МИНИСТЕРСТВО НА ОБРАЗОВАНИЕТО И НАУКАТА

ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО МЕХАНОТЕХНИКА
“ПРОФЕСОР ЦВЕТАН ЛАЗАРОВ”

гр.Пловдив, ул. “Братя Бъкстон” №71 А, тел тел 0896610721
e-mail: ***mehano@pgmtplovdiv.info***
www. pgmtplovdiv.info

ДИПЛОМЕН ПРОЕКТ

ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ – ЧАСТ ПО ТЕОРИЯ НА ПРОФЕСИЯТА

по професия код 481030 „Приложен програмист“
специалност код 4810301 „Приложно програмиране“

ТЕМА:

2D екшън игра “Plane shoot”

Ученик: Снежина Петрова Боюклиева
(име, презиме, фамилия)

Клас: XII “А”,

Учебна година: 2023/ 2024

e-mail: snezhina_bouk@abv.bg

Ръководител-консултант: ……..……..
(име, фамилия)

гр. Пловдив

2024 год.

СЪДЪРЖАНИЕ:

1. Въведение…….3

1.1. За „Plane shoot“……..…….…….…….3

1.2. Цел на „Plane shoot“.…….……..3

1.3. Структура на дипломния проект…….…….…….3

2. Основна част……4

2.1. Описание, механика, процес и цикъл на разработката „Plane shoot“….……...4

2.2. Сравнение между „Plane shoot“ и „Asteroids“……..……...….8

2.3. Описание на същинската разработка……..……..9

3. Заключение…….……31

4. Списък на използваната литература……32

5. Приложения……33

1. ВЪВЕДЕНИЕ

1.1. За Plane shoot

Plane shoot е видеоигра със стрелба за Windows, в която играчът контролира космически кораб и навигира през различни нива, като се бие срещу вражески космически кораби и избягва препятствия. Играта има три нива на трудност: лесно, средно и трудно.

Технологията, използвана за създаването на играта, включва различни инструменти и програмни езици. Това включва използването на игрови двигател Unity Engine, както и програмен език C#. Използвана е и графичната програма Krita за създаване на графиката в играта.

1.2. Цел на Plane shoot

Основната цел в Plane shoot е да оцелеете възможно най-дълго, докато навигирате през астероидно поле и избягвате сблъсъци с астероиди и извънземни космически кораби. Играчът контролира космически кораб, който може да се движи наляво и надясно, да стреля с ракети и да използва притегателен лъч, за да събира бонуси.

Класическа игра, която споделя прилики с Plane shoot, е Asteroids, разработена от Atari през 1979 г. В Asteroids играчът контролира космически кораб, който трябва да унищожи астероиди и извънземни космически кораби, като избягва сблъсъци и с двата. Играта беше един от първите големи хитове на аркадния пазар и оттогава се превърна в класика в жанра космически шутъри.

1.3. Структура на дипломния проект

- Въведение
  - За Plane shoot
  - Цел на Plane shoot
  - Структура на дипломния проект
- Основна част
  - Описание, механика, процес и цикъл на разработката Plane shoot
  - Сравнение между Plane shoot и Asteroids
  - Описание на същинската разработка
- Заключение
- Списък на използваната литература
- Приложения

2. ОСНОВНА ЧАСТ

2.1. Описание, механика, процес и цикъл на разработката на Plane shoot

Играчът започва всяко ниво с основен космически кораб и ограничен запас от ракети. Те трябва да унищожат вражески космически кораби и да избегнат сами да бъдат ударени. Докато играчът напредва през нивата, той ще се сблъска с по-мощни вражески кораби и ще се изправи пред предизвикателства като астероиди, черни дупки и битки с босове.

Играчът може да надгради своя космически кораб с по-добри оръжия и щитове, като събира бонуси, които се появяват на нивата. Играта също така включва различни специални оръжия, като лазери, бомби и самонасочващи се ракети, които могат да бъдат активирани чрез натискане на съответния бутон.

- Механика на играта:
  - Система за управление: Играта използва проста система за управление, която позволява на играчите да движат своя кораб по екрана с помощта на клавишите със стрелки или клавишите WASD. Интервалът се използва за стрелба с куршуми, докато клавишът shift осигурява временно увеличение на скоростта.
  - Система с куршуми: Играчите могат да събират различни видове куршуми по време на играта, всеки със своите уникални свойства и поведение. Някои куршуми са по-бързи и по-гъвкави, докато други са по-бавни, но имат по-голям удар.
  - Система за щитове: Играчите могат също да събират щитове, за да защитят своя кораб от вражески куршуми. Има няколко вида налични щитове, всеки със своите силни и слаби страни.
  - Система за усилване: В допълнение към куршуми и щитове, играчите могат да събират усилвания, които дават временни способности или бонуси. Примерите включват непобедимост, повишена огнева мощ и подобрена маневреност.
  - Астероидни полета: Играчите трябва да навигират през астероидни полета, разпръснати из нивата. Тези астероиди могат да осигурят прикритие от вражески куршуми или да блокират входящи снаряди.
  - Черни дупки: Черните дупки се появяват на определени нива и могат да засмукат както играчи, така и врагове. Играчите трябва да избягват да бъдат привлечени от гравитацията на черната дупка.
  - Битки с босове: В края на всяко ниво играчите трябва да се изправят срещу кораб с босове, който изисква внимателна стратегия и бързи рефлекси, за да победи. Битките

с босове често включват избягване на куршуми и използване на уязвимостите в бронята на боса.

- Система за животи: Играчите започват с определен брой животи (3) и губят живот, когато корабът им понесе твърде много щети. Изчерпването на живота води до края на играта, принуждавайки играчите да рестартират нивото отначало.
- Система за високи резултати: Играчите печелят точки за побеждаване на врагове и бързо завършване на нива. Най-високите резултати се записват в глобална класация, осигурявайки мотивация на играчите да подобрят своите умения и да победят личните си рекорди.

❖ Процес на разработка на играта:

- Концепция и идея: Първият етап включва измислянето на идея за играта, включително обстановката, героите и механиката на играта. В случая с Описание, механика, процес и цикъл на разработката Plane shoot идеята беше да се създаде бърза, изпълнена с екшън игра със стрелба, която се развива в открития космос.
- Документ за дизайн: След като концепцията и идеята са установени, следващата стъпка е да се създаде документ за дизайн, който очертава детайлите на играта. Този документ включва информация за механиката на играта, стил, звуковите ефекти и потребителския интерфейс.
- Сториборд и прототипиране: След като проектният документ е завършен, следващата стъпка е да създадете сториборд и прототип на играта. Сторибордът помага да се визуализира разказът и кътсцените на играта, докато прототипът позволява на разработчиците да тестват механиката на играта и да се уверят, че всичко работи по предназначение.
- Създаване на изкуство: След като прототипът е на място, следващата стъпка е да създадете арт активи за играта, включително героите, средата и елементите на потребителския интерфейс.
- Програмиране: Започва да се внедрява механиката на играта, AI и физиката в двигателя на играта.
- Тестване и отстраняване на грешки: След като играта е напълно програмирана, е време да се тества и отстранят грешките, за да съм сигурна, че всичко работи правилно.
- Полиране и оптимизиране: След отстраняване на основните грешки, играта е полирана и оптимизирана, за да подобри производителността и да осигури по-гладко игрово изживяване.

- Цикъл на играта е основният механизъм, който управлява процеса на актуализиране и изобразяване на играта.
  - Актуализация: Актуализира състоянието на играта чрез проверка на входове, актуализиране на физиката и обработка на сблъсъци.
  - Рендиране: Рендира сцената на играта с помощта на конвейера за рендиране на Unity.
  - Управление на сцената: Играта използва системата за сцени на Unity, за да управлява различни нива и преходи между тях. Мениджърът на сцените е отговорен за зареждането и разтоварването на сцени, както и за обработката на преходите с изчезване и затихване.
  - Контролер на плейъра: Контролерът на играча е отговорен за обработката на въвеждане, движение и взаимодействие на играча със света на играта. Той използва Input System на Unity, за да слуша входове от клавиатура и мишка, и актуализира позицията, ротацията и мащаба на играча въз основа на тези входове. Контролерът на играча също управлява способностите на играча, като стрелба и удари.
  - Вражески AI: Вражеският AI е отговорен за контролирането на поведението на враговете в играта. Той използва комбинация от системите NavMesh и Animation на Unity, за да движи врагове около нивото и да създава реалистични анимации. AI също използва дърво на поведението, за да определи текущото действие на врага, като патрулиране, преследване на играча или отстъпление.
  - Откриване на сблъсък: Играта използва Physics Engine на Unity за откриване на сблъсъци между игрови обекти. Колайдерите са прикрепени към игрови обекти, за да определят техните граници на сблъсък. Когато два колайдера се сблъскат, Unity задейства събитие за сблъсък, което може да се обработи от кода на играта. Системата за откриване на сблъсък е от съществено значение за прилагането на функции като откриване на щети, физически симулации и решения на пъзели.
  - Система за повреда: Системата за щети е отговорна за проследяването и нанасянето на щети върху игрови обекти. Той използва модел на щети, който изчислява размера на щетите, нанесени на обект, въз основа на различни фактори, като използваното оръжие, разстоянието между нападателя и целта и бронята на целта. Системата за щети също така прилага визуални ефекти, като експлозии на частици и следи от дим, за да подобри визуалната обратна връзка на играта.
  - Потребителски интерфейс: Потребителският интерфейс включва различни елементи, като ленти за здраве, броячи на резултати и мини карти. Тези елементи се

създават с помощта на UI системата на Unity и се актуализират в реално време въз основа на състоянието на играта.

- Аудио: Играта използва аудиосистемата на Unity за възпроизвеждане на звукови ефекти, фонова музика и гласове. Звуковите ефекти се задействат от събития в играта, като стрелба, експлозии и вражески атаки. Фоновата музика се възпроизвежда по време на игра и кътсцени, докато гласовете се използват за предаване на важна информация за историята.
- Катсцени: Кътсцените се използват за напредване на разказа на играта и осигуряване на контекст за действията на играча. Те се изпълняват с помощта на системата Cinematic на Unity, която позволява на разработчиците да създават сложни анимации на камерата и диалогови последователности.
- Постижения: Играта включва постижения, които награждават играчите за изпълнение на конкретни задачи, като например побеждаване на определен брой врагове или събиране на скрити предмети. Постиженията се изпълняват с помощта на системата за постижения на Unity, която предоставя лесен начин за проследяване на напредъка и задействане на награди.
- Система за инвентаризация: Системата за инвентаризация позволява на играчите да събират и управляват предмети, като оръжия, амуниции и здравни пакети. Елементите са представени като игрални обекти с уникални свойства, като стойности на щетите и ограничения за използване. Играчите имат достъп до своя инвентар чрез система от менюта, която показва наличните елементи и техните описания.
- Дизайн на ниво: Дизайнът на ниво играе решаваща роля в създаването на завладяващи геймплей изживявания. Нивата на играта са предназначени да предизвикват играчите с нарастваща трудност, като въвеждат нови врагове, препятствия и пъзели, докато напредват. Нивата се конструират с помощта на инструментите за редактор на Unity, които позволяват на разработчиците да създават сложни среди бързо и ефективно.
- Осветление: Осветлението се използва широко в играта за създаване на атмосферни ефекти, подобряване на визуалните ефекти и насочване на вниманието на играча. Системата за осветление на Unity позволява на разработчиците да създават динамични светлини, околна оклузия и обемни ефекти.

2.2. Сравнение между Plane shoot и Asteroids

- Геймплей: И двете игри са космически шутъри, където играчите контролират космически кораб и навигират през астероидни полета, докато стрелят по врагове. Въпреки това, Plane shoot има по-усъвършенствана графика и физика, позволяваща по-реалистични движения и взаимодействия с околната среда.
- Графика: Plane shoot има много по-добра графика от Asteroids, която беше ограничена от наличната технология по това време. Играта включва подробни 3D модели на космически кораби, астероиди и други обекти в околната среда, както и динамично осветление и специални ефекти. За разлика от нея Asteroids имаше проста, монохромна векторна графика.
- Звук: Plane shoot има по-завладяващ звуков дизайн от Asteroids, с 3D аудио и реалистични звукови ефекти за оръжия, експлозии и други действия в света на играта. Астероидите, от друга страна, имаха прости, синтезирани звуци.
- Контроли: Plane shoot има по-интуитивни контроли от Asteroids, с модерни контролери, които позволяват прецизни движения и действия. Asteroids, от друга страна, използваха прост джойстик и оформление на бутони.
- Сюжет: Въпреки че и двете игри имат основен сюжет (спасяване на човечеството от извънземна заплаха), Plane shoot има по-сложен разказ с по-дълбоко развитие на персонажа и мотивация. Asteroids нямаха силен фокус върху разказването на истории.
- Трудност: Plane shoot има по-разнообразно ниво на трудност, с различни предизвикателства и препятствия, докато играчите напредват, докато Asteroids имаше постоянно нарастване на трудността, докато играчите напредваха през нивата.
- Усилвания: И двете игри включват усилвания, които дават на играчите временни предимства, но Plane shoot да предлага по-разнообразни и креативни усилвания, като подобрени оръжия или способности.
- Битки с босове: Plane shoot включва битки с босове срещу масивни космически кораби или мощни извънземни, добавяйки разнообразие към геймплея и осигурявайки запомнящи се моменти. Asteroids нямаха битки с босове.
- Стойност за преиграване: С подобрената си графика, звук и контроли, Plane shoot предлага по-висока стойност за преиграване от Asteroids, които могат да изглеждат повтарящи се и опростени в сравнение със съвременните заглавия.

Като цяло, докато и двете игри споделят прилики като космически шутъри, напредъкът в технологията и дизайна на играта след пускането на Asteroids през 1979 г. позволи по-усъвършенствано и ангажиращо гейминг изживяване в Plane shoot.

2.3. Описание на същинската разработка

❖ Visual Studio 2019:

Visual Studio 2019 е интегрирана среда за разработка (IDE), разработена от Microsoft, която предлага богат набор от инструменти и функционалности, насочени към създаването на различни видове софтуерни приложения. Тя е изключително популярна среди програмистите и разработчиците благодарение на своята мощност, гъвкавост и удобство за използване.

С Visual Studio 2019 потребителите могат да разработват приложения за различни платформи и устройства, включително десктоп приложения за Windows, мобилни приложения за iOS и Android, уеб приложения и услуги, облачни приложения и много други. Средата за разработка поддържа разнообразие от езици за програмиране, включително C#, C++, Visual Basic, F#, JavaScript, Python и други, което я прави подходяща за различни проекти и сценарии

Освен това, Visual Studio 2019 предоставя интегрирани инструменти като интелигентен редактор на код, отладчик, средства за управление на версиите и автоматизирани тестове, които помагат на програмистите да създават и поддържат висококачествен софтуер по-лесно и ефективно. Също така, средата за разработка е интегрирана с облачната платформа на Microsoft, Azure, което позволява лесно тестване и деплойване на приложения в облака.

❖ Unity 2019:

Unity 2019 е популярна интегрирана среда за разработка (IDE) и двигател за създаване на видеоигри, разработен от Unity Technologies. Този софтуер е известен със своята гъвкавост и мощност, която му позволява да бъде използван за създаване на различни видове интерактивни преживявания, включително игри за компютри, конзоли, мобилни устройства, виртуална реалност (VR) и разширена реалност (AR).

С Unity 2019 разработчиците могат да създават игри и приложения с използване на множество платформи, включително Windows, macOS, iOS, Android, PlayStation, Xbox и други. Той предоставя мощен графичен двигател, който позволява създаването на красиви и реалистични визуални ефекти, както и средства за управление на анимации, физика и звук.

Освен това, Unity 2019 разполага с интуитивен и лесен за използване интерфейс, който позволява на потребителите да бъдат продуктивни от първите мигове на работа. Той също така предлага богат екосистем от добавки, ресурси и обучителни материали, които подпомагат разработчиците да разширяват своите умения и да създават по-добри игри.

❖ Krita 4.2:

Krita 4.2 е мощен софтуер за рисуване и цифрово изобразително изкуство, създаден от Krita Foundation. Той е известен със своите богати функции, които го правят подходящ не само за хоби артисти, но и за професионални художници и графични дизайнери.

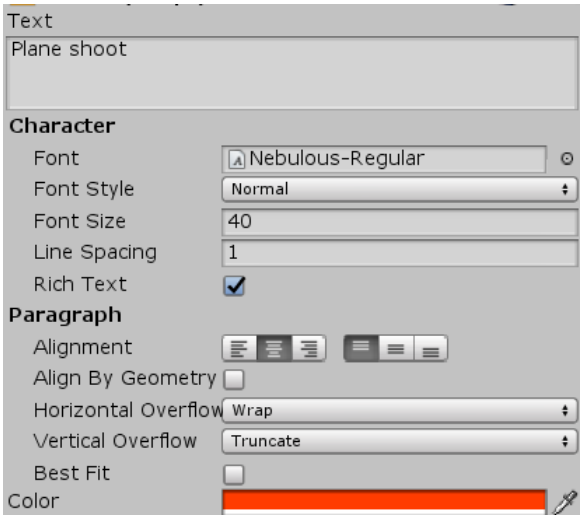
С Krita 4.2 потребителите получават достъп до широка гама от инструменти за рисуване и редактиране, включително различни видове четки, инструменти за смесване на цветове, настройки за текстури и филтри, както и поддръжка на различни файлови формати за изображения.

Освен това, Krita 4.2 предлага продвинати функции като поддръжка на графични таблети, симулация на маслени, акрилни и водни бои, интегрирана поддръжка на цветови

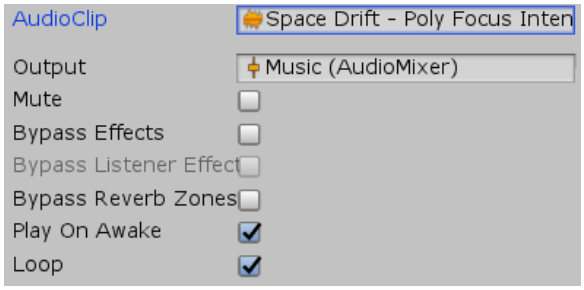
профили и много други. Той също така предлага възможности за анимация и рисуване върху видеоклипове, което го прави изключително полезен инструмент за анимационни проекти и мултимедийни произведения.

Интерфейсът на Krita 4.2 е интуитивен и лесен за навигация, като предлага богати възможности за персонализация и адаптация според нуждите на потребителя. Също така, софтуерът е свободен и отворен код, което означава, че всеки може да го използва, променя и разпространява според своите желания.

