Unity предоставляет разработчикам ряд стандартных объектов, которые можно использовать в проектах для создания различных аспектов игры или приложения.

2D объекты. Unity предоставляет специализированные инструменты и объекты для работы с 2D-графикой, что делает его популярной платформой для разработки 2D-игр.

* **«Sprites»:** Как уже упоминалось, это 2D-графические изображения, используемые для создания персонажей, фонов, элементов интерфейса и других визуальных аспектов 2D-игр.
* **«Sprite Renderer»:** Компонент для отображения спрайтов на сцене. Он контролирует визуализацию спрайтов, включая параметры, такие как цвет, масштаб и ориентация.
* **«2D Colliders»:** Unity предлагает различные типы 2D коллайдеров (например, «BoxCollider2D», «CircleCollider2D», «PolygonCollider2D»), которые используются для обработки физического взаимодействия в 2D-пространстве.
* **«2D Physics»:** Для управления физическими свойствами объектов в 2D-играх, Unity имеет компоненты, такие как «Rigidbody2D», которые добавляют реалистичное физическое поведение к объектам.

3D объекты. Для создания трехмерных сцен и объектов в Unity используется ряд стандартных объектов и компонентов, которые позволяют создавать более сложные и динамичные среды.

* **«Meshes»:** 3D модели, которые создаются с помощью вершин и полигонов. Эти объекты используются для создания сложных трехмерных форм, таких как персонажи, здания и природные объекты.
* **«Mesh Renderer»:** Компонент, который вместе с «Mesh Filter» используется для отображения 3D моделей на сцене.
* **«3D Colliders»:** Как и в случае с 2D, для 3D объектов есть свои коллайдеры, такие как «BoxCollider», «SphereCollider», «MeshCollider», которые помогают в обработке столкновений и физических взаимодействий.
* **«3D Physics»:** Компоненты, такие как «Rigidbody», которые добавляют динамику и реалистичное поведение к 3D объектам в игре.
* **«Lighting»:** Освещение в 3D сценах играет ключевую роль, так как оно влияет на восприятие глубины, текстур и атмосферы.

Оба типа объектов, 2D и 3D, поддерживают разнообразные стили и техники проектирования, предоставляя разработчикам гибкие инструменты для создания уникальных игровых миров. Unity эффективно интегрирует оба подхода, позволяя комбинировать элементы 2D и 3D в одном проекте, что расширяет творческие возможности разработчиков.

Вот так же еще некоторые из наиболее часто используемых стандартных объектов в Unity (Рис. 9):

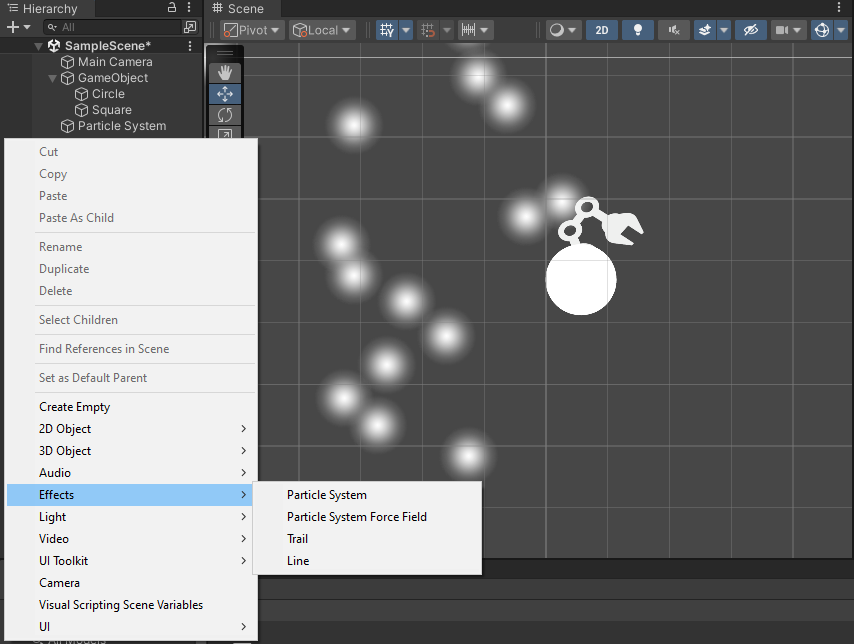


Рис. 9. Стандартные объекты

1. «**Empty GameObject».** Это базовый и самый простой объект в Unity, который не имеет визуального представления или предопределённой функциональности. Он часто используется как контейнер для компонентов или как родительский объект для организации других объектов в иерархии сцены.

2. «**Camera».** Камера — это объект, который определяет, какая часть сцены будет отображаться на экране. Unity поддерживает различные типы камер, включая перспективные и ортографические, которые используются для разных стилей визуализации, от 3D-сцен до 2D-игр.

3. «**Light».** Освещение важно для создания атмосферы и улучшения визуального восприятия сцены. Unity предлагает разные типы источников света, такие как «Directional Light», «Point Light», «Spot Light» и «Area Light», каждый из которых имеет свои особенности и настройки.

4. «**UI Elements».** Unity имеет мощную систему для создания пользовательского интерфейса (UI), включая элементы такие как «Text», «Image», «Button», «Slider» и многие другие. Эти элементы помогают создавать интерактивные меню, панели инструментов, информационные дисплеи и другие интерфейсные элементы.

5. «**Particle System».** Системы частиц используются для создания эффектов, таких как огонь, дым, туман или взрывы. Unity предлагает гибкий и мощный инструментарий для настройки поведения частиц, их внешнего вида и динамики.

6. «**Audio Source».** «Audio Source» — это компонент, который позволяет воспроизводить звуки в вашей сцене. Он может использоваться для добавления фоновой музыки, звуковых эффектов и других аудио элементов, которые улучшают атмосферу или интерактивность игры.

Эти стандартные объекты и их компоненты образуют основу для большинства проектов в Unity, позволяя разработчикам создавать сложные и интересные интерактивные приложения, и игры.

Рассмотрим основные компоненты Unity. Чтобы их добавить объекту необходимо просто нажать на кнопку «Add component» и выбрать необходимый. Добавим нашему основному объекту два компонента (Рис. 10):

1. **«Rigidbody 2D»**

«Rigidbody 2D» — это компонент в Unity, предназначенный для объектов, которые должны взаимодействовать с двумерной физикой. Прикрепление этого компонента к объекту позволяет ему участвовать в физической симуляции, что включает в себя гравитацию, столкновения, тягу и другие аспекты движения в 2D пространстве.

Основные свойства «Rigidbody 2D»:

* «Mass»: Масса объекта, которая влияет на то, как он реагирует на столкновения и силы.
* «Drag»: Сопротивление движению в воздухе. Большее значение уменьшает скорость объекта со временем.
* «Angular Drag»: Сопротивление вращению объекта.
* «Gravity Scale»: Коэффициент, который определяет, насколько сильно гравитация влияет на данный объект. Установка в ноль делает объект невосприимчивым к гравитации.
* «Body Type»: Указывает тип тела:
* «Dynamic»: Полностью подвержено воздействию физики.
* «Kinematic»: Не реагирует на столкновения, но все еще может быть перемещено с помощью скриптов.
* «Static»: Не двигается вовсе и не реагирует на силы.
* «Collision Detection»: Способ, которым Unity обнаруживает столкновения с этим объектом. Режимы включают «Discrete» (стандартный) и «Continuous» (для высокоскоростных объектов).
* «Interpolate»: Уменьшает дрожание объектов при перемещении за счет интерполяции между физическими обновлениями.

1. **«Collider 2D»**

«Collider 2D» — это компонент, который определяет форму объекта для целей физических столкновений и взаимодействий в 2D пространстве. Collider необходим для обработки входящих столкновений и может работать в сочетании с «Rigidbody 2D» для создания реалистичного физического поведения.

Типы «Collider 2D»:

* «BoxCollider2D»: Представляет собой прямоугольник, который можно настроить по размеру и положению.
* «CircleCollider2D»: Используется для объектов круглой формы.
* «PolygonCollider2D»: Автоматически или вручную определяет многоугольную форму, охватывающую объект.
* «EdgeCollider2D»: Создает коллайдер в форме линий или ребер, который не имеет твердой внутренней области и используется для создания границ или платформ.

Настройки «Collider 2D»:

* «Is Trigger»: Когда установлено в true, коллайдер не вызывает физического столкновения, но по-прежнему может срабатывать события, например, когда объект входит в триггерную зону.
* «Used By Effector»: Это свойство включает использование этого коллайдера с эффекторами, такими как платформенный эффектор, который изменяет поведение объектов, сталкивающихся или взаимодействующих с коллайдером.
* «Material»: Физический материал, который определяет трение и упругость коллайдера.

Использование «**Rigidbody 2D» и «Collider 2D»** вместе позволяет разработчикам создавать сложные и реалистичные физические сценарии в 2D играх, таких как платформеры, гонки и другие жанры, где важно точное и предсказуемое физическое поведение.

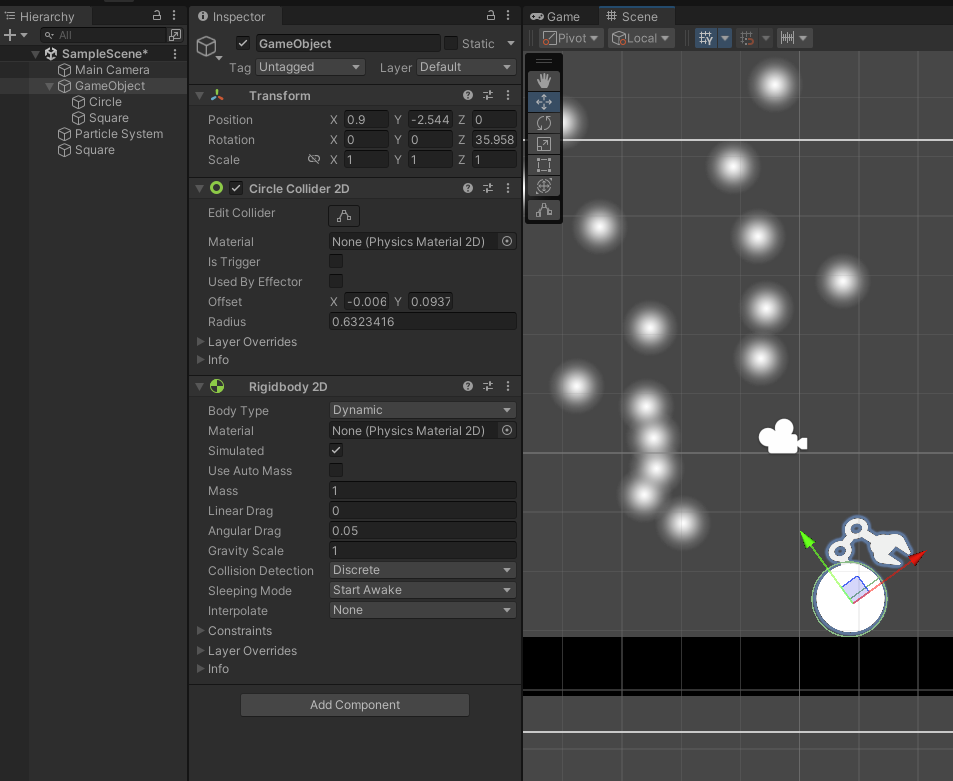


Рис. 10. Два компонента Rigidbody 2D и Collider 2D в действии.

В Unity множество интересных компонентов, которые значительно расширяют возможности разработчиков при создании игр и интерактивных приложений (Рис. 11). Вот несколько особенно замечательных компонентов, которые могут добавить уникальные функции в ваш проект:

**1. Particle System.** Это один из самых мощных и визуально привлекательных компонентов в Unity. «Particle System» позволяет создавать сложные эффекты, такие как дым, огонь, взрывы, туман или волшебные частицы. Система частиц в Unity гибка и настраиваема, что позволяет детально контролировать жизненный цикл каждой частицы, её поведение при столкновениях, влияние гравитации и многое другое.

**2. Nav Mesh Agent.** «Nav Mesh Agent» используется для имплементации искусственного интеллекта перемещения, позволяя персонажам автоматически находить путь в сложной среде, избегая препятствий и следуя к целям. Это особенно полезно для создания NPC (неигровых персонажей), которые могут перемещаться по игровому миру, следуя сложным маршрутам без прямого управления со стороны игрока.

**3. Animator.** Компонент «Animator» управляет анимациями персонажей и других объектов с использованием мощного Animator Controller, который позволяет настроить сложные схемы анимации с использованием состояний и переходов. Это ключевой инструмент для создания плавного и реалистичного движения в играх.

**4. Audio Source и Audio Listener.** Эти компоненты управляют всем, что связано с аудио в Unity. «Audio Source» позволяет объектам воспроизводить звуки, в то время как «Audio Listener» действует как уши игрока. Можно настроить звуки на 3D-пространственное аудио, чтобы усилить погружение в игру, делая звуковой ландшафт более реалистичным.

**5. Scriptable Objects.** Это не компонент в традиционном смысле, но «Scriptable Objects» позволяют разработчикам хранить данные независимо от классов сценариев, что является отличным способом управления конфигурациями, настройками и другими данными, которые должны быть легко доступны через разные части игры без необходимости дублирования объектов.

**6. Physics Materials.** Определяют, как объекты скользят и отскакивают друг от друга. Эти материалы могут быть применены к Collider, чтобы настроить трение и упругость, что делает взаимодействия более реалистичными или, наоборот, более стилизованными.

Каждый из этих компонентов открывает новые возможности для создания, оптимизации и усовершенствования игр и приложений в Unity, позволяя разработчикам тонко настраивать взаимодействие, поведение и визуальные эффекты.

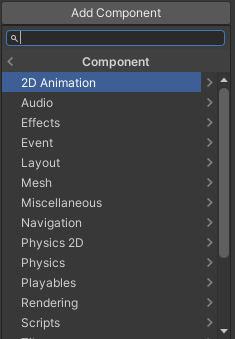


Рис. 11. Компоненты Unity.

**1.3 Настройка IDE\Console, скрипты**

В процессе разработки игр на Unity одним из ключевых аспектов является написание скриптов, которые контролируют логику игры, поведение объектов, взаимодействия и многое другое. Для эффективной работы с кодом в Unity необходимо выбрать подходящую интегрированную среду разработки (IDE), которая обеспечит удобные инструменты для написания, отладки и управления кодом.

Выбор IDE для Unity. Visual Studio является одним из наиболее популярных выборов среди разработчиков Unity, особенно на платформах Windows и macOS. Это мощная IDE поддерживает C#, предлагает обширные возможности для отладки, интеллектуальное автодополнение кода (IntelliSense), а также интеграцию с Unity, что делает её идеальным выбором для разработки игр. Её мы и будем использовать для проекта (Рис. 12).

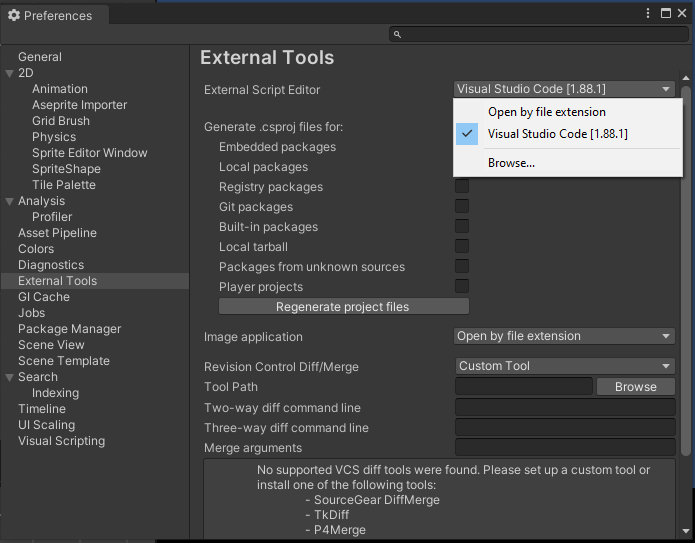


Рис. 12. Выбор IDE.

При использовании Visual Studio разработчики получают следующие преимущества:

* Тесная интеграция с Unity: Visual Studio может автоматически подхватывать все скрипты из проекта Unity, облегчая навигацию по проекту и управление его структурой.
* Отладка: Visual Studio предоставляет продвинутые функции отладки, позволяющие устанавливать точки останова, просматривать состояние переменных и выполнять код пошагово.
* Редактирование и рефакторинг кода: Мощные инструменты для редактирования и рефакторинга кода помогают улучшить качество кода и упростить его поддержку.

Необходимость консоли

Консоль в Unity играет важную роль в процессе разработки, так как она служит для вывода системных сообщений, предупреждений и ошибок. Это основное средство для отслеживания поведения игры и диагностики проблем. Разработчики используют консоль для вывода отладочной информации, что особенно полезно при тестировании и настройке игры (Рис 13).

Использование консоли позволяет разработчикам:

* Отслеживать и логировать ошибки: Быстро находить и исправлять ошибки в скриптах.
* Мониторинг состояния игры: Выводить значения переменных или состояние игровых объектов для анализа их поведения в реальном времени.
* Тестировать код: Проверять корректность работы отдельных функций и алгоритмов.

Выбор правильной IDE и использование консоли значительно упрощают процесс разработки в Unity, повышая эффективность и уменьшая вероятность ошибок в конечном продукте.

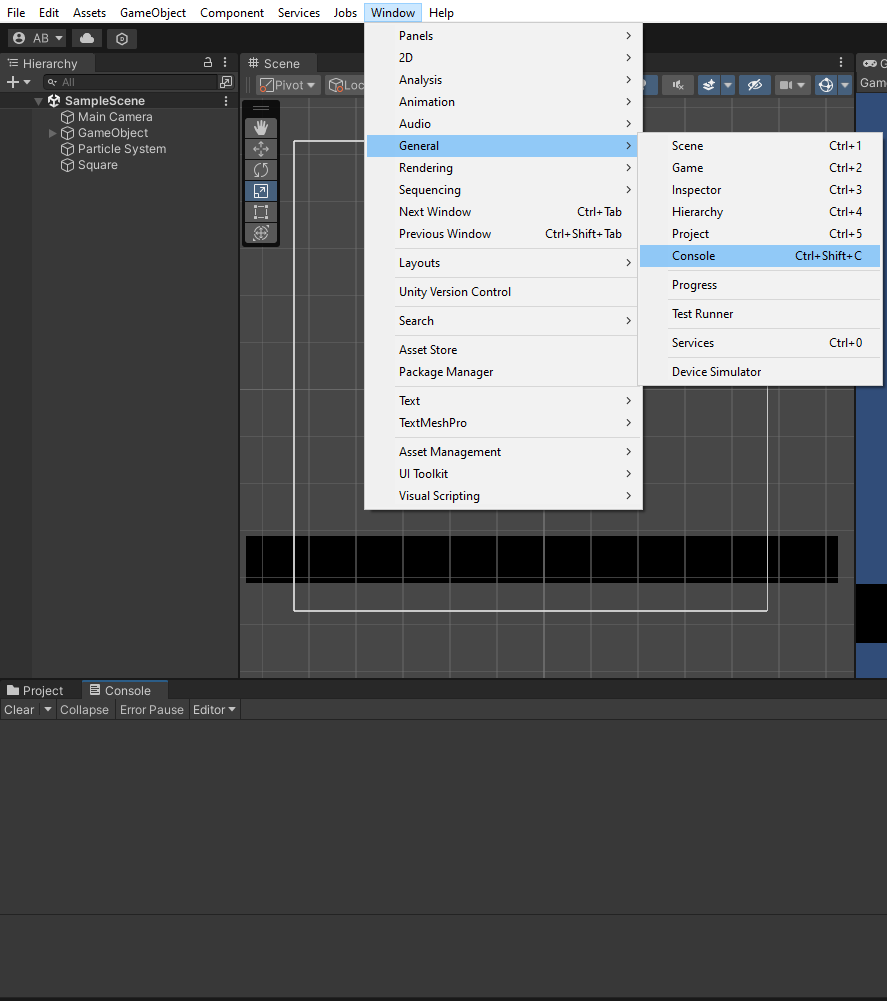


Рис. 13. Настройка консоли.

Для создания скриптов необходимо сначала создать папку с названием «Scripts» в нашем проекте «Project», расположенном в папке «Assets». Для этого кликните правой кнопкой мыши в свободном месте проекта, выберите «Create», а затем «Folder». После создания папки «Scripts», войдите в неё и, также используя контекстное меню (правая кнопка мыши), выберите «Create» и «C# Script». Затем дайте скрипту нужное название (Рис. 14).

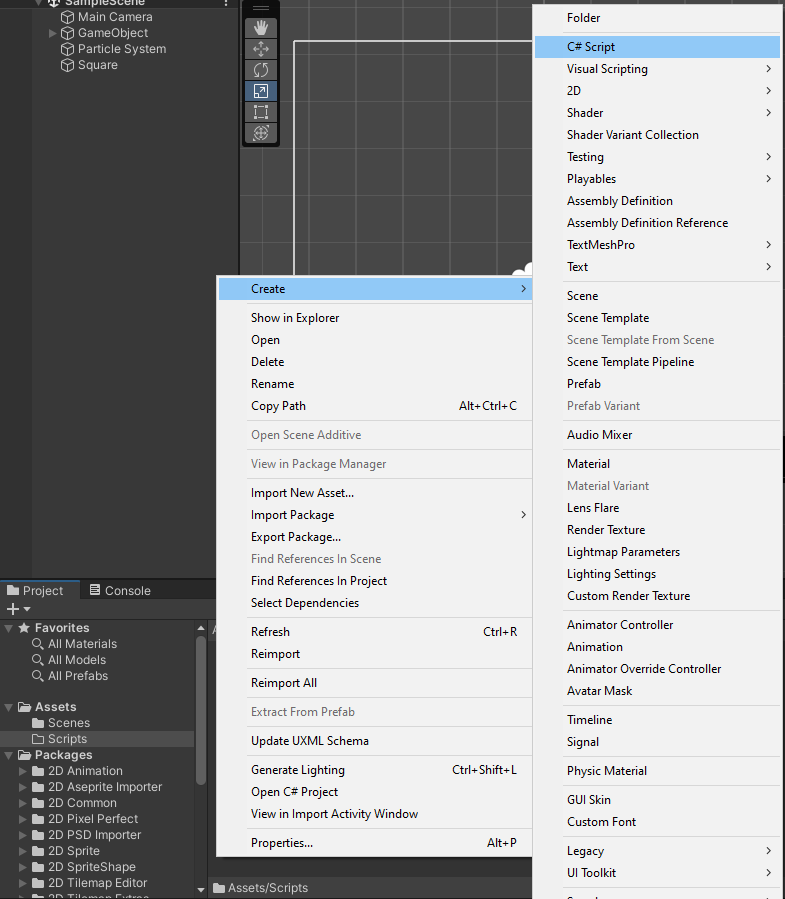


Рис. 14. Создание скрипта в проекте в папке «Scripts».

Когда мы создаем новый C# скрипт в Unity, IDE автоматически генерирует базовый шаблон кода. Этот шаблон содержит начальную структуру класса и основные методы, которые используются в большинстве игровых скриптов (Рис. 15). Вот разбор стандартного шаблона скрипта, который вы упомянули:

Импорт пространств имен

**using System.Collections;**

**using System.Collections.Generic;**

**using UnityEngine;**

Эти строки включают в скрипт различные пространства имен:

«System.Collections» и «System.Collections.Generic» предоставляют доступ к классам для работы с коллекциями данных, такими как списки и словари.

«UnityEngine» — это основное пространство имен, предоставляющее доступ к классам и функциям «Unity Engine».

Определение класса - **public class player : MonoBehaviour**

Здесь определяется класс player, который наследуется от класса «MonoBehaviour». «MonoBehaviour» — это базовый класс в Unity для всех скриптов, которые прикрепляются к игровым объектам. Классы, производные от «MonoBehaviour», могут взаимодействовать с системой игры через встроенные функции, такие как «Start» и «Update».

Метод «Start» - **void Start()**

Метод «Start()» вызывается один раз перед первым кадром, только после инициализации всех объектов. Этот метод обычно используется для инициализации переменных, настройки начального состояния объекта или вызова функций настройки, которые должны выполняться перед началом игры.

Метод «Update» - **void Update()**

Метод «Update()» вызывается каждый кадр, что делает его идеальным местом для реализации логики обработки ввода пользователя, движения персонажей, временных проверок и других операций, которые должны проверяться или выполняться очень часто.

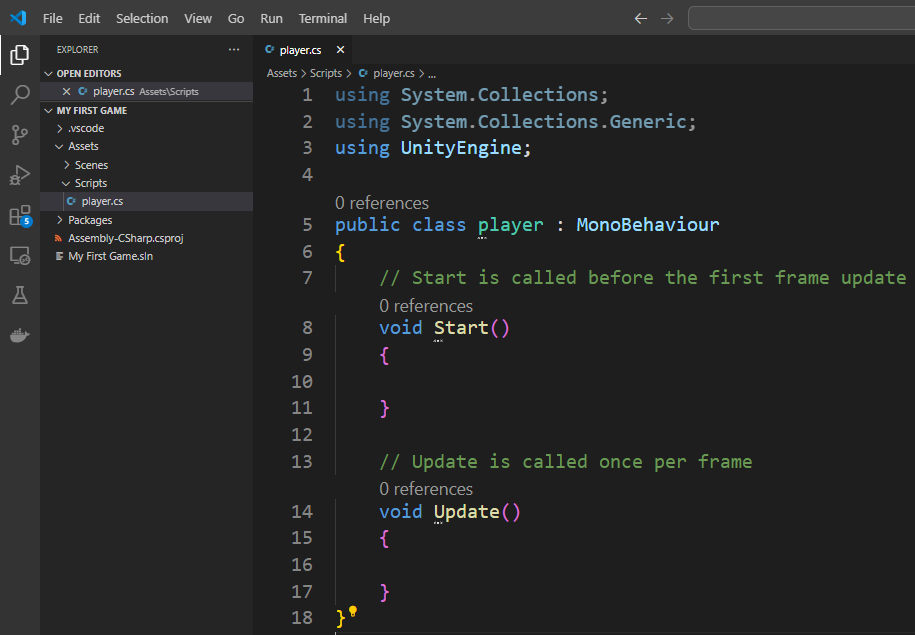


Рис. 15. Шаблон, созданный IDE Visual Studio.

Чтобы скрипт начал функционировать и взаимодействовать с игровым объектом в Unity, его необходимо прикрепить к этому объекту. В вашем случае, скрипт нужно вложить в игровой объект «Player», который представляет собой персонажа в игре (Рис. 16).

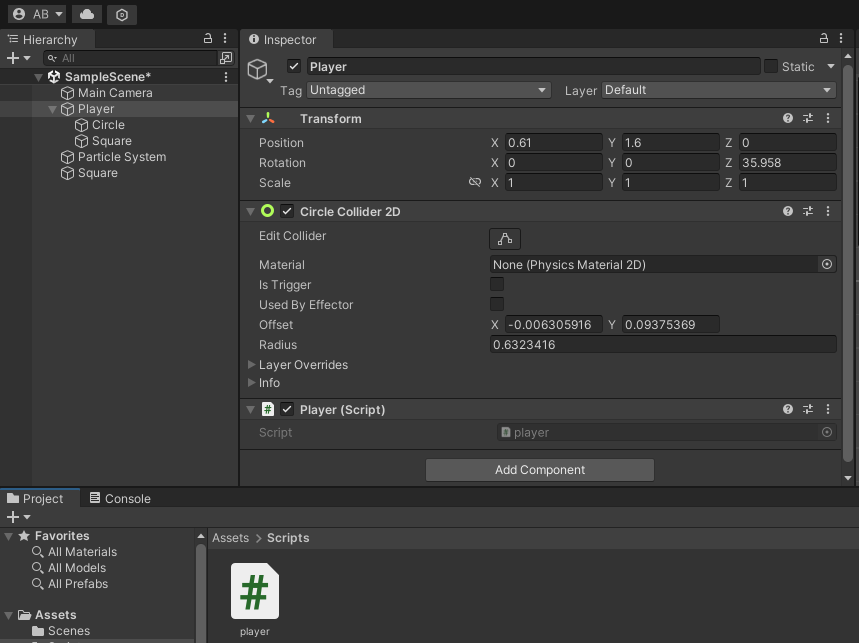


Рис. 16. Прикрепленный скрипт к объекту «Player».