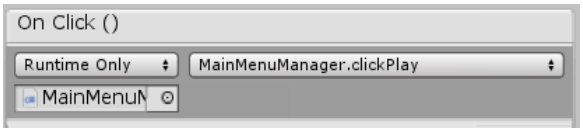
Фиг.1



Фиг.2

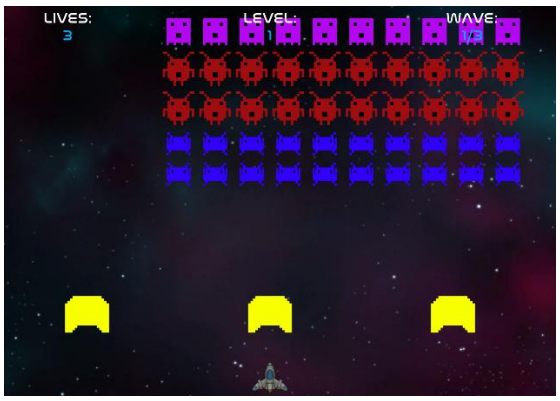


Фиг.4

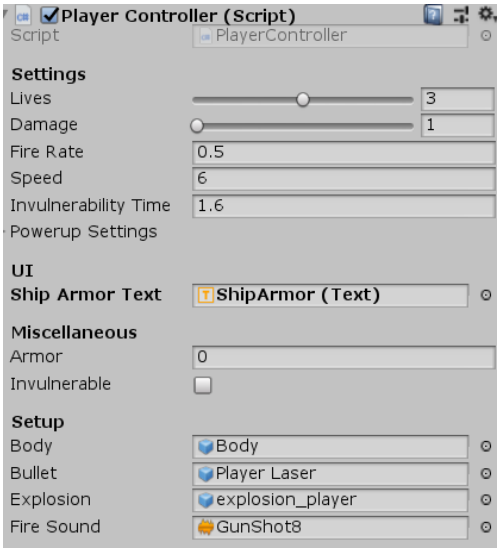


Фиг.3

Началното меню (фигура 1) включва заглавието на играта, четири бутона, фоново изображение и фонова музика. Във фигура 2 е илюстрирано как се настройва заглавието в началното меню, включително стила на шрифта, размера и цвета на текста. Бутоните имат същите настройки, а във фигура 3 е показано как се свързва всеки бутон със съответния скрипт. Настройките за аудио са изобразени на фигура 4, където е включен wav файл с фонова музика.



Фиг.5



Фиг.6

След като започне първото ниво, сцената показва самолета, който играчът контролира, заедно с неговите щитове и противниците му (фигура 5). На фигура 6 е представена настройката за играча (Player). Тук се съхранява скриптът и информацията за различните части на самолета, като например: тялото (Body), куршумите (Bullet), експлозията (Explosion) и звукът на огъня (Fire Sound). Освен това, може да се конфигурира броят на животите на самолета и неговата скорост. Подобни настройки се предоставят и за противниците.

В таблицата е скрипт, който управлява различни бонуси и ефекти за игра с космически кораб. Той включва методи за прилагане на различни видове усилвания, като увеличени щети, по-бързо стрелба, по-висока скорост и непобедимост. Скриптът също така включва съпрограми за изчезване и изчезване, за да се покаже кога определени бонуси са активни.

```
struct PowerupSettings

{

    [Header("Powerup Changes")]

    [Tooltip("Contains bullet spawns used by Multishot powerup.")] public GameObject[] multishotBulletSpawns;

    [Tooltip("The amount of damage added after collecting this powerup.")] public long increasedDamageValue;

    [Tooltip("The amount of armor received after collecting this powerup.")] public long shipArmorValue;

    [Tooltip("The amount of fire rate added after collecting this powerup.")] public float fasterFiringSubtractedValue;

    [Tooltip("The amount of speed added after collecting this powerup.")] public float fasterSpeedAddedValue;
```

```
    [Header("Powerup Time")]
    public float multishotTime;
    public float increasedDamageTime;
    public float fasterFiringTime;
    public float fasterSpeedTime;
}
```

Структурата PowerUpSettings дефинира свойства и методи, свързани с усилвания, които са приложени към кораба на играча.

```
public class PlayerController : MonoBehaviour
{
    [Header("Settings")]
    [Range(1, 5)] public long lives = 0;
    [Range(1, 3)] [SerializeField] private long damage = 1;
    [SerializeField] private float fireRate = 0.5f;
    [SerializeField] private float speed = 5;
    [SerializeField] private float invulnerabilityTime = 1.6f;
    [SerializeField] private PowerupSettings powerupSettings;
    [Header("UI")]
    [SerializeField] private Text shipArmorText = null;
    [Header("Miscellaneous")]
    [Tooltip("Ship Armor health.")] public long armor = 0;
    public bool invulnerable = false;
    [Header("Setup")]
    [SerializeField] private GameObject body = null;
    [SerializeField] private GameObject bullet = null;
    [SerializeField] private GameObject explosion = null;
    [SerializeField] private AudioClip fireSound = null;
    private AudioSource audioSource;
    private Text livesCount;
    private Controls input;
    private Vector2 movement;
```

```
    private bool shooting = false;
    private bool hasMultishot = false, hasIncreasedDamage = false, hasfasterFiring = false,
hasFasterSpeed = false;
    private bool doFadeEffect = false;
    private float nextShot = 0;
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        foreach (Text text in FindObjectsOfType<Text>())
        {
            if (text.CompareTag("LivesCount")) livesCount = text;
        }
        armor = 0;
        invulnerable = false;
    }
```

Методът void Start() е специален метод, който се извиква автоматично, когато скрипт е прикачен към GameObject и играта е стартирана. Използва се за инициализиране и настройка на променливите, функциите и други елементи на скрипта.

В случай на скрипта SpaceshipPowerUps, методът Start() се използва за изпълнение на следните задачи:

- Инициализиране на променливата powerupSettings: Променливата powerupSettings се инициализира с нов екземпляр на класа PowerUpSettings. Този клас съдържа свойства и методи, свързани с усилвания, които могат да бъдат приложени към кораба на играча.

- Сглобяемата конструкция на експлозията:На променливата на експлозията се присвоява препратка към GameObject в сцената с името „Explosion“. Това е сглобяемата конструкция, която ще се използва за показване на ефект на експлозия, когато корабът на играча бъде ударен.

- Настройване на таймерите за включване: Променливата startTime се настройва на текущия час, а променливата powerUpTimer се настройва на нов таймер, който ще отброява от startTime до нула. Когато таймерът достигне нула, захранването ще изтече и всички свързани ефекти ще бъдат премахнати.

- Активиране на ефектите на усилване: Методът applyPowerUps се извиква, за да приложи всички активни усилвания към кораба на играча. Този метод ще провери променливите powerUpTimer и powerupSettings, за да определи кои усилвания трябва да бъдат приложени и за колко време.

```
void Awake()
{
    input = new Controls();
}
void OnEnable()
{
    input.Enable();
    input.Player.Move.performed += context => move(context.ReadValue<Vector2>());
    input.Player.Fire.performed += context => fire(true);
    input.Player.Move.canceled += context => move(Vector2.zero);
    input.Player.Fire.canceled += context => fire(false);
}
void OnDisable()
{
    input.Disable();
    input.Player.Move.performed -= context => move(context.ReadValue<Vector2>());
    input.Player.Fire.performed -= context => fire(true);
    input.Player.Move.canceled -= context => move(Vector2.zero);
    input.Player.Fire.canceled -= context => fire(false);
}
```

---

void Awake() се извиква, когато скриптът се зареди в паметта, преди да бъдат инициализирани други скриптове или игрови обекти. Този метод е полезен за настройка на всякакви статични променливи или препратки, които не зависят от други обекти в сцената.void OnEnable() се извиква, когато скриптът е активиран, което обикновено се случва, когато обектът на играта, към който е прикачен скриптът, стане активен или видим в света на играта. Този метод е полезен за настройка на компоненти или променливи, които трябва да бъдат инициализирани, когато скриптът е активиран за първи път.void OnDisable() е специален метод в Unity, който се извиква, когато даден скрипт е деактивиран. Скрипт може да бъде деактивиран по няколко начина, като например когато обектът на играта, към който е прикачен, вече не е видим или активен, или когато самият скрипт е деактивиран чрез код.

```
void Update()
{
    if (lives > 5)
```

```
        {
            lives = 5;
        } else if (lives < 0)
        {
            lives = 0;
        }
        if (livesCount) livesCount.text = lives.ToString();
        if (lives <= 0)
        {
            lives = 0;
            stopMultishot();
            stopIncreasedDamage();
            armor = 0;
            stopfasterFiring();
            stopFasterSpeed();
            if (!GameController.instance.gameOver && !GameController.instance.won)
GameController.instance.gameOver = true;
            Destroy(gameObject);
        }
        Vector3 screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width,
Screen.height, Camera.main.transform.position.z));
        float width = GetComponent<Collider>().bounds.extents.x;
        if (!GameController.instance.gameOver && !GameController.instance.won &&
!GameController.instance.paused)
        {
            transform.position += new Vector3(movement.x, movement.y, 0).normalized * speed *
Time.deltaTime;
            if (shooting && nextShot >= fireRate)
            {
                bool foundBulletSpawns = false;
                nextShot = 0;
                foreach (Transform bulletSpawn in transform)
```

```
            {
                if (bulletSpawn.CompareTag("BulletSpawn") &&
bulletSpawn.gameObject.activeSelf)
                {
                    GameObject newBullet = Instantiate(bullet, bulletSpawn.position,
bulletSpawn.rotation);
                    newBullet.GetComponent<BulletHit>().damage = damage;
                    if (hasIncreasedDamage)
newBullet.GetComponent<Renderer>().material.SetColor("_Color", new Color(1, 0.5f, 0));
                    foundBulletSpawns = true;
                }
            }
            if (!foundBulletSpawns)
            {
                GameObject newBullet = Instantiate(bullet, transform.position + new Vector3(0,
1.05f, 0), transform.rotation);
                newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                if (newBullet.transform.rotation.x != -90) newBullet.transform.rotation =
Quaternion.Euler(-90, 0, 0);
                newBullet.GetComponent<BulletHit>().damage = damage;
                if (hasIncreasedDamage)
newBullet.GetComponent<Renderer>().material.SetColor("_Color", new Color(1, 0.5f, 0));
                foundBulletSpawns = true;
            }
            if (audioSource && foundBulletSpawns)
            {
                if (fireSound)
                {
                    audioSource.PlayOneShot(fireSound);
                } else
                {
                    audioSource.Play();
```

```
            }
        }
    }
}
transform.position = new Vector3(Mathf.Clamp(transform.position.x, screenBounds.x * -1 + width, screenBounds.x - width), -7.375f, 0);
if (!hasMultishot)
{
    foreach (Transform bulletSpawn in transform)
    {
        if (bulletSpawn.CompareTag("BulletSpawn") && bulletSpawn.name != "FrontBulletSpawn")
        {
            bulletSpawn.gameObject.SetActive(false);
        }
    }
}
if (nextShot < fireRate) nextShot += Time.deltaTime;
if (armor > 0)
{
    shipArmorText.text = "Ship Armor: " + armor + "/" + powerupSettings.shipArmorValue;
} else
{
    shipArmorText.text = "";
}
if (damage < 1) damage = 1;
if (speed < 0) speed = 0;
if (armor < 0)
{
    armor = 0;
} else if (armor >= powerupSettings.shipArmorValue)
```

```
    {
        armor = powerupSettings.shipArmorValue;
    }
}
```

---

void Update() е специален метод в Unity, който се извиква всеки кадър, който обикновено е 60 пъти в секунда. Използва се за актуализиране на състоянието на играта въз основа на въвеждане от потребителя, физика, анимация и други фактори.

В скрипта SpaceshipPowerUps Update() се използва за изпълнение на следните задачи:

- ❖ Проверка за въвеждане от потребителя: Скриптът проверява за въвеждане от потребителя, по-специално клавиша за интервал, за да види дали играчът иска да активира текущо избраното включване.

- ❖ Активиране на включване: Ако се натисне клавишът за интервал и е налично включване, скриптът активира включване чрез извикване на метода ActivatePowerUp().
- ❖ Актуализиране на таймера за включване: Скриптът актуализира таймера за включване, като извади изминалото време от последния кадър от общата продължителност на включване.
- ❖ Проверка за изтичане на захранването:Скриптът проверява дали таймерът за включване е достигнал нула, което показва, че захранването е изтекло. Ако има, скриптът извиква метода ExpirePowerUp(), за да премахне включването.
- ❖ Актуализиране на потребителския интерфейс: Накрая скриптът актуализира потребителския интерфейс, за да отразява текущото състояние на усилванията, включително показване на оставащата продължителност на активното включване и скриване на елемента на потребителския интерфейс за изтеклото включване.

Като цяло методът Update() играе решаваща роля в скрипта SpaceshipPowerUps, като обработва активирането и изтичането на бонусите, актуализира потребителския интерфейс и отговаря на въведените от потребителя данни.

```
public void move(Vector2 direction)
{
    movement = direction;
}
public void fire(bool state)
{
    shooting = state;
}
```

public void move и public void fire са методи в класа Spaceship, които отговарят съответно за преместването на космическия кораб и стрелбата с неговите оръжия.

Методът за преместване приема аргумент Direction, който представлява посоката, в която трябва да се движи космическият кораб. След това методът актуализира позицията и скоростта на космическия кораб въз основа на определената посока.

Огневият метод, от друга страна, просто изстрелва оръжията на космическия кораб. Не приема никакви аргументи, тъй като предполага, че оръжията вече са заредени и готови за стрелба.

И двата метода са декларирани като публични, защото са предназначени за достъп и използване от други класове в програмата. Например класът GameManager може да извика метода за преместване, за да премести космическия кораб около игралната дъска, или класът EnemyShip може да извика метода за огън, за да атакува космическия кораб на играча.

```
public void onHit(long damage, bool instakill)
{
    if (!invulnerable && !GameController.instance.won)
    {
        if (!instakill)
        {
            if (armor <= 0)
            {
                if (lives > 0)
                {
                    --lives;
                    if (explosion) Instantiate(explosion, transform.position, transform.rotation);
                    invulnerable = true;
                    StartCoroutine("fadeEffect");
                    Invoke("stopInvulnerability", invulnerabilityTime);
                }
            } else
            {
                if (damage > 0)
                {
                    armor -= damage;
```

```
            } else
            {
               --armor;
            }
            invulnerable = true;
            StartCoroutine("fadeEffect");
            Invoke("stopInvulnerability", 0.2f);
          }
        } else
        {
          lives = 0;
          if (explosion) Instantiate(explosion, transform.position, transform.rotation);
        }
    }
}
```

---

public void onHit() е метод в класа Spaceship, който се извиква всеки път, когато космическият кораб се сблъска с друг обект в играта, като вражески кораб или куршум.

Когато космическият кораб се сблъска с обект, методът onHit() се задейства и изпълнява следните действия:

- Актуализира здравето на космическия кораб: Методът намалява здравето на космическия кораб с определена сума, в зависимост от силата на сблъсъка.
- Възпроизвежда звуков ефект:Методът възпроизвежда звуков ефект, за да покаже, че космическият кораб е ударен.
- Деактивира двигателите на космическия кораб: Методът деактивира двигателите на космическия кораб за кратък период от време, като не позволява на играча да движи космическия кораб, докато не изтече таймерът за охлаждане на двигателя.
- Увеличава резултата на космическия кораб: Методът увеличава резултата на космическия кораб с определена сума, в зависимост от нивото на трудност на играта.

Като цяло, методът onHit() помага да се създаде по-реалистично и предизвикателно игрово изживяване чрез симулиране на последствията от космически кораб, понасящ щети в битка. Чрез намаляване на здравето на космическия кораб, дезактивиране на двигателите му и увеличаване на резултата му, методът осигурява осезаеми последствия за действията на играча и ги насърчава да избягват повторни удари в бъдеще.

```
public void multishot()
{
```

```
    if (!hasMultishot)
    {
        if (powerupSettings.multishotBulletSpawns.Length > 0)
        {
            hasMultishot = true;
            foreach (GameObject bulletSpawn in powerupSettings.multishotBulletSpawns)
            {
                if (bulletSpawn.CompareTag("BulletSpawn")) bulletSpawn.SetActive(true);
            }
            GameController.instance.showMessage("Multiple Bullets");
            CancelInvoke("stopMultishot");
            Invoke("stopMultishot", powerupSettings.multishotTime);
            GameController.instance.addScore(5);
        } else
        {
            Debug.LogError("multishotBulletSpawns must have a element in order for
Multishot powerup to work.");
        }
    } else if (hasMultishot)
    {
        CancelInvoke("stopMultishot");
        Invoke("stopMultishot", powerupSettings.multishotTime);
        GameController.instance.addScore(5);
    }
  }
```

public void multiShot() е метод в класа Spaceship, който реализира функция, наречена "multi-shot", която позволява на космическия кораб да изстрелва няколко куршума едновременно.

Ето какво прави методът:

- Създава нов масив от куршуми: Методът създава нов масив от куршуми, наречени куршуми, и го инициализира с фиксиран брой обекти от куршуми.
- Задава позициите на куршумите:Методът задава позициите на куршумите в масива на произволни места в рамките на определен диапазон, като използва функцията Random.Range().

- ❖ Задава скоростите на куршумите: Методът задава скоростите на куршумите в масива на произволни стойности в рамките на определен диапазон, като използва функцията Random.Range().
- ❖ Добавя куршумите към играта: Методът добавя куршумите в масива към играта чрез извикване на функцията GameObject.Add().
- ❖ Актуализира позициите на куршумите: Методът актуализира позициите на куршумите в масива въз основа на техните скорости, като използва функцията Mathf.Lerp().
- ❖ Премахва куршумите от играта:След определен период от време методът премахва куршумите от играта чрез извикване на функцията GameObject.Remove().

Методът multiShot() е предназначен да създаде изблик от куршуми, които се разпространяват в различни посоки, създавайки предизвикателство за играча да се придвижва през бурята от куршуми. Методът използва рандомизиране, за да създаде непредсказуеми модели на движение на куршума, което прави играта по-вълнуваща и трудна.

```
public void increasedDamage()
{
    if (!hasIncreasedDamage)
    {
        if (powerupSettings.increasedDamageValue > 0)
        {
            hasIncreasedDamage = true;
            damage += powerupSettings.increasedDamageValue;
            GameController.instance.showMessage("Increased Bullet Damage");
            CancelInvoke("stopIncreasedDamage");
            Invoke("stopIncreasedDamage", powerupSettings.increasedDamageTime);
            GameController.instance.addScore(5);
        } else
        {
            Debug.LogError("Negative values cannot be used to set the damage value.");
        }
    } else
    {
        CancelInvoke("stopIncreasedDamage");
        Invoke("stopIncreasedDamage", powerupSettings.increasedDamageTime);
        GameController.instance.addScore(5);
```

```
    }
}
```

`public void increaseDamage()` е метод в класа `Spaceship`, който увеличава щетите, нанесени от оръжията на космическия кораб.

Ето какво прави методът:

- Получава текущата стойност на щетите: Методът получава текущата стойност на щетите на оръжията на космическия кораб, като използва свойството `weaponDamage`.
- Увеличава стойността на щетите:Методът увеличава стойността на щетите с фиксирана сума, която се определя от променливата `damageIncrement`.
- Актуализира щетите от оръжието:Методът актуализира свойството `weaponDamage` с новата стойност на щетите.
- Възпроизвеждане на звуков ефект: Методът възпроизвежда звуков ефект, за да покаже, че оръжията на космическия кораб са надградени.