Для того чтобы проверить совместную работу консоли и IDE можно написать несколько простых тестов с выводом сообщения в консоль в методе «Start» (Рис. 17), а именно:

Debug.Log(1); //Выводит сообщение – «1»

Debug.LogWarning(1); //Выводит предупреждение – «1»

Debug.LogError(1); //Выводит ошибку – «1»

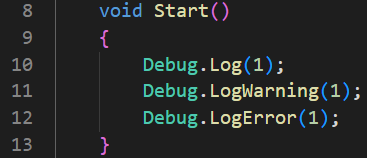


Рис. 17. VS Code.

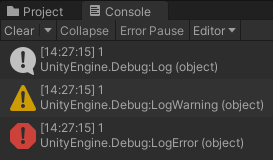


Рис. 18. Console Unity.

Так мы проверили взаимодействие между интегрированной средой разработки (IDE) и консолью Unity. Это подтверждает, что код, написанный в IDE, корректно взаимодействует с движком Unity и что вывод сообщений в консоль работает как ожидается (Рис. 18).

## **1.4 Применение C# в Unity**

Для создания простейшей 2D игры на Unity с использованием C# важно овладеть некоторыми основными аспектами языка программирования и понимать, как они применяются в контексте Unity.

Допустим мы знаем основы C#:

* Переменные и типы данных: Понимание различных типов данных (например, int, float, string, bool) и как их использовать для хранения информации.
* Управляющие структуры: Условные операторы (if, else) и циклы (for, while), которые помогут контролировать поток выполнения программы.
* Методы: Создание повторно используемых блоков кода (функций), которые выполняют конкретные задачи.
* Классы и объекты: Основы объектно-ориентированного программирования, включая создание и использование классов.

В дополнение к этим основам, существуют специализированные типы данных, такие как «Vector3» и «Vector2», которые часто используются для работы с компонентами «Transform»:

* «Vector3» используется для хранения трёхмерных координат (x, y, z).
* «Vector2» — для двухмерных координат (x, y), что актуально в разработке 2D игр.

Допустим, мы хотим изменить положение игрового объекта через скрипт. Вот пример того, как это можно сделать (Рис. 19, 20):

Объявляем переменную «Transform» в нашем скрипте:

* public «Transform a1»;
* В методе «Start», меняем позицию объекта:
* a1.position = new Vector2(2, 3);
* Перемещаем компонент «Transform» объекта «Player» в компонентах «Script».

После запуска игры, мы увидим, что объект «Player» изменил своё положение согласно заданным в скрипте координатам. Это демонстрирует, как можно изменять параметры объектов через скрипты в Visual Studio Code. Таким образом, мы практически применяем наши знания C# в разработке на Unity, настраивая поведение игровых объектов с помощью кода.

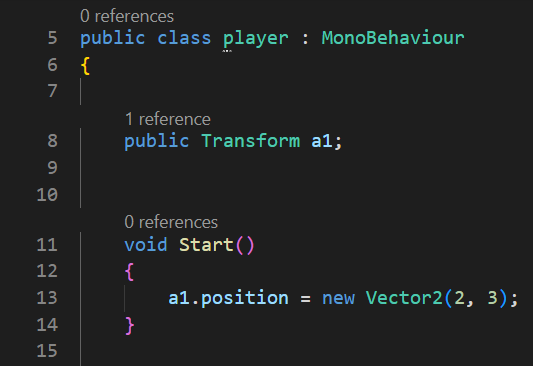


Рис. 19. Новая переменная Transform с Vector2.

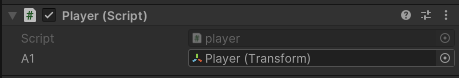


Рис. 20. Компонент «Transform» объекта «Player» в компонентах «Script».

Так же по аналогии можно делать и с условными операторами.  
Создадим цветовой светофор с помощью конструкции «if, else».

Этот пример предназначен для того, чтобы показать, как можно использовать условные операторы if, else в Unity для создания простой симуляции цветового светофора. Скрипт изменяет цвет спрайта в зависимости от значения таймера, который увеличивается с течением времени. Вот подробное описание работы этого скрипта:

Описание переменных

* «timer»: переменная типа float, которая служит для отсчёта времени.
* «spr»: переменная типа «SpriteRenderer» (Рис. 22), которая используется для управления свойствами спрайта, включая его цвет.

Логика работы скрипта. В методе «Start» задаётся начальное положение объекта с помощью координат вектора (2, 3).

В методе «Update», который вызывается каждый кадр к переменной timer прибавляется значение Time.deltaTime, что обеспечивает её увеличение в реальном времени.

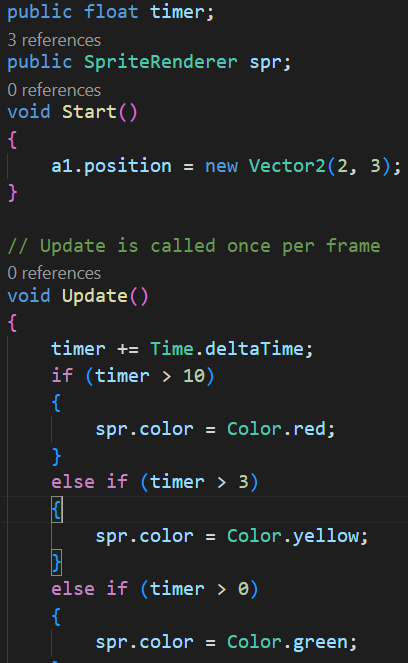


Рис. 21. Код цветового светофора.

С помощью условных операторов if и else if проверяется значение таймера (Рис. 21):

* Если timer больше 10 секунд, цвет спрайта устанавливается в красный (Color.red).
* Если timer больше 3 секунд, но меньше или равно 10 секундам, цвет спрайта устанавливается в жёлтый (Color.yellow).
* Если timer больше 0 секунд, но меньше или равно 3 секундам, цвет спрайта устанавливается в зелёный (Color.green).

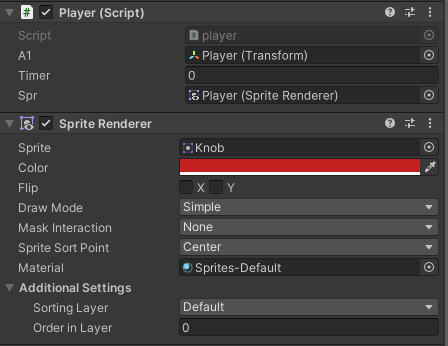


Рис. 22. Компонент Sprite Renderer со скриптом.

В процессе работы с двумя предложенными примерами скриптов на Unity с использованием C#, мы изучили такие вопросы как:

1. Основа работы с компонентами в Unity:

Понимание того, как скрипты взаимодействуют с компонентами игровых объектов, на примере «Transform» и «SpriteRenderer».

1. Применение основ программирования на C# в реальных задачах:

Использование переменных, условных операторов и циклов для управления поведением игровых объектов.

1. Разработка интерактивности в играх:

Создание скриптов, которые реагируют на изменения в игре в реальном времени, такие как перемещение объектов и изменение цветов.

Чему мы научились:

1. Управление положением объектов:

В первом примере мы научились изменять положение объекта с помощью компонента «Transform». Это позволило понять, как можно программно управлять позиционированием объектов на сцене, что является фундаментальной способностью для любой игры.

1. Использование таймеров и условных операторов для создания логики игры:

Во втором примере мы реализовали простую систему светофора, используя таймер и условные операторы для изменения цвета спрайта. Это демонстрирует, как можно использовать время и логические условия для управления потоком игры и взаимодействия пользователя.

1. Отладка и тестирование изменений в реальном времени:

Оба примера показали, как важно тестировать изменения в поведении объектов в Unity, используя игровую среду для непосредственного просмотра результатов кода.

1. Разработка адаптивных систем реагирования на изменения:

Понимание того, как динамически адаптировать поведение объектов в игре в зависимости от различных условий и входных данных.

Эти упражнения предоставили ценный опыт в области программирования и дизайна игр, укрепив знания и умения, необходимые для разработки более сложных игровых проектов в Unity.

**1.5 Префабы**

Префабы (prefabs) в Unity — это мощный инструмент, который позволяет разработчикам создавать, настраивать и повторно использовать игровые объекты и их конфигурации в различных сценах и проектах. Префаб представляет собой шаблон объекта, который содержит компоненты и настройки, необходимые для определенного элемента игры, будь то персонаж, предмет, декорация или интерактивный объект. В контексте создания простенького рогалика префабы могут играть ключевую роль в эффективности разработки и управлении контентом. Вот основные аспекты префабов, которые могут быть полезны для вашей игры:

1. Создание префабов

Чтобы создать префаб, вы начинаете с настройки игрового объекта в сцене Unity, добавляя все необходимые компоненты, такие как спрайты, коллайдеры, скрипты и другие элементы. Затем перетаскиваете этот объект из иерархии сцены в папку в вашем проекте. Таким образом, объект становится префабом, который можно повторно использовать, перетаскивая его из проекта обратно в любую сцену.

2. Преимущества использования префабов

* Консистентность: Все копии префаба будут иметь одинаковые настройки, что обеспечивает единообразие и уменьшает вероятность ошибок.
* Экономия времени: Вы создаете элемент один раз, а затем можете использовать его множество раз без дополнительных усилий на настройку.
* Легкость обновлений: Изменения в префабе автоматически применяются ко всем его экземплярам в проекте, что упрощает масштабирование и изменение игры.

3. Применение в нашей игре.

В игре, где генерация уровней и множество элементов повторяются, префабы становятся особенно полезными:

* Персонажи и враги: Создайте префабы для различных типов врагов и персонажей, управляемых AI.
* Элементы уровней: Разные виды стен, полов, ловушек и декоративных объектов могут быть реализованы через префабы, что упрощает их расстановку при генерации уровней.
* Интерактивные объекты: Двери, сундуки, ключи и другие интерактивные элементы могут быть легко добавлены и настроены через префабы.

4. Управление префабами

Варианты префабов (Prefab Variants): Если вы хотите создать объекты, которые слегка отличаются от основного префаба (например, враги с разными способностями), можно использовать варианты префабов. Это позволяет наследовать и изменять свойства от базового префаба без изменения самого префаба.

Использование префабов в Unity для создания рогалика значительно упростит процесс разработки, обеспечив эффективное повторное использование активов и унифицированное управление элементами игры. Это позволяет сосредоточиться на дизайне и геймплее, минимизируя рутинную работу по настройке объектов.

Создаем наш первый префаб (Рис. 23):

* Добавляем любой объект, даем ему компоненты «rigibody 2d» и «collider 2d»;
* Создаем папку Prefabs и перемещаем наш объект туда.

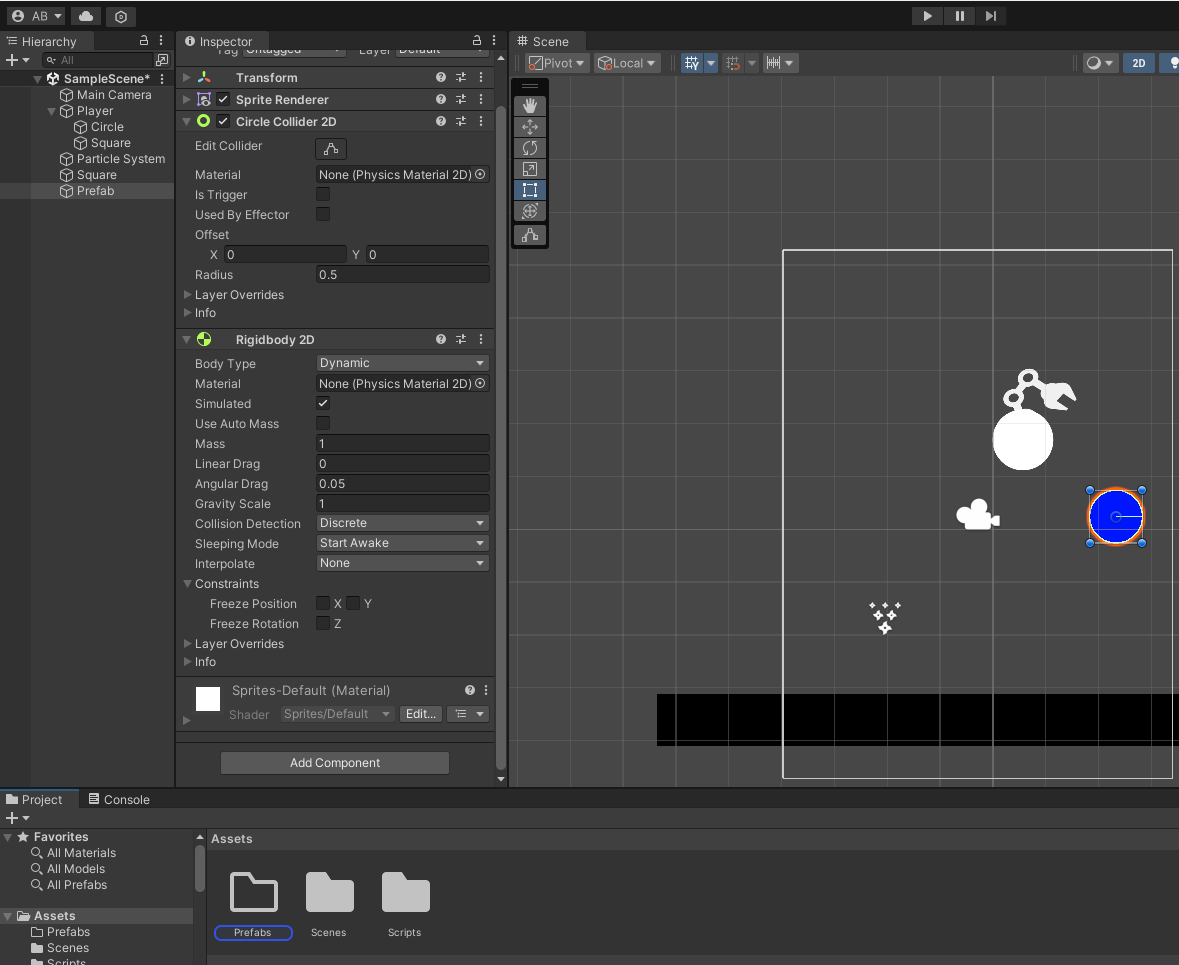


Рис. 23. Создание префаба.

После создания префаба вы можете использовать его по своему усмотрению. Можно бесконечно добавлять этот объект на сцену или использовать его в более сложных скриптах. Например, создайте скрипт, который будет генерировать ваш префаб во время игры при нажатии на определенную клавишу (Рис. 24).

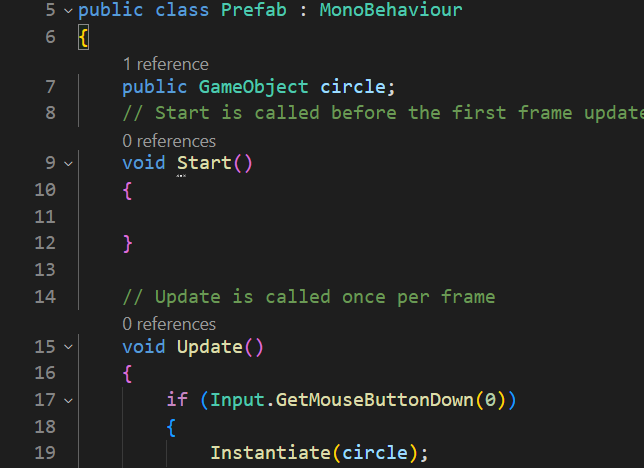


Рис. 24. Создание префаба по нажатии кнопки мыши.

Для работы этого скрипта, необходимо прикрепить его к игровому объекту, например, к игроку. Также убедитесь, что в объект скрипта добавлен ваш префаб. После этих шагов, в игровом окне вы сможете создавать префабы по нажатию левой кнопки мыши (Рис. 25).

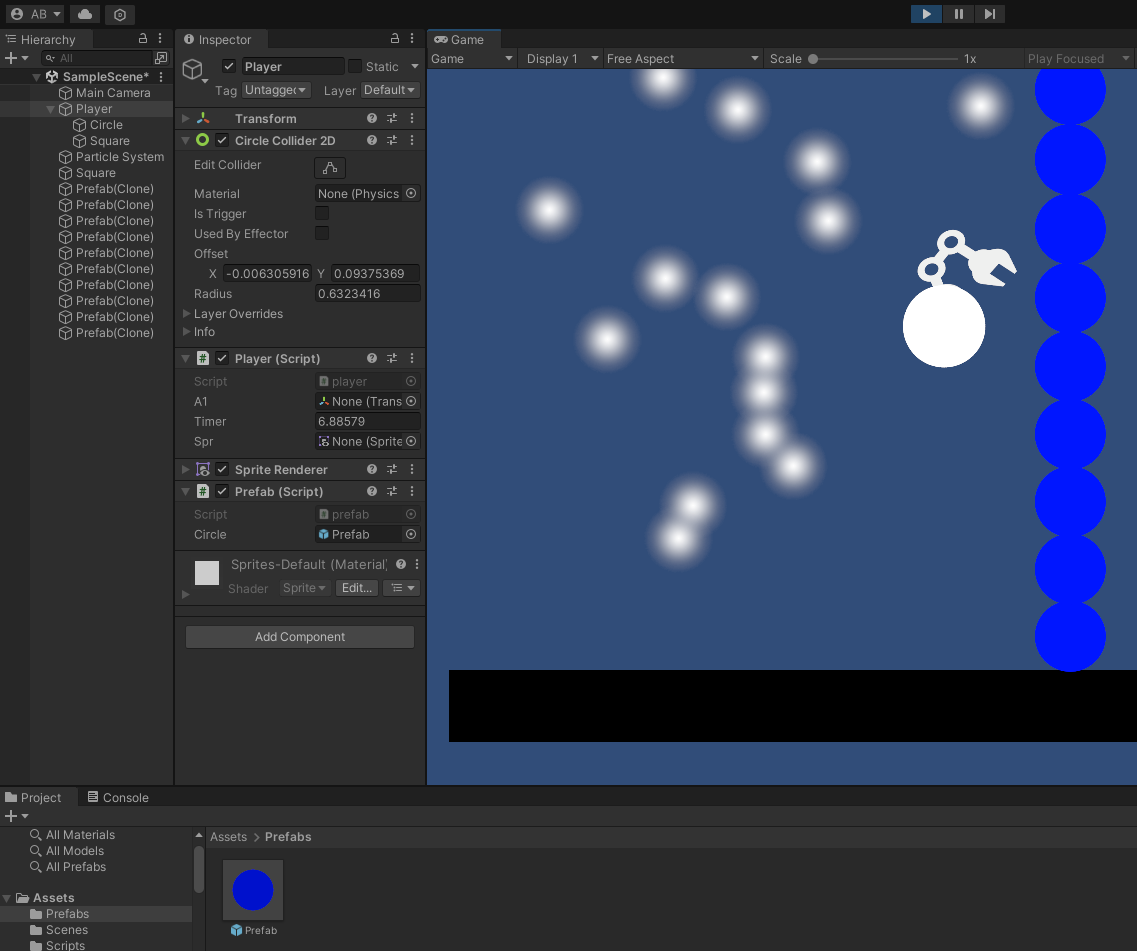


Рис. 25. Результат работы скрипта «Prefab».

**1.6 GetComponent, GameObject и функции**

Функция «GetComponent» в Unity — один из самых мощных инструментов для программирования на C#. Эта функция позволяет получить доступ к компонентам, которые присоединены к «GameObject» в ваших сценах.

На простом примере мы можем изменить имя объекта, используя класс «GameObject». Чтобы продемонстрировать, как работает «GetComponent», мы воспользуемся компонентом Rigidbody2D и изменим его массу на 5. Все эти изменения будут произведены в момент старта игры (Рис. 26).

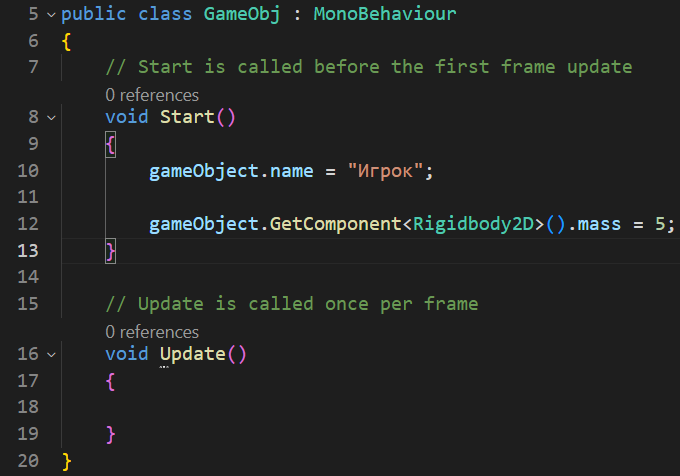


Рис. 26. Изменение компонентов с помощью «GetComponent», «GameObject».

Изменения мы сможем увидеть в инспекторе (Рис. 27).

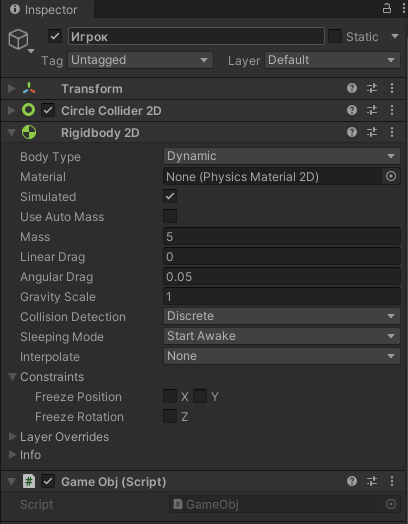


Рис. 27. Инспектор объекта.

Рассмотрим функции в рамках нашей игры. Функции в программировании играют критически важную роль, позволяя структурировать код, избегать дублирования и делать программу более читаемой и управляемой. В контексте разработки игр на Unity, использование функций C# особенно полезно для организации игровой логики, обработки событий, управления состояниями игровых объектов и взаимодействия между различными компонентами системы. Вот несколько примеров того, как функции могут быть использованы в нашей 2D игре:

1. Обработка игровых событий

Функции могут быть использованы для обработки событий в игре, таких как атака, смерть персонажа или сбор предметов. Например, функция «Shoot()» (Рис. 28) может вызываться, когда игрок нажимает кнопку атаки:

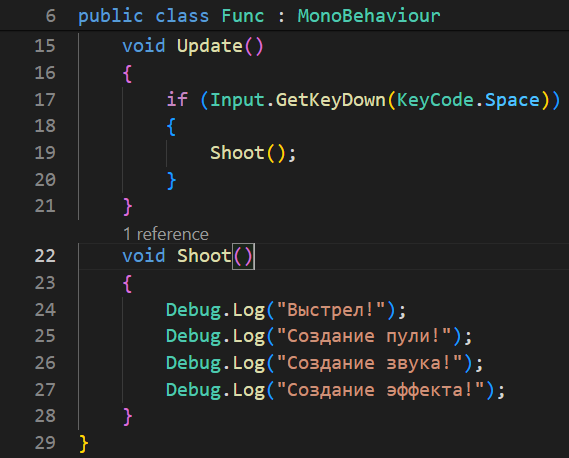


Рис. 28. Самый простой код функции «Shoot()».

После этого если мы во время игры будем нажимать на пробел, то в консоль будут выводиться 3 сообщения (Рис. 29).

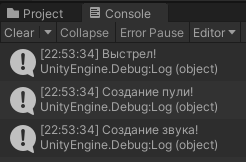


Рис. 29. Консоль после нажатия пробела, используя функцию «Shoot()».

2. Управление состоянием игры

Вы можете использовать функции для изменения или проверки состояний игры, таких как начало и конец уровня, пауза или продолжение игры. Например, функция «PauseGame()» может останавливать течение игры.

3. Анимация и визуальные эффекты

Функции могут управлять анимациями или запускать визуальные эффекты в ответ на определенные действия или события. Например, функция «PlayAnimation()» может активировать анимацию победы.

4. Интерфейс пользователя

Функции также могут управлять элементами интерфейса пользователя, например, обновлять счет, жизни игрока или другие индикаторы.

Все эти функции могут быть совершенно различными и выполнять уникальные задачи, наполняя нашу игру контентом. Всё зависит от нашего креатива.

Поэтому использование функций в игре позволяет не только сделать код более организованным и поддерживаемым, но и упрощает расширение игры новыми фичами и контентом, делая разработку более гибкой и эффективной.

**1.7 Корутины**

Корутины в Unity — это инструмент, позволяющий управлять последовательностями операций, время выполнения которых распространяется на несколько кадров или секунд. В Unity корутины особенно полезны для создания задержек, асинхронной логики или для организации сложного потока выполнения без блокировки основного потока игры.

Что такое корутины?

Корутина — это функция в Unity, которая может приостанавливать своё выполнение в определённой точке с помощью оператора «yield», а затем продолжать с этого места в следующих кадрах. Корутины пишутся как методы, возвращающие «IEnumerator», и они используют ключевые слова «yield return» для управления своим выполнением.

Как работают корутины?

Корутины выполняются внутри компонента «MonoBehaviour» и запускаются с помощью метода «StartCoroutine()». В теле корутины можно использовать различные варианты «yield», такие как:

* «yield return null» — приостановить корутину и возобновить её на следующем кадре.
* «yield return new WaitForSeconds(n)» — приостановить корутину на n секунд.
* «yield return StartCoroutine(anotherCoroutine())» — приостановить текущую корутину до тех пор, пока не завершится другая корутина.
* «yield break» — преждевременно завершить корутину.

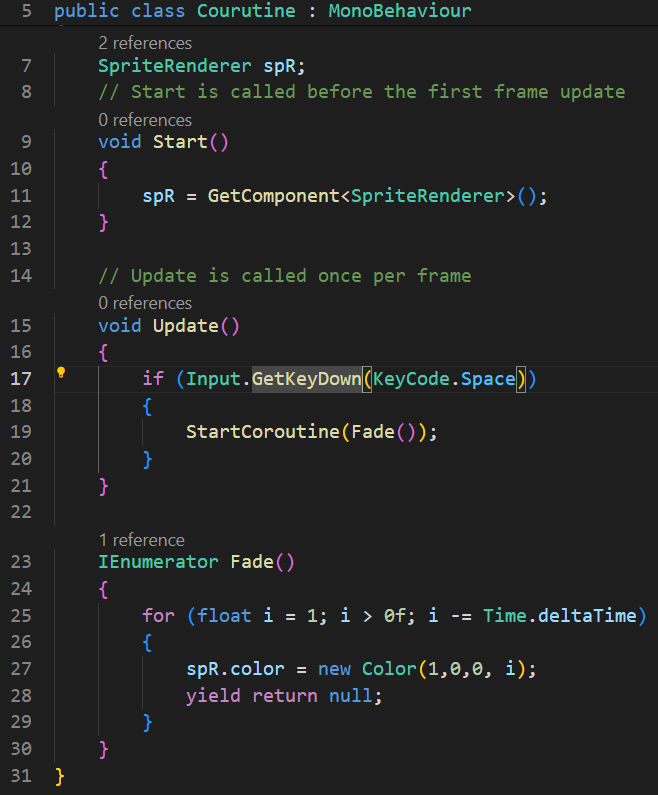


Рис. 30. Пример использования корутины.

Этот код (Рис. 30) пример использования корутины в Unity для создания эффекта плавного исчезновения (fade out) спрайта при нажатии клавиши пробела. Вот более подробное описание каждой части кода:

Класс «Courutine» наследуется от «MonoBehaviour», что позволяет ему взаимодействовать с компонентами Unity и использовать функции жизненного цикла, такие как «Start()» и «Update()».

Объявляется переменная «spR» типа «SpriteRenderer». Этот компонент используется для управления визуальными свойствами спрайта, такими как его цвет.

В методе «Start()», который вызывается один раз перед первым кадром, происходит инициализация переменной «spR». С помощью метода «GetComponent<>» из текущего объекта извлекается компонент «SpriteRenderer», который позволяет манипулировать спрайтом.

Метод «Update()» вызывается каждый кадр и проверяет, была ли нажата клавиша пробел (KeyCode.Space). Если клавиша нажата, то запускается корутина Fade().

Корутина «Fade()» создает плавное уменьшение прозрачности спрайта, что создает эффект исчезновения. В цикле for переменная i начинается с 1 (полностью непрозрачный) и уменьшается до 0 (полностью прозрачный) с шагом Time.deltaTime, который представляет собой время в секундах, прошедшее с последнего кадра. Это обеспечивает плавное изменение. В каждой итерации цикла цвет спрайта обновляется с новым уровнем альфа-канала (i), где new Color(1,0,0,i) представляет красный цвет с изменяемой прозрачностью. yield return null; означает, что корутина должна приостановить выполнение и продолжить с следующего кадра.

В результате при нажатии клавиши пробел спрайт будет плавно терять свою видимость, изменяя свою прозрачность от полностью видимого к полностью прозрачному. Этот эффект часто используется в играх для анимации исчезновения объектов.

**1.8 Массивы и списки**

Массивы в C# (и в Unity) представляют собой структуры данных, позволяющие хранить наборы элементов одного типа. В контексте Unity массивы часто используются для управления коллекциями объектов, таких как игровые объекты, компоненты или любые другие типы данных, которые нужно группировать и обрабатывать совместно. Рассмотрим код, который демонстрирует работу с массивами в Unity (Рис. 31):



Рис. 31. Пример применения массива.

Здесь «rbs» — это массив компонентов «Rigidbody2D», а objects — массив игровых объектов (GameObject). Объявление массивов публичными позволяет их видеть и настраивать в инспекторе Unity, что удобно для присваивания значений вручную или для проверки и отладки.

Метод «Start()» вызывается перед первым кадром, где происходит выполнение начальной настройки:

Цикл for используется для итерации по каждому элементу массива objects. В каждой итерации:

* Изменяется имя объекта, добавляя его индекс (i + " объект"), что упрощает его идентификацию в иерархии Unity.
* Устанавливается красный цвет для компонента «SpriteRenderer» каждого объекта.
* К каждому объекту добавляется компонент «Rigidbody2D», что делает объекты участниками физической симуляции.

Заполнение массива rbs

Использование FindObjectsOfType<Rigidbody2D>() автоматически находит все объекты в сцене, которые имеют компонент «Rigidbody2D», и возвращает массив этих компонентов. Это позволяет легко получить доступ ко всем «Rigidbody2D» в сцене для дальнейшей обработки или управления.

Преимущества использования массивов в Unity

* Организация: Массивы позволяют удобно хранить и управлять группами объектов.
* Повторное использование кода: Циклы вместе с массивами позволяют применять одни и те же операции ко многим объектам, сокращая дублирование кода.
* Производительность: Эффективное использование массивов может повысить производительность, особенно при обработке больших данных.

Классы списков в C# (в данном случае List<T>) представляют собой динамические коллекции, которые могут расширяться и сокращаться в размере по мере необходимости, что делает их чрезвычайно удобными для работы с коллекциями данных, размер которых заранее неизвестен или может изменяться.

В примере кода (Рис. 32) используется «List<GameObject>» для управления коллекцией игровых объектов в Unity. Вот подробное описание работы с этим списком:

Класс «List» наследуется от «MonoBehaviour», что позволяет ему взаимодействовать с компонентами Unity и использовать функции жизненного цикла.

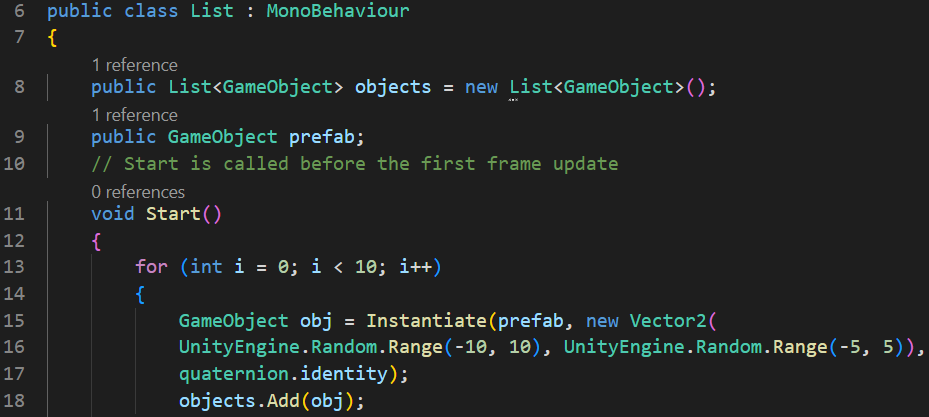


Рис. 32. Пример применения списка.

Сначала объявляется список «objects», предназначенный для хранения объектов «GameObject». Также объявлен «prefab», который будет использоваться как шаблон для создания новых объектов.

В методе Start(), который вызывается перед первым кадром, происходит создание и добавление новых объектов в список «objects». Используя цикл for, код создаёт 10 новых объектов из «prefab». Каждый объект инстанциируется в случайной позиции на сцене. После создания каждый новый объект добавляется в список «objects».

В общем, и массивы, и списки предоставляют мощные средства для работы с коллекциями объектов в Unity. Выбор между использованием массива или списка зависит от специфических потребностей проекта и предпочтений в управлении данными. Для статических данных, размер которых не изменяется, подойдут массивы, а для динамичных коллекций, где элементы часто добавляются или удаляются, лучше использовать списки.

# **2. Практическая часть. Создание игры.**

**2.1 Перемещение и вращение игрока, Input**

В рамках разработки 2D игры на Unity, управление игроком, включая перемещение и вращение, является ключевым элементом геймплея. Для реализации этих механик в Unity используются встроенные средства ввода (Input) и компоненты физики, такие как «Rigidbody2D». Вот как можно реализовать базовое управление игроком, используя C# и Unity.

Работа с «Input»

Unity предоставляет класс «Input», который позволяет получать входные данные от клавиатуры, мыши, геймпада и других устройств. Для перемещения и вращения игрока обычно используются оси, настроенные в менеджере ввода Unity (Edit > Project Settings > Input Manager).

Получение входных данных

float horizontal = Input.GetAxis("Horizontal");

float vertical = Input.GetAxis("Vertical");

Эти строки кода получают значения осей, которые по умолчанию связаны с клавишами стрелок и W, A, S, D на клавиатуре. Значения будут варьироваться от -1 до 1, где -1 это полное движение влево или вниз, 1 — вправо или вверх.

Перемещение игрока

Для перемещения игрока можно использовать компонент «Rigidbody2D», который позволяет объекту взаимодействовать с физической системой Unity.

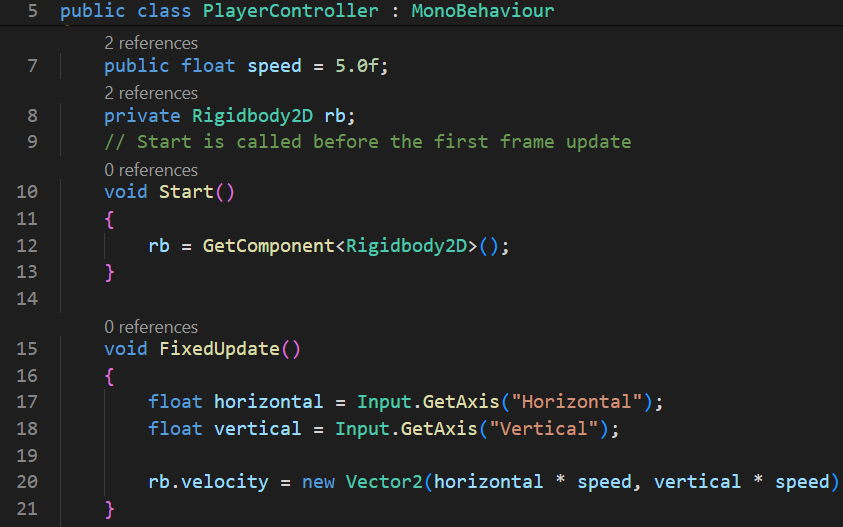


Рис. 33. Пример скрипта для перемещения.

В этом примере скрипт изменяет скорость (velocity) «Rigidbody2D», что приводит к перемещению игрока в зависимости от ввода пользователя. Метод «FixedUpdate» используется для обновления физических свойств, так как он работает в соответствии с физическим таймером Unity.

Для создания механики, при которой игрок поворачивается вслед за курсором мыши в игре, можно использовать несколько подходов в Unity с использованием C# (Рис. 34). Вот пример того, как это можно реализовать:

Основные шаги для поворота персонажа за мышкой

1. Определение положения мыши: Нужно получить положение курсора мыши в экранных координатах и преобразовать его в мировые координаты.
2. Вычисление угла поворота: Нужно вычислить угол между текущим направлением игрока и направлением на курсор мыши.
3. Поворот игрока: Применить вычисленный угол для поворота игрока так, чтобы он смотрел на курсор.

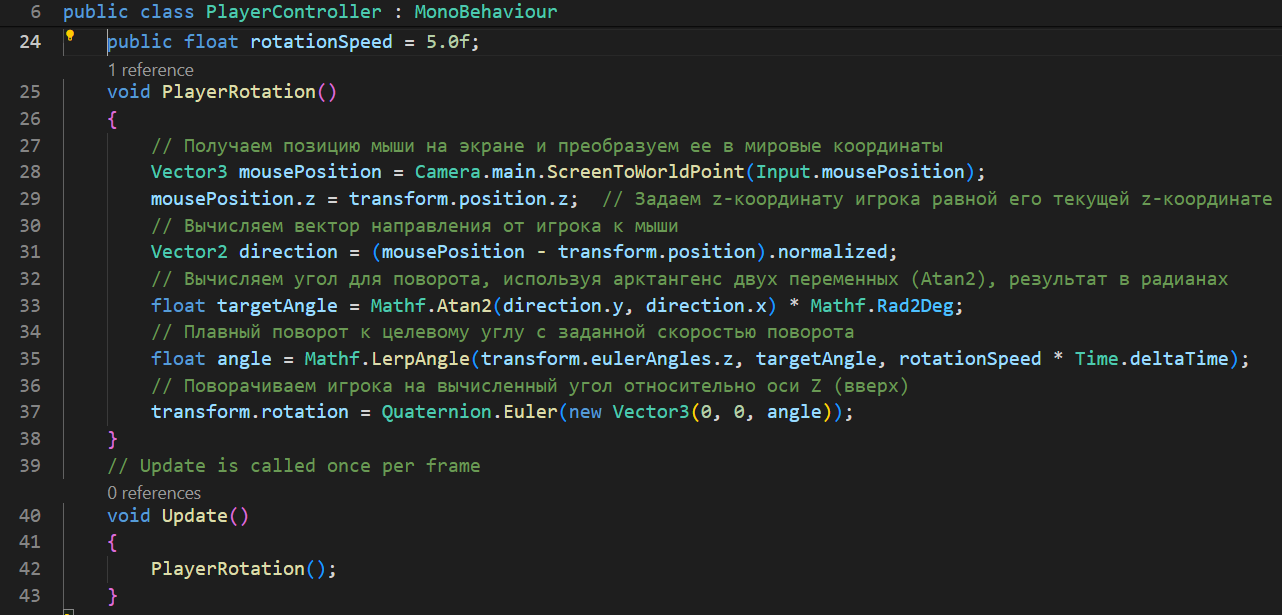


Рис. 34. Пример скрипта для поворота игрока за мышкой.

Как работает код:

1. Публичная переменная «rotationSpeed»:

* Это переменная, которую можно настроить прямо в Unity Editor, определяя, насколько быстро будет поворачиваться персонаж. Большее значение увеличивает скорость поворота.

1. Вычисление целевого угла:

* Целевой угол вычисляется на основе позиции мыши. Это направление, куда должен смотреть персонаж.

1. Плавный поворот с «Mathf.LerpAngle»:

* Вместо прямого присваивания угла используется функция «Mathf.LerpAngle», которая плавно интерполирует углы от текущего угла поворота к целевому. Это создает более естественное движение поворота.
* Функция «Mathf.LerpAngle» учитывает наименьшее расстояние для поворота, что устраняет возможные проблемы при переходе через угол 360°/0°.

В Unity одной из ключевых возможностей является удобство настройки параметров через инспектор, без необходимости изменения кода. Использование публичных переменных, таких как public float speed = 5.0f; и public float rotationSpeed = 5.0f;, позволяет разработчикам и дизайнерам игр гибко настраивать поведение игровых объектов прямо в редакторе Unity. Вот как это работает и почему это важно:

Когда мы объявляем переменные в скрипте как «public» или используете «[SerializeField]» для приватных переменных, Unity автоматически отображает эти переменные в инспекторе, прикрепленном к объекту, который использует данный скрипт (Рис. 35). Это делает переменные доступными для редактирования в редакторе Unity.

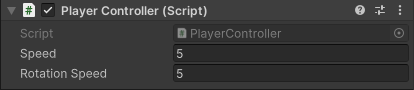


Рис. 35. Изменение параметров speed и «rotationSpeed» через инспектор.

**2.2 Стрельба, создание пули и дополнительных функций**

Напишем код, который реализует базовый механизм стрельбы для игрока и движение пули в простой 2D игре в Unity. Давайте разберем, как мы его написали и что делали в Unity (Рис. 36, 37).

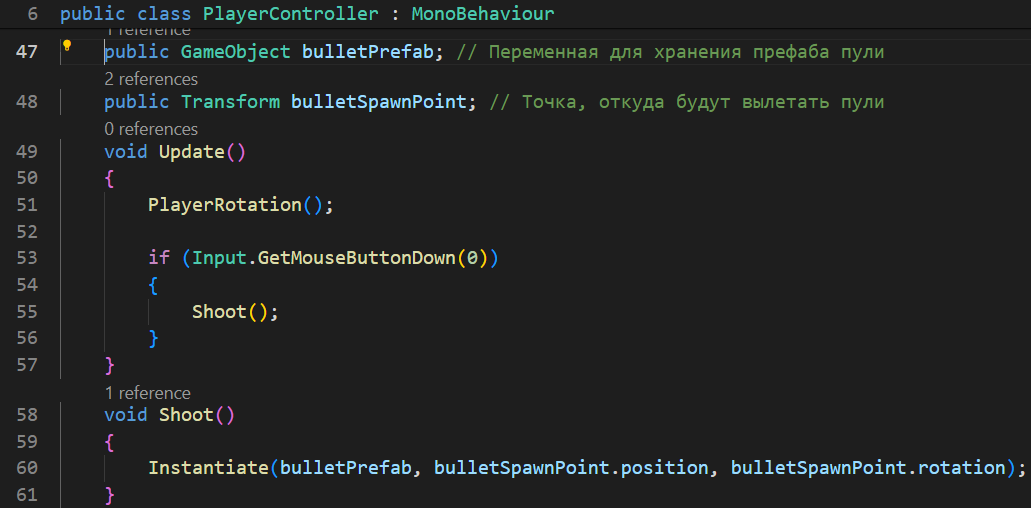


Рис. 36. Класс «PlayerController» с функцией «Shoot».

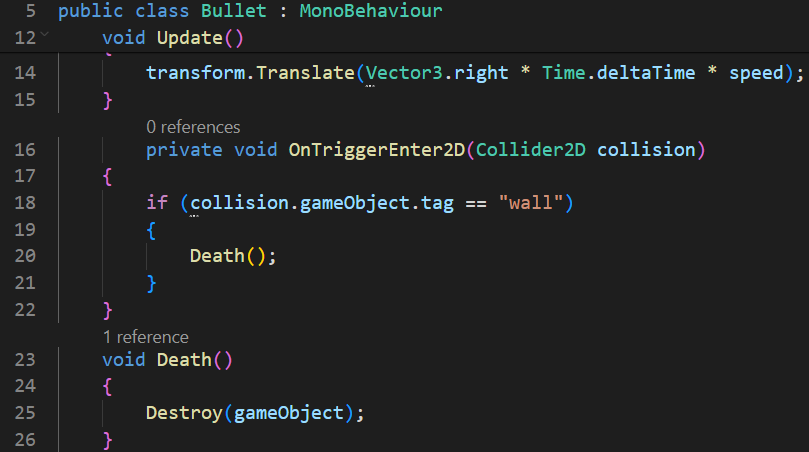


Рис. 37. Класс «Bullet» с функцией «Death/Destroy».

Шаг 1: Подготовка префабов.

1. В Unity мы создали префабы для игрока и пули. Префаб игрока содержит компоненты, необходимые для его управления, например, «Rigidbody2D» для физики, а также «SpriteRenderer» и «Collider2D» для визуального отображения и обнаружения столкновений.
2. Создали префаб пули, включив в него компоненты, такие как «Rigidbody2D» для обеспечения физического движения и «Collider2D» для обнаружения столкновений.

Шаг 2: Написание кода.

1. Создали скрипт «PlayerController», который отвечает за управление игроком и стрельбой.
2. В этом скрипте мы определили переменные «bulletPrefab» и «bulletSpawnPoint». Первая хранит префаб пули, а вторая - точку, откуда будут вылетать пули.
3. В методе «Update» реализовали проверку нажатия кнопки мыши (Input.GetMouseButtonDown(0)) и вызов функции «Shoot», если нажата левая кнопка мыши.
4. Функция «Shoot» создает новую пулю, используя «Instantiate», и устанавливает ее позицию и поворот на основе точки спавна «bulletSpawnPoint».
5. Создали скрипт «Bullet», который отвечает за движение и поведение пули.
6. В скрипте «Bullet» определили переменную «speed», которая отвечает за скорость движения пули.
7. В методе «Update» используем «transform.Translate» для перемещения пули вправо с учетом ее скорости.
8. Добавили метод OnTriggerEnter2D, который вызывается при столкновении пули с другим коллайдером. Если столкнулась с объектом с тегом «wall», вызывается функция «Death», которая уничтожает пулю с помощью Destroy(gameObject).

Шаг 3: Настройка в Unity.

1. Привязали скрипты «PlayerController» и «Bullet» к соответствующим объектам в Unity (игроку и префабу пули).
2. Установили значение переменной «bulletPrefab» в скрипте «PlayerController», присвоив ей префаб пули через редактор Unity.
3. Установили значение переменной «bulletSpawnPoint», указав точку спавна пули через редактор Unity.
4. Указали скорость движения пули в переменной «speed» на объекте пули в редакторе Unity.

Этот код и настройки в Unity обеспечивают базовый механизм стрельбы игрока и движение пули в вашей игре. Он легко настраиваем и может быть дополнен другими механиками, такими как управление врагами и обработка столкновений.

Так же мы добавили код для уничтожения наших пуль с задержкой по времени чтобы игровой объект "жил" определенное время после своего создания и не создавал дополнительных нагрузок на компьютер, а затем был уничтожен (Рис. 38).

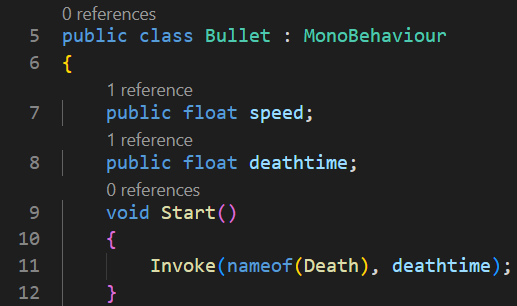


Рис. 38. Добавление времени жизни пули.

«public float deathtime;» - объявляем публичную переменную deathtime типа float. Поскольку переменная объявлена как публичная, вы можете установить её значение в инспекторе Unity. Это значение определяет, через сколько времени (в секундах) после начала работы скрипта должна быть вызвана функция Death.

Invoke(nameof(Death), deathtime); - Этот вызов метода «Invoke» планирует выполнение метода «Death» через количество секунд, указанное в переменной «deathtime».

Метод «Invoke» принимает два параметра:

* «nameof(Death)» - это строковое имя метода, который нужно вызвать. Использование «nameof» помогает избежать ошибок, связанных с неправильным написанием имени функции, поскольку оно обеспечивает получение имени метода напрямую из кода, что уменьшает вероятность опечаток.
* «deathtime» - время в секундах, после которого метод должен быть вызван.

Этот подход часто используется для создания задержек в выполнении функций. Это особенно полезно в случаях, когда нужно управлять временем жизни временных объектов, таких как эффекты, анимации или враги.

Еще мы добавили функциональность задержки между выстрелами в игровой механике стрельбы для игрока, что позволяет контролировать частоту стрельбы и предотвращает непрерывный спам пулями. Давайте подробнее рассмотрим реализацию этой функции (Рис. 39).

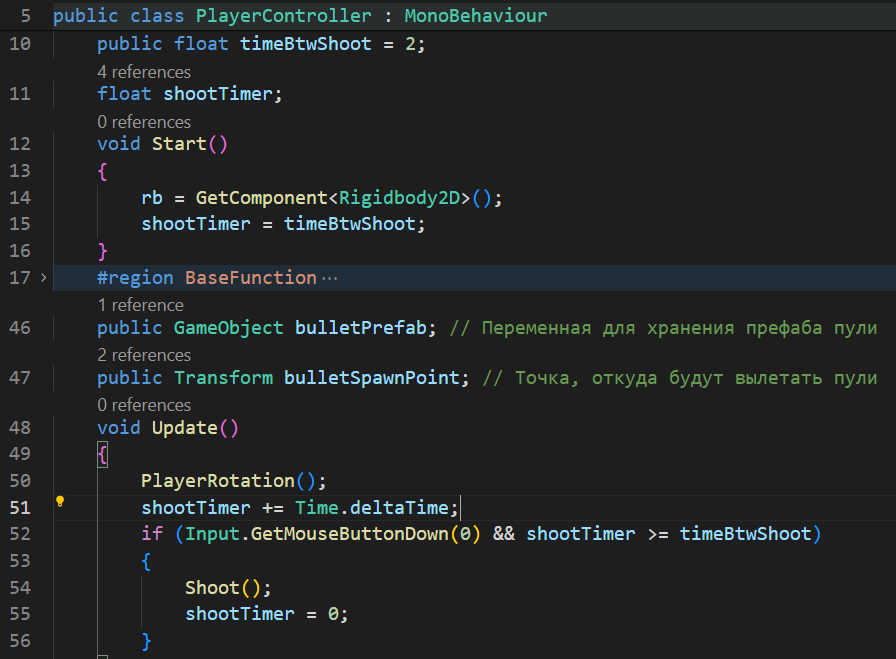


Рис. 39. Добавление задержки после выстрела.

«timeBtwShoot»: Это публичная переменная, которая определяет время в секундах между выстрелами. Её значение можно настроить в инспекторе Unity, что обеспечивает гибкость в балансировке игры.

«shootTimer»: Приватная переменная, которая служит таймером, отслеживающим время, прошедшее с последнего выстрела.

В методе «Start()», помимо инициализации компонента Rigidbody2D, также задается начальное значение «shootTimer» равным «timeBtwShoot», что означает, что первый выстрел может быть произведен сразу после начала игры.

shootTimer += Time.deltaTime;: Эта строка постоянно увеличивает shootTimer на время, прошедшее с последнего кадра, обеспечивая точный отсчет времени.

Условие для стрельбы: Чтобы произвести выстрел, необходимо выполнение двух условий: нажатие левой кнопки мыши (Input.GetMouseButtonDown(0)) и то, что таймер «shootTimer» достиг или превысил значение «timeBtwShoot». Это гарантирует, что между выстрелами будет задержка в соответствии с установленным интервалом.

Добавление задержки между выстрелами — важный элемент для создания сбалансированной и интересной игровой механики. Это не только предотвращает непрерывную стрельбу, что может быть нереалистичным или слишком легким для игры, но и добавляет стратегический элемент, поскольку игроку нужно теперь думать, когда и как лучше стрелять. Эта функция также может быть дополнена различными улучшениями, такими как различные типы оружия с разными интервалами стрельбы, что добавит глубины геймплею.

**2.3 Следование камеры за игроком**

Скрипт «CameraFollow» для Unity предназначен для управления позицией камеры так, чтобы она следовала за игровым объектом (обычно игроком), но при этом оставалась в пределах заданных границ. Код содержит несколько настраиваемых публичных полей, которые позволяют точно настроить поведение камеры в редакторе Unity (Рис. 40).

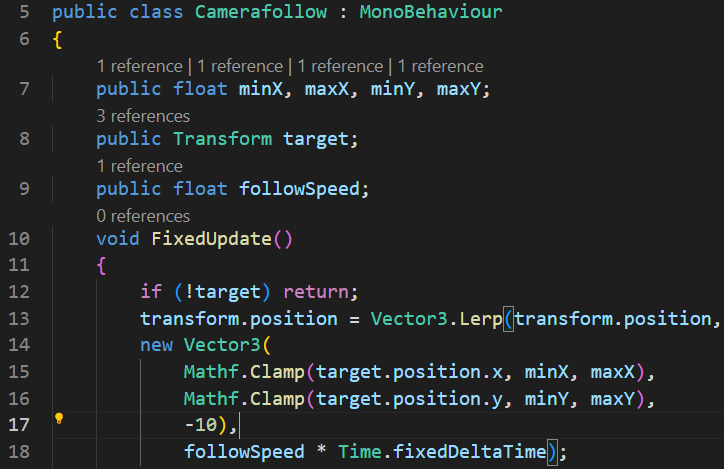


Рис. 40. Код для перемещения камеры за игроком.

1. Поля класса:

«minX», «maxX», «minY», «maxY»: Эти переменные определяют минимальные и максимальные значения координат X и Y, в которых может находиться камера. Это позволяет ограничить перемещение камеры определенной областью игрового мира.

«target»: Объект «Transform», за которым должна следовать камера. Обычно это игрок или другой важный объект сцены.

«followSpeed»: Скорость, с которой камера следует за целью. Большее значение приведет к более быстрому реагированию камеры на перемещения цели.

1. Метод «FixedUpdate»:

Проверяется наличие цели (target). Если цель не установлена, метод прерывается командой «return».

Vector3.Lerp используется для плавного перемещения камеры от текущей позиции к новой позиции, которая вычисляется с учетом границ (minX, maxX, minY, maxY) и заданной глубины камеры (-10 для Z).

Mathf.Clamp используется для ограничения координат X и Y цели внутри заданных границ. Это гарантирует, что камера не выйдет за пределы разрешенной области.

Настройка в Unity. Для работы скрипта его нужно присоединить к объекту камеры в вашей сцене Unity. После этого:

* В инспекторе Unity для объекта камеры установите значения «minX», «maxX», «minY», «maxY» для определения границ, в которых может двигаться камера.
* Установите объект «target», за которым должна следовать камера, перетащив нужный объект на поле «target» в инспекторе.
* Задайте «followSpeed», чтобы контролировать, насколько быстро камера будет реагировать на перемещение цели.

Этот скрипт идеально подходит для игр, где необходимо, чтобы камера следовала за игроком, но при этом не выходила за пределы заданной игровой зоны.

## **2.4 Создание уровня, ассеты**

Создание уровней в видеоиграх — это захватывающий процесс, который включает в себя не только техническое исполнение, но и большую долю креативности. Для начала работы над уровнями с использованием графических ассетов в Unity вам понадобится ряд ресурсов и инструментов. Вот несколько советов и рекомендаций по этапам создания уровней и источникам, где можно получить качественные ассеты.

1. Планирование Уровня.

Перед тем как начать работу с ассетами, важно спланировать структуру уровня:

* Концепт и дизайн: Определите тему и стиль вашего уровня. Это может быть фэнтези, киберпанк, постапокалипсис и т.д.
* Макет уровня: Создайте макет уровня на бумаге или в любом графическом редакторе. Это поможет вам визуализировать расположение ключевых объектов и путей движения игрока.

2. Поиск и Подбор Графических Ассетов

Выбор правильных ассетов важен для создания визуального стиля и атмосферы игры. Вот несколько рекомендуемых источников:

* «Unity Asset Store»: Одно из самых популярных мест для поиска ассетов. Здесь вы найдете бесплатные и платные ассеты, включая текстуры, модели, анимации и готовые киты для создания уровней.
* «Kenney.nl»: Предлагает большое количество бесплатных 2D и 3D ассетов, которые отлично подходят для прототипирования и использования в финальных проектах.
* «OpenGameArt»: Платформа для обмена бесплатными графическими ассетами, где можно найти работы различных художников, подходящие для разных видов игр.
* Itch.io: Помимо того, что Itch.io известен как платформа для инди-игр, здесь также можно найти ассеты, предложенные сообществом.

3. Работа с Ассетами в Unity

После того, как мы выбрали и загрузили необходимые ассеты, пора начать работу над уровнем в Unity:

Импорт ассетов: Перетащите ассеты в папку «Assets» вашего проекта Unity.

* Создание префабов: Соберите объекты уровня в префабы для удобства использования и повторного использования.
* Сценография: Разместите объекты на сцене, используя ранее созданный макет в качестве руководства.
* Освещение и эффекты: Добавьте освещение, тени и другие визуальные эффекты для создания атмосферы.
* Оптимизация: Необходимо убедиться, что ваш уровень хорошо оптимизирован для целевых платформ. Используйте инструменты профилирования Unity для анализа производительности.

4. Тестирование Уровня

Регулярно тестируйте уровень, играя на нем. Это поможет вам выявить любые проблемы с дизайном или производительностью и убедиться, что уровень интересен и приятен для игры.

Создание уровней — это комплексный процесс, требующий творческого подхода и технических знаний. Использование качественных ассетов и эффективных инструментов может значительно упростить этот процесс и помочь создать увлекательные и запоминающиеся игровые уровни (Рис. 41).