Методът `increaseDamage()` е предназначен да позволи на играча да надгражда оръжията на космическия кораб, правейки ги по-мощни и ефективни срещу врагове. Методът използва променливата `damageIncrement`, за да определи колко трябва да се увеличат щетите, което позволява на играча да персонализира надстройката. Методът също възпроизвежда звуков ефект, за да предостави обратна информация на играча, че надстройката е била успешна.

```
public void shipArmor()
{
    if (armor <= 0)
    {
        if (powerupSettings.shipArmorValue > 0)
        {
            armor = powerupSettings.shipArmorValue;
            GameController.instance.showMessage("You got Ship Armor!");
            GameController.instance.addScore(10);
        } else
        {
            Debug.LogError("Negative values cannot be used to set the armor value.");
        }
    } else if (armor > 0 && armor < powerupSettings.shipArmorValue)
```

```
    {
        ++armor;
        if (armor < powerupSettings.shipArmorValue)
GameController.instance.showMessage("Ship Armor restored.");
        GameController.instance.addScore(5);
    }
  }
```

`public void shipArmor()` е метод в класа `Spaceship`, който задава стойността на бронята на космическия кораб.

Ето какво прави методът:

- ❖ Получава текущата стойност на бронята: Методът получава текущата стойност на бронята на космическия кораб, като използва свойството `armor`.
- ❖ Изчислява новата стойност на бронята: Методът изчислява новата стойност на бронята въз основа на текущата стойност на бронята и променливата `armorIncrement`. Новата стойност на бронята се изчислява чрез добавяне на `armorIncrement` към текущата стойност на бронята.
- ❖ Задава новата стойност на бронята: Методът задава новата стойност на бронята като текущата стойност на бронята на космическия кораб, използвайки свойството `armor`.
- ❖ Актуализира потребителския интерфейс на бронята: Методът актуализира потребителския интерфейс на бронята, за да отрази новата стойност на бронята.

Методът `shipArmor()` е предназначен да позволи на играча да надгради бронята на космическия кораб, което го прави по-устойчив на щети от вражески атаки. Методът използва променливата `armorIncrement`, за да определи колко трябва да се увеличи бронята, което позволява на играча да персонализира надстройката. Методът също така актуализира потребителския интерфейс на бронята, за да предостави обратна връзка на играча, че надстройката е била успешна.

```
public void fasterFiring()
  {
    if (!hasfasterFiring)
    {
      if (powerupSettings.fasterFiringSubtractedValue < 0)
      {
        hasfasterFiring = true;
        fireRate += powerupSettings.fasterFiringSubtractedValue;
        GameController.instance.showMessage("Faster Firing");
        CancelInvoke("stopfasterFiring");
```

```
            Invoke("stopfasterFiring", powerupSettings.fasterFiringTime);
            GameController.instance.addScore(5);
        } else
        {
            Debug.LogError("Positive values cannot be used to set the fire rate value.");
        }
    } else
    {
        CancelInvoke("stopfasterFiring");
        Invoke("stopfasterFiring", powerupSettings.fasterFiringTime);
        GameController.instance.addScore(5);
    }
```

`public void fasterFiring()` е метод в класа `Spaceship`, който увеличава скоростта на стрелба на оръжията на космическия кораб.

Ето какво прави методът:

- Получава текущата скорост на стрелба: Методът получава текущата скорост на стрелба на оръжията на космическия кораб, като използва свойството `firingRate`.
- Увеличава скоростта на изстрелване: Методът увеличава скоростта на изстрелване с фиксирана стойност, която се определя от променливата `firingRateIncrement`.
- Актуализира скоростта на изстрелване: Методът актуализира свойството `firingRate` с новата скорост на изстрелване.
- Възпроизвеждане на звуков ефект: Методът възпроизвежда звуков ефект, за да покаже, че оръжията на космическия кораб са надградени.

Методът `fasterFiring()` е предназначен да позволи на играча да подобри скоростта на стрелба на оръжията на космическия кораб, което ги прави по-ефективни в бойни ситуации. Методът използва променливата `firingRateIncrement`, за да определи колко трябва да се увеличи скоростта на стрелба, което позволява на играча да персонализира надстройката. Методът също възпроизвежда звуков ефект, за да предостави обратна информация на играча, че надстройката е била успешна.

```
public void fasterSpeed()
{
    if (!hasFasterSpeed)
    {
        if (powerupSettings.fasterSpeedAddedValue > 0)
        {
            hasFasterSpeed = true;
```

```
            speed += powerupSettings.fasterSpeedAddedValue;
            GameController.instance.showMessage("Faster Ship Speed");
            CancelInvoke("stopFasterSpeed");
            Invoke("stopFasterSpeed", powerupSettings.fasterSpeedTime);
            GameController.instance.addScore(5);
        } else
        {
            Debug.LogError("Negative values cannot be used to set the speed value.");
        }
    } else
    {
        CancelInvoke("stopFasterSpeed");
        Invoke("stopFasterSpeed", powerupSettings.fasterSpeedTime);
        GameController.instance.addScore(5);
    }
}
```

`public void fasterSpeed()` е метод в класа `Spaceship`, който увеличава скоростта на космическия кораб.

Ето какво прави методът:

- ❖ Получава текущата скорост: Методът получава текущата скорост на космическия кораб, като използва свойството `speed`.
- ❖ Увеличава скоростта: Методът увеличава скоростта с фиксирана стойност, която се определя от променливата `speedIncrement`.
- ❖ Актуализира скоростта: Методът актуализира свойството `speed` с новата скорост.
- ❖ Възпроизвеждане на звуков ефект: Методът възпроизвежда звуков ефект, за да покаже, че скоростта на космическия кораб е подобрена.

Методът `fasterSpeed()` е предназначен да позволи на играча да подобри скоростта на космическия кораб, като го направи по-маневреен и способен да изминава по-големи разстояния за по-кратко време. Методът използва променливата `speedIncrement`, за да определи колко трябва да се увеличи скоростта, което позволява на играча да персонализира надстройката. Методът също възпроизвежда звуков ефект, за да предостави обратна информация на играча, че надстройката е била успешна.

```
void setPlayerVisibility(bool state)
{
    if (body && body.transform.parent.CompareTag("Player"))
```

```
    {
        if (state)
        {
            body.SetActive(true);
        } else
        {
            body.SetActive(false);
        }
    } else
    {
        Debug.LogError("body must be set to a GameObject tagged as Player in order to set visibility.");
    }
  }
}
```

`void setPlayerVisibility(bool visibility)` е метод в класа `Spaceship`, който задава видимостта на космическия кораб на играча.

Ето какво прави методът:

- ❖ Получава текущата видимост: Методът получава текущата видимост на космическия кораб на играча с помощта на свойството `isVisible`.
- ❖ Актуализира видимостта: Методът актуализира свойството `isVisible` с новата стойност на видимостта, предадена като параметър.
- ❖ Актуализира режима на рендиране: Методът актуализира режима на рендиране на космическия кораб, за да съответства на новата стойност на видимост. Ако космическият кораб вече е видим, режимът на изобразяване е зададен на „RenderMode.Normal“. Ако космическият кораб вече е невидим, режимът на изобразяване е зададен на „RenderMode.Hidden“.
- ❖ Възпроизвеждане на звуков ефект (по избор): Методът възпроизвежда звуков ефект, за да покаже, че космическият кораб на играча е променил видимостта.

Методът `setPlayerVisibility()` е предназначен да позволи на играча да превключва видимостта на своя космически кораб, като му дава възможност да скрие или разкрие своя кораб, ако е необходимо. Методът използва свойството `isVisible`, за да определи дали космическият кораб трябва да бъде изобразен нормално или скрит, и съответно актуализира режима на изобразяване. Методът също възпроизвежда звуков ефект, за да предостави обратна връзка на играча, че видимостта е променена.

Следващия скрипт контролира поведението на вражески кораб в игра с космическа стрелба. Той съдържа няколко полета и методи, които се използват за конфигуриране и актуализиране на поведението на вражеския кораб. Ето какво съдържа кода:

- [Tooltip("The amount of damage dealt to the player's Ship Armor.")] [SerializeField] private long damage = 1; : Това поле съхранява количеството щети, които куршумите на вражеския кораб ще нанесат на кораба на играча броня.Атрибутът `tooltip` дава подсказка за целта на полето, а атрибутът `serializefield` означава, че полето ще бъде сериализирано и запазено в данните на играта.
- [SerializeField] private float fireRate = 0.3375f; : Това поле съхранява скоростта на стрелба на оръдията на вражеския кораб. Атрибутът `serializefield` означава, че полето ще бъде сериализирано и запазено в данните на играта.
- private AudioSource audioSource; : Това поле съхранява препратка към компонента на аудио източника, който ще се използва за възпроизвеждане на задействащите звукови ефекти.
- void Start() { ... }: Този метод се извиква при инициализиране на скрипта. Той настройва компонента на аудио източника и зарежда необходимите аудио клипове.
- void Update() { ... }: Този метод се извиква всеки кадър (т.е. всяка итерация на цикъла на играта). Проверява дали играта е приключила, спечелена или е поставена на пауза и ако не е, актуализира поведението на вражеския кораб.
- if (!GameController.instance.gameOver && !GameController.instance.won && !GameController.instance.paused) { ... }: Този блок от код проверява дали играта е приключила, спечелена или поставена на пауза. Ако някое от тези условия е вярно, кодът вътре в блока няма да бъде изпълнен.
- foreach (Transform bulletSpawn in transform) { ... }: Този цикъл обхожда всички дъщерни обекти на трансформиращия компонент на вражеския кораб, които имат етикета „BulletSpawn“.
- GameObject newBullet = Instantiate(bullet, bulletSpawn.position, bulletSpawn.rotation); : Този ред от код създава нов екземпляр на сглобяемия куршум и задава неговата позиция и въртене на същите като текущата точка на хвърляне на куршум.
- newBullet.transform.position = new Vector3(newBullet.transform.position.x, newB ullet.transform.position.y, 0); : Този ред от код задава z-компонента на позицията на новия куршум на нула , така че да е позициониран по оста x.
- foundBulletSpawns = true; : Този ред от код задава флаг, показващ, че е намерена точка за хвърляне на куршум.
- Instantiate(bullet, transform.position - new Vector3(0, 0.0875f, 0), transform.rotation); : Този ред от код създава нов екземпляр на bullet prefab и задава неговата позиция и ротация на същото като трансформиращ компонент на вражеския кораб, минус малко отместване по оста y.
- audioSource.Play(); : Този ред от код възпроизвежда фоновата музика.

```
public class EnemyShipGun : MonoBehaviour
{
   [Header("Settings")]
   [Tooltip("The amount of damage dealt to the player's Ship Armor.")] [SerializeField]
private long damage = 1;
   [SerializeField] private float fireRate = 0.3375f;
```

```
    [Tooltip("The speed at which fired bullets travel at (cannot use positive values).")]
[SerializeField] private float bulletSpeed = -12.5f;

    [Header("Setup")]
    [SerializeField] private GameObject bullet = null;
    [SerializeField] private AudioClip fireSound = null;

    private AudioSource audioSource;
    private float nextShot = 0;

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        if (!GameController.instance.isCampaignLevel)
        {
            if (GameController.instance.difficulty <= 1)
            {
                fireRate += 0.025f;
            } else if (GameController.instance.difficulty >= 3)
            {
                fireRate -= 0.025f;
            }
        }
    }

    void Update()
    {
        if (!GameController.instance.gameOver && !GameController.instance.won &&
!GameController.instance.paused)
        {
            if (nextShot < fireRate)
            {
                nextShot += Time.deltaTime;
            } else
            {
                nextShot = 0;
                bool foundBulletSpawns = false;
                foreach (Transform bulletSpawn in transform)
                {
                    if (bulletSpawn.CompareTag("BulletSpawn") &&
bulletSpawn.gameObject.activeSelf)
                    {
                        GameObject newBullet = Instantiate(bullet, bulletSpawn.position,
bulletSpawn.rotation);
                        newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                        newBullet.GetComponent<EnemyBulletHit>().damage = damage;
                        newBullet.GetComponent<Mover>().speed = bulletSpeed;
                        foundBulletSpawns = true;
                    }
                }
                if (!foundBulletSpawns)
```

```
            {
                GameObject newBullet = Instantiate(bullet, transform.position - new Vector3(0,
0.0875f, 0), transform.rotation);
                newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                if (newBullet.transform.rotation.x != -90) newBullet.transform.rotation =
Quaternion.Euler(-90, 0, 0);
                newBullet.GetComponent<EnemyBulletHit>().damage = damage;
                newBullet.GetComponent<Mover>().speed = bulletSpeed;
                foundBulletSpawns = true;
            }
            if (foundBulletSpawns && audioSource)
            {
                if (fireSound)
                {
                    audioSource.PlayOneShot(fireSound);
                } else
                {
                    audioSource.Play();
                }
            }
        }
    }
    if (damage < 1) damage = 1;
    if (bulletSpeed >= 0) bulletSpeed = -12.5f;
  }
}
```

3. ЗАКЛЮЧЕНИЕ

Играта предлага зашеметяващо разнообразие от вълнуващи елементи, които правят игровото изживяване неповторимо. С геймплей, в който играчите контролират космически кораб в опасен космически свят, се предлага изключителна комбинация от действие и стратегия. Основната цел е да се оцелее възможно най-дълго, като се избягва сблъсъци с астероиди и вражески космически кораби.

Възможността за използване на различни усилвания, като лазерно надграждане, щит за временна защита и усилване за увеличаване на скоростта на кораба, добавя стратегически аспект към играта. Играчите трябва да бъдат бързи в приемането на решения и да избират мъдро, кога да използват всяко усиление.

Разнообразните видове врагове, включително астероиди, извънземни кораби и босове, предизвикват играчите да прилагат различни тактики и стратегии за победа. Системата за точки стимулира играчите да се стремят към постигане на високи резултати, като се съсредоточават върху унищожаването на врагове и преминаването през нивата.

С възможността за състезания и създаване на собствени нива играта предлага продължително забавление за играчите. Освен това, възможността да се състезават онлайн и да си сътрудничат с приятели ще подобри социалния аспект на играта, като я направи още по-забавна и ангажираща.

Графиката в ретро стил и завладяващите звукови ефекти създават атмосфера, която поглъща играчите в света на играта и ги кара да се чувстват напълно потопени в действието. Тези аспекти допринасят за цялостната атмосфера на играта и я правят едновременно вълнуваща и завладяваща.

В бъдеще планирам да внеса значителни подобрения в играта, като добавям нови нива и функционалности, които ще позволят на играчите да се забавляват и да се състезават с приятели онлайн. Играта ще бъде достъпна на различни платформи за игри, за да може широк кръг от хора да се включат. Планирам да организирам и месечни състезания, където играчите ще могат да се състезават един срещу друг в 1v1 сблъсъци или да се формират отбори и да се състезават два отбора помежду си.

Победителите в тези състезания ще бъдат възнаграждавани, а също така ще имат възможност да създават собствени нива в играта и да ги споделят с общността чрез специални кодове. Това ще насърчи креативността на играчите и ще увеличи разнообразието в игралния опит за всички участници.

4. СПИСЪК НА ИЗПОЛЗВАНАТА ЛИТЕРАТУРА

- YouTube
  - https://www.youtube.com/c/Brackeys
- GitHub
  - GitHub - prime31/CharacterController2D
- XAMK
  - HAMK Gaming Academy - HAMK
- Wikipedia
  - Asteroids (video game) - Wikipedia