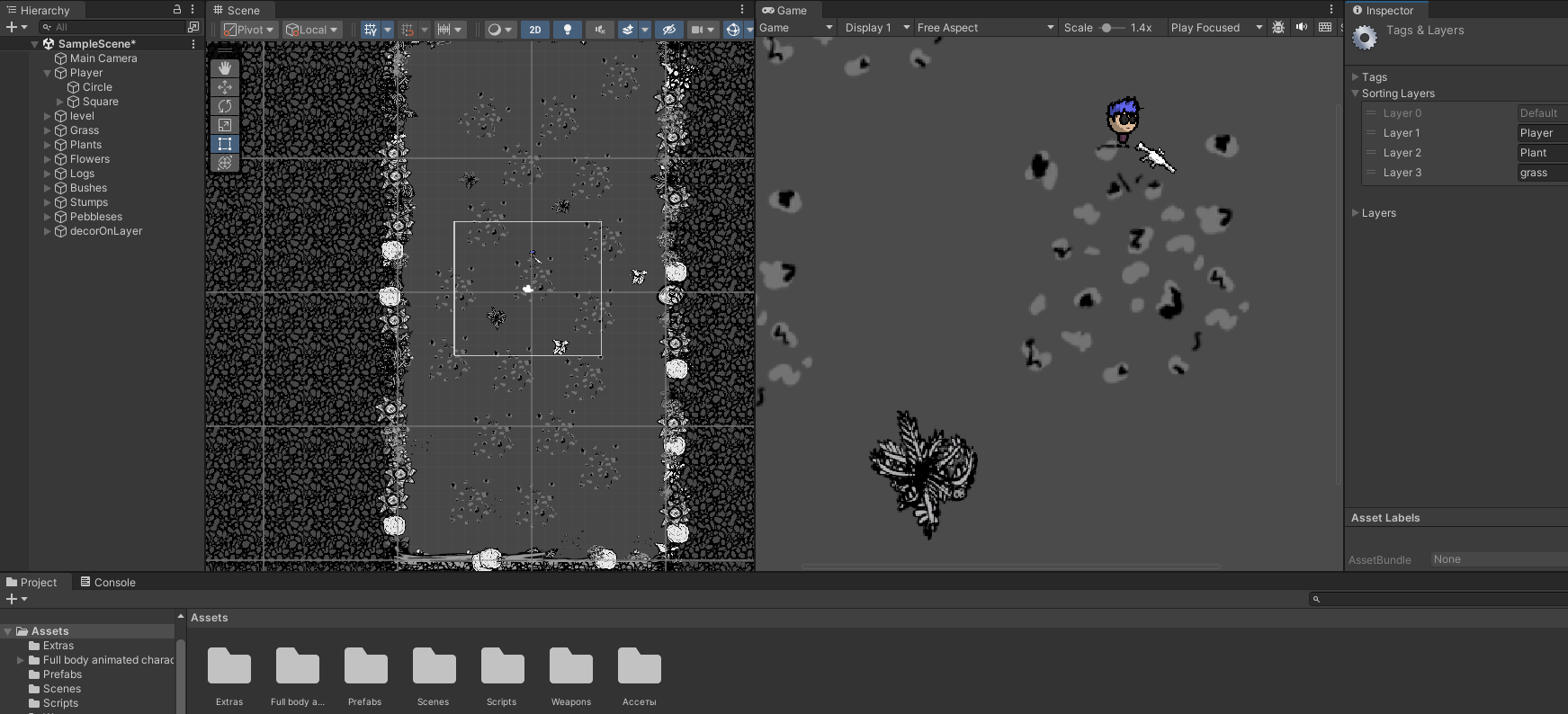


Рис. 41. Наш текущий прогресс в разработке уровня для игры.

Обзор Развития Уровня

Разработка уровня включала ряд ключевых этапов, начиная от дизайна и расстановки ассетов до настройки управления камерой и механики персонажа:

1. Дизайн и Размещение Ассетов:

* Ограниченное Пространство: Уровень ограничен специально размещенным бокс коллайдером, который задает границы для перемещения игрока.
* Ассеты Уровня: Пространство уровня активно заполнено различными ассетами, такими как листья, камни, деревья и кусты, что создает живую и убедительную игровую среду.

1. Настройка Персонажа и Оружия:

* Модели Персонажа и Оружия: На уровень добавлены модельки персонажа, его оружия и пули. Эти элементы были тщательно настроены для корректного взаимодействия.
* Функция Поворота Оружия: Для обеспечения реалистичного поведения оружия, функция поворота была перенесена в отдельный скрипт Weapon, который теперь управляет вращением оружия вокруг игрока. Оружие реагирует на движения мыши, плавно поворачиваясь к курсору с заданной скоростью вращения.

1. Камера и Управление Коллизиями:

* Настройка Камеры: Камера настроена таким образом, чтобы оставаться в пределах заданных границ уровня, что предотвращает отображение внешних или недоработанных участков сцены.
* Коллизии: Внедрена система коллизий с использованием бокс коллайдера, которая позволяет персонажу аккуратно сталкиваться с границами без лишнего "перекручивания" или других нежелательных артефактов движения.

1. Организация Ассетов:

* Сортировка Ассетов: Все элементы на уровне были тщательно отсортированы по видам для удобства управления и возможности быстрого доступа и модификации.

Все реализованные настройки и доработки уровня направлены на создание увлекательной и функциональной игровой среды, обеспечивая плавное управление персонажем и его взаимодействие с объектами на уровне. Продуманное размещение интерактивных и статичных объектов, а также эффективное управление камерой способствуют созданию глубокого и погружающего игрового опыта.

## **2.5 Создание анимации игрока**

Для начала создаем «Animation Controller» в директории проекта, добавляем компонент Animator к объекту игрока, и открываем окна Animator и Animation в Unity. В окне «Animation» кликаем на кнопку «Create», давая новой анимации название «PlayerIdleAnimation». После этого новый объект анимации появится в окне Animator и в директории проекта под названием «PlayerIdleAnimation» (Рис. 42).

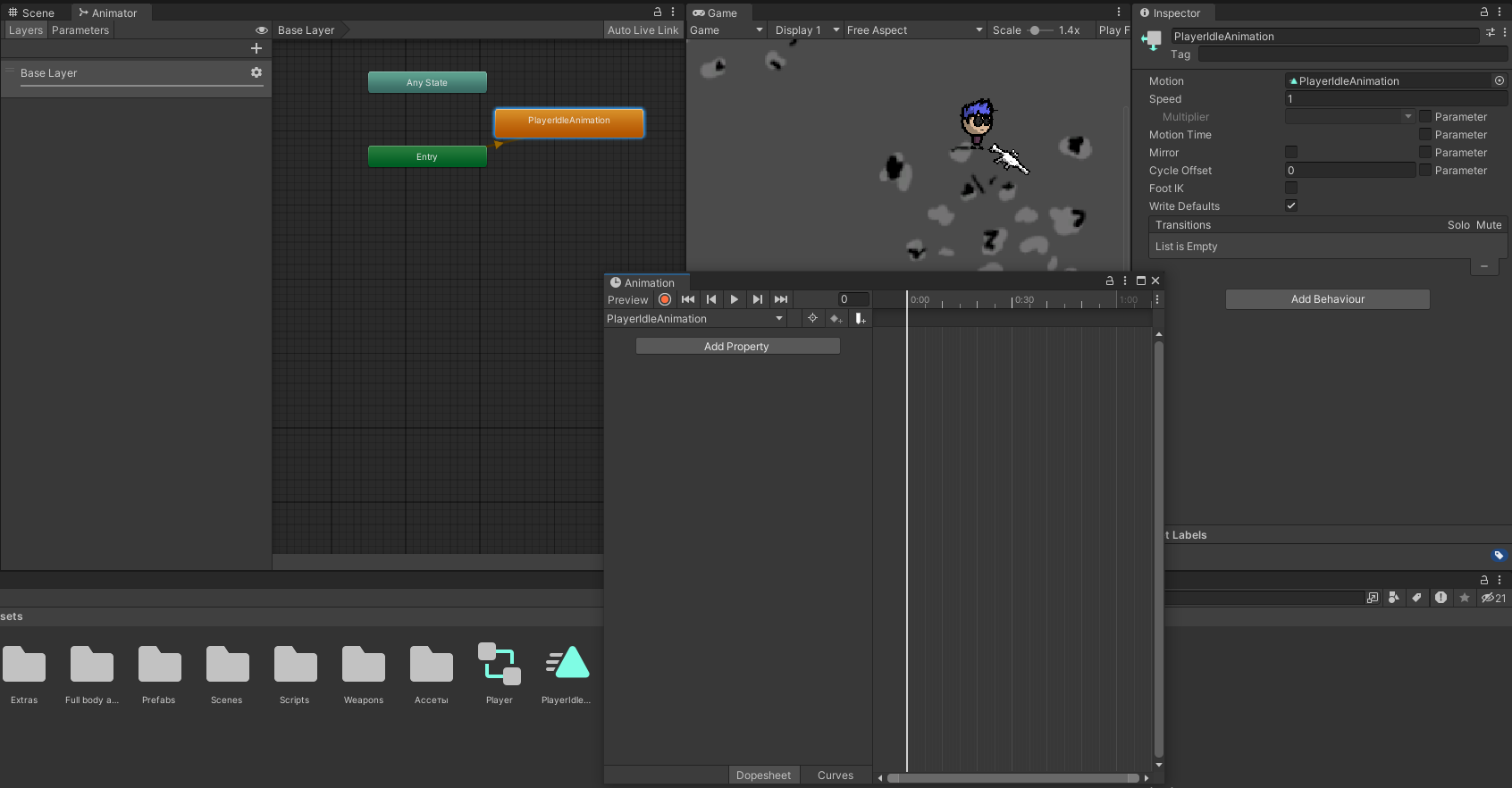


Рис. 42. Окно Unity после добавления анимации.

Затем добавляем еще одну анимацию — «PlayerWalkAnimation», которая также отображается в окне Animator (Рис. 43). Теперь необходимо настроить переход между анимациями: от «PlayerIdleAnimation», когда персонаж стоит, до «PlayerWalkAnimation», когда игрок начинает двигаться. Для этого в скрипт игрока добавляем код, который будет переключать эти два состояния (Рис. 45). В Animator создаем булевый параметр для управления этими состояниями. Правой кнопкой мыши кликаем на «PlayerIdleAnimation», выбираем опцию «Make Transition», создаем стрелку от данного состояния к «PlayerWalkAnimation» и обратно (Рис. 44). В инспекторе, в настройках перехода, устанавливаем «Transition Duration» в ноль, чтобы переход между состояниями происходил мгновенно.

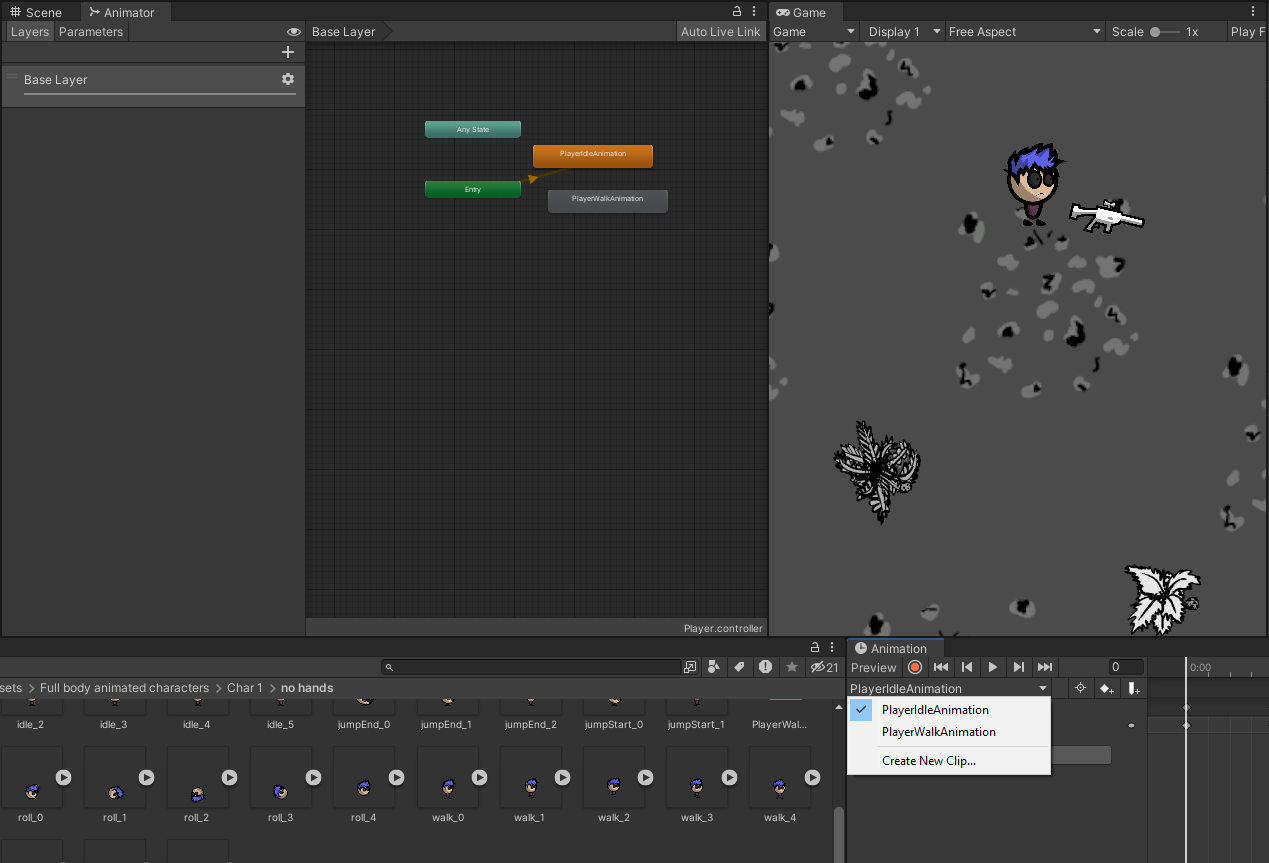


Рис. 43. Добавление новой анимации «PlayerWalkAnimation».

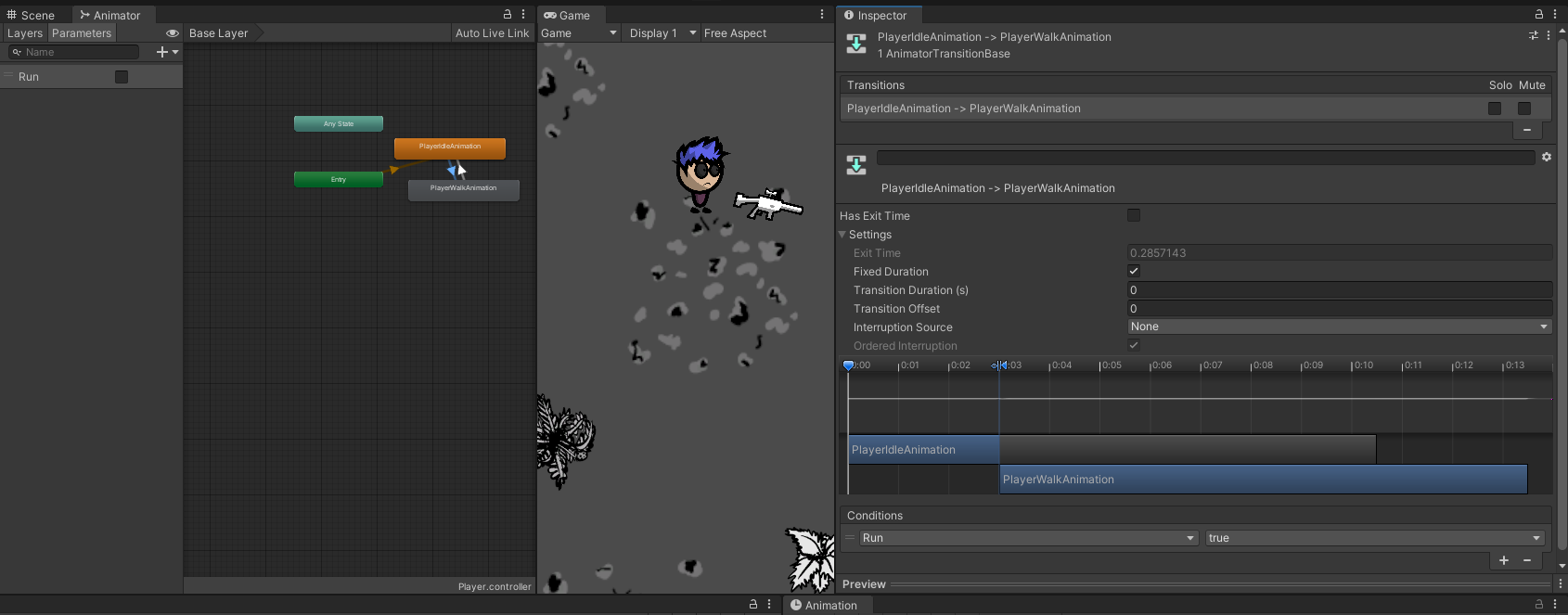


Рис. 44. Настройка перехода между двумя состояниями.

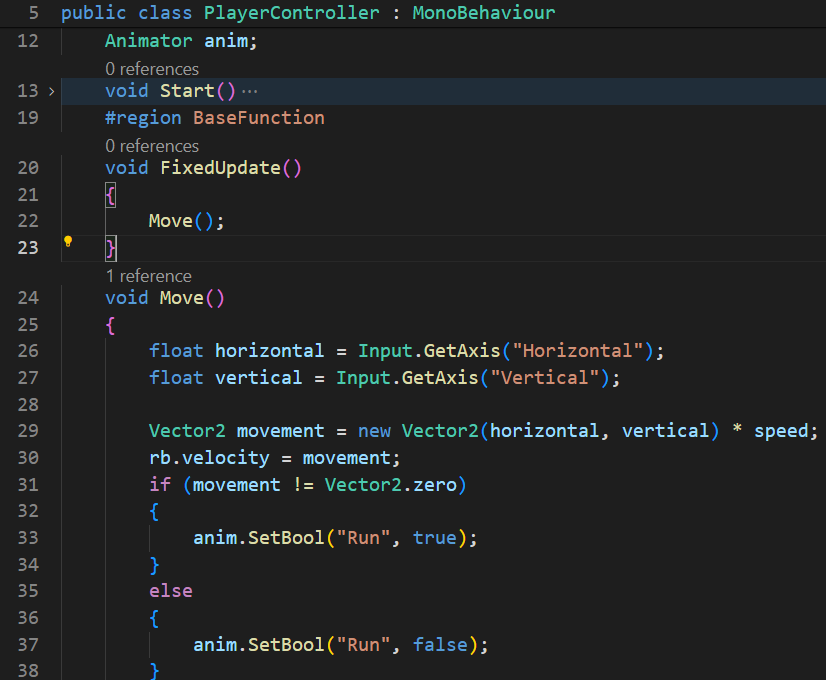


Рис. 45. Код перехода анимации.

Давайте более детально рассмотрим, как работает код:

Получение ввода пользователя: Используется Input.GetAxis для получения ввода от пользователя по горизонтали и вертикали. Это обычный и эффективный способ управления движением в Unity.

Применение движения: Создаем вектор «movement», который умножает полученные значения на скорость «speed», и применяете этот вектор к «velocity» компонента «Rigidbody2D». Это обеспечивает физически основанное перемещение персонажа.

Управление анимацией: Используем условие для проверки, движется ли персонаж (movement != Vector2.zero). Если персонаж движется, то активируется анимация бега, в противном случае — останавливается. Это делается через установку булевого параметра «Run» в Animator.



Рис. 46. Поворот игрока.

В этом фрагменте кода мы реализовали функцию ScalePlayer, которая управляет направлением, в котором смотрит спрайт персонажа, в зависимости от направления его движения. Функция принимает один параметр: x, который представляет горизонтальное движение персонажа (его скорость по оси X).

Детали реализации:

* ScalePlayer(movement.x);: Этот вызов функции передаёт текущее горизонтальное движение персонажа в функцию «ScalePlayer». Значение movement.x берётся из вектора движения, который обычно определяется через пользовательский ввод или физические расчёты движения персонажа.
* void ScalePlayer(float x): Определение функции, которая управляет ориентацией спрайта. Функция проверяет знак переменной x:
* Если x > 0, это означает, что персонаж движется вправо. В этом случае свойство flipX компонента SpriteRenderer устанавливается в false, чтобы спрайт смотрел вправо.
* Если x < 0, персонаж движется влево. Свойство flipX устанавливается в true, заставляя спрайт смотреть влево.

## **2.6 Пользовательский интерфейс (UI)**

Создание пользовательского интерфейса (UI) в Unity включает в себя несколько ключевых аспектов, таких как работа с канвасом («Canvas»), UI элементами (кнопки, текст, изображения и т.д.) и управление взаимодействием пользователя. Unity предоставляет мощные и гибкие инструменты через систему UI, которая позволяет создавать интерактивные и визуально привлекательные интерфейсы.

Основы создания UI в Unity

1. Создание «Canvas»

«Canvas» является основой для всех UI элементов. Это контейнер, в котором размещаются все UI компоненты.

Создание «Canvas»: В Unity перейдите в GameObject > UI > Canvas. Это создаст новый канвас, который будет служить основой для UI элементов. «Canvas» имеет несколько режимов отображения (Render Modes), таких как «Screen Space» и «World Space», которые определяют, как и где элементы будут отображаться в игре.

2. Добавление UI элементов

На «Canvas» можно размещать различные элементы, такие как кнопки, текст, слайдеры и т.д.

Добавление элементов: Чтобы добавить элемент, выберите Canvas и перейдите в GameObject > UI и выберите нужный тип элемента, например, Button или Text.

3. Настройка элементов

Каждый UI элемент имеет свои настройки, которые можно изменять в инспекторе. Например, для текста можно настроить шрифт, размер, цвет, выравнивание и т.д.

«RectTransform»: У каждого UI элемента есть компонент «RectTransform», который управляет размером, позицией и ориентацией элемента на «Canvas». Это ключевой инструмент для позиционирования UI.

4. Работа с событиями

UI элементы могут взаимодействовать с пользователем через систему событий.

«Event System»: Для работы событий в UI необходим «Event System». Обычно он добавляется автоматически при создании первого «Canvas», но если его нет, добавьте его через GameObject > UI > Event System.

Настройка событий: Вы можете настроить действия, которые будут выполняться при нажатии кнопок, изменении слайдеров и других взаимодействиях пользователя. Это делается через компонент, например, Button, в разделе On Click() в инспекторе, где можно добавить функции, которые будут вызваны при активации события.

Мы установим режим отображения канваса (Render mode) в значение «Screen Space – Camera». Для этого перетащим нашу камеру в поле «Render Camera» настроек канваса. После этой настройки канвас будет выровнен по границам камеры и будет следовать за ней во время игры (Рис. 47).

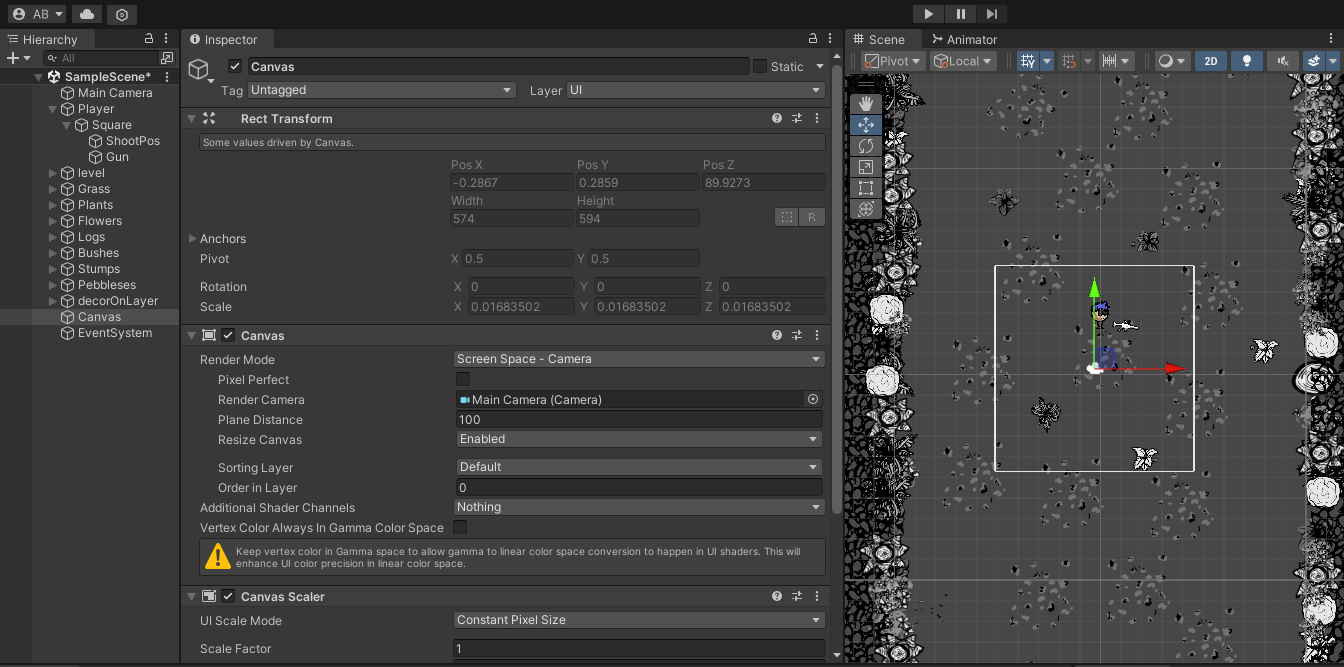


Рис. 47. Настройка Canvas.

Давайте добавим скачанный шрифт для текста (Font Asset Style), чтобы текст в игре выглядел более атмосферно. Сначала добавьте шрифт в папку с игрой. Затем перейдите в меню Window > TextMeshPro > Font Asset Creator. В открывшемся окне перенесите шрифт в поле «Source Font File». Задайте «Character Set» как «Unicode Range (Hex)». Введите в поле значение «0000-4FF», чтобы добавить поддержку символов для русского языка. Дополнительные настройки шрифта можно увидеть на Рис. 48.

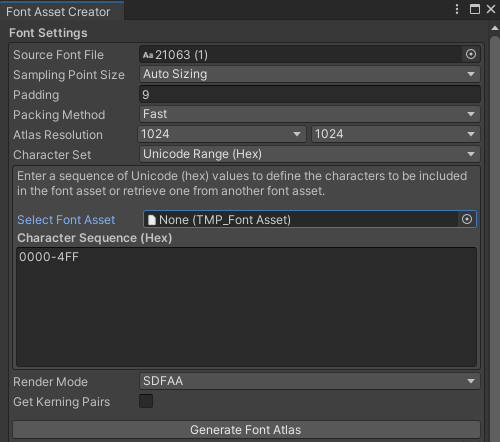


Рис. 48. Настройки создания шрифта.

После настройки нажмите на кнопку «Generate Font Atlas», чтобы создать шрифтовый атлас, и сохраните его в папке с игрой для последующего использования.

## **2.7 Создание врага, его передвижение и анимация**

Опишем процесс создания врага, его передвижения и анимации в игре. Этот процесс включает несколько ключевых этапов, начиная от копирования и адаптации существующих ассетов персонажа до программирования логики поведения и анимации.

1. Копирование и Модификация:

* Начнем с копирования объекта игрока для создания базовой структуры врага.
* Удаляем компоненты, связанные с оружием и другие ненужные элементы, которые не относятся к базовой функциональности врага.
* Добавляем модель врага и настриваем его анимационные клипы, аналогично тем, что используются для игрока.

1. Создание и Настройка Скрипта (Рис. 49):

* Разрабатываем новый скрипт «Enemy», который будет управлять поведением врага.
* Определяем основные параметры врага, такие как «health», «stopdistance», «distanceToRunOut» и «speed», для контроля здоровья, дистанции остановки, дистанции убегания и скорости передвижения соответственно.

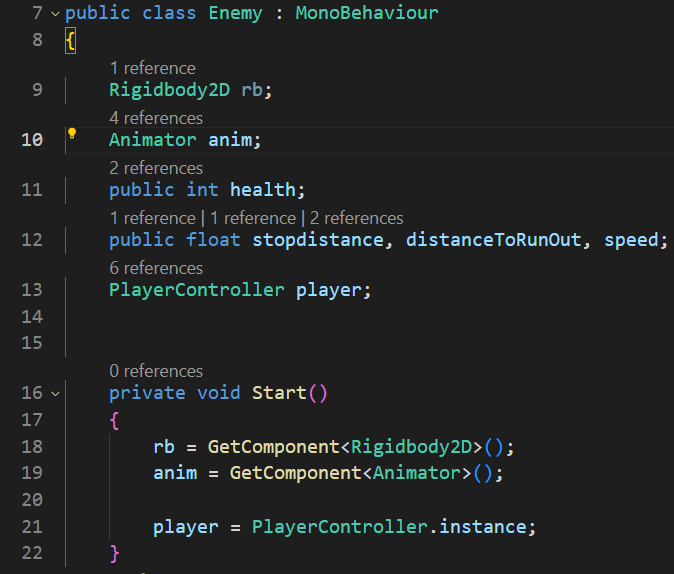


Рис. 49. Определение основных параметров врага.