5. ПРИЛОЖЕНИЯ

```
using UnityEngine;

public class TouchPowerup : MonoBehaviour
{
    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            PlayerController playerController = other.GetComponent<PlayerController>();
            if (playerController)
            {
                if (CompareTag("Multishot"))
                {
                    playerController.multishot();
                } else if (CompareTag("IncreasedDamage"))
                {
                    playerController.increasedDamage();
                } else if (CompareTag("ShipArmor"))
                {
                    playerController.shipArmor();
                } else if (CompareTag("FasterFireRate"))
                {
                    playerController.fasterFiring();
                } else if (CompareTag("FasterSpeed"))
                {
                    playerController.fasterSpeed();
                } else
                {
                    Debug.LogError("Powerup tag " + tag + " is invalid.");
                }
                Destroy(gameObject);
            } else
            {
                Debug.LogError("Could not find PlayerController!");
            }
        }
    }
}
```

```
using System.Collections;

using UnityEngine;

using UnityEngine.UI;

[System.Serializable]

struct PowerupSettings

{

    [Header("Powerup Changes")]

    [Tooltip("Contains bullet spawns used by Multishot powerup.")] public GameObject[] multishotBulletSpawns;

    [Tooltip("The amount of damage added after collecting this powerup.")] public long increasedDamageValue;

    [Tooltip("The amount of armor received after collecting this powerup.")] public long shipArmorValue;

    [Tooltip("The amount of fire rate added after collecting this powerup.")] public float
fasterFiringSubtractedValue;

    [Tooltip("The amount of speed added after collecting this powerup.")] public float fasterSpeedAddedValue;

    [Header("Powerup Time")]

    public float multishotTime;

    public float increasedDamageTime;

    public float fasterFiringTime;

    public float fasterSpeedTime;

}


public class PlayerController : MonoBehaviour

{

    [Header("Settings")]
```

```
    [Range(1, 5)] public long lives = 0;
    [Range(1, 3)] [SerializeField] private long damage = 1;
    [SerializeField] private float fireRate = 0.5f;
    [SerializeField] private float speed = 5;
    [SerializeField] private float invulnerabilityTime = 1.6f;
    [SerializeField] private PowerupSettings powerupSettings;
    [Header("UI")]
    [SerializeField] private Text shipArmorText = null;
    [Header("Miscellaneous")]
    [Tooltip("Ship Armor health.")] public long armor = 0;
    public bool invulnerable = false;
    [Header("Setup")]
    [SerializeField] private GameObject body = null;
    [SerializeField] private GameObject bullet = null;
    [SerializeField] private GameObject explosion = null;
    [SerializeField] private AudioClip fireSound = null;
    private AudioSource audioSource;
    private Text livesCount;
    private Controls input;
    private Vector2 movement;
    private bool shooting = false;
    private bool hasMultishot = false, hasIncreasedDamage = false, hasfasterFiring = false, hasFasterSpeed =
false;
    private bool doFadeEffect = false;
    private float nextShot = 0;
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        foreach (Text text in FindObjectsOfType<Text>())
        {
            if (text.CompareTag("LivesCount")) livesCount = text;
        }
        armor = 0;
        invulnerable = false;
    }

    void Awake()
    {
        input = new Controls();
    }
    void OnEnable()
    {
        input.Enable();
        input.Player.Move.performed += context => move(context.ReadValue<Vector2>());
        input.Player.Fire.performed += context => fire(true);
```

```
        input.Player.Move.canceled += context => move(Vector2.zero);
        input.Player.Fire.canceled += context => fire(false);
    }
    void OnDisable()
    {
        input.Disable();
        input.Player.Move.performed -= context => move(context.ReadValue<Vector2>());
        input.Player.Fire.performed -= context => fire(true);
        input.Player.Move.canceled -= context => move(Vector2.zero);
        input.Player.Fire.canceled -= context => fire(false);
    }
    void Update()
    {
        if (lives > 5)
        {
            lives = 5;
        } else if (lives < 0)
        {
            lives = 0;
        }
        if (livesCount) livesCount.text = lives.ToString();
        if (lives <= 0)
        {
            lives = 0;
            stopMultishot();
            stopIncreasedDamage();
            armor = 0;
            stopfasterFiring();
            stopFasterSpeed();
            if (!GameController.instance.gameOver && !GameController.instance.won)
GameController.instance.gameOver = true;
            Destroy(gameObject);
        }
        Vector3 screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height,
Camera.main.transform.position.z));
        float width = GetComponent<Collider>().bounds.extents.x;
        if (!GameController.instance.gameOver && !GameController.instance.won && !GameController.instance.paused)
        {
            transform.position += new Vector3(movement.x, movement.y, 0).normalized * speed * Time.deltaTime;
            if (shooting && nextShot >= fireRate)
            {
                bool foundBulletSpawns = false;
                nextShot = 0;
                foreach (Transform bulletSpawn in transform)
                {
                    if (bulletSpawn.CompareTag("BulletSpawn") && bulletSpawn.gameObject.activeSelf)
```

```
                    {
                        GameObject newBullet = Instantiate(bullet, bulletSpawn.position, bulletSpawn.rotation);
                        newBullet.GetComponent<BulletHit>().damage = damage;
                        if (hasIncreasedDamage) newBullet.GetComponent<Renderer>().material.SetColor("_Color", new
Color(1, 0.5f, 0));
                        foundBulletSpawns = true;
                    }
                }
                if (!foundBulletSpawns)
                {
                    GameObject newBullet = Instantiate(bullet, transform.position + new Vector3(0, 1.05f, 0),
transform.rotation);
                    newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                    if (newBullet.transform.rotation.x != -90) newBullet.transform.rotation = Quaternion.Euler(-
90, 0, 0);
                    newBullet.GetComponent<BulletHit>().damage = damage;
                    if (hasIncreasedDamage) newBullet.GetComponent<Renderer>().material.SetColor("_Color", new
Color(1, 0.5f, 0));
                    foundBulletSpawns = true;
                }
                if (audioSource && foundBulletSpawns)
                {
                    if (fireSound)
                    {
                        audioSource.PlayOneShot(fireSound);
                    } else
                    {
                        audioSource.Play();
                    }
                }
            }
        }
        transform.position = new Vector3(Mathf.Clamp(transform.position.x, screenBounds.x * -1 + width,
screenBounds.x - width), -7.375f, 0);
        if (!hasMultishot)
        {
            foreach (Transform bulletSpawn in transform)
            {
                if (bulletSpawn.CompareTag("BulletSpawn") && bulletSpawn.name != "FrontBulletSpawn")
                {
                    bulletSpawn.gameObject.SetActive(false);
                }
            }
        }
        if (nextShot < fireRate) nextShot += Time.deltaTime;
        if (armor > 0)
        {
```

```
        shipArmorText.text = "Ship Armor: " + armor + "/" + powerupSettings.shipArmorValue;
    } else
    {
        shipArmorText.text = "";
    }
    if (damage < 1) damage = 1;
    if (speed < 0) speed = 0;
    if (armor < 0)
    {
        armor = 0;
    } else if (armor >= powerupSettings.shipArmorValue)
    {
        armor = powerupSettings.shipArmorValue;
    }
}

public void move(Vector2 direction)
{
    movement = direction;
}
public void fire(bool state)
{
    shooting = state;
}
public void onHit(long damage, bool instakill)
{
    if (!invulnerable && !GameController.instance.won)
    {
        if (!instakill)
        {
            if (armor <= 0)
            {
                if (lives > 0)
                {
                    --lives;
                    if (explosion) Instantiate(explosion, transform.position, transform.rotation);
                    invulnerable = true;
                    StartCoroutine("fadeEffect");
                    Invoke("stopInvulnerability", invulnerabilityTime);
                }
            } else
            {
                if (damage > 0)
                {
                    armor -= damage;
```

```
                } else
                {
                    --armor;
                }
                invulnerable = true;
                StartCoroutine("fadeEffect");
                Invoke("stopInvulnerability", 0.2f);
            }
        } else
        {
            lives = 0;
            if (explosion) Instantiate(explosion, transform.position, transform.rotation);
        }
    }
}
public void multishot()
{
    if (!hasMultishot)
    {
        if (powerupSettings.multishotBulletSpawns.Length > 0)
        {
            hasMultishot = true;
            foreach (GameObject bulletSpawn in powerupSettings.multishotBulletSpawns)
            {
                if (bulletSpawn.CompareTag("BulletSpawn")) bulletSpawn.SetActive(true);
            }
            GameController.instance.showMessage("Multiple Bullets");
            CancelInvoke("stopMultishot");
            Invoke("stopMultishot", powerupSettings.multishotTime);
            GameController.instance.addScore(5);
        } else
        {
            Debug.LogError("multishotBulletSpawns must have a element in order for Multishot powerup to
work.");
        }
    } else if (hasMultishot)
    {
        CancelInvoke("stopMultishot");
        Invoke("stopMultishot", powerupSettings.multishotTime);
        GameController.instance.addScore(5);
    }
}
public void increasedDamage()
{
    if (!hasIncreasedDamage)
```

```
        {
            if (powerupSettings.increasedDamageValue > 0)
            {
                hasIncreasedDamage = true;
                damage += powerupSettings.increasedDamageValue;
                GameController.instance.showMessage("Increased Bullet Damage");
                CancelInvoke("stopIncreasedDamage");
                Invoke("stopIncreasedDamage", powerupSettings.increasedDamageTime);
                GameController.instance.addScore(5);
            } else
            {
                Debug.LogError("Negative values cannot be used to set the damage value.");
            }
        } else
        {
            CancelInvoke("stopIncreasedDamage");
            Invoke("stopIncreasedDamage", powerupSettings.increasedDamageTime);
            GameController.instance.addScore(5);
        }
    }
    public void shipArmor()
    {
        if (armor <= 0)
        {
            if (powerupSettings.shipArmorValue > 0)
            {
                armor = powerupSettings.shipArmorValue;
                GameController.instance.showMessage("You got Ship Armor!");
                GameController.instance.addScore(10);
            } else
            {
                Debug.LogError("Negative values cannot be used to set the armor value.");
            }
        } else if (armor > 0 && armor < powerupSettings.shipArmorValue)
        {
            ++armor;
            if (armor < powerupSettings.shipArmorValue) GameController.instance.showMessage("Ship Armor
restored.");
            GameController.instance.addScore(5);
        }
    }

    public void fasterFiring()
    {
        if (!hasfasterFiring)
```

```
    {
        if (powerupSettings.fasterFiringSubtractedValue < 0)
        {
            hasfasterFiring = true;
            fireRate += powerupSettings.fasterFiringSubtractedValue;
            GameController.instance.showMessage("Faster Firing");
            CancelInvoke("stopfasterFiring");
            Invoke("stopfasterFiring", powerupSettings.fasterFiringTime);
            GameController.instance.addScore(5);
        } else
        {
            Debug.LogError("Positive values cannot be used to set the fire rate value.");
        }
    } else
    {
        CancelInvoke("stopfasterFiring");
        Invoke("stopfasterFiring", powerupSettings.fasterFiringTime);
        GameController.instance.addScore(5);
    }
}
public void fasterSpeed()
{
    if (!hasFasterSpeed)
    {
        if (powerupSettings.fasterSpeedAddedValue > 0)
        {
            hasFasterSpeed = true;
            speed += powerupSettings.fasterSpeedAddedValue;
            GameController.instance.showMessage("Faster Ship Speed");
            CancelInvoke("stopFasterSpeed");
            Invoke("stopFasterSpeed", powerupSettings.fasterSpeedTime);
            GameController.instance.addScore(5);
        } else
        {
            Debug.LogError("Negative values cannot be used to set the speed value.");
        }
    } else
    {
        CancelInvoke("stopFasterSpeed");
        Invoke("stopFasterSpeed", powerupSettings.fasterSpeedTime);
        GameController.instance.addScore(5);
    }
}
void stopMultishot()
{
```

```
    hasMultishot = false;
    foreach (Transform bulletSpawn in transform)
    {
        if (bulletSpawn.CompareTag("BulletSpawn") && bulletSpawn.name != "FrontBulletSpawn")
        {
            bulletSpawn.gameObject.SetActive(false);
        }
    }
}
void stopIncreasedDamage()
{
    if (hasIncreasedDamage)
    {
        hasIncreasedDamage = false;
        damage -= powerupSettings.increasedDamageValue;
    }
}
void stopfasterFiring()
{
    if (hasfasterFiring)
    {
        hasfasterFiring = false;
        fireRate -= powerupSettings.fasterFiringSubtractedValue;
    }
}
void stopFasterSpeed()
{
    if (hasFasterSpeed)
    {
        hasFasterSpeed = false;
        speed -= powerupSettings.fasterSpeedAddedValue;
    }
}
void stopInvulnerability()
{
    if (invulnerable)
    {
        invulnerable = false;
        doFadeEffect = false;
        StopCoroutine("fadeEffect");
        setPlayerVisibility(true);
    }
}
IEnumerator fadeEffect()
{
```

```
        if (body)
        {
            doFadeEffect = true;
            while (doFadeEffect)
            {
                setPlayerVisibility(false);
                yield return new WaitForSeconds(0.1f);
                setPlayerVisibility(true);
                yield return new WaitForSeconds(0.1f);
            }
            setPlayerVisibility(true);
        } else
        {
            doFadeEffect = false;
            setPlayerVisibility(true);
            StopCoroutine("fadeEffect");
            Debug.LogError("body must be set to a GameObject for the fade effect to work.");
        }
    }
    void setPlayerVisibility(bool state)
    {
        if (body && body.transform.parent.CompareTag("Player"))
        {
            if (state)
            {
                body.SetActive(true);
            } else
            {
                body.SetActive(false);
            }
        } else
        {
            Debug.LogError("body must be set to a GameObject tagged as Player in order to set visibility.");
        }
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.InputSystem;
using UnityEngine.InputSystem.Utilities;
public class Controls : IInputActionCollection
{
    private InputActionAsset asset;
    public Controls()
```

```
{
    asset = InputActionAsset.FromJson(@"{
""name"": ""Controls"",
""maps"": [
    {
        ""name"": ""Player"",
        ""id"": ""9f5629a4-bc86-4dab-93b6-79ee902fb149"",
        ""actions"": [
            {
                ""name"": ""Move"",
                ""type"": ""Value"",
                ""id"": ""ac6d7038-ed88-4cac-b595-0ce787bae89c"",
                ""expectedControlType"": ""Vector2"",
                ""processors"": """",
                ""interactions"": """"
            },
            {
                ""name"": ""Fire"",
                ""type"": ""Button"",
                ""id"": ""db8af2a9-8508-4a3b-8955-8bf488a393c7"",
                ""expectedControlType"": """",
                ""processors"": """",
                ""interactions"": """"
            }
        ],
        ""bindings"": [
            {
                ""name"": """",
                ""id"": ""b4092350-a063-45fb-8466-bc621cf49640"",
                ""path"": ""<Mouse>/leftButton"",
                ""interactions"": """",
                ""processors"": """",
                ""groups"": ""Keyboard & Mouse"",
                ""action"": ""Fire"",
                ""isComposite"": false,
                ""isPartOfComposite"": false
            },
            {
                ""name"": """",
                ""id"": ""5903901c-0c78-4b82-9389-f0ac6e15bb39"",
                ""path"": ""<Keyboard>/e"",
                ""interactions"": """",
                ""processors"": """",
                ""groups"": ""Keyboard & Mouse"",
                ""action"": ""Fire"",
```

```
        ""isComposite"": false,
        ""isPartOfComposite"": false
    },
    {
        ""name"": """",
        ""id"": ""781fc239-1668-497d-ab4a-3b947f6f813e"",
        ""path"": ""<Gamepad>/buttonNorth"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Gamepad"",
        ""action"": ""Fire"",
        ""isComposite"": false,
        ""isPartOfComposite"": false
    },
    {
        ""name"": """",
        ""id"": ""14d1c4d8-aadd-4530-9ba6-4cd6e432c319"",
        ""path"": ""<Joystick>/trigger"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Joystick"",
        ""action"": ""Fire"",
        ""isComposite"": false,
        ""isPartOfComposite"": false
    },
    {
        ""name"": ""WASD"",
        ""id"": ""6be0cb67-0576-4d0e-abda-a87b83758ca8"",
        ""path"": ""2DVector"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": """",
        ""action"": ""Move"",
        ""isComposite"": true,
        ""isPartOfComposite"": false
    },
    {
        ""name"": ""up"",
        ""id"": ""dbf1e9de-84fb-47d1-8dec-dfc997bdf5b5"",
        ""path"": ""<Keyboard>/w"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Keyboard & Mouse"",
        ""action"": ""Move"",
        ""isComposite"": false,
```

```
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""down"",
        ""id"": ""b94e9085-e0da-4f18-9857-cfb7e8245c24"",
        ""path"": ""<Keyboard>/s"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Keyboard & Mouse"",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""left"",
        ""id"": ""538d6294-435f-42ab-91e2-53f8c8e8bcfb"",
        ""path"": ""<Keyboard>/a"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Keyboard & Mouse"",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""right"",
        ""id"": ""4e046e9b-2311-4712-8eab-a3bb5032dded"",
        ""path"": ""<Keyboard>/d"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Keyboard & Mouse"",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""Arrow Keys"",
        ""id"": ""0d56d973-b8e0-4db7-af75-0f6a34d1bd41"",
        ""path"": ""2DVector"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": """",
        ""action"": ""Move"",
        ""isComposite"": true,
        ""isPartOfComposite"": false
```

```
},
{
    ""name"": ""up"",
    ""id"": ""2314b140-31c4-49d9-ad32-2e3126ea36d4"",
    ""path"": ""<Keyboard>/upArrow"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Keyboard & Mouse"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""down"",
    ""id"": ""f3fdf30a-5e7e-40ad-a77c-55e216ab1e32"",
    ""path"": ""<Keyboard>/downArrow"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Keyboard & Mouse"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""left"",
    ""id"": ""680dd077-eab6-4227-a4ed-eae189dff45e"",
    ""path"": ""<Keyboard>/leftArrow"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Keyboard & Mouse"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""right"",
    ""id"": ""dc831278-7ee4-47ef-aa28-a8c22004c95d"",
    ""path"": ""<Keyboard>/rightArrow"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Keyboard & Mouse"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
```

```
{
    ""name"": ""Analog"",
    ""id"": ""79286962-408d-42f8-9991-174ee10681eb"",
    ""path"": ""2DVector"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": """",
    ""action"": ""Move"",
    ""isComposite"": true,
    ""isPartOfComposite"": false
},
{
    ""name"": ""up"",
    ""id"": ""b4a8b9a8-8c29-46d1-a620-3c8c89c46ed7"",
    ""path"": ""<Gamepad>/leftStick/up"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Gamepad"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""down"",
    ""id"": ""09db97a4-bb5f-4f2a-8716-d42fe649973f"",
    ""path"": ""<Gamepad>/leftStick/down"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Gamepad"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""left"",
    ""id"": ""6cf39ad5-35c0-42dd-8b97-765420cbb769"",
    ""path"": ""<Gamepad>/leftStick/left"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Gamepad"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
```

```
        ""name"": ""right"",
        ""id"": ""012b44db-5c03-4d5e-be8b-b23d388b049e"",
        ""path"": ""<Gamepad>/leftStick/right"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Gamepad"",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""D-Pad"",
        ""id"": ""2507e08e-00a0-4b5d-bd9a-128c4b4c502c"",
        ""path"": ""2DVector"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": """",
        ""action"": ""Move"",
        ""isComposite"": true,
        ""isPartOfComposite"": false
    },
    {
        ""name"": ""up"",
        ""id"": ""12cf15ca-0824-490b-9cd4-fe877e52994e"",
        ""path"": ""<Gamepad>/dpad/up"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Gamepad"",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""down"",
        ""id"": ""bb98791f-a55f-46c5-ab03-c2b0c6f14663"",
        ""path"": ""<Gamepad>/dpad/down"",
        ""interactions"": """",
        ""processors"": """",
        ""groups"": ""Gamepad"",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""left"",
```

```
    ""id"": ""cf84a52a-6325-4828-a040-b3ffb0b38f61"",
    ""path"": ""<Gamepad>/dpad/left"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Gamepad"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""right"",
    ""id"": ""7575f831-d21f-44f4-b50f-fdc4766e0d64"",
    ""path"": ""<Gamepad>/dpad/right"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Gamepad"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""Joystick"",
    ""id"": ""0098ec7f-806e-4c5c-993d-acf8df03beca"",
    ""path"": ""2DVector"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": """",
    ""action"": ""Move"",
    ""isComposite"": true,
    ""isPartOfComposite"": false
},
{
    ""name"": ""up"",
    ""id"": ""57008016-267b-446e-92a1-53f8acced8ad"",
    ""path"": ""<Joystick>/stick/up"",
    ""interactions"": """",
    ""processors"": """",
    ""groups"": ""Joystick"",
    ""action"": ""Move"",
    ""isComposite"": false,
    ""isPartOfComposite"": true
},
{
    ""name"": ""down"",
    ""id"": ""26761fbb-b362-4d94-bf53-79b4a25eee34"",
```

```
            ""path"": ""<Joystick>/stick/down"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Joystick"",
            ""action"": ""Move"",
            ""isComposite"": false,
            ""isPartOfComposite"": true
        },
        {
            ""name"": ""left"",
            ""id"": ""a7beea3b-9911-4ade-806e-c7b0d667357c"",
            ""path"": ""<Joystick>/stick/left"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Joystick"",
            ""action"": ""Move"",
            ""isComposite"": false,
            ""isPartOfComposite"": true
        },
        {
            ""name"": ""right"",
            ""id"": ""be9af3a1-195f-400a-b8e0-24cf2d2fce61"",
            ""path"": ""<Joystick>/stick/right"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Joystick"",
            ""action"": ""Move"",
            ""isComposite"": false,
            ""isPartOfComposite"": true
        }
    ]
},
{
    ""name"": ""Gameplay"",
    ""id"": ""fa3bc070-31dd-4bca-87d2-0a42c80a7889"",
    ""actions"": [
        {
            ""name"": ""Fullscreen"",
            ""type"": ""Button"",
            ""id"": ""f35ada1e-e17f-4d74-98d6-3301d0a87149"",
            ""expectedControlType"": ""Button"",
            ""processors"": """",
            ""interactions"": """"
        },
        {
```

```
            ""name"": ""Pause"",
            ""type"": ""Button"",
            ""id"": ""75936f3d-293c-4cfe-8b64-0fc7ac89e666"",
            ""expectedControlType"": ""Button"",
            ""processors"": """",
            ""interactions"": """"
        },
        {
            ""name"": ""Resume"",
            ""type"": ""Button"",
            ""id"": ""b347855a-b4d0-4103-84ba-3cabc3ce7638"",
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        },
        {
            ""name"": ""Restart"",
            ""type"": ""Button"",
            ""id"": ""640e2732-a080-4111-b213-9f1be1181560"",
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        }
    ],
    ""bindings"": [
        {
            ""name"": """",
            ""id"": ""5865f171-9870-4b5a-972f-3063c41de6eb"",
            ""path"": ""<Keyboard>/f11"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Keyboard & Mouse"",
            ""action"": ""Fullscreen"",
            ""isComposite"": false,
            ""isPartOfComposite"": false
        },
        {
            ""name"": """",
            ""id"": ""2b145fb7-a4d1-458d-82fc-a0d321c812bc"",
            ""path"": ""<Keyboard>/escape"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Keyboard & Mouse"",
            ""action"": ""Pause"",
            ""isComposite"": false,
```

```
            ""isPartOfComposite"": false
        },
        {
            ""name"": """",
            ""id"": ""509e1651-66c5-46e5-8fbd-369de4e1c31e"",
            ""path"": ""<Gamepad>/start"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Gamepad"",
            ""action"": ""Pause"",
            ""isComposite"": false,
            ""isPartOfComposite"": false
        },
        {
            ""name"": """",
            ""id"": ""817eedbe-8e2b-4230-b7bd-7609ff87be3c"",
            ""path"": ""<Gamepad>/buttonEast"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Gamepad"",
            ""action"": ""Resume"",
            ""isComposite"": false,
            ""isPartOfComposite"": false
        },
        {
            ""name"": """",
            ""id"": ""e4db20af-453e-4545-9586-b5b9db1bdd71"",
            ""path"": ""<Gamepad>/buttonSouth"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Gamepad"",
            ""action"": ""Restart"",
            ""isComposite"": false,
            ""isPartOfComposite"": false
        }
    ]
},
{
    ""name"": ""Menu"",
    ""id"": ""ae75ae21-49d0-4b8c-b4e5-2f67a02d55c4"",
    ""actions"": [
        {
            ""name"": ""ClearHighScores"",
            ""type"": ""Button"",
            ""id"": ""ae97a100-87eb-455f-a0ba-399a4159f02b"",
```

```
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        },
        {
            ""name"": ""CloseMenu"",
            ""type"": ""Button"",
            ""id"": ""9c831bc1-5b26-47e1-9878-328ab65c0aa5"",
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        }
    ],
    ""bindings"": [
        {
            ""name"": """",
            ""id"": ""62573ea4-2fcb-421e-8b7b-241109429663"",
            ""path"": ""<Gamepad>/buttonSouth"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Gamepad"",
            ""action"": ""ClearHighScores"",
            ""isComposite"": false,
            ""isPartOfComposite"": false
        },
        {
            ""name"": """",
            ""id"": ""6463830c-802c-4dc2-bac7-77102fda1ae7"",
            ""path"": ""<Keyboard>/escape"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Keyboard & Mouse"",
            ""action"": ""CloseMenu"",
            ""isComposite"": false,
            ""isPartOfComposite"": false
        },
        {
            ""name"": """",
            ""id"": ""df511a31-de65-4670-a83f-be289923daf8"",
            ""path"": ""<Gamepad>/buttonEast"",
            ""interactions"": """",
            ""processors"": """",
            ""groups"": ""Gamepad"",
            ""action"": ""CloseMenu"",
            ""isComposite"": false,
```

```
            ""isPartOfComposite"": false
        }
    ]
},
{
    ""name"": ""Sound"",
    ""id"": ""6d17b40a-b39d-4407-bb73-ff7246bc8258"",
    ""actions"": [
        {
            ""name"": ""LowerSound"",
            ""type"": ""Button"",
            ""id"": ""d407e0a4-8fe8-45e4-a949-c7f4556dd7e2"",
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        },
        {
            ""name"": ""LowerMusic"",
            ""type"": ""Button"",
            ""id"": ""3dbf0c0f-8dd5-4a0e-89a4-1a7bb838cae5"",
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        },
        {
            ""name"": ""IncreaseSound"",
            ""type"": ""Button"",
            ""id"": ""e1f8988c-0816-4c3e-b1cf-ed8a3522999e"",
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        },
        {
            ""name"": ""IncreaseMusic"",
            ""type"": ""Button"",
            ""id"": ""3fd865eb-9ba5-4c33-a065-e59e46cf7062"",
            ""expectedControlType"": """",
            ""processors"": """",
            ""interactions"": """"
        }
    ],
    ""bindings"": [
        {
            ""name"": """",
            ""id"": ""82c9d527-07e3-4f63-add9-d4adcc49024b"",
```

```
                ""path"": ""<Gamepad>/leftShoulder"",
                ""interactions"": """",
                ""processors"": """",
                ""groups"": ""Gamepad"",
                ""action"": ""LowerSound"",
                ""isComposite"": false,
                ""isPartOfComposite"": false
            },
            {
                ""name"": """",
                ""id"": ""65e80a46-d425-4a5d-b725-4b492c28eb41"",
                ""path"": ""<Gamepad>/leftTrigger"",
                ""interactions"": """",
                ""processors"": """",
                ""groups"": ""Gamepad"",
                ""action"": ""LowerMusic"",
                ""isComposite"": false,
                ""isPartOfComposite"": false
            },
            {
                ""name"": """",
                ""id"": ""6d633747-6384-4eed-995a-c7d2909bd8d1"",
                ""path"": ""<Gamepad>/rightShoulder"",
                ""interactions"": """",
                ""processors"": """",
                ""groups"": ""Gamepad"",
                ""action"": ""IncreaseSound"",
                ""isComposite"": false,
                ""isPartOfComposite"": false
            },
            {
                ""name"": """",
                ""id"": ""a16b0de9-415d-46a5-a9eb-7bde9301776a"",
                ""path"": ""<Gamepad>/rightTrigger"",
                ""interactions"": """",
                ""processors"": """",
                ""groups"": ""Gamepad"",
                ""action"": ""IncreaseMusic"",
                ""isComposite"": false,
                ""isPartOfComposite"": false
            }
        ]
    }
],
""controlSchemes"": [
```

```
        {
            ""name"": ""Keyboard & Mouse"",
            ""bindingGroup"": ""Keyboard & Mouse"",
            ""devices"": [
                {
                    ""devicePath"": ""<Keyboard>"",
                    ""isOptional"": false,
                    ""isOR"": false
                },
                {
                    ""devicePath"": ""<Mouse>"",
                    ""isOptional"": false,
                    ""isOR"": false
                }
            ]
        },
        {
            ""name"": ""Gamepad"",
            ""bindingGroup"": ""Gamepad"",
            ""devices"": [
                {
                    ""devicePath"": ""<Gamepad>"",
                    ""isOptional"": false,
                    ""isOR"": false
                }
            ]
        },
        {
            ""name"": ""Joystick"",
            ""bindingGroup"": ""Joystick"",
            ""devices"": [
                {
                    ""devicePath"": ""<Joystick>"",
                    ""isOptional"": false,
                    ""isOR"": false
                }
            ]
        }
    ]
}");
        // Player
        m_Player = asset.FindActionMap("Player", throwIfNotFound: true);
        m_Player_Move = m_Player.FindAction("Move", throwIfNotFound: true);
        m_Player_Fire = m_Player.FindAction("Fire", throwIfNotFound: true);
        // Gameplay
```

```
        m_Gameplay = asset.FindActionMap("Gameplay", throwIfNotFound: true);
        m_Gameplay_Fullscreen = m_Gameplay.FindAction("Fullscreen", throwIfNotFound: true);
        m_Gameplay_Pause = m_Gameplay.FindAction("Pause", throwIfNotFound: true);
        m_Gameplay_Resume = m_Gameplay.FindAction("Resume", throwIfNotFound: true);
        m_Gameplay_Restart = m_Gameplay.FindAction("Restart", throwIfNotFound: true);
        // Menu
        m_Menu = asset.FindActionMap("Menu", throwIfNotFound: true);
        m_Menu_ClearHighScores = m_Menu.FindAction("ClearHighScores", throwIfNotFound: true);
        m_Menu_CloseMenu = m_Menu.FindAction("CloseMenu", throwIfNotFound: true);
        // Sound
        m_Sound = asset.FindActionMap("Sound", throwIfNotFound: true);
        m_Sound_LowerSound = m_Sound.FindAction("LowerSound", throwIfNotFound: true);
        m_Sound_LowerMusic = m_Sound.FindAction("LowerMusic", throwIfNotFound: true);
        m_Sound_IncreaseSound = m_Sound.FindAction("IncreaseSound", throwIfNotFound: true);
        m_Sound_IncreaseMusic = m_Sound.FindAction("IncreaseMusic", throwIfNotFound: true);
    }
    ~Controls()
    {
        UnityEngine.Object.Destroy(asset);
    }
    public InputBinding? bindingMask
    {
        get => asset.bindingMask;
        set => asset.bindingMask = value;
    }
    public ReadOnlyArray<InputDevice>? devices
    {
        get => asset.devices;
        set => asset.devices = value;
    }
    public ReadOnlyArray<InputControlScheme> controlSchemes => asset.controlSchemes;
    public bool Contains(InputAction action)
    {
        return asset.Contains(action);
    }
    public IEnumerator<InputAction> GetEnumerator()
    {
        return asset.GetEnumerator();
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
    public void Enable()
    {
```

```
    asset.Enable();
}
public void Disable()
{
    asset.Disable();
}
// Player
private readonly InputActionMap m_Player;
private IPlayerActions m_PlayerActionsCallbackInterface;
private readonly InputAction m_Player_Move;
private readonly InputAction m_Player_Fire;
public struct PlayerActions
{
    private Controls m_Wrapper;
    public PlayerActions(Controls wrapper) { m_Wrapper = wrapper; }
    public InputAction @Move => m_Wrapper.m_Player_Move;
    public InputAction @Fire => m_Wrapper.m_Player_Fire;
    public InputActionMap Get() { return m_Wrapper.m_Player; }
    public void Enable() { Get().Enable(); }
    public void Disable() { Get().Disable(); }
    public bool enabled => Get().enabled;
    public static implicit operator InputActionMap(PlayerActions set) { return set.Get(); }
    public void SetCallbacks(IPlayerActions instance)
    {
        if (m_Wrapper.m_PlayerActionsCallbackInterface != null)
        {
            Move.started -= m_Wrapper.m_PlayerActionsCallbackInterface.OnMove;
            Move.performed -= m_Wrapper.m_PlayerActionsCallbackInterface.OnMove;
            Move.canceled -= m_Wrapper.m_PlayerActionsCallbackInterface.OnMove;
            Fire.started -= m_Wrapper.m_PlayerActionsCallbackInterface.OnFire;
            Fire.performed -= m_Wrapper.m_PlayerActionsCallbackInterface.OnFire;
            Fire.canceled -= m_Wrapper.m_PlayerActionsCallbackInterface.OnFire;
        }
        m_Wrapper.m_PlayerActionsCallbackInterface = instance;
        if (instance != null)
        {
            Move.started += instance.OnMove;
            Move.performed += instance.OnMove;
            Move.canceled += instance.OnMove;
            Fire.started += instance.OnFire;
            Fire.performed += instance.OnFire;
            Fire.canceled += instance.OnFire;
        }
    }
}
```

```
public PlayerActions @Player => new PlayerActions(this);
// Gameplay
private readonly InputActionMap m_Gameplay;
private IGameplayActions m_GameplayActionsCallbackInterface;
private readonly InputAction m_Gameplay_Fullscreen;
private readonly InputAction m_Gameplay_Pause;
private readonly InputAction m_Gameplay_Resume;
private readonly InputAction m_Gameplay_Restart;
public struct GameplayActions
{
    private Controls m_Wrapper;
    public GameplayActions(Controls wrapper) { m_Wrapper = wrapper; }
    public InputAction @Fullscreen => m_Wrapper.m_Gameplay_Fullscreen;
    public InputAction @Pause => m_Wrapper.m_Gameplay_Pause;
    public InputAction @Resume => m_Wrapper.m_Gameplay_Resume;
    public InputAction @Restart => m_Wrapper.m_Gameplay_Restart;
    public InputActionMap Get() { return m_Wrapper.m_Gameplay; }
    public void Enable() { Get().Enable(); }
    public void Disable() { Get().Disable(); }
    public bool enabled => Get().enabled;
    public static implicit operator InputActionMap(GameplayActions set) { return set.Get(); }
    public void SetCallbacks(IGameplayActions instance)
    {
        if (m_Wrapper.m_GameplayActionsCallbackInterface != null)
        {
            Fullscreen.started -= m_Wrapper.m_GameplayActionsCallbackInterface.OnFullscreen;
            Fullscreen.performed -= m_Wrapper.m_GameplayActionsCallbackInterface.OnFullscreen;
            Fullscreen.canceled -= m_Wrapper.m_GameplayActionsCallbackInterface.OnFullscreen;
            Pause.started -= m_Wrapper.m_GameplayActionsCallbackInterface.OnPause;
            Pause.performed -= m_Wrapper.m_GameplayActionsCallbackInterface.OnPause;
            Pause.canceled -= m_Wrapper.m_GameplayActionsCallbackInterface.OnPause;
            Resume.started -= m_Wrapper.m_GameplayActionsCallbackInterface.OnResume;
            Resume.performed -= m_Wrapper.m_GameplayActionsCallbackInterface.OnResume;
            Resume.canceled -= m_Wrapper.m_GameplayActionsCallbackInterface.OnResume;
            Restart.started -= m_Wrapper.m_GameplayActionsCallbackInterface.OnRestart;
            Restart.performed -= m_Wrapper.m_GameplayActionsCallbackInterface.OnRestart;
            Restart.canceled -= m_Wrapper.m_GameplayActionsCallbackInterface.OnRestart;
        }
        m_Wrapper.m_GameplayActionsCallbackInterface = instance;
        if (instance != null)
        {
            Fullscreen.started += instance.OnFullscreen;
            Fullscreen.performed += instance.OnFullscreen;
            Fullscreen.canceled += instance.OnFullscreen;
            Pause.started += instance.OnPause;
```

```
            Pause.performed += instance.OnPause;
            Pause.canceled += instance.OnPause;
            Resume.started += instance.OnResume;
            Resume.performed += instance.OnResume;
            Resume.canceled += instance.OnResume;
            Restart.started += instance.OnRestart;
            Restart.performed += instance.OnRestart;
            Restart.canceled += instance.OnRestart;
        }
    }
}
public GameplayActions @Gameplay => new GameplayActions(this);
// Menu
private readonly InputActionMap m_Menu;
private IMenuActions m_MenuActionsCallbackInterface;
private readonly InputAction m_Menu_ClearHighScores;
private readonly InputAction m_Menu_CloseMenu;
public struct MenuActions
{
    private Controls m_Wrapper;
    public MenuActions(Controls wrapper) { m_Wrapper = wrapper; }
    public InputAction @ClearHighScores => m_Wrapper.m_Menu_ClearHighScores;
    public InputAction @CloseMenu => m_Wrapper.m_Menu_CloseMenu;
    public InputActionMap Get() { return m_Wrapper.m_Menu; }
    public void Enable() { Get().Enable(); }
    public void Disable() { Get().Disable(); }
    public bool enabled => Get().enabled;
    public static implicit operator InputActionMap(MenuActions set) { return set.Get(); }
    public void SetCallbacks(IMenuActions instance)
    {
        if (m_Wrapper.m_MenuActionsCallbackInterface != null)
        {
            ClearHighScores.started -= m_Wrapper.m_MenuActionsCallbackInterface.OnClearHighScores;
            ClearHighScores.performed -= m_Wrapper.m_MenuActionsCallbackInterface.OnClearHighScores;
            ClearHighScores.canceled -= m_Wrapper.m_MenuActionsCallbackInterface.OnClearHighScores;
            CloseMenu.started -= m_Wrapper.m_MenuActionsCallbackInterface.OnCloseMenu;
            CloseMenu.performed -= m_Wrapper.m_MenuActionsCallbackInterface.OnCloseMenu;
            CloseMenu.canceled -= m_Wrapper.m_MenuActionsCallbackInterface.OnCloseMenu;
        }
        m_Wrapper.m_MenuActionsCallbackInterface = instance;
        if (instance != null)
        {
            ClearHighScores.started += instance.OnClearHighScores;
            ClearHighScores.performed += instance.OnClearHighScores;
            ClearHighScores.canceled += instance.OnClearHighScores;
```

```
            CloseMenu.started += instance.OnCloseMenu;
            CloseMenu.performed += instance.OnCloseMenu;
            CloseMenu.canceled += instance.OnCloseMenu;
        }
    }
}
public MenuActions @Menu => new MenuActions(this);
// Sound
private readonly InputActionMap m_Sound;
private ISoundActions m_SoundActionsCallbackInterface;
private readonly InputAction m_Sound_LowerSound;
private readonly InputAction m_Sound_LowerMusic;
private readonly InputAction m_Sound_IncreaseSound;
private readonly InputAction m_Sound_IncreaseMusic;
public struct SoundActions
{
    private Controls m_Wrapper;
    public SoundActions(Controls wrapper) { m_Wrapper = wrapper; }
    public InputAction @LowerSound => m_Wrapper.m_Sound_LowerSound;
    public InputAction @LowerMusic => m_Wrapper.m_Sound_LowerMusic;
    public InputAction @IncreaseSound => m_Wrapper.m_Sound_IncreaseSound;
    public InputAction @IncreaseMusic => m_Wrapper.m_Sound_IncreaseMusic;
    public InputActionMap Get() { return m_Wrapper.m_Sound; }
    public void Enable() { Get().Enable(); }
    public void Disable() { Get().Disable(); }
    public bool enabled => Get().enabled;
    public static implicit operator InputActionMap(SoundActions set) { return set.Get(); }
    public void SetCallbacks(ISoundActions instance)
    {
        if (m_Wrapper.m_SoundActionsCallbackInterface != null)
        {
            LowerSound.started -= m_Wrapper.m_SoundActionsCallbackInterface.OnLowerSound;
            LowerSound.performed -= m_Wrapper.m_SoundActionsCallbackInterface.OnLowerSound;
            LowerSound.canceled -= m_Wrapper.m_SoundActionsCallbackInterface.OnLowerSound;
            LowerMusic.started -= m_Wrapper.m_SoundActionsCallbackInterface.OnLowerMusic;
            LowerMusic.performed -= m_Wrapper.m_SoundActionsCallbackInterface.OnLowerMusic;
            LowerMusic.canceled -= m_Wrapper.m_SoundActionsCallbackInterface.OnLowerMusic;
            IncreaseSound.started -= m_Wrapper.m_SoundActionsCallbackInterface.OnIncreaseSound;
            IncreaseSound.performed -= m_Wrapper.m_SoundActionsCallbackInterface.OnIncreaseSound;
            IncreaseSound.canceled -= m_Wrapper.m_SoundActionsCallbackInterface.OnIncreaseSound;
            IncreaseMusic.started -= m_Wrapper.m_SoundActionsCallbackInterface.OnIncreaseMusic;
            IncreaseMusic.performed -= m_Wrapper.m_SoundActionsCallbackInterface.OnIncreaseMusic;
            IncreaseMusic.canceled -= m_Wrapper.m_SoundActionsCallbackInterface.OnIncreaseMusic;
        }
        m_Wrapper.m_SoundActionsCallbackInterface = instance;
```

```
            if (instance != null)
            {
                LowerSound.started += instance.OnLowerSound;
                LowerSound.performed += instance.OnLowerSound;
                LowerSound.canceled += instance.OnLowerSound;
                LowerMusic.started += instance.OnLowerMusic;
                LowerMusic.performed += instance.OnLowerMusic;
                LowerMusic.canceled += instance.OnLowerMusic;
                IncreaseSound.started += instance.OnIncreaseSound;
                IncreaseSound.performed += instance.OnIncreaseSound;
                IncreaseSound.canceled += instance.OnIncreaseSound;
                IncreaseMusic.started += instance.OnIncreaseMusic;
                IncreaseMusic.performed += instance.OnIncreaseMusic;
                IncreaseMusic.canceled += instance.OnIncreaseMusic;
            }
        }
    }
    public SoundActions @Sound => new SoundActions(this);
    private int m_KeyboardMouseSchemeIndex = -1;
    public InputControlScheme KeyboardMouseScheme
    {
        get
        {
            if (m_KeyboardMouseSchemeIndex == -1) m_KeyboardMouseSchemeIndex =
asset.FindControlSchemeIndex("Keyboard & Mouse");
            return asset.controlSchemes[m_KeyboardMouseSchemeIndex];
        }
    }
    private int m_GamepadSchemeIndex = -1;
    public InputControlScheme GamepadScheme
    {
        get
        {
            if (m_GamepadSchemeIndex == -1) m_GamepadSchemeIndex = asset.FindControlSchemeIndex("Gamepad");
            return asset.controlSchemes[m_GamepadSchemeIndex];
        }
    }
    private int m_JoystickSchemeIndex = -1;
    public InputControlScheme JoystickScheme
    {
        get
        {
            if (m_JoystickSchemeIndex == -1) m_JoystickSchemeIndex = asset.FindControlSchemeIndex("Joystick");
            return asset.controlSchemes[m_JoystickSchemeIndex];
        }
```

```
    }
    public interface IPlayerActions
    {
        void OnMove(InputAction.CallbackContext context);
        void OnFire(InputAction.CallbackContext context);
    }
    public interface IGameplayActions
    {
        void OnFullscreen(InputAction.CallbackContext context);
        void OnPause(InputAction.CallbackContext context);
        void OnResume(InputAction.CallbackContext context);
        void OnRestart(InputAction.CallbackContext context);
    }
    public interface IMenuActions
    {
        void OnClearHighScores(InputAction.CallbackContext context);
        void OnCloseMenu(InputAction.CallbackContext context);
    }
    public interface ISoundActions
    {
        void OnLowerSound(InputAction.CallbackContext context);
        void OnLowerMusic(InputAction.CallbackContext context);
        void OnIncreaseSound(InputAction.CallbackContext context);
        void OnIncreaseMusic(InputAction.CallbackContext context);
    }
}
```

```
using UnityEngine;

public class BulletHit : MonoBehaviour
{
    public long damage = 1;
    [SerializeField] private GameObject explosion = null;

    void Update()
    {
        if (damage < 1) damage = 1; //Checks if damage is below 1
    }

    void OnTriggerStay(Collider other)
    {
        EnemyHealth enemyHealth = other.GetComponent<EnemyHealth>();
        ShieldHealth shieldHealth = other.GetComponent<ShieldHealth>();
        if (other.CompareTag("Enemy") && enemyHealth)
        {
            enemyHealth.takeDamage(damage);
            if (explosion) Instantiate(explosion, transform.position, transform.rotation);
            Destroy(gameObject);
        } else if (other.CompareTag("Shield"))
        {
            shieldHealth.takeDamage(damage);
            if (explosion) Instantiate(explosion, transform.position, transform.rotation);
            Destroy(gameObject);
        }
    }
}
```

```
using System.Collections;
```

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.Audio;
using UnityEngine.UI;
public class MainMenuManager : MonoBehaviour
{
    [Header("High Scores Menu")]
    [SerializeField] private Text endlessEasyHighScore = null;
    [SerializeField] private Text endlessNormalHighScore = null;
    [SerializeField] private Text endlessHardHighScore = null;
    [SerializeField] private Text endlessNIGHTMAREHighScore = null;
    [Header("Sound Effects")]
    [SerializeField] private AudioClip buttonClick = null;
    [Header("Setup")]
    [SerializeField] private Canvas mainMenu = null;
    [SerializeField] private Canvas highScoresMenu = null;
    [SerializeField] private Canvas settingsMenu = null;
    [SerializeField] private Canvas gamemodesMenu = null;
    [SerializeField] private Canvas selectDifficultyMenu = null;
    [SerializeField] private Canvas clearHighScoresPrompt = null;
    [SerializeField] private Text loadingText = null;
    [SerializeField] private AudioMixer audioMixer = null;
    private AudioSource audioSource;
    private Controls input;
    private bool loading = false;
    void Awake()
    {
        audioSource = GetComponent<AudioSource>();
        input = new Controls();
        if (audioSource) audioSource.ignoreListenerPause = true;
        loading = false;
        Time.timeScale = 1;
        AudioListener.pause = false;
        if (!PlayerPrefs.HasKey("SoundVolume"))
        {
            PlayerPrefs.SetFloat("SoundVolume", 1);
            PlayerPrefs.Save();
        } else
        {
            audioMixer.SetFloat("SoundVolume", Mathf.Log10(PlayerPrefs.GetFloat("SoundVolume")) * 20);
        }
        if (!PlayerPrefs.HasKey("MusicVolume"))
        {
            PlayerPrefs.SetFloat("MusicVolume", 1);
            PlayerPrefs.Save();
```