Логика Передвижения и Анимация

1. Инициализация Компонентов:

* В методе «Start()» инициализируем компоненты «Rigidbody2D» и «Animator».
* Получаем ссылку на контроллер игрока, чтобы использовать его позицию для логики движения врага.

1. Обработка Урона:

* Реализуем метод «damage()», который уменьшает здоровье врага при получении урона и вызывает метод «Death()», если здоровье падает до нуля или ниже.

1. Логика Передвижения:

* В методе «FixedUpdate()» проверяем дистанцию до игрока. Если она больше заданной «stopdistance», враг движется к игроку. Если дистанция меньше «distanceToRunOut», враг отходит от игрока.
* Используем Vector2.MoveTowards для перемещения врага, регулируя его скорость в зависимости от ситуации.
* Контролируем анимацию бега с помощью параметра "Run" в Animator.

1. Поворот Врага:

* В методе «Scale()», враг поворачивается лицом к игроку или от него в зависимости от его текущего положения относительно игрока, используя «Quaternion.Euler» для установки ориентации врага.

1. Обработка Смерти:

* В методе «Death()», уничтожаем объект врага, что удаляет его с игрового поля.

Таким образом, враг в игре настроен на активное взаимодействие с игроком, реагируя на его движения и изменения в окружающей среде. Это включает в себя логическое поведение для преследования или отступления, а также реакцию на получение урона (Рис. 50). Все анимации и поведения врага тщательно синхронизированы для создания вызова и интереса в игровом процессе.

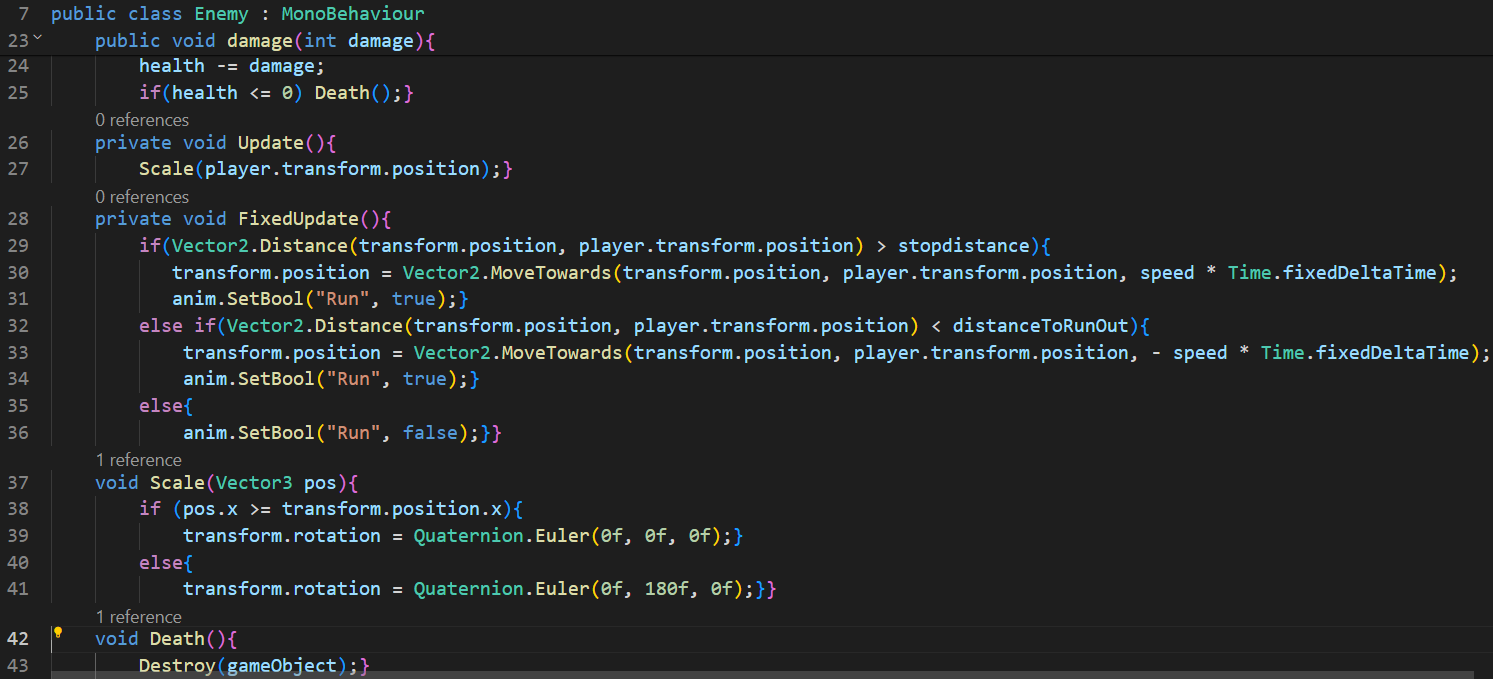


Рис. 50. Код поведения «Enemy».

В разработке игрового врага в Unity, все ключевые переменные, которые определяют его поведение, настроены для контроля через редактор Unity. Это позволяет легко адаптировать и тестировать различные аспекты поведения врага без необходимости изменения кода. Ниже описаны дополнения и детали по настройке и анимации смерти врага.

Настройка переменных через Unity

Все переменные, такие как «health», «stopdistance», «distanceToRunOut», и «speed», объявленные в скрипте «Enemy», экспонированы в инспекторе Unity. Это значит, что вы можете легко изменять эти значения в редакторе Unity, чтобы настроить поведение врага под конкретные требования игрового уровня или сценария без перекомпиляции кода. Для этого:

* Выберите объект врага в иерархии Unity.
* Перейдите в Инспектор, где вы увидите компонент скрипта Enemy.
* Измените значения переменных напрямую в Инспекторе, чтобы они лучше соответствовали вашему игровому дизайну.

Мы также добавили анимацию смерти для врага, которая активируется, когда здоровье врага падает до нуля или ниже. После проигрывания анимации смерти следует исчезновение объекта врага из игры (Рис. 51).

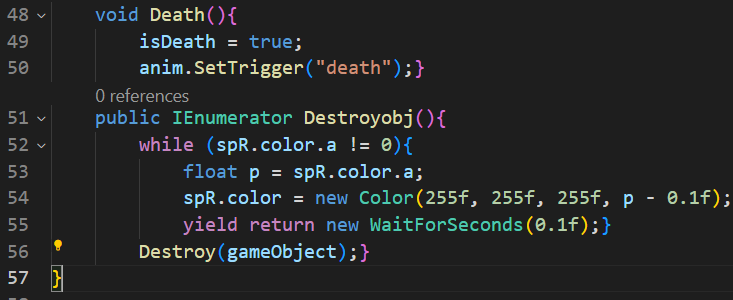


Рис. 51. Анимация смерти «Enemy».

Метод «Death()» вызывает анимацию смерти, используя «SetTrigger("death")» на компоненте Animator. Это переключает состояние аниматора на анимацию смерти в «Animator Controller».

Мы реализовали функцию «Destroyobj» как корутину, что позволяет использовать временные задержки внутри процесса исчезновения. Вот детальное объяснение того, как функция работает:

Определение функции как корутины: Вы объявили «Destroyobj» как «IEnumerator», что позволяет использовать оператор «yield» для создания задержек между изменениями прозрачности спрайта.

В цикле while проверяется текущая прозрачность спрайта врага (spR.color.a), который не должен быть равен нулю для продолжения цикла.

Значение прозрачности уменьшается на каждом шаге цикла. Код spR.color = new Color(255f, 255f, 255f, p - 0.1f); задаёт новый цвет спрайта, уменьшая альфа-канал (p - 0.1f), что делает спрайт более прозрачным.

Задержка между итерациями: yield return new WaitForSeconds(0.1f); вставляет паузу в 0.1 секунды между каждым изменением прозрачности, создавая визуальный эффект затухания.

Уничтожение объекта: После завершения цикла, когда альфа-канал достигает нуля и спрайт становится полностью прозрачным, объект врага уничтожается с помощью «Destroy(gameObject)»;.

Таким образом, мы создали полный жизненный цикл для врага, от реагирования на атаки до визуально заметной смерти и исчезновения, обеспечивая глубокую интеграцию поведения и визуальных эффектов.

## **2.8 Настройка врага с ближней атакой**

Создаем новый скрипт «MeleeEnemy».

Этот реализует механизм атаки ближнего боя для врага в игре. Он расширяет базовый класс «Enemy», добавляя специфическую логику для врагов, способных наносить урон в ближнем бою. Давайте подробно разберем, что делает каждая часть кода (Рис. 52).

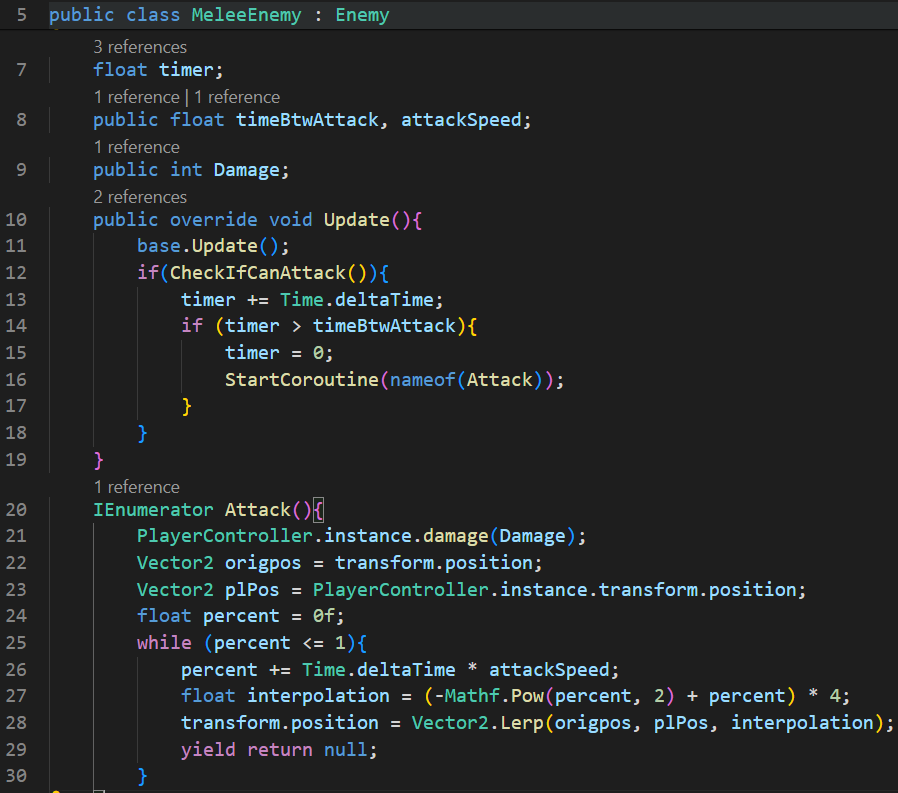


Рис. 52. Код скрипта «MeleeEnemy»

Основные Параметры:

* «timer» - используется для отсчета времени между атаками.
* «timeBtwAttack» - время в секундах между атаками.
* «attackSpeed» - скорость анимации атаки или перемещения к игроку.
* «Damage» - количество урона, которое наносит враг при атаке.

Метод «Update»:

* «base.Update()» - вызывает реализацию «Update» из базового класса «Enemy», что позволяет сохранить общую логику обновления, наследуемую от класса «Enemy».
* «CheckIfCanAttack()» - метод, который должен определять, может ли враг атаковать (не реализован в вашем фрагменте, предположительно проверяет дистанцию до игрока или другие условия).
* Если условие атаки выполняется, таймер увеличивается на время, прошедшее с последнего кадра («Time.deltaTime»).
* Когда «timer» превышает «timeBtwAttack», таймер сбрасывается и начинается корутина «Attack».

Корутина «Attack»:

* Нанесение урона: немедленно вызывается метод «damage» игрока для нанесения урона.
* Движение к игроку: анимированное движение от текущей позиции врага (origpos) к позиции игрока (plPos).
* Используется линейная интерполяция (Vector2.Lerp) с параметром интерполяции, который изменяется по параболической функции для создания более динамичного движения.
* percent увеличивается с каждым кадром, что обеспечивает постепенное изменение позиции.
* Перемещение продолжается, пока percent не достигнет или не превысит 1, после чего корутина завершается.

Добавление Логики Урона Игроку

В коде игрока добавлен метод «damage», который уменьшает здоровье игрока на значение «damage». Если здоровье падает до нуля или ниже, игрок уничтожается («Destroy(gameObject)»), что обрабатывает его "смерть" в игре (Рис. 53).

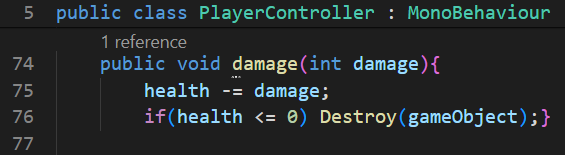


Рис. 53. Получение урона игроком.

Скрипт «MeleeEnemy» эффективно моделирует поведение врага, способного к атаке в ближнем бою. Использование корутин для анимации движения врага при атаке добавляет плавности и реализма. Управление временем между атаками с помощью таймера помогает сделать поведение врага предсказуемым и настраиваемым. Эти элементы вместе формируют основу для дальнейшей настройки и дополнения поведения врагов в вашей игре.

## **2.9 Система частиц, анимация получения урона**

Реализуем получение урона персонажем, которое включает в себя анимацию камеры, визуальные эффекты и тряску камеры для усиления впечатления от удара. Вот подробное описание различных компонентов системы и их взаимодействия.

Когда персонаж получает урон, его здоровье уменьшается на значение «damage». В момент получения урона создаётся визуальный эффект («hiteffect»), который инстанцируется в позиции персонажа. Этот эффект может представлять из себя систему частиц, которая визуализирует удар или ранение (Рис. 54).

Сразу после получения урона вызывается функция «CamShake()» через экземпляр класса «Camerafollow», что инициирует анимацию тряски камеры. Это усиливает визуальное и эмоциональное восприятие удара, делая момент более драматичным.

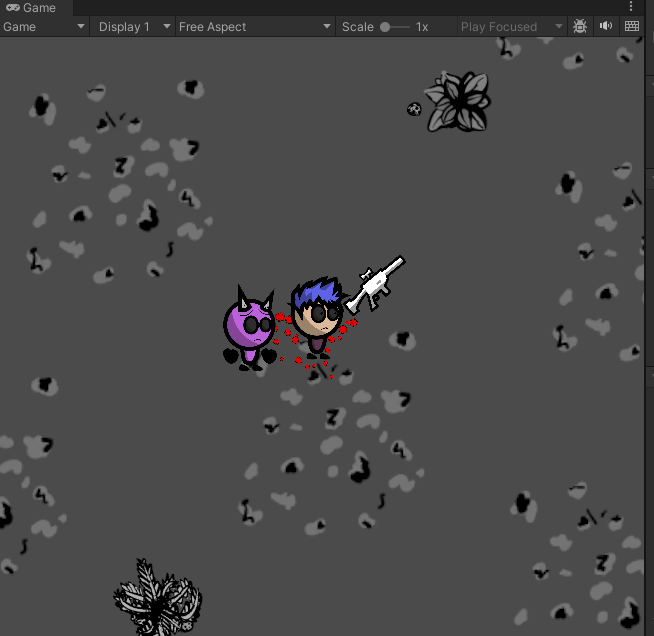


Рис. 54. Появление системы частиц после удара противником.

Анимация тряски камеры:

Аниматор, прикреплённый к камере, содержит анимацию с именем «ShakeCamera», которая активируется функцией «CamShake()». Это создаёт эффект тряски, который временно меняет положение и ориентацию камеры, добавляя динамизм в моменты интенсивных действий (Рис. 55).

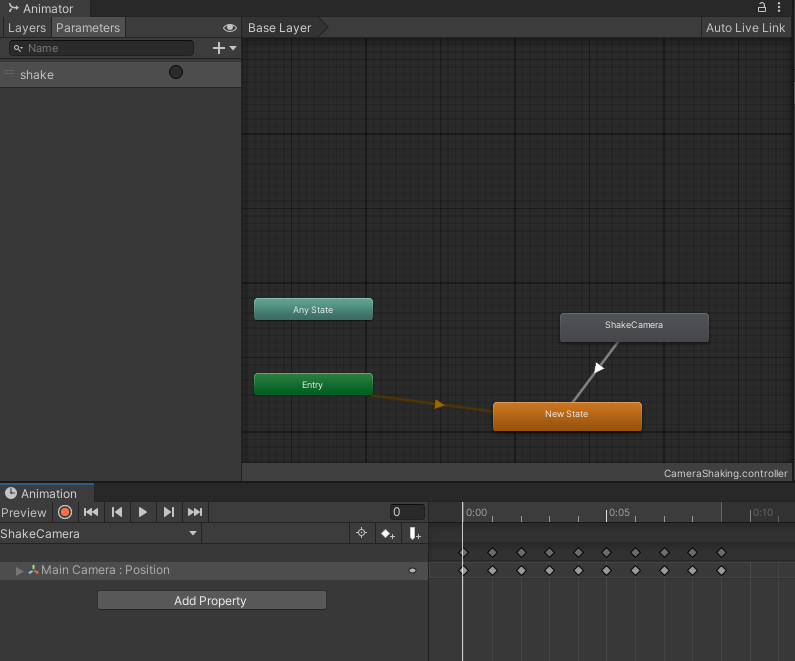


Рис. 55. Кадры тряски камеры в «Animation».

Добавление анимации получения урона врагу аналогично тому, как это реализовано для игрока. Вот как это было реализовано и описано в коде для класса Enemy (Рис. 56).

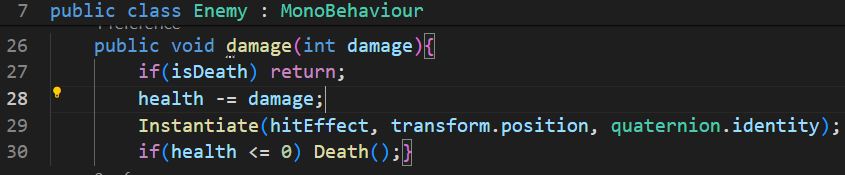


Рис. 56. Анимация получения урона врагом.

Эти эффекты делают взаимодействия в игре более реалистичными и эмоционально насыщенными, значительно улучшая игровой опыт. Визуальные эффекты в момент получения урона, вместе с анимацией тряски камеры, помогают создать ощущение напряженности и опасности боевых действий. Эти механизмы не только повышают визуальное разнообразие сцен, но и делают каждое столкновение более значимым и запоминающимся.

## **2.10 Создание анимации выстрела игрока.**

В данном разделе описывается реализация анимации выстрела для игрока в Unity, используя систему спрайтов для визуализации вспышки от дула оружия (Рис. 57). Это важный аспект создания реалистичных и визуально привлекательных боевых эффектов в играх.

Основные Компоненты

«spritesMuzzFlash»: Массив спрайтов, содержащий различные изображения вспышек от дула, что позволяет добавить вариативность визуальным эффектам выстрела.

«muzzleFlashSpR»: Компонент «SpriteRenderer», который используется для отображения спрайта вспышки. Этот компонент должен быть прикреплен к объекту в сцене, который находится в месте выхода пули из оружия.

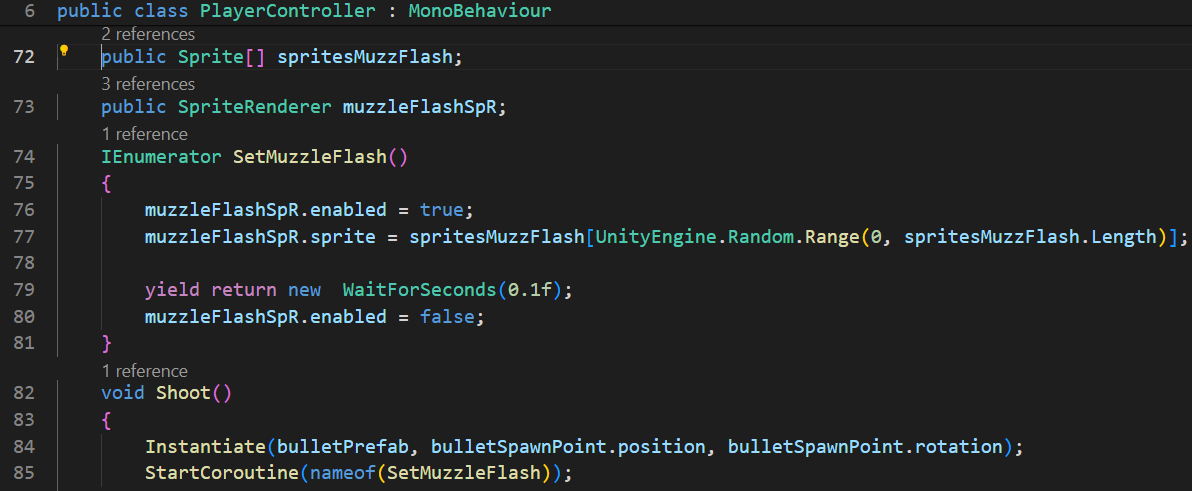


Рис. 57. Добавление анимации выстрела «MuzzleFlash».

Корутина «SetMuzzleFlash» управляет отображением и скрытием вспышки при выстреле:

1. Активация Вспышки:

Включение «SpriteRenderer» (muzzleFlashSpR.enabled = true), делает вспышку видимой.

Спрайт для вспышки выбирается случайным образом из предоставленного массива «spritesMuzzFlash», что добавляет элемент случайности и делает каждый выстрел уникальным.

1. Задержка:

«yield return new WaitForSeconds(0.1f)»: Корутина ожидает 0.1 секунды, в течение которых вспышка видна на экране. Это кратковременное появление создает эффект реального выстрела.

1. Деактивация Вспышки:

После задержки вспышка скрывается (muzzleFlashSpR.enabled = false), возвращая состояние объекта в исходное, до выстрела.

Функция «Shoot» отвечает за логику выстрела:

Создание Пули: «Instantiate» - этот вызов создает экземпляр пули в определенной точке («bulletSpawnPoint»), используя заданное вращение. Пуля будет двигаться в направлении, заданном этим вращением.

Активация Корутины:

«StartCoroutine(nameof(SetMuzzleFlash))»: Непосредственно после создания пули запускается корутина «SetMuzzleFlash», которая отвечает за отображение вспышки от выстрела.

Реализация анимации выстрела с визуальными эффектами вспышки значительно улучшает визуальное восприятие игрового процесса и делает действия игрока более заметными и удовлетворительными. Использование корутин позволяет эффективно управлять временными интервалами отображения вспышки, а выбор случайного спрайта добавляет разнообразие и реалистичность каждому выстрелу.

Так же проекте реализована система для управления сортировкой и распределением врагов на экране, чтобы избежать визуального наложения моделей друг на друга. Эта система особенно важна в играх с большим количеством вражеских единиц, где перекрытие моделей может снизить читаемость действий на экране и ухудшить визуальное восприятие игры. Давайте подробно рассмотрим реализованные функции.

Случайное Распределение Позиций

Корутина «SetRandomPos», которая периодически (каждые 1.5 секунды) назначает врагам небольшое случайное смещение от их текущих позиций. Это помогает разбавить статичность их расположения, делая поведение врагов менее предсказуемым и более естественным (Рис. 58).

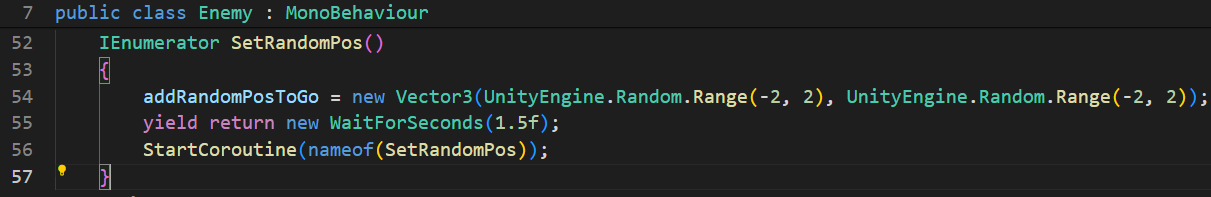


Рис. 58. Код распределения врагов между собой.

Управление Порядком Слойности Врагов

Скрипт «EnemyOrderInLayerManager» управляет порядком отрисовки врагов в зависимости от их вертикального положения на экране, что особенно актуально для 2D и изометрических 3D игр, где важно правильно отображать объекты, расположенные на разной высоте.

Регистрация и Удаление Врагов

Добавление и удаление спрайтов: Враги регистрируются в списке «enemyesSpR» при их создании и удаляются при уничтожении (Рис. 59). Это позволяет динамически управлять списком отслеживаемых объектов.

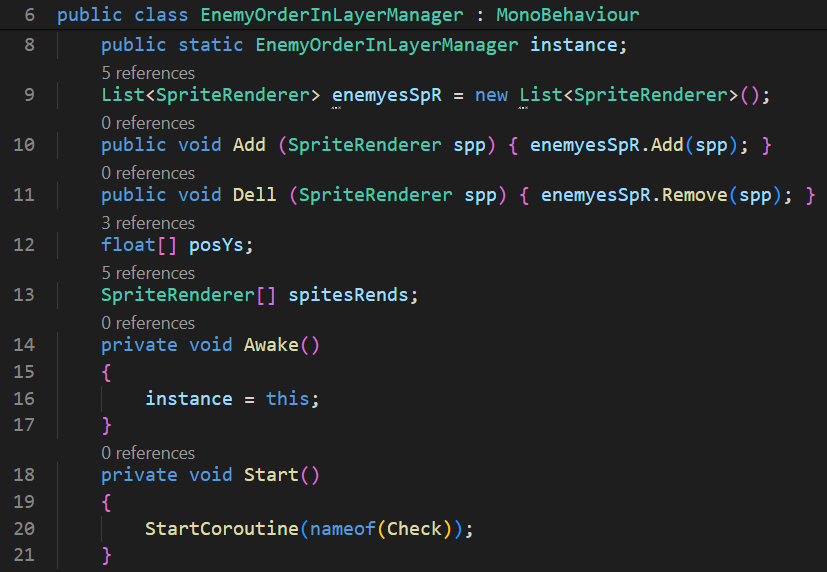


Рис. 59. Начало скрипта «EnemyOrderInLayerManager».

Периодическая Проверка и Сортировка

Корутина «Check»: Эта функция регулярно (раз в секунду) обновляет порядок слоёв врагов, основываясь на их позициях по оси Y.

После сортировки массива по Y-координатам, порядок отрисовки (sortingOrder) каждого врага обновляется так, что более низко расположенные объекты рисуются поверх выше стоящих. Это создаёт иллюзию глубины и объёмности в 2D пространстве (Рис. 60).

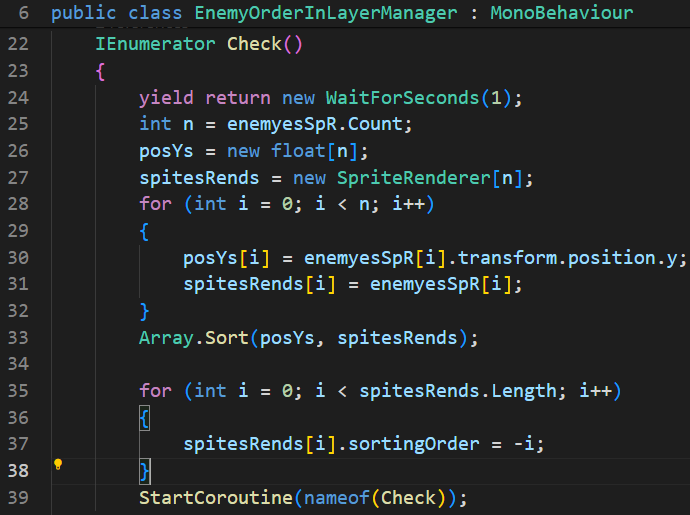


Рис. 60. Основная корутина «Check» скрипта «EnemyOrderInLayerManager».