```
    } else
    {
        audioMixer.SetFloat("MusicVolume", Mathf.Log10(PlayerPrefs.GetFloat("MusicVolume")) * 20);
    }
    mainMenu.enabled = true;
    highScoresMenu.enabled = false;
    settingsMenu.enabled = false;
    gamemodesMenu.enabled = false;
    selectDifficultyMenu.enabled = false;
    clearHighScoresPrompt.enabled = false;
}
void OnEnable()
{
    input.Enable();
    input.Gameplay.Fullscreen.performed += context => toggleFullscreen();
    input.Menu.ClearHighScores.performed += context => clearHighScores(false);
    input.Menu.CloseMenu.performed += context => closeMenu();
}
void OnDisable()
{
    input.Disable();
    input.Gameplay.Fullscreen.performed -= context => toggleFullscreen();
    input.Menu.ClearHighScores.performed -= ContextMenu => clearHighScores(false);
    input.Menu.CloseMenu.performed -= context => closeMenu();
}
void Update()
{
    if (Input.GetKeyDown(KeyCode.JoystickButton0) && clearHighScoresPrompt.enabled) clearHighScores(false);
    updateHighScore(endlessEasyHighScore, "EasyHighScore", "Easy");
    updateHighScore(endlessNormalHighScore, "NormalHighScore", "Normal");
    updateHighScore(endlessHardHighScore, "HardHighScore", "Hard");
    updateHighScore(endlessNIGHTMAREHighScore, "NightmareHighScore", "NIGHTMARE!");
    if (!loading)
    {
        loadingText.enabled = false;
    } else
    {
        loadingText.enabled = true;
    }
}
void toggleFullscreen()
{
    Screen.fullScreen = !Screen.fullScreen;
}
void closeMenu()
```

```
{
    if (highScoresMenu.enabled)
    {
        highScoresMenu.enabled = false;
        mainMenu.enabled = true;
    } else if (settingsMenu.enabled)
    {
        settingsMenu.enabled = false;
        mainMenu.enabled = true;
    } else if (gamemodesMenu.enabled)
    {
        gamemodesMenu.enabled = false;
        mainMenu.enabled = true;
    } else if (selectDifficultyMenu.enabled)
    {
        selectDifficultyMenu.enabled = false;
        gamemodesMenu.enabled = true;
    } else if (clearHighScoresPrompt.enabled)
    {
        clearHighScoresPrompt.enabled = false;
        highScoresMenu.enabled = true;
    }
}
public void clickPlay()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    if (!gamemodesMenu.enabled)
    {
        mainMenu.enabled = false;
        gamemodesMenu.enabled = true;
    } else
    {
        mainMenu.enabled = true;
        gamemodesMenu.enabled = false;
    }
}
```

```
public void clickHighScores()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    if (!highScoresMenu.enabled)
    {
        mainMenu.enabled = false;
        highScoresMenu.enabled = true;
    } else
    {
        mainMenu.enabled = true;
        highScoresMenu.enabled = false;
    }
}
public void clickSettings()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    if (!settingsMenu.enabled)
    {
        mainMenu.enabled = false;
        settingsMenu.enabled = true;
    } else
    {
        mainMenu.enabled = true;
        settingsMenu.enabled = false;
    }
}
public void clickSurvivalMode()
```

```
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    if (!selectDifficultyMenu.enabled)
    {
        gamemodesMenu.enabled = false;
        selectDifficultyMenu.enabled = true;
    } else
    {
        gamemodesMenu.enabled = true;
        selectDifficultyMenu.enabled = false;
    }
}
public void clickClearHighScore()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    if (!clearHighScoresPrompt.enabled)
    {
        highScoresMenu.enabled = false;
        clearHighScoresPrompt.enabled = true;
    } else
    {
        highScoresMenu.enabled = true;
        clearHighScoresPrompt.enabled = false;
    }
}
public void clickQuitGame()
{
```

```
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    Application.Quit();
    #if UNITY_EDITOR
    UnityEditor.EditorApplication.isPlaying = false;
    #endif
}
public void startCampaign()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    if (PlayerPrefs.GetInt("StandardLevel") > 0)
    {
        StartCoroutine(loadScene("Level " + PlayerPrefs.GetInt("StandardLevel")));
    } else
    {
        PlayerPrefs.SetInt("StandardLevel", 1);
        PlayerPrefs.Save();
        StartCoroutine(loadScene("Level 1"));
    }
}
public void startSurvival(string difficulty)
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
```

```
        {
            audioSource.Play();
        }
    }
    if (difficulty != "")
    {
        StartCoroutine(loadScene("Survival " + difficulty));
    } else
    {
        StartCoroutine(loadScene("Survival Normal"));
    }
}
public void clearHighScores(bool wasClicked)
{
    if (clearHighScoresPrompt.enabled)
    {
        if (audioSource && wasClicked)
        {
            if (buttonClick)
            {
                audioSource.PlayOneShot(buttonClick);
            } else
            {
                audioSource.Play();
            }
        }
        PlayerPrefs.DeleteKey("EasyHighScore");
        PlayerPrefs.DeleteKey("NormalHighScore");
        PlayerPrefs.DeleteKey("HardHighScore");
        PlayerPrefs.DeleteKey("NightmareHighScore");
        PlayerPrefs.Save();
        clearHighScoresPrompt.enabled = false;
        highScoresMenu.enabled = true;
    }
}
void updateHighScore(Text main, string key, string difficulty)
{
    if (main && key != "" && difficulty != "")
    {
        if (PlayerPrefs.HasKey(key))
        {
            main.text = "High Score on " + difficulty + ": " + PlayerPrefs.GetString(key);
        } else
        {
            main.text = "High Score on " + difficulty + ": 0";
```

```
            }
        }
    }
    IEnumerator loadScene(string scene)
    {
        if (!loading)
        {
            loading = true;
            AsyncOperation load = SceneManager.LoadSceneAsync(scene);
            if (Camera.main.GetComponent<AudioSource>()) Camera.main.GetComponent<AudioSource>().Stop();
            while (!load.isDone)
            {
                Time.timeScale = 0;
                AudioListener.pause = true;
                loadingText.text = "Loading: " + Mathf.Floor(load.progress * 100) + "%";
                mainMenu.enabled = false;
                highScoresMenu.enabled = false;
                settingsMenu.enabled = false;
                gamemodesMenu.enabled = false;
                selectDifficultyMenu.enabled = false;
                clearHighScoresPrompt.enabled = false;
                yield return null;
            }
        }
    }
}
```

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.Audio;
using UnityEngine.UI;
public class GameController : MonoBehaviour
{
    public static GameController instance;
    [Header("Settings")]
    [SerializeField] private long wavesToClearForShieldRespawn = 0;
    public float enemyMoveTime = 0.3f;
    public float enemyFireRate = 3;
    public float enemyBulletSpeedIncrement = 0;
    public float maxEnemyMoveTime = 0.12f;
    public float maxEnemyFireRate = 0.6f;
    public float maxEnemyBulletSpeedIncrement = 2.5f;
    [SerializeField] private Vector2 enemyShipSpawnTime = new Vector2(45, 60);
    [SerializeField] private Vector2 asteroidSpawnTime = new Vector2(1, 8);
```

```
    [SerializeField] private Vector2 powerupSpawnTime = new Vector2(12, 18);

    [SerializeField] private bool canEnemyShipsSpawn = true;

    [SerializeField] private bool canAsteroidsSpawn = true;

    [Header("Survival-Only Settings")]

    [SerializeField] private float scoreMultiplier = 1;

    [Tooltip("1 is Easy, 2 is Normal, 3 is Hard, 4 is NIGHTMARE!.")] [Range(1, 4)] public int difficulty = 1;

    [Tooltip("NIGHTMARE! only.")] [SerializeField] private GameObject[] NIGHTMAREBosses = new GameObject[0];

    [Header("Sound Effects")]

    [SerializeField] private AudioClip buttonClick = null;

    [SerializeField] private AudioClip gameOverJingle = null;

    [SerializeField] private AudioClip winJingle = null;

    [Header("UI")]

    [SerializeField] private Canvas gameHUD = null;

    [SerializeField] private Canvas gamePausedMenu = null;

    [SerializeField] private Canvas gameOverMenu = null;

    [SerializeField] private Canvas levelCompletedMenu = null;

    [SerializeField] private Canvas settingsMenu = null;

    [SerializeField] private Canvas quitGameMenu = null;

    [SerializeField] private Canvas restartPrompt = null;

    [SerializeField] private Text levelCount = null;

    [SerializeField] private Text scoreCount = null;

    [SerializeField] private Text waveCount = null;

    [SerializeField] private Text highScoreIndicator = null;

    [SerializeField] private Text message = null;

    [SerializeField] private Text loadingText = null;

    [Header("Miscellaneous")]

    public bool isCampaignLevel = true;

    [Tooltip("Gives a life after the kill goal is reached (only used in Survival Mode NIGHTMARE!).")] public int
killsForLife = 0;

    public bool gameOver = false;

    public bool won = false;

    public bool paused = false;

    [Header("Setup")]

    [SerializeField] private GameObject enemyHolder = null;

    [SerializeField] private GameObject shieldGroup = null;

    [SerializeField] private GameObject shieldGroupToRespawn = null;

    [SerializeField] private GameObject[] enemyPatterns = new GameObject[0];

    [SerializeField] private GameObject enemyShip = null;

    [SerializeField] private GameObject[] asteroids = new GameObject[0];

    [SerializeField] private GameObject[] powerups = new GameObject[0];

    [SerializeField] private AudioMixer audioMixer = null;

    private AudioSource audioSource;

    private Controls input;

    private long score = 0;

    private long wave = 1;
```

```
private bool loading = false;
private long wavesCleared = 0; //Used for when the shields should be respawned
private long wavesTillBoss = 0; //Only used in Survival Mode NIGHTMARE!
private int clickSource = 1; //1 is pause menu, 2 is game over menu, 3 is level completed menu
private bool showedNewHighScore = false;
private bool playedLoseJingle = false, playedWinJingle = false;
void Awake()
{
    if (instance == null)
    {
        instance = this;
    } else if (instance != this)
    {
        Destroy(gameObject);
    }
    audioSource = GetComponent<AudioSource>();
    input = new Controls();
    if (audioSource) audioSource.ignoreListenerPause = true;
    loading = false;
    Time.timeScale = 1;
    AudioListener.pause = false;
    if (!PlayerPrefs.HasKey("SoundVolume"))
    {
        PlayerPrefs.SetFloat("SoundVolume", 1);
        PlayerPrefs.Save();
    } else
    {
        audioMixer.SetFloat("SoundVolume", Mathf.Log10(PlayerPrefs.GetFloat("SoundVolume")) * 20);
    }
    if (!PlayerPrefs.HasKey("MusicVolume"))
    {
        PlayerPrefs.SetFloat("MusicVolume", 1);
        PlayerPrefs.Save();
    } else
    {
        audioMixer.SetFloat("MusicVolume", Mathf.Log10(PlayerPrefs.GetFloat("MusicVolume")) * 20);
    }
    gameOver = false;
    won = false;
    paused = false;
    killsForLife = 0;
    gameHUD.enabled = true;
    gamePausedMenu.enabled = false;
    gameOverMenu.enabled = false;
    levelCompletedMenu.enabled = false;
```

```
    settingsMenu.enabled = false;
    quitGameMenu.enabled = false;
    restartPrompt.enabled = false;
    spawnEnemyPattern();
    if (canEnemyShipsSpawn) StartCoroutine(spawnEnemyShips());
    if (canAsteroidsSpawn) StartCoroutine(spawnAsteroids());
    StartCoroutine(spawnPowerups());
}
void OnEnable()
{
    input.Enable();
    input.Gameplay.Fullscreen.performed += context => toggleFullscreen();
    input.Gameplay.Pause.performed += context => pause();
    input.Gameplay.Resume.performed += context => resumeGame(false);
    input.Gameplay.Restart.performed += context => restart(false);
    input.Menu.CloseMenu.performed += context => closeMenu();
}
void OnDisable()
{
    input.Disable();
    input.Gameplay.Fullscreen.performed -= context => toggleFullscreen();
    input.Gameplay.Pause.performed -= context => pause();
    input.Gameplay.Resume.performed -= context => resumeGame(false);
    input.Gameplay.Restart.performed -= context => restart(false);
    input.Menu.CloseMenu.performed -= context => closeMenu();
}
void Update()
{
    if (!gameOver && !won && enemyHolder.transform.childCount <= 0)
    {
        if (isCampaignLevel)
        {
            if (wave < enemyPatterns.Length)
            {
                ++wave;
                ++wavesCleared;
                spawnEnemyPattern();
            } else
            {
                won = true;
                if (PlayerPrefs.GetInt("StandardLevel") >= PlayerPrefs.GetInt("MaxCampaignLevels"))
                {
                    PlayerPrefs.SetInt("StandardLevel", 1);
                    levelCompletedMenu.enabled = false;
                    StartCoroutine(loadScene("Campaign Ending"));
```

```
                }
                PlayerPrefs.Save();
            }
        } else
        {
            ++wave;
            ++wavesCleared;
            if (difficulty >= 4) ++wavesTillBoss;
            scoreMultiplier += 0.01f;
            spawnEnemyPattern();
        }
        if (!gameOver && !won && !isCampaignLevel)
        {
            if (difficulty <= 1)
            {
                enemyMoveTime -= 0.015f;
                enemyFireRate -= 0.15f;
                enemyBulletSpeedIncrement -= 0.0875f;
            } else if (difficulty == 2)
            {
                enemyMoveTime -= 0.02f;
                enemyFireRate -= 0.2f;
                enemyBulletSpeedIncrement += 0.125f;
            } else if (difficulty == 3)
            {
                enemyMoveTime -= 0.025f;
                enemyFireRate -= 0.3f;
                enemyBulletSpeedIncrement -= 0.15f;
            } else if (difficulty >= 4)
            {
                enemyMoveTime -= 0.035f;
                enemyFireRate -= 0.4f;
                enemyBulletSpeedIncrement -= 0.2f;
            }
        }
    }
    if (shieldGroupToRespawn && wavesToClearForShieldRespawn > 0 && wavesCleared >=
wavesToClearForShieldRespawn)
    {
        wavesCleared = 0;
        if (shieldGroup)
        {
            Destroy(shieldGroup);
            shieldGroup = Instantiate(shieldGroupToRespawn, new Vector3(0, -4.5f, 0), Quaternion.Euler(-90, 0,
0));
        } else
```

```
            {
                shieldGroup = Instantiate(shieldGroupToRespawn, new Vector3(0, -4.5f, 0), Quaternion.Euler(-90, 0,
0));
            }
        }
        if (gameOver && !won)
        {
            if (!quitGameMenu.enabled && !restartPrompt.enabled) gameOverMenu.enabled = true;
            clickSource = 2;
            if (audioSource && gameOverJingle && !playedLoseJingle)
            {
                playedLoseJingle = true;
                audioSource.PlayOneShot(gameOverJingle);
            }
            if (Camera.main.GetComponent<AudioSource>()) Camera.main.GetComponent<AudioSource>().Stop();
            if (!isCampaignLevel)
            {
                if (difficulty <= 1)
                {
                    setNewHighScore("EasyHighScore");
                } else if (difficulty == 2)
                {
                    setNewHighScore("NormalHighScore");
                } else if (difficulty == 3)
                {
                    setNewHighScore("HardHighScore");
                } else if (difficulty >= 4)
                {
                    setNewHighScore("NightmareHighScore");
                }
            }
        } else
        {
            gameOverMenu.enabled = false;
        }
        if (isCampaignLevel && !gameOver && won)
        {
            if (!loading && !quitGameMenu.enabled && !restartPrompt.enabled) levelCompletedMenu.enabled = true;
            clickSource = 3;
            if (PlayerPrefs.GetInt("StandardLevel") < PlayerPrefs.GetInt("MaxLevels"))
            {
                if (audioSource && winJingle && !playedWinJingle)
                {
                    playedWinJingle = true;
                    audioSource.PlayOneShot(winJingle);
```

```
        }
    } else
    {
        StartCoroutine(loadScene("Ending"));
    }
    if (Camera.main.GetComponent<AudioSource>()) Camera.main.GetComponent<AudioSource>().Stop();
}
if (isCampaignLevel)
{
    levelCount.transform.parent.gameObject.SetActive(true);
    if (PlayerPrefs.HasKey("StandardLevel"))
    {
        levelCount.text = PlayerPrefs.GetInt("StandardLevel").ToString();
    } else
    {
        levelCount.text = "1";
    }
    scoreCount.transform.parent.gameObject.SetActive(false);
    waveCount.text = wave + "/" + enemyPatterns.Length;
} else
{
    levelCount.transform.parent.gameObject.SetActive(false);
    scoreCount.transform.parent.gameObject.SetActive(true);
    scoreCount.text = score.ToString();
    waveCount.text = wave.ToString();
}
if (!loading)
{
    Camera.main.transform.position = new Vector3(0, 0, -10);
    foreach (GameObject player in GameObject.FindGameObjectsWithTag("Player"))
    {
        if (player) player.SetActive(true);
    }
    foreach (GameObject background in GameObject.FindGameObjectsWithTag("Background"))
    {
        if (background)
        {
            background.transform.position = new Vector3(0, background.transform.position.y, 5);
            background.GetComponent<BackgroundScroll>().enabled = true;
        }
    }
} else
{
    Camera.main.transform.position = new Vector3(500, 0, -10);
    foreach (GameObject player in GameObject.FindGameObjectsWithTag("Player"))
```

```
            {
                if (player) player.SetActive(false);
            }
            foreach (GameObject background in GameObject.FindGameObjectsWithTag("Background"))
            {
                if (background)
                {
                    background.transform.position = new Vector3(500, background.transform.position.y, 5);
                    background.GetComponent<BackgroundScroll>().enabled = false;
                }
            }
        }
        if (enemyMoveTime < maxEnemyMoveTime) enemyMoveTime = maxEnemyMoveTime; //Checks if enemy move time is
exceeding the maximum
        if (enemyFireRate < maxEnemyFireRate) enemyFireRate = maxEnemyFireRate; //Checks if enemy fire rate is
exceeding the maximum
        if (enemyBulletSpeedIncrement > maxEnemyBulletSpeedIncrement) enemyBulletSpeedIncrement =
maxEnemyBulletSpeedIncrement; //Checks if enemy bullet speed increment is exceeding the maximum
        if (isCampaignLevel) scoreMultiplier = 0; //Checks if the gamemode being played is Campaign
        if (!isCampaignLevel && difficulty < 4) //Checks if Survival Mode is being played on difficulties below
NIGHTMARE!
        {
            NIGHTMAREBosses = new GameObject[0];
            killsForLife = 0;
        }
        if (scoreMultiplier < 0) scoreMultiplier = 0; //Checks if the score multiplier is below 0
    }

    void toggleFullscreen()
    {
        Screen.fullScreen = !Screen.fullScreen;
    }
    void closeMenu()
    {
        if (paused)
        {
            if (settingsMenu.enabled)
            {
                settingsMenu.enabled = false;
                if (clickSource <= 1)
                {
                    gamePausedMenu.enabled = true;
                } else if (clickSource == 2)
                {
                    gameOverMenu.enabled = true;
                } else if (clickSource >= 3)
                {
```

```
                    levelCompletedMenu.enabled = true;
                }
            } else if (quitGameMenu.enabled)
            {
                quitGameMenu.enabled = false;
                if (clickSource <= 1)
                {
                    gamePausedMenu.enabled = true;
                } else if (clickSource == 2)
                {
                    gameOverMenu.enabled = true;
                } else if (clickSource >= 3)
                {
                    levelCompletedMenu.enabled = true;
                }
            } else if (restartPrompt.enabled)
            {
                restartPrompt.enabled = false;
                if (clickSource <= 1)
                {
                    gamePausedMenu.enabled = true;
                } else if (clickSource == 2)
                {
                    gameOverMenu.enabled = true;
                } else if (clickSource >= 3)
                {
                    levelCompletedMenu.enabled = true;
                }
            }
        }
    }
    void spawnEnemyPattern()
    {
        GameObject pattern;
        if (isCampaignLevel)
        {
            if (wave < enemyPatterns.Length + 1)
            {
                pattern = Instantiate(enemyPatterns[wave - 1], new Vector3(0, 19.5f, 0),
enemyHolder.transform.rotation);
            } else
            {
                pattern = null;
            }
        } else
```

```
        {
            if (difficulty < 4)
            {
                pattern = Instantiate(enemyPatterns[Random.Range(0, enemyPatterns.Length)], new Vector3(0, 19.5f,
0), enemyHolder.transform.rotation);
            } else
            {
                if (wavesTillBoss < 5)
                {
                    pattern = Instantiate(enemyPatterns[Random.Range(0, enemyPatterns.Length)], new Vector3(0,
19.5f, 0), enemyHolder.transform.rotation);
                    if (!pattern.CompareTag("ExcludeAlwaysFast"))
                    {
                        pattern.GetComponent<EnemyMover>().fast = true;
                        pattern.GetComponent<EnemyGun>().fast = true;
                    }
                } else
                {
                    pattern = Instantiate(NIGHTMAREBosses[Random.Range(0, NIGHTMAREBosses.Length)], new Vector3(0,
19.5f, 0), enemyHolder.transform.rotation);
                    if (!pattern.CompareTag("ExcludeAlwaysFast"))
                    {
                        pattern.GetComponent<EnemyMover>().fast = true;
                        pattern.GetComponent<EnemyGun>().fast = true;
                    }
                    wavesTillBoss = 0;
                }
            }
        }
        if (pattern) pattern.transform.SetParent(enemyHolder.transform);
    }


    IEnumerator spawnEnemyShips()
    {
        while (!gameOver && !won && enemyShip)
        {
            if (!paused)
            {
                yield return new WaitForSeconds(Random.Range(enemyShipSpawnTime.x, enemyShipSpawnTime.y));
                if (!gameOver && !won && !paused)
                {
                    GameObject ship;
                    float random = Random.value;
                    if (random <= 0.5f) //Left
                    {
                        ship = Instantiate(enemyShip, new Vector3(-18, 7.5f, 0), Quaternion.Euler(-90, 0, 0));
```

```
                        if (!isCampaignLevel && difficulty <= 1) ship.GetComponent<EnemyHealth>().health = 1;
                        ship.GetComponent<Mover>().speed = 5;
                    } else //Right
                    {
                        ship = Instantiate(enemyShip, new Vector3(18, 7.5f, 0), Quaternion.Euler(-90, 0, 0));
                        if (!isCampaignLevel && difficulty <= 1) ship.GetComponent<EnemyHealth>().health = 1;
                        ship.GetComponent<Mover>().speed = -5;
                    }
                }
            } else
            {
                yield return new WaitForEndOfFrame();
            }
        }
    }


    IEnumerator spawnAsteroids()
    {
        while (!gameOver && !won)
        {
            if (!paused)
            {
                yield return new WaitForSeconds(Random.Range(asteroidSpawnTime.x, asteroidSpawnTime.y));
                if (!gameOver && !won && !paused)
                {
                    Instantiate(asteroids[Random.Range(0, asteroids.Length)], new Vector3(Random.Range(-10.5f,
10.5f), 9.5f, 0), Quaternion.Euler(-90, 90, -90));
                }
            } else
            {
                yield return new WaitForEndOfFrame();
            }
        }
    }


    IEnumerator spawnPowerups()
    {
        while (!gameOver && !won)
        {
            if (!paused)
            {
                yield return new WaitForSeconds(Random.Range(powerupSpawnTime.x, powerupSpawnTime.y));
                if (!gameOver && !won && !paused)
                {
                    Instantiate(powerups[Random.Range(0, powerups.Length)], new Vector3(Random.Range(-11, 11), 9,
0), Quaternion.Euler(0, 0, 0));
```

```
            }
        } else
        {
            yield return new WaitForEndOfFrame();
        }
    }
}

public void addScore(long value)
{
    if (!isCampaignLevel && !gameOver && value > 0) score += (long)(value * scoreMultiplier);
}

void setNewHighScore(string key)
{
    if (key != "" && !isCampaignLevel)
    {
        if (!PlayerPrefs.HasKey(key) && score > 0)
        {
            PlayerPrefs.SetString(key, score.ToString());
            if (!showedNewHighScore)
            {
                showedNewHighScore = true;
                highScoreIndicator.text = "NEW HIGH SCORE!";
                Invoke("stopHighScoreIndicator", 3);
            }
        } else if (PlayerPrefs.HasKey(key) && score > long.Parse(PlayerPrefs.GetString(key)))
        {
            PlayerPrefs.SetString(key, long.Parse(PlayerPrefs.GetString(key)).ToString());
            if (!showedNewHighScore)
            {
                showedNewHighScore = true;
                highScoreIndicator.text = "NEW HIGH SCORE!";
                Invoke("stopHighScoreIndicator", 3);
            }
        }
        PlayerPrefs.Save();
    }
}
public void showMessage(string t)
{
    if (message)
    {
        message.text = t;
        CancelInvoke("resetMessage");
```

```
        Invoke("resetMessage", 1);
    }
}
void resetMessage()
{
    if (message) message.text = "";
}
void stopHighScoreIndicator()
{
    highScoreIndicator.text = "";
}
void pause()
{
    if (!gameOver && !won && !gameOverMenu.enabled && !levelCompletedMenu.enabled)
    {
        if (!paused) //Pauses the game
        {
            clickSource = 1;
            paused = true;
            Time.timeScale = 0;
            AudioListener.pause = true;
            gamePausedMenu.enabled = true;
        } else //Unpauses the game
        {
            if (!settingsMenu.enabled && !quitGameMenu.enabled && !restartPrompt.enabled)
            {
                paused = false;
                Time.timeScale = 1;
                AudioListener.pause = false;
                gamePausedMenu.enabled = false;
            }
        }
    }
}
public void resumeGame(bool wasClicked)
{
    if (!settingsMenu.enabled && !quitGameMenu.enabled && !restartPrompt.enabled)
    {
        if (audioSource && wasClicked)
        {
            if (buttonClick)
            {
                audioSource.PlayOneShot(buttonClick);
            } else
            {
```

```
                audioSource.Play();
            }
        }
        paused = false;
        Time.timeScale = 1;
        AudioListener.pause = false;
        gamePausedMenu.enabled = false;
    }
}
public void toNextLevel()
{
    if (isCampaignLevel && won && levelCompletedMenu.enabled)
    {
        if (audioSource)
        {
            if (buttonClick)
            {
                audioSource.PlayOneShot(buttonClick);
            } else
            {
                audioSource.Play();
            }
        }
        if (PlayerPrefs.GetInt("StandardLevel") < PlayerPrefs.GetInt("MaxCampaignLevels"))
        {
            PlayerPrefs.SetInt("StandardLevel", PlayerPrefs.GetInt("StandardLevel") + 1);
            StartCoroutine(loadScene("Level " + PlayerPrefs.GetInt("StandardLevel")));
        } else
        {
            PlayerPrefs.SetInt("StandardLevel", 1);
            StartCoroutine(loadScene("Ending"));
        }
        PlayerPrefs.Save();
    }
}
public void restart(bool wasClicked)
{
    if (gameOverMenu.enabled || restartPrompt.enabled)
    {
        if (audioSource && wasClicked)
        {
            if (buttonClick)
            {
                audioSource.PlayOneShot(buttonClick);
            } else
```

```
            {
                audioSource.Play();
            }
        }
        StartCoroutine(loadScene(SceneManager.GetActiveScene().name));
    }
}
public void exitGame()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    Application.Quit();
    #if UNITY_EDITOR
    UnityEditor.EditorApplication.isPlaying = false;
    #endif
}
public void exitToMainMenu()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    StartCoroutine(loadScene("Main Menu"));
}
public void openCanvasFromClickSource(Canvas canvas)
{
    if (canvas)
    {
        if (audioSource)
        {
            if (buttonClick)
```

```
            {
                audioSource.PlayOneShot(buttonClick);
            } else
            {
                audioSource.Play();
            }
        }
        if (!canvas.enabled)
        {
            canvas.enabled = true;
            if (clickSource <= 1)
            {
                gamePausedMenu.enabled = false;
            } else if (clickSource == 2)
            {
                gameOverMenu.enabled = false;
            } else if (clickSource >= 3)
            {
                levelCompletedMenu.enabled = false;
            }
        } else
        {
            canvas.enabled = false;
            if (clickSource <= 1)
            {
                gamePausedMenu.enabled = true;
            } else if (clickSource == 2)
            {
                gameOverMenu.enabled = true;
            } else if (clickSource >= 3)
            {
                levelCompletedMenu.enabled = true;
            }
        }
    }
}
IEnumerator loadScene(string scene)
{
    if (!loading)
    {
        loading = true;
        AsyncOperation load = SceneManager.LoadSceneAsync(scene);
        if (Camera.main.GetComponent<AudioSource>()) Camera.main.GetComponent<AudioSource>().Stop();
        while (!load.isDone)
        {
```

```
                Time.timeScale = 0;

                AudioListener.pause = true;

                loadingText.text = "Loading: " + Mathf.Floor(load.progress * 100) + "%";

                gameHUD.enabled = false;

                gamePausedMenu.enabled = false;

                gameOverMenu.enabled = false;

                levelCompletedMenu.enabled = false;

                settingsMenu.enabled = false;

                quitGameMenu.enabled = false;

                restartPrompt.enabled = false;

                yield return null;

            }

        }

    }

}
```

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.Audio;
using UnityEngine.UI;
public class EndingManager : MonoBehaviour
{
    [Header("Credits Settings")]
    [Tooltip("The Y position credits start at.")] [SerializeField] private float creditsY = 600;
    [SerializeField] private float creditsScrollSpeed = 0.5f;
    [Header("Sound Effects")]
    [SerializeField] private AudioClip buttonClick = null;
    [Header("Setup")]
    [SerializeField] private Canvas endingMenu = null;
    [SerializeField] private Canvas creditsMenu = null;
    [SerializeField] private RectTransform credits = null;
    [SerializeField] private Text loadingText = null;
    [SerializeField] private AudioMixer audioMixer = null;
    private AudioSource audioSource;
    private Controls input;
    private bool loading = false;
    void Awake()
    {
        audioSource = GetComponent<AudioSource>();
        input = new Controls();
        if (audioSource) audioSource.ignoreListenerPause = true;
        loading = false;
        Time.timeScale = 1;
        AudioListener.pause = false;
        if (!PlayerPrefs.HasKey("SoundVolume"))
        {
            PlayerPrefs.SetFloat("SoundVolume", 1);
        } else
        {
            audioMixer.SetFloat("SoundVolume", Mathf.Log10(PlayerPrefs.GetFloat("SoundVolume")) * 20);
        }
        if (!PlayerPrefs.HasKey("MusicVolume"))
        {
            PlayerPrefs.SetFloat("MusicVolume", 1);
        } else
        {
            audioMixer.SetFloat("MusicVolume", Mathf.Log10(PlayerPrefs.GetFloat("MusicVolume")) * 20);
        }
        PlayerPrefs.SetInt("StandardLevel", 1);
        PlayerPrefs.Save();
        endingMenu.enabled = true;
        creditsMenu.enabled = false;
    }
    void OnEnable()
    {
        input.Enable();
        input.Gameplay.Fullscreen.performed += context => toggleFullscreen();
        input.Menu.CloseMenu.performed += context => stopCredits();
    }
    void OnDisable()
    {
```