Такая система управления врагами делает визуальное восприятие игры более организованным и атмосферным, улучшая общее игровое ощущение и помогая избежать визуального хаоса, особенно в массовых сценах боя или в сценах с большим скоплением NPC.

Для полной интеграции системы управления порядком отрисовки врагов на основе их позиции в пространстве игры, важно корректно регистрировать и удалять каждый экземпляр врага в глобальном списке управления. Это обеспечивает актуальность списка врагов в «EnemyOrderInLayerManager» и позволяет правильно обновлять порядок их отображения (Рис. 61).

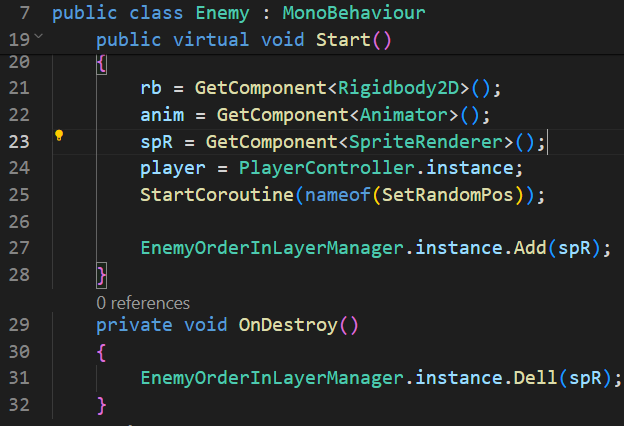


Рис. 61. Добавление и удаление врагов в порядковый список.

С этими изменениями система становится способной эффективно обрабатывать множество экземпляров врагов, управляя их порядком отрисовки на основе их вертикального положения в игровом мире. Это позволяет динамически добавлять и удалять врагов из сцены, сохраняя при этом корректный визуальный порядок их отображения.

Эти дополнения делают игру более оптимизированной для сцен с большим количеством враждебных NPC, гарантируя, что визуальное представление остается чистым и организованным, даже в самых интенсивных боевых сценах.

## **2.11 Оставшиеся враги: стрелок и подрывник**

Создание дальнего противника (Ranged Enemy) в игре представляет собой такой же процесс, как для «MeleeEnemy», который включает в себя настройку врага с возможностью стрелять в игрока издалека. Это требует не только программирования логики атаки, но и настройки компонентов, связанных с позиционированием и выстрелами. Давайте подробно рассмотрим основные аспекты реализации класса «RangedEnemy».

Описание класса «RangedEnemy»

«RangedEnemy» наследуется от базового класса «Enemy», что позволяет использовать общую логику и функциональность, такие как здоровье, получение урона и базовое поведение (Рис. 62).

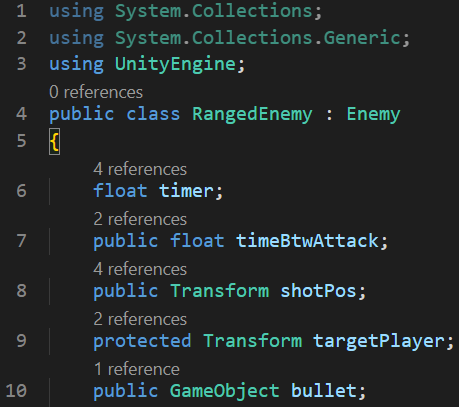


Рис. 62. Инициализация переменных.

Инициализация переменных:

* «timer» – переменная для отслеживания времени между атаками.
* «timeBtwAttack» – время в секундах, которое должно пройти между двумя атаками.
* «shotPos» – трансформ, который определяет позицию, откуда будет производиться выстрел.
* «targetPlayer» – цель врага, обычно это игрок. Задается как позиция игрока в момент начала игры.
* «bullet» – префаб пули, который будет инстанциирован при каждом выстреле.

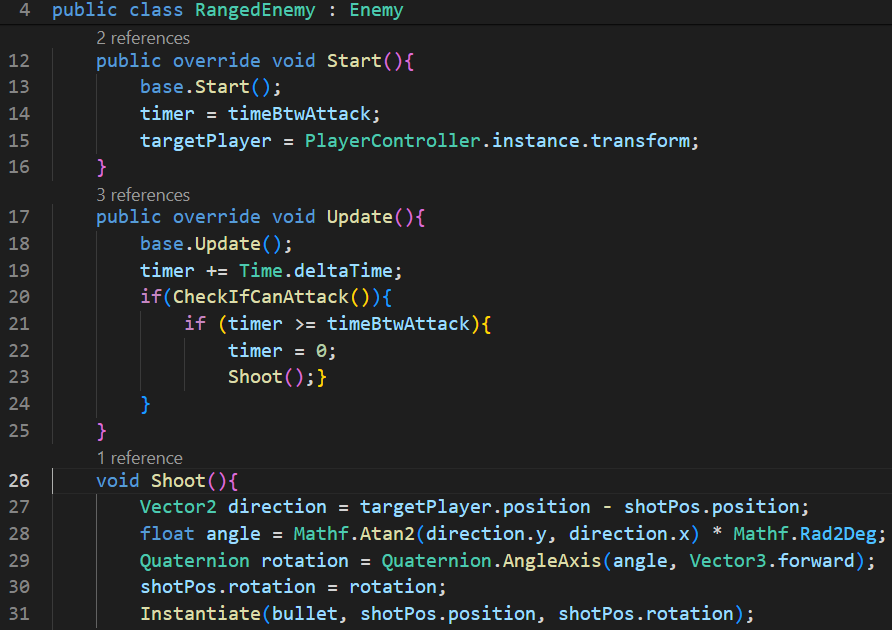


Рис. 63. Методы класса RangedEnemy.

Методы класса

1. «Start»:

* Вызывает «Start» из базового класса для инициализации общих параметров.
* Инициализирует таймер, устанавливая его равным времени между атаками.
* Устанавливает цель (игрока) для атак.

1. «Update»:

* Вызывает «Update» из базового класса для выполнения общих обновлений состояния.
* Увеличивает таймер каждый кадр и проверяет, готов ли враг к новой атаке. Если время между атаками прошло, сбрасывает таймер и вызывает метод «Shoot».

1. «Shoot»:

* Вычисляет направление от позиции выстрела к позиции игрока.
* Рассчитывает угол для корректного направления пули.
* Устанавливает ориентацию «shotPos» в соответствии с вычисленным углом.
* Создает пулю в позиции выстрела, используя угол выстрела для правильной ориентации пули.

Реализация стрельбы

Процесс стрельбы осуществляется путем создания экземпляра префаба пули в заданной позиции («shotPos») и с заданным вращением. Это вращение корректируется таким образом, чтобы пуля летела прямо к цели. Данный подход обеспечивает реалистичное и точное поведение стрельбы, что делает врага более опасным и интересным противником.

Класс RangedEnemy является отличным примером того, как можно расширять функциональность врагов в игре, добавляя им способность атаковать игрока на расстоянии. Такой тип врага требует от игрока большей осторожности и стратегического подхода.

Создание дальнего противника в игре не только включает в себя программирование его поведения и функциональности, но также требует детальной настройки его визуальных и анимационных аспектов. Включение анимации ходьбы, смерти, настройка модельки и корректная конфигурация в Unity — это ключевые этапы, которые делают персонажа полноценным и интерактивным (Рис. 64).

Создание Модели:

Для врага была выбрана модель, которая соответствует его роли в игре. Модель должна визуально отличаться от других врагов, чтобы игрок мог легко идентифицировать угрозу на расстоянии.

Анимация Ходьбы:

Анимация ходьбы имеет ключевое значение для демонстрации движения врага по сцене. Она была создана и настроена для того, чтобы отображать реалистичные движения персонажа при приближении к игроку или перемещении по игровому миру.

Анимация Смерти:

Анимация смерти используется для визуализации момента уничтожения врага. Это важно для поддержания реалистичности и для подтверждения игроку, что враг был успешно побежден.

Настройка в Unity

Импорт и Расстановка Компонентов:

Модель и анимации были импортированы в Unity и присвоены соответствующему объекту врага. Каждая анимация была настроена с использованием Animator Controller, который управляет переключениями между различными анимационными состояниями в зависимости от действий врага.

Настройка Анимационных Состояний:

Были созданы и настроены анимационные состояния для ходьбы, атаки и смерти, с соответствующими переходами, основанными на логике поведения врага (например, переход в состояние смерти при достижении нулевого здоровья).

Тестирование и Отладка:

После настройки, проведено тестирование врага в игровой сцене для убеждения в корректности его анимаций и поведения. Это включает проверку правильности переходов между анимационными состояниями и адекватности реакции на действия игрока.

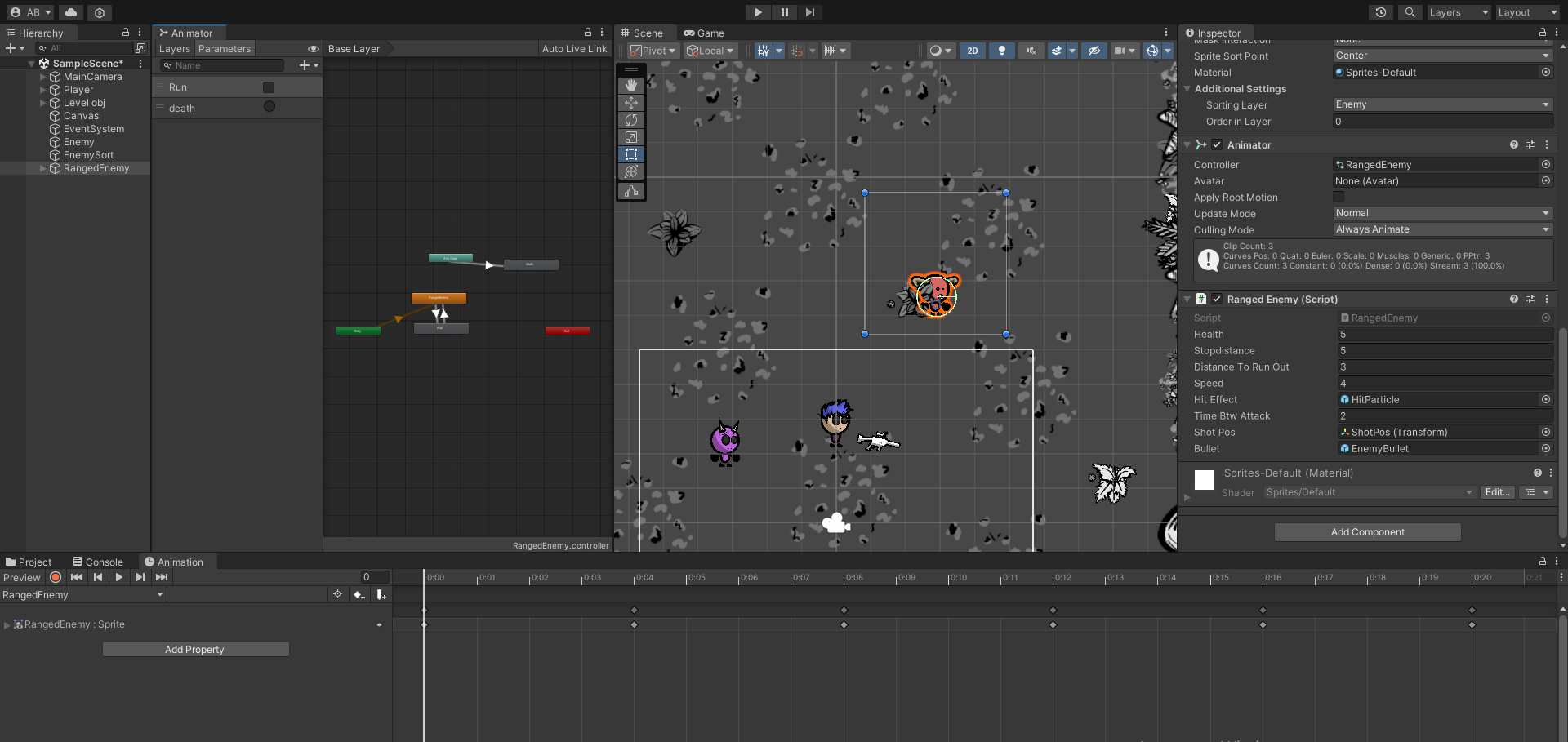


Рис. 64. Окно Unity с настройками анимаций.

Важность Качественной Настройки

Качественная настройка анимаций и моделей врагов улучшает визуальное восприятие игры и углубляет игровой опыт, делая противостояния с врагами более захватывающими и эмоционально насыщенными. Тщательная проработка деталей каждого типа врага способствует созданию уникальной атмосферы и поддерживает общую концепцию дизайна игры.

Еще один новый тип врага, «BoomEnemy», представляет собой уникального персонажа, который использует область взрыва для нанесения урона, что делает его отличным от дальнобойного и ближнего противника. Этот класс обладает специфической механикой атаки, которая активируется при определенных условиях и взаимодействует с игроком через область действия.

Основные Параметры и Компоненты

* «attackRadius»: радиус, в пределах которого враг наносит урон всем персонажам, определяемым через «LayerMask».
* «whatIsPlayer»: слой («LayerMask»), который определяет, какие объекты следует рассматривать как цели для атаки.
* «Damage»: количество урона, которое наносится каждому объекту в радиусе атаки.
* «boomEffect»: Префаб визуального эффекта взрыва, который инстанцируется в момент активации взрыва. Это улучшает визуальное восприятие атаки и делает её более зрелищной.

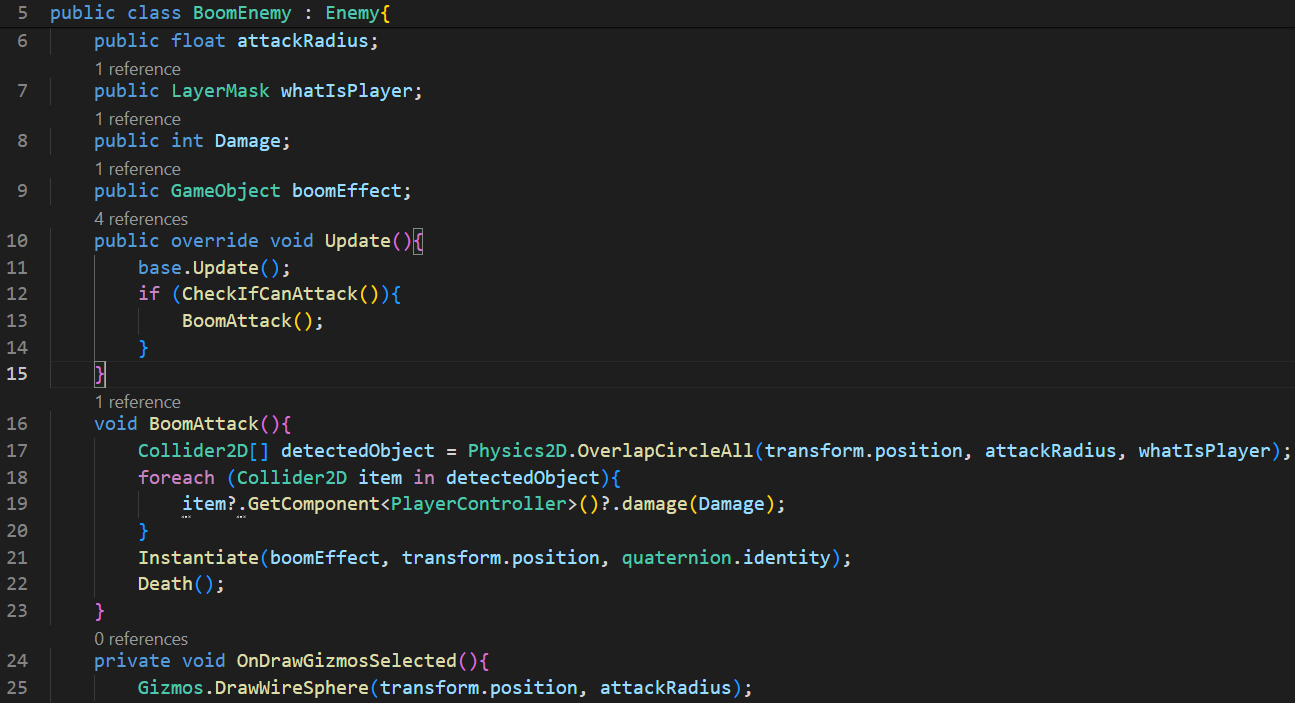


Рис. 65. Код класса «BoomEnemy».

1. «Update»:

* Метод «Update» переопределяется для включения логики проверки возможности атаки. Если условия для атаки выполнены («CheckIfCanAttack()» возвращает true), вызывается метод «BoomAttack()».

1. «BoomAttack»:

* В этом методе используется Physics2D.OverlapCircleAll для обнаружения всех коллайдеров в заданном радиусе «attackRadius» вокруг позиции врага, которые принадлежат слою «whatIsPlayer».
* Для каждого обнаруженного коллайдера вызывается метод «damage()» компонента «PlayerController», если таковой присутствует, что позволяет нанести урон всем игрокам в зоне поражения.
* После выполнения атаки немедленно вызывается метод «Death()», что предполагает самоуничтожение врага после осуществления взрывной атаки.

1. «OnDrawGizmosSelected»:

* Этот метод используется для визуализации радиуса атаки в редакторе Unity, что помогает разработчику визуально оценить область воздействия взрыва. «Gizmos.DrawWireSphere» отображает сферу вокруг врага, обозначая радиус «attackRadius».

Использование Gizmos в методе «OnDrawGizmosSelected» является чрезвычайно полезным для отладки и настройки поведения врага, так как позволяет разработчикам точно настроить радиус атаки и визуально проверить его на сцене перед тестированием игры.

BoomEnemy представляет собой интересный тип вражеского персонажа, который добавляет дополнительные стратегические и тактические аспекты к игровому процессу. Способность наносить урон в радиусе и последующее самоуничтожение делают этого врага уникальной угрозой, требующей от игроков особого внимания и стратегического планирования.

## **2.12 Рывок, полоса здоровья игрока, генератор волн врагов**

В игре реализована механика рывка (Dash), которая добавляет динамичности и стратегического разнообразия геймплею, позволяя игроку быстро перемещаться или уклоняться от атак. Давайте подробно рассмотрим реализацию этой функции в вашем коде.

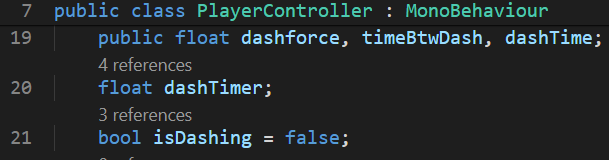


Рис. 66 Новые переменные для реализации рывка.

Концепция и переменные (Рис. 66)

* «dashforce» - Сила, с которой игрок будет "рвануться" вперед. Это значение определяет, насколько мощным будет рывок.
* «timeBtwDash» - Время в секундах между возможностью использовать рывок. Это предотвращает постоянное использование рывка, добавляя тактический элемент управления способностями.
* «dashTime» - Продолжительность рывка.
* «dashTimer» - Таймер, отсчитывающий время до следующего возможного рывка.
* «isDashing» - Булевая переменная, указывающая, находится ли игрок в состоянии рывка.

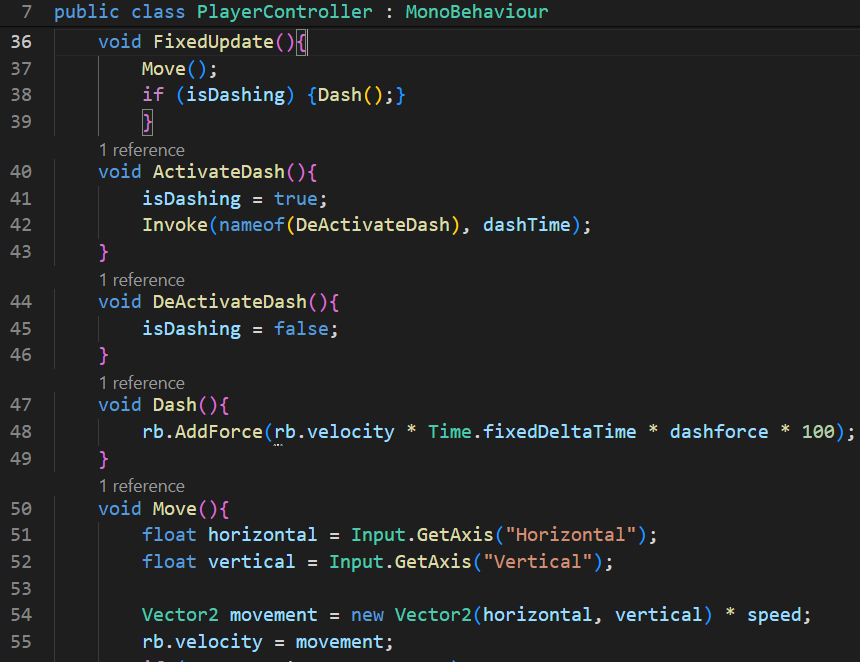


Рис. 67. Основной код рывка

Логика рывка (Рис. 67)

Активация и деактивация рывка

«ActivateDash()» - Функция, которая активирует рывок. Она устанавливает «isDashing» в true, что инициирует рывок в методе «FixedUpdate». Также она вызывает «DeActivateDash()» через заданный «dashTime», используя «Invoke», что обеспечивает временную продолжительность рывка.

«DeActivateDash()» - Деактивирует режим рывка, устанавливая «isDashing» обратно в false. Это возвращает управление игрока к обычному режиму движения.

Реализация рывка

«Dash()» - Вызывается в «FixedUpdate» при условии «isDashing» == true. Метод использует Rigidbody2D.AddForce для придания игроку импульса в направлении его текущего движения, умножая текущую скорость на «dashforce» и время. Множитель 100 применяется для усиления эффекта, учитывая масштабы и значения в Unity.

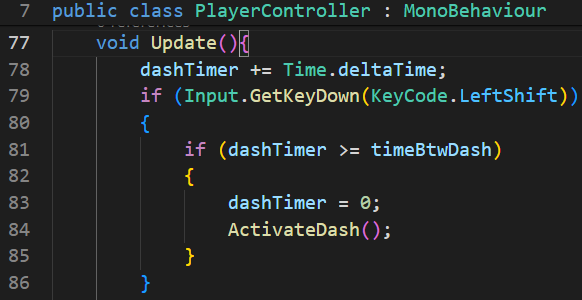


Рис. 68. Код рывка в методе «Update».

Инициация рывка

В методе Update() проверяется, нажата ли клавиша для рывка (KeyCode.LeftShift). Если с момента последнего рывка прошло достаточно времени (dashTimer >= timeBtwDash), таймер сбрасывается, и вызывается ActivateDash(), инициирующий рывок (Рис. 68).

Реализация рывка позволяет игрокам использовать эту способность для стратегического маневрирования в бою, быстрого уклонения от атак или закрытия дистанции до врага. Это добавляет слой стратегического глубины в игровой процесс, делая его более интерактивным и захватывающим.

Механика рывка значительно обогащает динамику игры, делая каждое столкновение более непредсказуемым и захватывающим. Она требует от игроков не только реакции, но и просчета лучшего момента для ее использования, что улучшает общий игровой опыт.

Так же в проекте добавили интерфейс пользователя для отображения здоровья и таймера рывка с помощью слайдеров. Это позволяет игрокам визуально отслеживать эти важные параметры, что улучшает управление ресурсами и стратегию игры. Давайте рассмотрим, как вы реализовали слайдеры здоровья и рывка в вашем коде.

Инициализация Слайдеров

Объявление Слайдеров (Рис. 69):

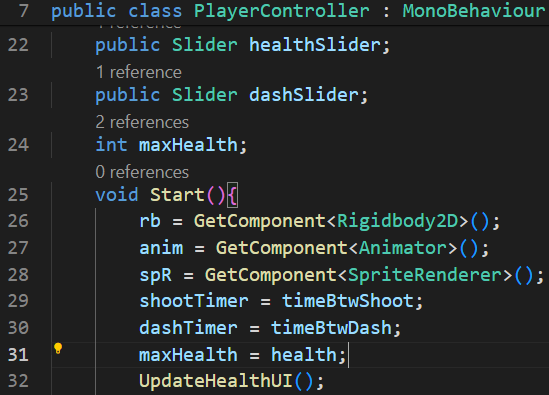


Рис. 69. Слайдеры и метод «Start».

В классе объявляем два объекта «Slider» — «healthSlider» для отображения текущего здоровья и «dashSlider» для индикации времени до следующего доступного рывка.

Инициализация в «Start()»:

В методе «Start», помимо инициализации других компонентов, также устанавливаем начальные значения для слайдеров. Для «healthSlider» вы вызываете «UpdateHealthUI()», чтобы установить начальное значение, соответствующее максимальному здоровью персонажа.

Обновление Слайдера Рывка

В методе «Update()» обновляем значение «dashSlider» в зависимости от времени, прошедшего с последнего рывка, относительно интервала времени между рывками («timeBtwDash»). Это позволяет игрокам видеть, когда рывок снова будет доступен.

Обработка и Обновление Слайдера Здоровья (Рис. 70)

Когда персонаж получает урон через метод «damage()», вы обновляете текущее здоровье, вызываете эффекты получения урона и тряски камеры, а затем обновляете слайдер здоровья для отражения нового уровня здоровья.



Рис. 70. Принцип работы слайдера «healthSlider».

Использование слайдеров для визуального отображения здоровья и времени восстановления способностей является отличным способом улучшить интерфейс пользователя. Это не только повышает удобство и информативность игрового процесса, но и помогает игрокам принимать обоснованные решения в бою, основываясь на доступных ресурсах и времени восстановления ключевых способностей. Такие интерфейсные элементы особенно важны в динамичных и тактически сложных играх, где правильное управление ресурсами может определить исход битвы.

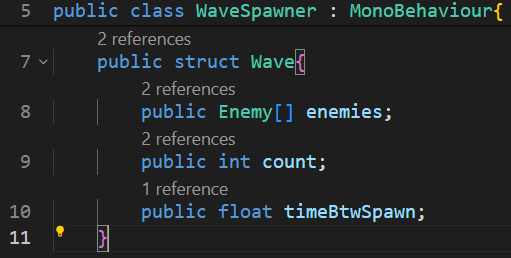


Рис. 71. Структура «Wave».

Так же создал скрипт «WaveSpawner», который представляет собой систему для спавна волн врагов в игре, использующую Unity. Этот скрипт управляет появлением врагов на различных этапах игры, обеспечивает последовательное увеличение сложности и динамизм игрового процесса. Давайте подробно разберем его ключевые элементы и функциональность.

Структура «Wave» (Рис. 71)

Структура «Wave» хранит информацию о каждой волне врагов:

* «enemies»: Массив типов врагов, которые могут появляться в этой волне.
* «count»: Количество врагов в волне.
* «timeBtwSpawn»: Время между спавном отдельных врагов в волне.

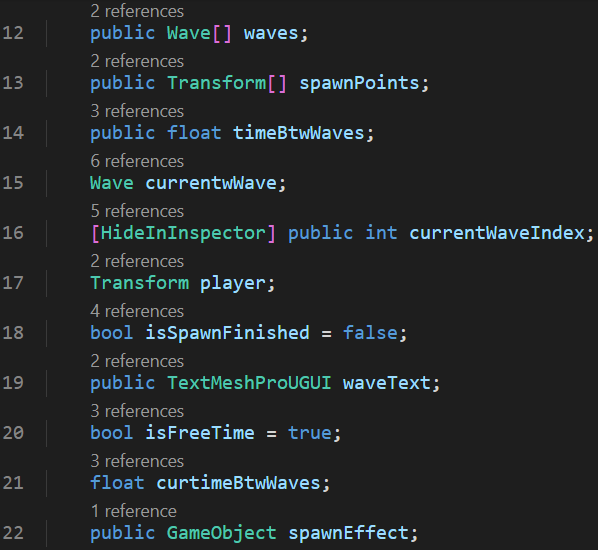


Рис. 72. Основные Переменные и Объекты скрипта «WaveSpawner».

Основные Переменные и Объекты (Рис. 72)

* «waves»: Массив волн, которые должны быть вызваны в течение игры.
* «spawnPoints»: Массив точек спавна, откуда враги будут появляться.
* «timeBtwWaves»: Время между волнами врагов.
* «currentWaveIndex»: Индекс текущей волны в массиве волн.
* «player»: Ссылка на игрока, используется для проверки его состояния.
* «isSpawnFinished»: Флаг, указывающий, завершился ли спавн текущей волны.
* «waveText»: Текстовое поле для вывода информации о текущем состоянии волн.
* «isFreeTime»: Флаг, показывающий, находится ли игра в режиме паузы между волнами.
* «spawnEffect»: Эффект, который проигрывается при спавне врагов.

Логика Работы



Рис. 73. Основные методы скрипта «WaveSpawner».

Метод «Start()»

* Инициализирует начальные параметры и запускает первую волну врагов.

Метод «Update()»

* Обновляет текстовое поле «waveText», отображая время до начала следующей волны или номер текущей волны.
* Если все враги текущей волны уничтожены и волны ещё остались, запускает следующую волну.

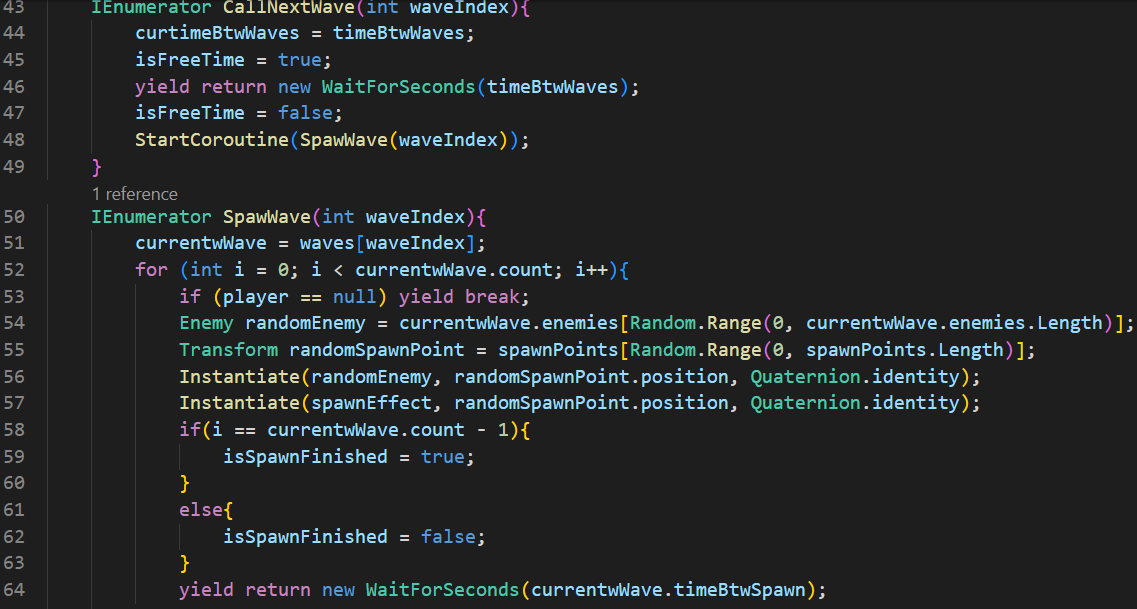


Рис. 74. Корутины скрипта «WaveSpawner».

Корутины «CallNextWave()», «SpawWave()» (Рис.74)

Метод «CallNextWave()»

* Ожидает заданное время между волнами, затем начинает спавн следующей волны.

Метод «SpawWave()»

* Последовательно спавнит врагов текущей волны в случайно выбранных точках спавна.
* Для каждого врага инстанцирует объект врага и объект визуального эффекта спавна.
* Управляет временем спавна между врагами в волне.

Визуальные и Игровые Эффекты

* Визуальное отображение информации о волнах и времени через интерфейс пользователя улучшает взаимодействие с игроком и помогает поддерживать его в курсе текущего состояния игры.
* Использование визуальных эффектов при спавне врагов делает процесс более зрелищным и информативным.

Система WaveSpawner является ключевым компонентом для управления сложностью и динамикой в игре, где волны врагов — это основная часть игрового процесса. Она не только добавляет ритмичность и предсказуемость вызовов для игрока, но и позволяет разработчикам легко настраивать сложность игры, изменяя параметры волн и их тайминги.

## **2.13 Неуязвимость игрока во время рывка, следы ног, экран проигрыша, монеты**

В этом игровом проекте на Unity необходимо было добавить ключевое улучшение к механике рывка (Dash), а именно - временная неуязвимость персонажа во время активации этой способности. Это изменение повышает стратегическую ценность использования рывка, позволяя игрокам использовать его для избегания урона в критических ситуациях. Давайте подробно рассмотрим добавленный код и его функциональность.

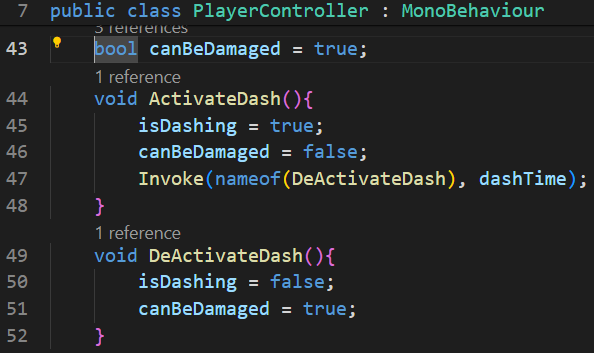


Рис. 75. Код механики рывка.

Переменная «canBeDamaged»:

* Добавлена булевая переменная «canBeDamaged», которая контролирует, может ли персонаж получать урон. По умолчанию она установлена в true, что означает, что персонаж может быть поврежден.

Метод «ActivateDash()»:

* Когда рывок активируется, переменная «canBeDamaged» устанавливается в «false», что делает персонажа временно неуязвимым для урона.
* В этом же методе, «isDashing» устанавливается в «true», запуская состояние рывка.
* Метод «Invoke(nameof(DeActivateDash), dashTime)» планирует вызов метода «DeActivateDash» через заданное время «dashTime», которое определяет длительность рывка.

Метод «DeActivateDash()»:

* После завершения времени действия рывка, метод возвращает isDashing в «false», завершая состояние рывка.
* В то же время, «canBeDamaged» возвращается в true, снова разрешая получение урона персонажем.

Метод «damage(int damage)» (Рис. 76):

* В методе, отвечающем за получение урона, добавлена проверка переменной «canBeDamaged». Если она установлена в «false», метод немедленно прерывается с помощью «return», не позволяя игроку получить урон.

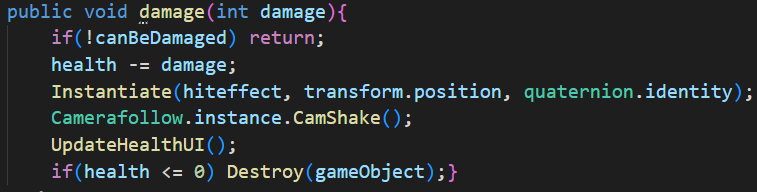


Рис. 76. Код метода «damage» для механики рывка.

Таким образом, вся последующая логика по обработке урона пропускается, пока персонаж находится в состоянии неуязвимости.

Добавление временной неуязвимости во время рывка значительно улучшает игровой процесс, предоставляя игрокам мощный инструмент для управления боевыми ситуациями. Это позволяет более агрессивно использовать рывок для прорыва через группы врагов или избежания урона во время интенсивных боев. Включение этой механики требует от игроков более глубокого понимания таймингов и управления способностями и делает процесс игры более динамичным и интересным.

Было добавлены следы шагов через систему частиц (ParticleSystem), которые значительно улучшают визуальную составляющую игры, делая движения персонажа более реалистичными и заметными. Этот подход к интеграции следов шагов с использованием частиц включает в себя контроль за активацией и интенсивностью эмиссии частиц в зависимости от действий персонажа.

Интеграция «ParticleSystem» для Следов персонажа.

В коде предполагается, что «footParticle» — это ссылка на компонент «ParticleSystem», прикреплённый к персонажу. Эта система частиц настроена таким образом, чтобы визуализировать следы шагов при движении персонажа.

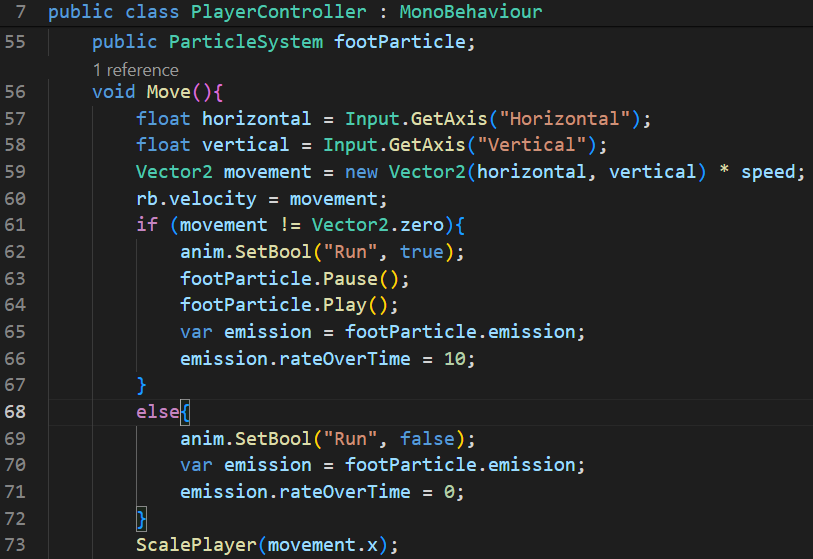


Рис. 77. Код «ParticleSystem» - «footParticle».

При движении персонажа (когда вектор движения не равен нулю) система частиц активируется. Использование методов «Pause()» и «Play()» предназначено для обеспечения непрерывности визуализации частиц при каждом шаге.

Эмиссия частиц регулируется настройкой «rateOverTime» свойства «emission» системы частиц. Этот параметр управляет скоростью, с которой частицы генерируются, и устанавливается равным 10 в момент активного движения.

Когда персонаж останавливается («movement» равен нулю), генерация частиц приостанавливается путем установки «rateOverTime» в 0. Это предотвращает создание следов шагов, когда персонаж не движется.

Интеграция следов шагов через «ParticleSystem» является отличным примером того, как можно использовать встроенные инструменты Unity для улучшения визуальной составляющей игры. Это повышает уровень детализации среды и персонажей, делая игровой мир более живым и интерактивным.

Таким же образом были добавлены следы всем врагам в игре через систему частиц (ParticleSystem) (Рис. 78), который аналогичен тому, как реализовали это для главного персонажа. Использование следов шагов улучшает визуальную обратную связь и помогает создать более динамичную и живую игровую среду.

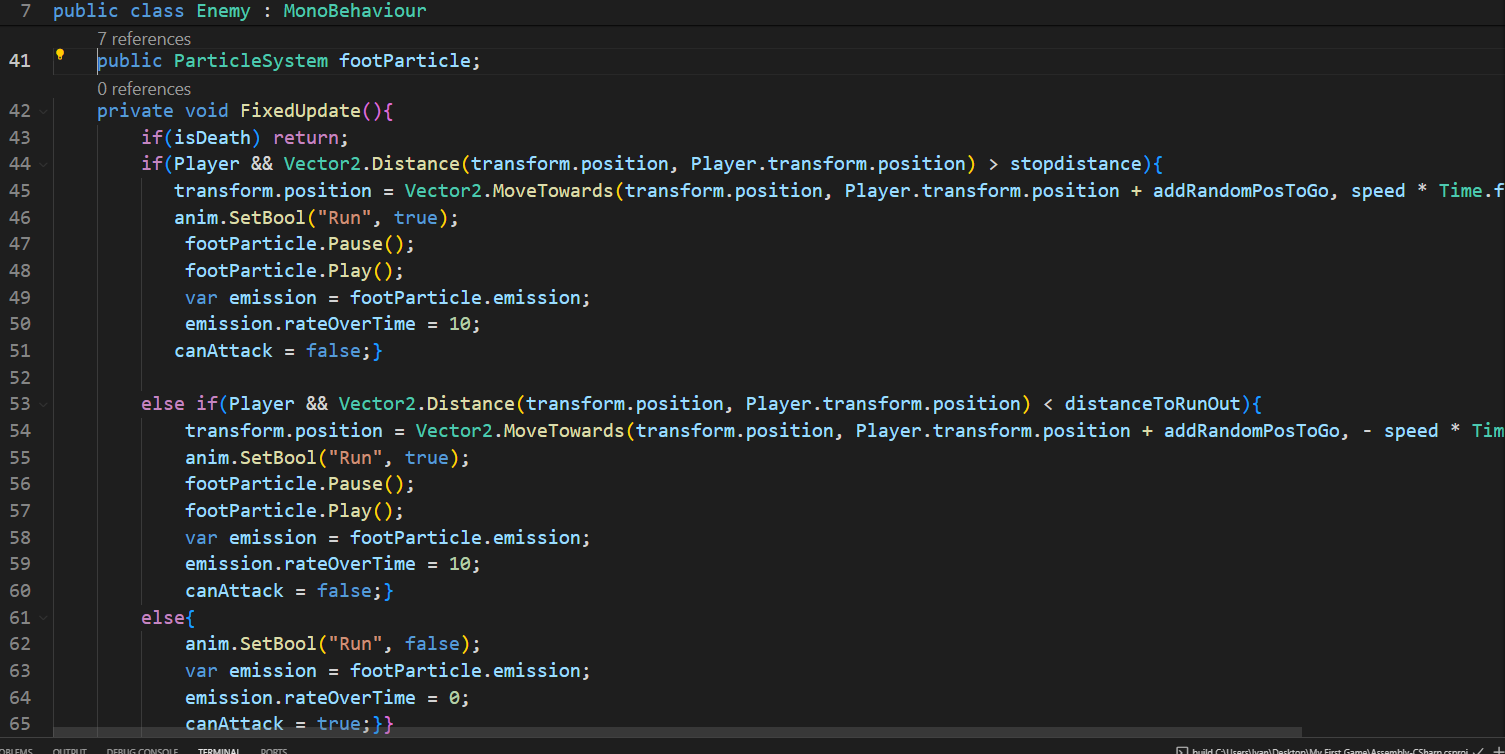


Рис. 78. Аналогичный код для «Enemy» «footParticle».

Инициализация «ParticleSystem»:

Каждый враг оснащен компонентом «ParticleSystem», сохраненным в переменной «footParticle». Эта система частиц настроена для визуализации следов шагов, когда враг перемещается.

Управление системой частиц в «FixedUpdate»:

В методе «FixedUpdate» происходит управление активностью системы частиц в зависимости от состояния и движения врага.

Логика визуализации следов шагов:

* Когда враг движется к игроку или отходит от него (за исключением состояния смерти «isDeath», система частиц активируется для визуализации движения:
* Методы «Pause()» и «Play()» используются для обновления и перезапуска эмиссии частиц, что обеспечивает непрерывное отображение следов.
* Свойство «rateOverTime» объекта emission системы частиц устанавливается в значение 10, что соответствует желаемой частоте генерации частиц.
* Если враг перестает двигаться, частота генерации частиц снижается до нуля (emission.rateOverTime = 0), эффективно останавливая генерацию следов шагов.

Использование следов шагов у врагов и главного героя повышает уровень детализации и визуальной привлекательности игры, делая каждое действие врагов и героя заметнее и информативнее. Это может помочь игрокам лучше ориентироваться в тактической ситуации, осознавая направление движения врагов даже в сложных или хаотичных моментах игры.

Так же была реализована панель проигрыша (death panel), которая активируется при смерти игрока. Эта панель содержит информацию о достигнутом уровне (волне), сообщение о проигрыше и кнопку для перезапуска уровня. Это важный элемент пользовательского интерфейса, который информирует игрока о его проигрыше и предоставляет возможность быстро начать игру заново.

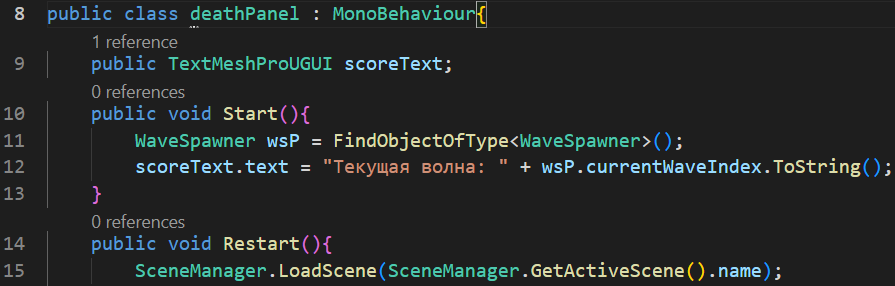


Рис.79. Код «deathPanel».

Компоненты «deathPanel» (Рис. 79).

Инициализация и отображение информации о волне:

В методе «Start» панели смерти используется компонент «TextMeshProUGUI» с именем «scoreText» для отображения информации о том, на какой волне игрок проиграл. Это делается путем поиска активного компонента «WaveSpawner» на сцене и получения из него индекса текущей волны.

Метод «Restart» для перезапуска уровня:

Метод «Restart» вызывает перезагрузку текущей сцены, используя «SceneManager». Это позволяет игроку начать игру заново с того же уровня, на котором он проиграл.

Активация «deathPanel» (Рис. 80).

Панель смерти активируется в скрипте управления игроком (PlayerController), когда здоровье игрока падает до нуля или ниже. В этот момент панель делается активной, а объект игрока уничтожается.



Рис. 80. Код «deathPanel» в скрипте «PlayerController».

Использование «deathPanel» улучшает взаимодействие с игроком, предоставляя четкую обратную связь о проигрыше и простую опцию для быстрого перезапуска. Это помогает поддерживать игровой интерес и мотивацию, позволяя игрокам немедленно попробовать пройти уровень заново без необходимости проходить через дополнительные меню или задержки.

Была еще реализована система наград в виде монеток, которые игрок зарабатывает, убивая врагов. Эта функциональность добавляет дополнительный элемент прогрессии для игроков, улучшая игровой опыт.

Код в «PlayerController» (Рис. 81).

Переменные и UI элементы:

«currentMoney»: Хранит текущее количество монеток, которые игрок заработал.

coinsText: Элемент интерфейса TextMeshProUGUI, который отображает количество монеток игрока.

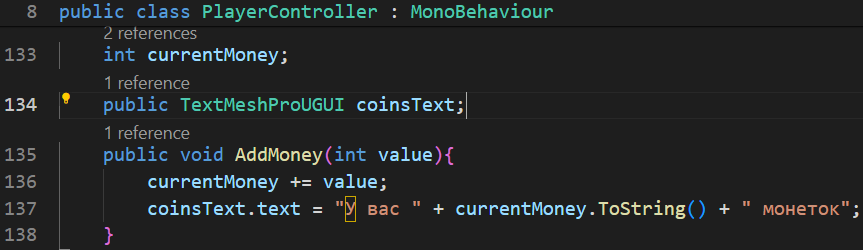


Рис. 81. Код в скрипте «PlayerController».

Метод «AddMoney»:

Этот метод принимает целочисленное значение «value», которое добавляется к текущему балансу монет (currentMoney).

После добавления суммы, текст на интерфейсе обновляется, чтобы отразить новое количество монеток, используя строку "У вас X монеток".

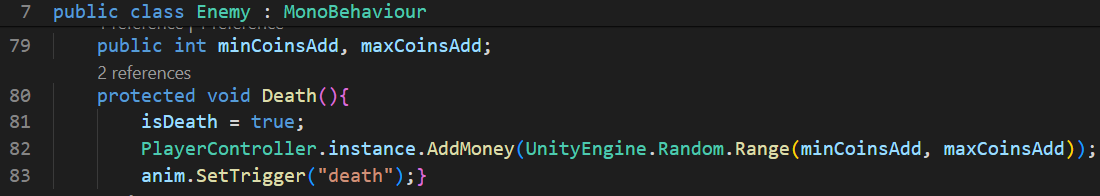


Рис. 82. Код в скрипте «Enemy».

Код в «Enemy» (Рис. 82).

Переменные для монеток:

«minCoinsAdd» и «maxCoinsAdd»: Устанавливают минимальное и максимальное количество монеток, которые можно получить за убийство данного врага.

Метод «Death»:

При смерти врага, метод «Death» вызывается. В нем, с помощью PlayerController.instance.AddMoney, добавляется случайное количество монеток в диапазоне от «minCoinsAdd» до «maxCoinsAdd».

Используется UnityEngine.Random.Range для определения случайного количества монеток.

Добавление монеток при убийстве врагов вводит в игру дополнительный элемент стратегии и вознаграждения, предоставляя игрокам конкретную награду за их успехи в бою. Это не только стимулирует игроков активнее участвовать в боевых действиях, но и может быть использовано для развития персонажа или покупки улучшений и предметов, углубляя механику прогрессии в игре.

Система монеток при убийстве врагов эффективно увеличивает заинтересованность и вовлеченность игроков, делая каждую победу над врагом более значимой и выгодной.

Вот так теперь выглядит окно Unity с иерархией, когда персонаж погибает:

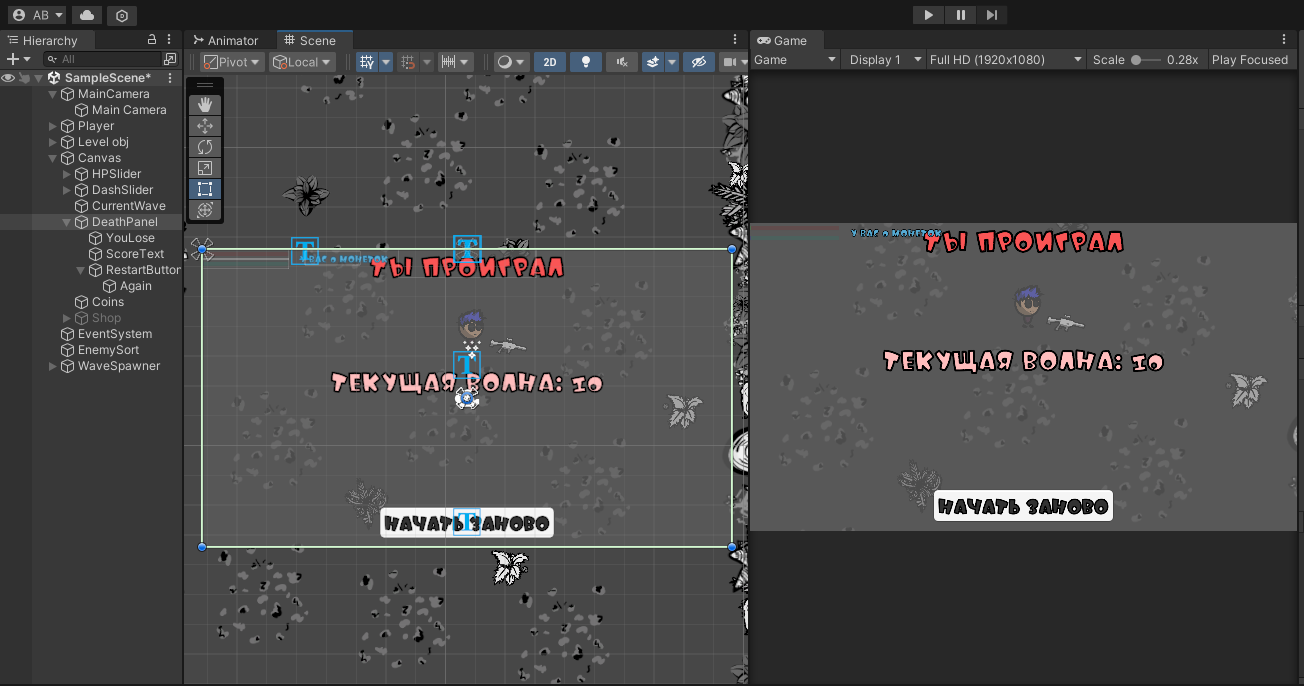


Рис. 83. Окно Unity со всеми добавленными функциями.

## **2.14 Внутриигровой магазин**

Внутриигровой магазин представляет собой систему, позволяющую игрокам приобретать улучшения за игровую валюту, полученную в ходе игры. Магазин активируется и деактивируется с помощью клавиши «Escape», что делает его легкодоступным в любой момент игры. Рассмотрим структуру и функциональность вашего магазина.

Основные компоненты магазина (Рис. 84):

1. «buyButtons»:

Массив кнопок, каждая из которых предназначена для покупки конкретного улучшения. Кнопки становятся неактивными (недоступными для нажатия), если улучшение уже приобретено или у игрока недостаточно монет для покупки.

1. «boughtTexts»:

Массив текстовых полей «TextMeshProUGUI», отображающих статус каждого улучшения (например, "Куплено" или "Мало монет").

1. «prices»:

Массив, содержащий стоимость каждого улучшения. Используется для проверки достаточности средств у игрока для совершения покупки.

1. «shopPanel»:

«GameObject», который представляет собой панель магазина. Его видимость переключается нажатием «Escape».

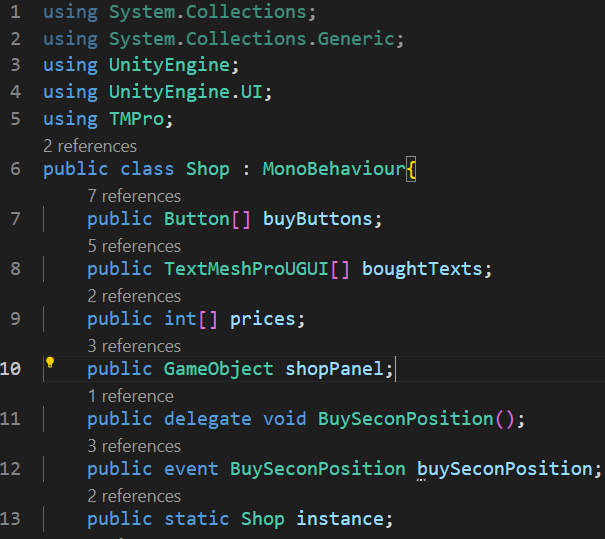


Рис. 84. Основные переменных.

Логика работы:

1. Инициализация и настройка:

В методе «Awake» устанавливается статическая ссылка на экземпляр магазина (instance), что позволяет другим скриптам обращаться к магазину.

В методе «Start» происходит инициализация состояний кнопок и текстов в зависимости от сохраненных настроек (используя «PlayerPrefs»). Если улучшение уже куплено, кнопка становится неактивной, и текст изменяется на "Куплено".

1. Обработка ввода пользователя:

В методе «Update» слушается нажатие клавиши «Escape» для переключения видимости «shopPanel». При открытии магазина время в игре останавливается (Time.timeScale = 0), а при закрытии возобновляется (Рис. 85).



Рис. 85. Код методов «Start» и «Update».

1. Проверка возможности покупки:

Метод «Check» переоценивает возможность покупки каждого улучшения, обновляя интерактивность кнопок и тексты в зависимости от текущего баланса монет игрока и статуса покупки улучшений.

1. Покупка улучшений:

Метод «Buy» вызывается при нажатии на соответствующую кнопку. Этот метод обновляет состояние улучшения в «PlayerPrefs», вычитает стоимость из баланса монет и вызывает проверку доступности улучшений.

1. Удаление настроек:

Метод «DeletePlayerPrefs» предназначен для сброса всех сохраненных настроек (PlayerPrefs.DeleteAll()), что может быть полезно во время разработки или тестирования.

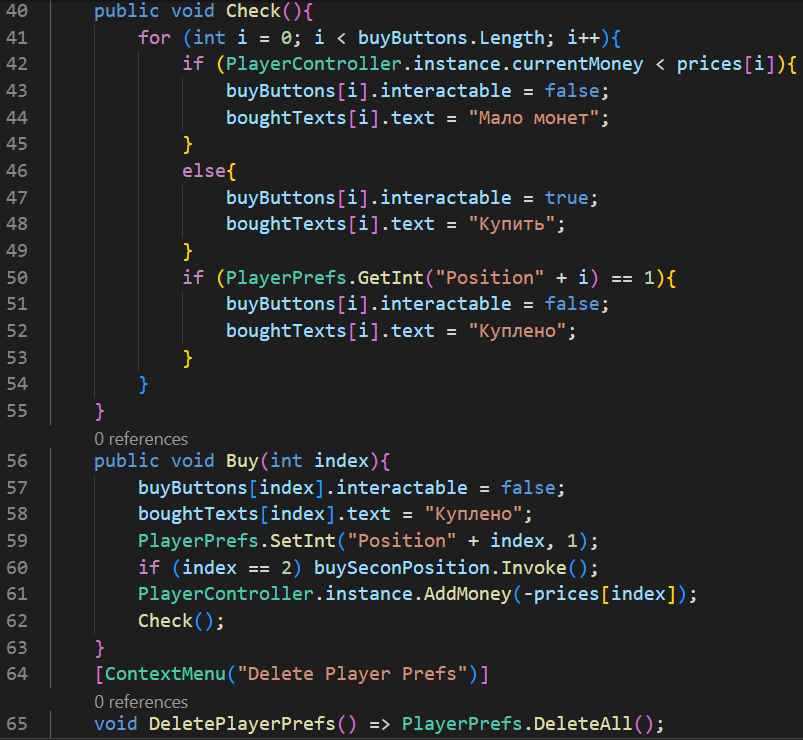


Рис. 86. Код методов «Check», «Buy» и «DeletePlayerPrefs».

Визуальное и функциональное оформление:

Магазин обладает самым простым и интуитивно понятным интерфейсом, позволяя игрокам легко управлять своими ресурсами и улучшениями. Он добавляет в игру дополнительные возможности для стратегии и улучшает общее взаимодействие пользователя с игрой.

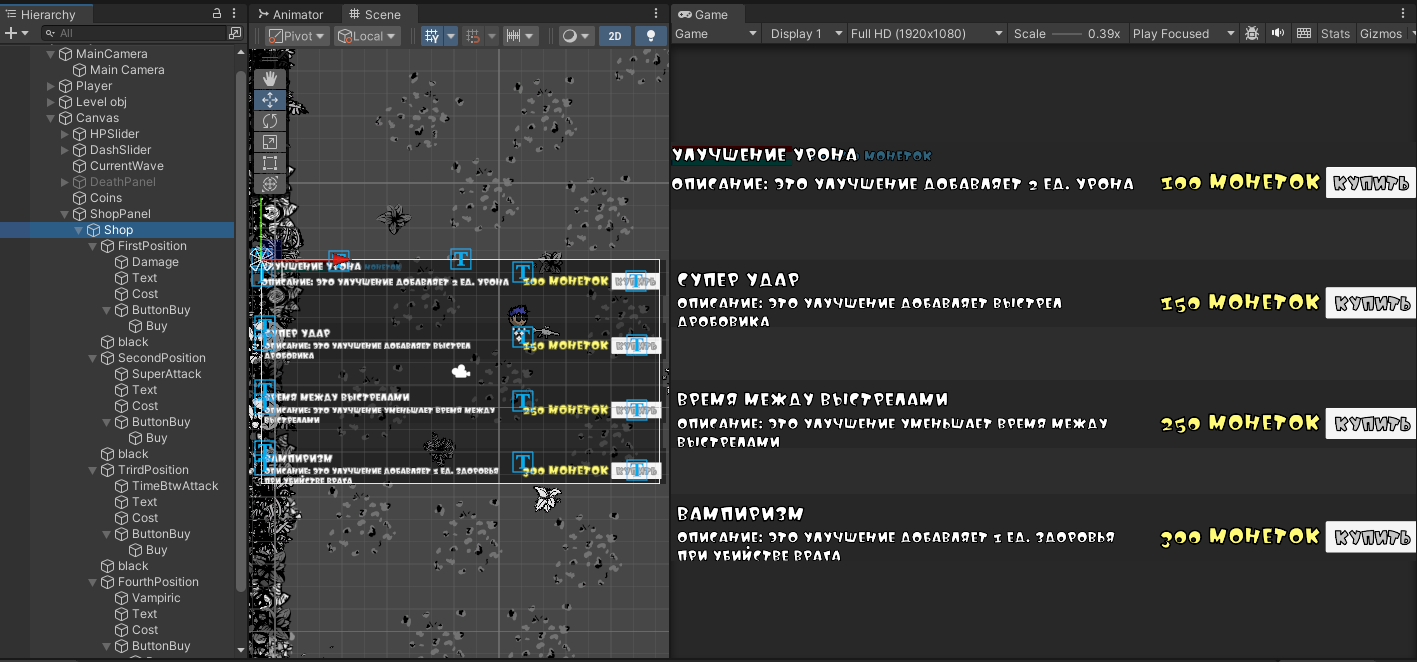


Рис. 87. Окно внутриигрового магазина «Shop».

Первое улучшение в магазине это - увеличение урона снаряда (пули). Представляет собой решение для динамического изменения урона в зависимости от покупки улучшения через внутриигровой магазин.

Описание Кода Улучшения Урона

Проверка Покупки Улучшения:

При столкновении пули с объектом проверяется, куплено ли улучшение урона (Position0 в PlayerPrefs).

Если улучшение куплено (PlayerPrefs.GetInt("Position0") == 1), к базовому урону (damage) добавляется дополнительные 2 единицы урона.



Рис. 87. Код увеличения урона в скрипте «Bullet».

int damagee = PlayerPrefs.GetInt("Position0") == 1 ? damage += 2 : damage;

Эта строка кода использует тернарный оператор для определения, следует ли увеличить урон. Если улучшение куплено, то к текущему урону прибавляется 2 единицы, в противном случае урон остается базовым.

Применение Урона:

После расчета урона с учетом возможного улучшения, урон применяется к врагу (Enemy), в который попала пуля. Это делается вызовом метода damage(damagee) у компонента Enemy целевого объекта.

Это улучшение имеет значительное влияние на игровой процесс, так как позволяет игроку наносить больший урон врагам, что может быть критически важно для преодоления более сложных уровней или волн врагов. Увеличение урона напрямую влияет на способность игрока эффективно справляться с вызовами, предлагаемыми игрой, и повышает его шансы на успех.

Следующее улучшение для внутриигрового магазина представляет собой «супер-выстрел», который позволяет выпускать три пули одновременно, имитируя эффект дробовика. Два из этих выстрелов направлены под углом в 15 градусов относительно центральной оси, создавая расширенный угол атаки. Эта функциональность улучшает боевые возможности персонажа, позволяя эффективно атаковать группы врагов или увеличивать шанс попадания по быстро движущимся целям. Давайте подробно рассмотрим реализацию этого улучшения.

Реализация «Супер-Выстрела»

Массив «Transform»:

«shootSuperPos» представляет собой массив трансформов, которые определяют позиции и направления трех различных пушек или точек выстрела. Эти позиции настроены так, чтобы две из них были повернуты на 15 градусов влево и вправо соответственно от центральной оси.

Метод «SuperShoot»:

В этом методе происходит инстанцирование пуль из всех трех позиций «shootSuperPos». Для каждой позиции создается пуля с соответствующим направлением, заданным ее трансформацией.

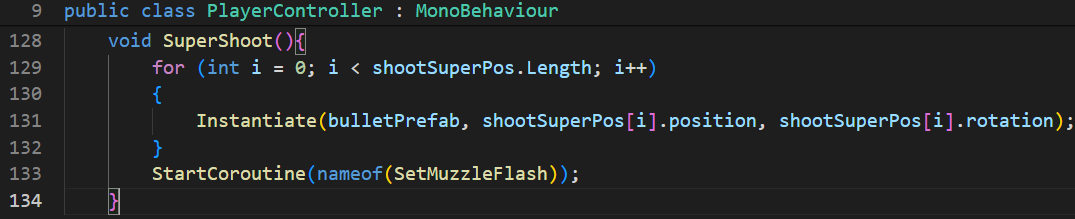


Рис. 88. Метод «SuperShoot».

Активация «Супер-Выстрела» (Рис. 89):

«Супер-выстрел» активируется при нажатии правой кнопки мыши (или другой заданной кнопки), если таймер «shootSuperTimer» позволяет это сделать (т.е., больше времени «timeBtwSuperShoot»), и, если улучшение было приобретено (проверка через PlayerPrefs.GetInt("Position1") == 1).

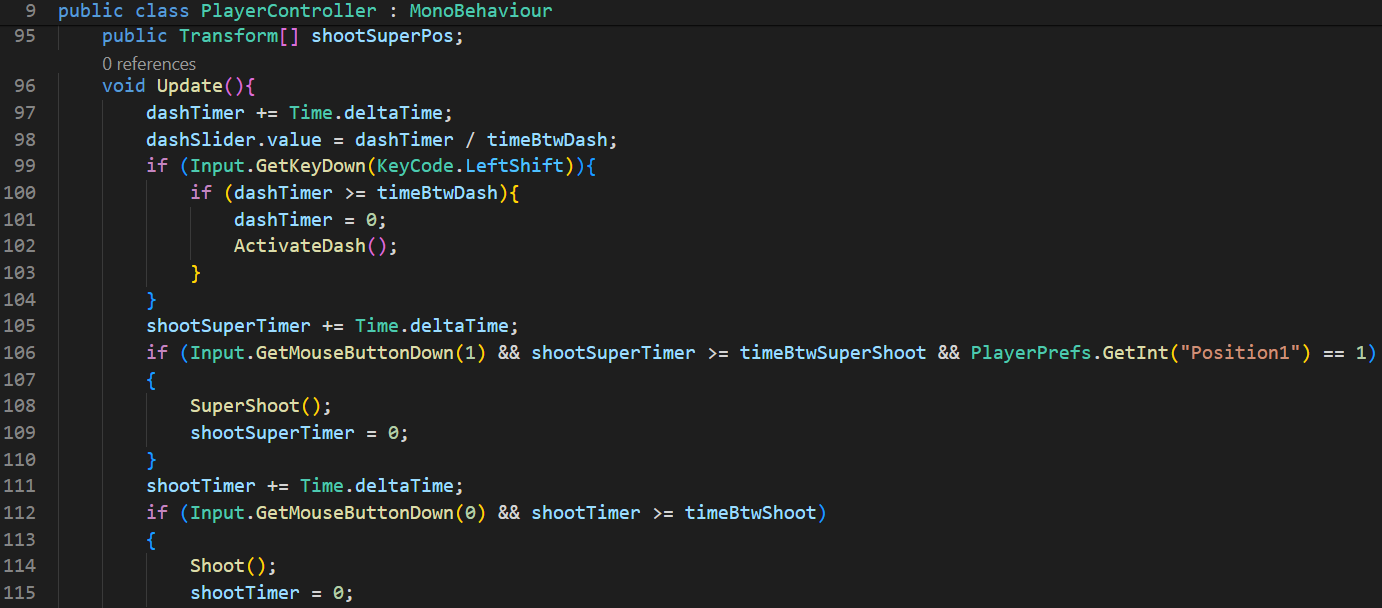


Рис. 89. Код «Супер-выстрела» в методе «Update» скрипта «PlayerController».

Значение улучшения для игры

Добавление «супер-выстрела» значительно усиливает атакующий потенциал игрока, делая возможным нанесение урона сразу нескольким целям или увеличивая вероятность поражения быстро движущихся или сложно целеуказываемых противников. Это улучшение особенно полезно в ситуациях, когда игрок окружен или, когда нужно быстро "очистить" зону от большого количества врагов.

Следующее улучшение во внутриигровом магазине направлено на ускорение стрельбы, включая как обычные выстрелы, так и супер-выстрелы. Это улучшение уменьшает время между выстрелами, позволяя игроку стрелять чаще.

Реализация Ускорения Стрельбы

Событие «buySecondPosition»:

В классе «Shop» объявлено событие «buySecondPosition», которое вызывается при покупке определённого улучшения (в данном случае, когда покупается улучшение под индексом 2).

Это событие используется для триггера изменений в других частях игры, а именно в контроллере игрока.

public delegate void BuySecondPosition();

public event BuySecondPosition buySecondPosition;

Метод «UpdateTimeBtwShoot» в «PlayerController» (Рис. 90):

Этот метод отвечает за изменение переменных «timeBtwShoot» и «timeBtwSuperShoot», которые контролируют задержку между обычными и супер-выстрелами соответственно.

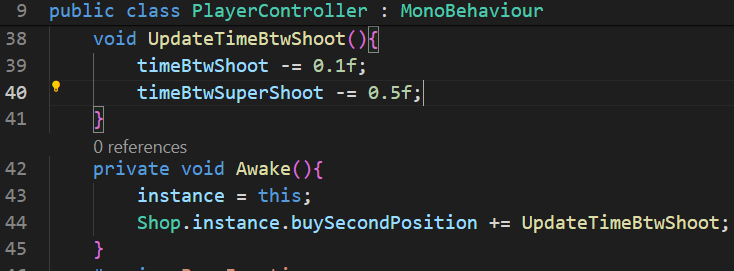


Рис. 90. Метод «UpdateTimeBtwShoot» в «PlayerController».

При активации улучшения время между выстрелами уменьшается на 0.1 секунду для обычных выстрелов и на 0.5 секунды для супер-выстрелов.

Подписка на Событие:

В методе Awake «PlayerController» подписывается на событие «buySecondPosition» из «Shop». Это означает, что каждый раз, когда улучшение покупается в магазине, вызывается метод «UpdateTimeBtwShoot», который обновляет интервалы стрельбы.

Это улучшение значительно повышает боевой потенциал игрока, позволяя ему выпускать пули быстрее, что особенно важно в моменты интенсивных боёв или, когда необходимо быстро реагировать на появление новых врагов. Уменьшение времени между выстрелами позволяет игроку более эффективно контролировать поле боя и быстрее реагировать на угрозы.

Последнее улучшение «вампиризм» представляет собой механику, позволяющую игроку восстанавливать здоровье при каждом убийстве врага.

Реализация Вампиризма

Добавление здоровья при смерти врага:

В методе «Death» класса «Enemy» (Рис. 91), помимо добавления денег за убийство, теперь также проверяется, куплено ли улучшение вампиризма (проверка через PlayerPrefs.GetInt("Position3")).

Если улучшение вампиризма активировано (Position3 == 1), то вызывается метод «AddHealth(1)» на экземпляре «PlayerController», который увеличивает здоровье игрока на 1 единицу при каждом убийстве врага.

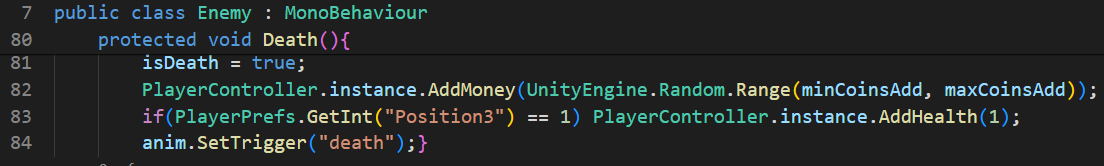


Рис. 91. Код в методе «Death» класса «Enemy».

Метод «AddHealth» в «PlayerController» (Рис. 92):

Этот метод отвечает за увеличение текущего здоровья игрока. Он также гарантирует, что здоровье не превысит максимальный уровень (maxHealth).

После добавления здоровья вызывается метод UpdateHealthUI(), который обновляет пользовательский интерфейс для отображения текущего состояния здоровья игрока.

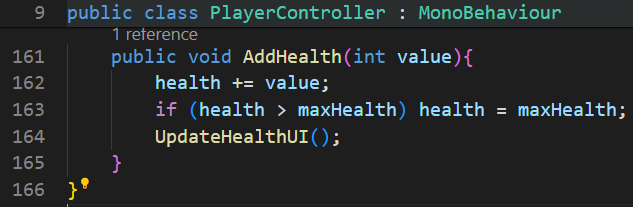


Рис. 92. Код в методе «AddHealth» в «PlayerController».

Улучшение вампиризма предоставляет игроку дополнительные средства для поддержания своего здоровья в течение боя, что особенно полезно в ситуациях, когда ресурсы для восстановления здоровья.

## **2.15 Post Processing (эффекты)**

Добавление «Post Processing» (постобработки) в Unity значительно улучшает визуальное восприятие игры, добавляя реалистичности и атмосферности сценам. Постобработка позволяет применять различные визуальные эффекты после рендеринга сцены, но перед тем как она будет окончательно отображена на экране. В Unity для работы с постобработкой часто используется «Post Processing Stack», который можно добавить через «Package Manager».

Как добавляются «Post Processing» в Unity

Добавление «Post Processing Stack»:

* Откройте «Window» -> «Package Manager».
* Найдите и установите пакет «Post Processing».

Создание «Post Processing Layer»:

* Добавьте новый слой в Edit -> Project Settings -> Tags and Layers -> «PP».
* Присвойте этот слой камере, которая будет использоваться для рендеринга постобработки.

Настройка камеры:

* Выберите камеру в вашей сцене.
* Добавьте компонент «Post-process Layer» из раздела «Post Processing».
* Установите созданный ранее слой в поле «Layer».

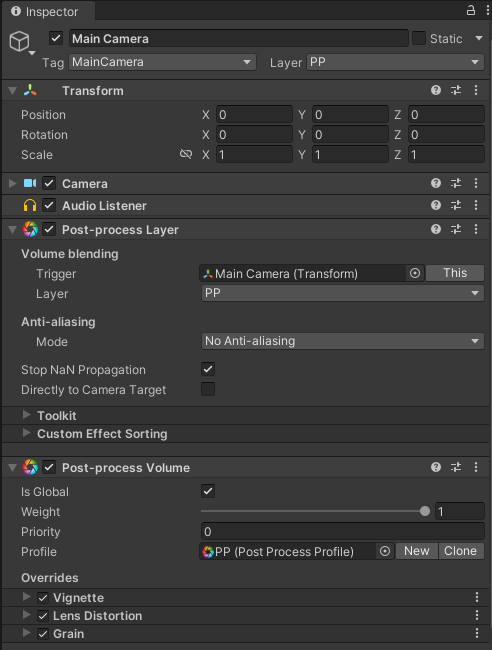


Рис. 93. Инспектор главной камеры с Post Processing.

Создание «Post Processing» Profile:

* В панели Project создайте новый «Post-process Profile» через «Create» -> «Post-processing Profile».
* Присвойте этот профиль компоненту «Post-process Volume», который можно добавить на камеру или создать отдельный объект с этим компонентом для глобальных эффектов.

В профиле постобработки можно добавлять и настраивать различные эффекты.

Эффекты «Post Processing»

В Unity существует множество эффектов постобработки, каждый из которых предназначен для достижения определённых визуальных целей:

1. «Vignette» (Виньетка):

Этот эффект добавляет темные углы к изображению, что создает ощущение фокусировки в центре. Он часто используется для усиления кинематографического ощущения или для привлечения внимания к центральной части кадра.

1. «Lens Distortion» (Искажение):

Имитирует искажение, создаваемое реальными камерными объективами. Этот эффект может добавлять выпуклость или вогнутость к изображению, что используется для создания уникальных визуальных стилей или имитации определенного типа камеры.

1. «Grain» (Зернистость):

Добавляет зернистость к изображению, что придает ему текстуру фильма или старинной фотографии. Этот эффект улучшает реалистичность визуализации и часто используется в хоррор-играх или проектах в ретро-стиле для усиления атмосферности.

Использование «Post Processing» в Unity позволяет разработчикам значительно улучшить визуальное качество игр, добавляя глубину и атмосферу. Постобработка может трансформировать обычные сцены в визуально богатые и эмоционально вовлекающие изображения, делая игровой процесс более запоминающимся и впечатляющим.

## **2.16 Universal RP (свет)**

Universal Render Pipeline (URP), ранее известный как Lightweight Render Pipeline (LWRP), является частью новых рендеринговых пайплайнов Unity, предназначенных для более гибкой и легкой оптимизации под различные платформы. URP особенно полезен для проектов, ориентированных на мобильные устройства и AR/VR, но его можно эффективно использовать и в высококачественных проектах для PC и консолей. Одной из ключевых особенностей URP является его поддержка улучшенного освещения и шейдеров.

Добавление Universal Render Pipeline в проект Unity

Установка Universal RP через Package Manager:

* Перейдите в Window -> Package Manager.
* В списке пакетов выберите "Packages: Unity Registry".
* Найдите "Universal RP" и нажмите "Install".

Настройка URP:

* После установки URP, создайте новый URP asset (Render Pipeline Asset). Перейдите в Assets -> Create -> Rendering -> Universal Render Pipeline -> Pipeline Asset.
* Дополнительно можно создать Universal Render Pipeline -> Renderer Feature для кастомизации рендерера.

Применение URP к вашему проекту:

* Перейдите в Edit -> Project Settings -> Graphics.
* В разделе "Scriptable Render Pipeline Settings" перетащите созданный URP asset.

Настройка освещения с Global Light

В URP Global Light обычно представляет собой Directional Light (направленный свет), который используется для имитации солнца или основного источника света в сцене.

1. Добавление Global Light:

* В иерархии вашего проекта создайте новый объект (GameObject -> Light -> Directional Light). Этот свет будет вашим Global Light.
* По умолчанию, направленный свет в URP настроен как основной источник света, освещающий всю сцену.

1. Настройка света:

* Вы можете настроить параметры света в инспекторе, изменяя такие параметры, как Intensity, Color и Shadow Type.
* Для реализации более сложных эффектов, таких как мягкие тени или контроль над освещением на разных уровнях, используйте Light Layers.

Световые компоненты:

* Directional Lights (Направленный свет): Используется для имитации далеких источников света, таких как солнце. В URP направленный свет может использовать каскадные тени для улучшенного отображения теней на различных дистанциях.
* Point Lights (Точечный свет): Источник света, испускающий свет во всех направлениях от данной точки. В URP поддерживается возможность использования различных режимов освещения для точечных источников света.
* Spot Lights (Прожекторы): Источники света, которые излучают свет в определенном направлении, с углом распространения. Они идеальны для создания эффектов фокусированного освещения.

Universal Render Pipeline предоставляет мощные инструменты и возможности для создания красивых и производительных игр. Особенности освещения в URP позволяют создавать более динамичные и реалистичные сцены, что повышает общее качество игрового процесса и визуального опыта. URP является отличным выбором для проектов, требующих как высокой производительности, так и превосходной визуализации.

Осталось добавить аудио в игру через «SoundManager», которая представляет собой хорошую практику управления всеми звуковыми эффектами в Unity. «SoundManager» действует как централизованная система для воспроизведения звуков, что упрощает контроль над аудио в вашей игре и делает код более чистым и организованным. Давайте подробнее разберем реализацию и использование «SoundManager».

Создание «SoundManager»

Создание объекта «SoundManager»:

В Unity создайте пустой объект (GameObject), который будет служить контейнером для компонента «SoundManager» и «AudioSource».

Назовите этот объект, например, «SoundManager», что поможет легко идентифицировать его в иерархии проекта.

Добавление компонентов:

К объекту «SoundManager» добавляем компонент «AudioSource», который будет использоваться для воспроизведения всех аудио клипов.

Создаем новый скрипт «SoundManager» и прикрепляем его к тому же объекту.



Рис. 94. Код «SoundManager».

audS.pitch = Random.Range(0.9f, 1.1f); - Случайно изменяем высоту тона для большей вариативности звука

audS.PlayOneShot(value); - Воспроизводим звук однократно

Использование «SoundManager»

Интеграция звуков в игровые события:

В любом месте вашего кода, где требуется воспроизвести звук (например, при получении урона, выстреле или иных событиях), вы можете легко вызвать метод «PlaySound», передав нужный аудио клип.

Пример вызова звука рывка:

SoundManager.instance.PlaySound(dashSound);

Централизованное управление:

Все звуковые эффекты управляются из одного места, что облегчает их изменение и обновление.

Легкость доступа:

Использование статической переменной «instance» позволяет обращаться к «SoundManager» из любого места кода без необходимости передавать ссылки или использовать другие сложные механизмы.

Гибкость:

Метод «PlaySound» можно расширять и модифицировать для воспроизведения различных звуков в зависимости от контекста, добавлять настройки громкости, группировать звуки по категориям и т.д.

«SoundManager» предоставляет мощный и гибкий инструмент для управления аудио в вашей игре на Unity. Использование такого подхода не только упрощает разработку, но и делает процесс более организованным, а саму игру — более динамичной и эмоциональной за счет качественного и уместного звукового сопровождения.

## **2.18 Прицел**

Добавление пользовательского курсора в форме прицела — это отличный способ улучшить интерфейс и игровой процесс, особенно в играх, где важна точность и реакция. Мы создали скрипт и настроили объект для отображения прицела вместо стандартного курсора мыши.

Создание и настройка объекта прицела

1. Создание объекта прицела:

* В Unity создайте новый «GameObject» в вашей иерархии. Назовите его, например, «Cursor».
* К этому объекту добавьте компонент Sprite Renderer. Это позволит вам установить спрайт, который будет использоваться как изображение прицела.

1. Добавление спрайта прицела:

* В компоненте Sprite Renderer установите спрайт прицела. Это может быть любое изображение, которое вы хотите использовать в качестве прицела. Спрайт должен быть четким и заметным на разных фонах игровой сцены.

1. Добавление и настройка скрипта:

* Создайте новый скрипт C# с именем Cursor, как в вашем примере, и прикрепите его к только что созданному GameObject.
* Скрипт управляет позиционированием прицела и состоянием видимости стандартного курсора мыши.

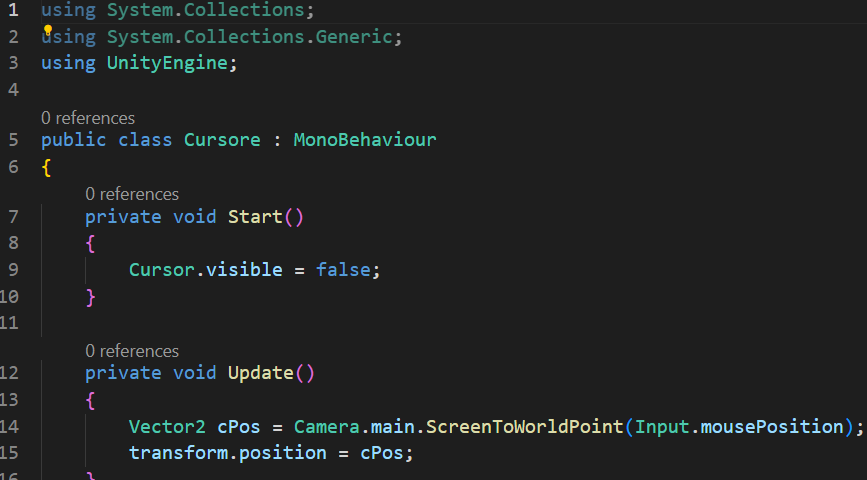


Рис. 95. Код управления прицелом

В методе «Start()» скрипт делает стандартный курсор мыши невидимым, чтобы пользователи видели только ваш прицел.

В методе «Update()», который вызывается каждый кадр, скрипт читает позицию курсора мыши, преобразует её из экранных координат в координаты мира (используя камеру) и перемещает объект прицела в эту позицию. Это обеспечивает, что прицел всегда следует за курсором мыши.

Преимущества

* Улучшенный визуальный контроль: Игроки получают более точный и интуитивно понятный контроль над действиями в игре.
* Повышенная реакция: Мгновенное отображение изменения положения прицела улучшает отзывчивость игры.
* Адаптация под стиль игры: Прицел можно легко адаптировать под визуальный стиль вашей игры, что улучшает общую атмосферу и погружение.

Использование пользовательского прицела улучшает взаимодействие игрока с игрой и повышает общую атмосферность и функциональность игрового процесса. Этот метод особенно ценен в играх, требующих точности и быстрых реакций.

# **Заключение**

На протяжении нашей работы над созданием 2D игры на платформе Unity с использованием C#, мы рассмотрели и реализовали множество важных аспектов игрового дизайна и программирования. Ниже приведено краткое заключение о проделанной работе и изученных темах:

1. Настройка и создание проекта

* Установка и настройка Unity: Вы настроили Unity Hub, установили необходимую версию Unity и создали новый 2D проект.
* Основы Unity Editor: Изучили интерфейс Unity, в том числе вкладки и основные инструменты, такие как Hierarchy, Scene, Game, и Inspector.

1. Разработка игровых механик

* Скриптинг на C#: Написание скриптов для управления персонажами, врагами, и игровыми событиями.
* Управление персонажем: Реализация перемещения, поворота, стрельбы, а также системы здоровья и взаимодействия с объектами игрового мира.
* Враги: Создание врагов с различными поведенческими моделями, включая преследование и атаку.

1. Визуальные и аудио эффекты

* Система частиц: Изучение и использование системы частиц для создания визуальных эффектов, таких как взрывы и трейлы.
* Post Processing и URP: Настройка Universal Render Pipeline для улучшения графики и введение пост-обработки для создания атмосферы.
* Аудио: Разработка звуковой схемы игры, включая эффекты выстрелов, ударов и фоновую музыку с помощью AudioSource и SoundManager.

1. Интерфейс пользователя

* UI элементы: Создание интерфейса пользователя, включая экраны меню, индикаторы здоровья и другие игровые худы.
* Управление сценами: Реализация переходов между сценами и управления состояниями игры.

1. Оптимизация и тестирование

* Профилирование и оптимизация: Использование инструментов Unity для профилирования производительности и оптимизации игры.
* Тестирование: Проведение тестов для выявления и устранения ошибок, улучшение стабильности и производительности игры.

На протяжении проекта я не только реализовали функционал по заданным требованиям, но и значительно углубили свои знания и навыки в области программирования на C#, работы с Unity Engine, и основ игрового дизайна. Я изучили и успешно применили ряд продвинутых техник и инструментов, которые необходимы для создания современных видеоигр. Этот проект помог сформировать твердую основу для будущих разработок и проектов в области игровой индустрии.