```
    input.Disable();
    input.Gameplay.Fullscreen.performed -= context => toggleFullscreen();
    input.Menu.CloseMenu.performed -= context => stopCredits();
}
void Update()
{
    if (Input.GetKeyDown(KeyCode.Escape) || Input.GetKeyDown(KeyCode.JoystickButton1))
    {
        creditsMenu.enabled = false;
        endingMenu.enabled = true;
    }
    if (!creditsMenu.enabled) credits.anchoredPosition = new Vector2(0, creditsY);
    if (!loading)
    {
        loadingText.enabled = false;
    } else
    {
        loadingText.enabled = true;
    }
}
void toggleFullscreen()
{
    Screen.fullScreen = !Screen.fullScreen;
}
void stopCredits()
{
    if (creditsMenu.enabled)
    {
        creditsMenu.enabled = false;
        endingMenu.enabled = true;
        StopCoroutine(scrollCredits());
    }
}
public void clickCredits()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    if (!creditsMenu.enabled)
    {
        endingMenu.enabled = false;
        creditsMenu.enabled = true;
        StartCoroutine(scrollCredits());
    } else
    {
        endingMenu.enabled = true;
        creditsMenu.enabled = false;
        StopCoroutine(scrollCredits());
    }
}
public void exitToMainMenu()
{
    if (audioSource)
    {
        if (buttonClick)
        {
            audioSource.PlayOneShot(buttonClick);
        } else
        {
            audioSource.Play();
        }
    }
    StartCoroutine(loadScene("Main Menu"));
}
IEnumerator scrollCredits()
{
    while (creditsMenu.enabled)
    {
        yield return new WaitForEndOfFrame();
        if (creditsMenu.enabled) credits.anchoredPosition -= new Vector2(0, creditsScrollSpeed);
        if (credits.anchoredPosition.y <= -creditsY)
        {
            endingMenu.enabled = true;
            creditsMenu.enabled = false;
            yield break;
        }
    }
}
IEnumerator loadScene(string scene)
{
    if (!loading)
```

```
        {
            loading = true;
            AsyncOperation load = SceneManager.LoadSceneAsync(scene);
            if (Camera.main.GetComponent<AudioSource>()) Camera.main.GetComponent<AudioSource>().Stop();
            while (!load.isDone)
            {
                Time.timeScale = 0;
                AudioListener.pause = true;
                loadingText.text = "Loading: " + Mathf.Floor(load.progress * 100) + "%";
                endingMenu.enabled = false;
                creditsMenu.enabled = false;
                yield return null;
            }
        }
    }
}
```

```
using System.Collections;
using UnityEngine;
public class EnemyShipGun : MonoBehaviour
{
    [Header("Settings")]
    [Tooltip("The amount of damage dealt to the player's Ship Armor.")] [SerializeField] private long damage = 1;
    [SerializeField] private float fireRate = 0.3375f;
    [Tooltip("The speed at which fired bullets travel at (cannot use positive values).")] [SerializeField] private
float bulletSpeed = -12.5f;
    [Header("Setup")]
    [SerializeField] private GameObject bullet = null;
    [SerializeField] private AudioClip fireSound = null;
    private AudioSource audioSource;
    private float nextShot = 0;

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        if (!GameController.instance.isCampaignLevel)
        {
            if (GameController.instance.difficulty <= 1)
            {
                fireRate += 0.025f;
            } else if (GameController.instance.difficulty >= 3)
            {
                fireRate -= 0.025f;
            }
        }
    }
    void Update()
    {
        if (!GameController.instance.gameOver && !GameController.instance.won && !GameController.instance.paused)
        {
            if (nextShot < fireRate)
            {
                nextShot += Time.deltaTime;
            } else
            {
                nextShot = 0;
                bool foundBulletSpawns = false;
                foreach (Transform bulletSpawn in transform)
                {
                    if (bulletSpawn.CompareTag("BulletSpawn") && bulletSpawn.gameObject.activeSelf)
                    {
                        GameObject newBullet = Instantiate(bullet, bulletSpawn.position, bulletSpawn.rotation);
                        newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                        newBullet.GetComponent<EnemyBulletHit>().damage = damage;
                        newBullet.GetComponent<Mover>().speed = bulletSpeed;
                        foundBulletSpawns = true;
                    }
                }
                if (!foundBulletSpawns)
                {
                    GameObject newBullet = Instantiate(bullet, transform.position - new Vector3(0, 0.0875f, 0),
transform.rotation);
                    newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                    if (newBullet.transform.rotation.x != -90) newBullet.transform.rotation = Quaternion.Euler(-
90, 0, 0);
                    newBullet.GetComponent<EnemyBulletHit>().damage = damage;
                    newBullet.GetComponent<Mover>().speed = bulletSpeed;
                    foundBulletSpawns = true;
                }
                if (foundBulletSpawns && audioSource)
                {
                    if (fireSound)
                    {
                        audioSource.PlayOneShot(fireSound);
```

```
                    } else
                    {
                        audioSource.Play();
                    }
                }
            }
        }
        if (damage < 1) damage = 1;
        if (bulletSpeed >= 0) bulletSpeed = -12.5f;
    }
}
```

```
using System.Collections;
using UnityEngine;
public class EnemyMover : MonoBehaviour
{
    [SerializeField] private float speed = 0.5f;
    public bool fast = false;

    void Start()
    {
        StartCoroutine(move());
    }

    IEnumerator move()
    {
        while (transform.childCount > 0)
        {
            if (!GameController.instance.paused)
            {
                if (transform.position.y <= 7.5f)
                {
                    if (!fast)
                    {
                        if (transform.childCount > 5)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime);
                        } else if (transform.childCount == 5)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.01f);
                        } else if (transform.childCount == 4)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.02f);
                        } else if (transform.childCount == 3)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.03f);
                        } else if (transform.childCount == 2)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.04f);
                        } else if (transform.childCount <= 1)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.05f);
                        }
                    } else
                    {
                        if (GameController.instance.enemyMoveTime >= 0.18f)
                        {
                            if (transform.childCount > 5)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.0175f);
                            } else if (transform.childCount == 5)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.035f);
                            } else if (transform.childCount == 4)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.0525f);
                            } else if (transform.childCount == 3)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.07f);
                            } else if (transform.childCount == 2)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.0875f);
                            } else if (transform.childCount <= 1)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.105f);
                            }
                        } else
                        {
                            if (transform.childCount > 5)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.0125f);
                            } else if (transform.childCount == 5)
                            {
                                yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.025f);
                            } else if (transform.childCount == 4)
                            {
```

```
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.0375f);
                        } else if (transform.childCount == 3)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.05f);
                        } else if (transform.childCount == 2)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.0625f);
                        } else if (transform.childCount <= 1)
                        {
                            yield return new WaitForSeconds(GameController.instance.enemyMoveTime - 0.05f);
                        }
                    }
                }
            } else
            {
                yield return new WaitForSeconds(0.1f);
            }
            if (transform.position.y <= 7.5f)
            {
                Vector3 screenBounds = Camera.main.ScreenToWorldPoint(new Vector3(Screen.width, Screen.height,
Camera.main.transform.position.z));
                float[] widths = new float[transform.childCount];
                foreach (Transform enemy in transform)
                {
                    if (enemy && enemy.CompareTag("Enemy") && enemy.GetComponent<Collider>())
                    {
                        for (int i = 0; i < transform.childCount; i++) widths[i] =
enemy.GetComponent<Collider>().bounds.extents.x;
                    }
                }
                transform.position += Vector3.right * speed;
                foreach (Transform enemy in transform)
                {
                    for (int i = 0; i < widths.Length; i++)
                    {
                        if (enemy.position.x <= screenBounds.x * -1 + widths[i] || enemy.position.x >=
screenBounds.x - widths[i])
                        {
                            speed = -speed;
                            transform.position += Vector3.right * speed;
                            transform.position = new Vector3(transform.position.x, transform.position.y -
0.5f, 0);
                        }
                    }
                }
            } else
            {
                transform.position += Vector3.down * speed;
            }
        } else
        {
            yield return null;
        }
    }
}
```

```
using UnityEngine;
public class EnemyHit : MonoBehaviour
{
    void OnTriggerStay(Collider other)
    {
        if (!GameController.instance.gameOver)
        {
            PlayerController playerController = other.GetComponent<PlayerController>();
            ShieldHealth shieldHealth = other.GetComponent<ShieldHealth>();
            if (other.CompareTag("Player") && playerController)
            {
                playerController.onHit(9223372036854775807, true);
            } else if (other.CompareTag("Shield") && shieldHealth)
            {
                shieldHealth.takeDamage(9223372036854775807);
            }
        }
    }
}
```

```
using UnityEngine;
public class EnemyHealth : MonoBehaviour
{
    [Header("Stats")]
    public long health = 0;
    [Tooltip("Amount of score earned from killing this enemy (Survival only).")] [SerializeField] private long
score = 0;
```

```
    [Header("Additional Enemy Spawning")]
    public bool spawnEnemiesOnDeath = false;
    [SerializeField] private long enemyAmount = 0;
    [SerializeField] private GameObject[] enemiesToSpawn = new GameObject[0];
    [SerializeField] private Vector2 randomSpawnX = Vector2.zero;
    [SerializeField] private Vector2 randomSpawnY = Vector2.zero;
    [Header("Extra Life")]
    [Tooltip("Survival only.")] [SerializeField] private bool giveLivesOnDeath = false;
    [Tooltip("Amount of lives given upon killing this enemy (Survival only).")] [SerializeField] private long
livesGiven = 1;
    [Header("Setup")]
    [SerializeField] private GameObject explosion = null;
    void Update()
    {
        if (health <= 0)
        {
            GameController.instance.addScore(score);
            if (explosion) Instantiate(explosion, transform.position, transform.rotation);
            if (spawnEnemiesOnDeath && enemiesToSpawn.Length > 0 && enemyAmount > 0)
            {
                for (int i = 0; i < enemyAmount; i++)
                {
                    Instantiate(enemiesToSpawn[Random.Range(0, enemiesToSpawn.Length)], transform.position + new
Vector3(Random.Range(randomSpawnX.x, randomSpawnX.y), Random.Range(randomSpawnY.x, randomSpawnY.y), 0),
Quaternion.Euler(Random.Range(-180, 180), -90, 90));
                }
            }
            if (!GameController.instance.isCampaignLevel && giveLivesOnDeath && livesGiven > 0)
            {
                PlayerController playerController = FindObjectOfType<PlayerController>();
                if (playerController && playerController.lives > 0)
                {
                    if (GameController.instance.difficulty < 4)
                    {
                        giveLife(playerController);
                    } else
                        ++GameController.instance.killsForLife;
                        if (GameController.instance.killsForLife >= 2)
                        {
                            giveLife(playerController);
                            GameController.instance.killsForLife = 0;
                        }
                    }
                }
            }
            Destroy(gameObject);
        }
    }
    public void takeDamage(long damage)
    {
        if (damage > 0)
        {
            health -= damage;
        } else
        {
            --health;
        }
    }
    void giveLife(PlayerController playerController)
    {
        if (playerController && playerController.lives > 0 && livesGiven > 0 && health <= 0)
        {
            playerController.lives += livesGiven;
            if (livesGiven == 1)
            {
                GameController.instance.showMessage("You got 1 Life!");
            } else if (livesGiven > 1)
            {
                GameController.instance.showMessage("You got " + livesGiven + " Lives!");
            }
        }
    }
}
```

```
using System.Collections;
using UnityEngine;
public class EnemyGun : MonoBehaviour
{
    [Header("Settings")]
    [Tooltip("Amount of damage dealt to the player (only affects armor).")] [SerializeField] private long damage =
1;
    [Tooltip("The speed at which fired bullets travel at (cannot use positive values).")] [SerializeField] private
float bulletSpeed = -12.5f;
    [Tooltip("Should this alien group shoot faster?")] public bool fast = false;
    [Header("Setup")]
    [SerializeField] private GameObject bullet = null;
```

```
    [SerializeField] private AudioClip fireSound = null;
    private AudioSource audioSource;
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        bulletSpeed -= GameController.instance.enemyBulletSpeedIncrement;
        StartCoroutine(shoot());
    }
    void Update()
    {
        if (damage < 1) damage = 1;
        if (bulletSpeed == 0)
        {
            bulletSpeed = -12.5f;
        } else if (bulletSpeed > 0)
        {
            bulletSpeed = -bulletSpeed;
        }
    }
    IEnumerator shoot()
    {
        while (transform.childCount > 0 && !GameController.instance.gameOver && !GameController.instance.won)
        {
            if (!GameController.instance.paused)
            {
                if (transform.position.y <= 7.5f)
                {
                    if (!fast)
                    {
                        if (transform.childCount > 5)
                        {
                            yield return new WaitForSeconds(Random.Range(0.5f,
GameController.instance.enemyFireRate));
                        } else if (transform.childCount == 5)
                        {
                            yield return new WaitForSeconds(Random.Range(0.5f,
GameController.instance.enemyFireRate - 0.02f));
                        } else if (transform.childCount == 4)
                        {
                            yield return new WaitForSeconds(Random.Range(0.5f,
GameController.instance.enemyFireRate - 0.04f));
                        } else if (transform.childCount == 3)
                        {
                            yield return new WaitForSeconds(Random.Range(0.5f,
GameController.instance.enemyFireRate - 0.06f));
                        } else if (transform.childCount == 2)
                        {
                            yield return new WaitForSeconds(Random.Range(0.5f,
GameController.instance.enemyFireRate - 0.08f));
                        } else if (transform.childCount <= 1)
                        {
                            yield return new WaitForSeconds(Random.Range(0.5f,
GameController.instance.enemyFireRate - 0.1f));
                        }
                    } else
                    {
                        if (transform.childCount > 5)
                        {
                            yield return new WaitForSeconds(Random.Range(0.25f,
GameController.instance.enemyFireRate - 0.035f));
                        } else if (transform.childCount == 5)
                        {
                            yield return new WaitForSeconds(Random.Range(0.25f,
GameController.instance.enemyFireRate - 0.07f));
                        } else if (transform.childCount == 4)
                        {
                            yield return new WaitForSeconds(Random.Range(0.25f,
GameController.instance.enemyFireRate - 0.105f));
                        } else if (transform.childCount == 3)
                        {
                            yield return new WaitForSeconds(Random.Range(0.25f,
GameController.instance.enemyFireRate - 0.14f));
                        } else if (transform.childCount == 2)
                        {
                            yield return new WaitForSeconds(Random.Range(0.25f,
GameController.instance.enemyFireRate - 0.175f));
                        } else if (transform.childCount <= 1)
                        {
                            yield return new WaitForSeconds(Random.Range(0.25f,
GameController.instance.enemyFireRate - 0.21f));
                        }
                    }
                    if (transform.childCount > 0 && !GameController.instance.gameOver &&
!GameController.instance.won && !GameController.instance.paused)
                    {
                        //Gets children and picks a random one for firing
                        bool foundBulletSpawns = false;
                        Transform[] enemies = GetComponentsInChildren<Transform>();
```

```
                        Transform enemy = enemies[Random.Range(0, enemies.Length)];
                        foreach (Transform bulletSpawn in enemy.parent.transform)
                        {
                            if (bulletSpawn.CompareTag("BulletSpawn") && bulletSpawn.gameObject.activeSelf &&
enemy.transform != transform)
                            {
                                GameObject newBullet = Instantiate(bullet, bulletSpawn.position,
bulletSpawn.rotation);
                                newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                                newBullet.GetComponent<EnemyBulletHit>().damage = damage;
                                newBullet.GetComponent<Mover>().speed = bulletSpeed;
                                foundBulletSpawns = true;
                            }
                        }
                        if (!foundBulletSpawns)
                        {
                            GameObject newBullet = Instantiate(bullet, enemy.position - new Vector3(0, 1, 0),
enemy.rotation);
                            newBullet.transform.position = new Vector3(newBullet.transform.position.x,
newBullet.transform.position.y, 0);
                            if (newBullet.transform.rotation.x != -90) newBullet.transform.rotation =
Quaternion.Euler(-90, 0, 0);
                            newBullet.GetComponent<EnemyBulletHit>().damage = damage;
                            newBullet.GetComponent<Mover>().speed = bulletSpeed;
                            foundBulletSpawns = true;
                        }
                        if (foundBulletSpawns && audioSource)
                        {
                            if (fireSound)
                            {
                                audioSource.PlayOneShot(fireSound);
                            } else
                            {
                                audioSource.Play();
                            }
                        }
                    }
                } else
                {
                    yield return null;
                }
            } else
            {
                yield return null;
            }
        }
    }
}
```

```
using UnityEngine;
public class EnemyBulletHit : MonoBehaviour
{
    [Tooltip("How much damage this projectile deals to the player's Ship Armor.")] public long damage = 1;
    [Tooltip("Damage multiplier on Hard mode (Survival only).")] public float hardDamageMultiplier = 1;
    [SerializeField] private GameObject explosion = null;
    void Start()
    {
        if (!GameController.instance.isCampaignLevel && GameController.instance.difficulty >= 3) damage =
(long)(damage * hardDamageMultiplier);
    }
    void Update()
    {
        if (damage < 1) damage = 1; //Checks if damage is below 1
    }
    void OnTriggerStay(Collider other)
    {
        if (!GameController.instance.gameOver)
        {
            PlayerController playerController = other.GetComponent<PlayerController>();
            ShieldHealth shieldHealth = other.GetComponent<ShieldHealth>();
            if (other.CompareTag("Player") && playerController && !playerController.invulnerable)
            {
                playerController.onHit(damage, false);
                if (explosion) Instantiate(explosion, transform.position, transform.rotation);
                Destroy(gameObject);
            } else if (other.CompareTag("Shield") && shieldHealth)
            {
                shieldHealth.takeDamage(damage);
                if (explosion) Instantiate(explosion, transform.position, transform.rotation);
                Destroy(gameObject);
            }
        }
    }
}
```

```
using UnityEngine;
using UnityEngine.UI;
public class ToggleFullscreen : MonoBehaviour
{
    [SerializeField] private Vector2 fullscreenTextSize = new Vector2(262, 30);
    [SerializeField] private Vector2 windowedModeTextSize = new Vector2(345, 30);
    [SerializeField] private AudioClip buttonClick = null;
    private Text fullscreenText;
    private AudioSource audioSource;
    void Start()
    {
        fullscreenText = GetComponent<Text>();
        audioSource = GetComponent<AudioSource>();
        if (audioSource) audioSource.ignoreListenerPause = true;
    }
    void Update()
    {
        if (!Screen.fullScreen)
        {
            fullscreenText.text = "Change to Fullscreen";
            fullscreenText.rectTransform.sizeDelta = fullscreenTextSize;
        } else
        {
            fullscreenText.text = "Change to Windowed Mode";
            fullscreenText.rectTransform.sizeDelta = windowedModeTextSize;
        }
    }
    public void changeFullscreen()
    {
        if (audioSource)
        {
            if (buttonClick)
            {
                audioSource.PlayOneShot(buttonClick);
            } else
            {
                audioSource.Play();
            }
        }
        Screen.fullScreen = !Screen.fullScreen;
    }
}
```

```
using UnityEngine;
public class ShieldHealth : MonoBehaviour
{
    [SerializeField] private long maxHealth = 0;
    [SerializeField] private Color32 shieldColor = new Color32(0, 100, 255, 255);
    [SerializeField] private Color32 damagedColor = new Color32(0, 75, 170, 255);
    private long health = 0;
    void Start()
    {
        health = maxHealth;
    }
    void Update()
    {
        if (health <= 0) Destroy(gameObject);
        foreach (Transform part in transform)
        {
            if (part.GetComponent<Renderer>())
            {
                if (health <= maxHealth * 0.5)
                {
                    part.GetComponent<Renderer>().material.SetColor("_Color", damagedColor);
                } else
                {
                    part.GetComponent<Renderer>().material.SetColor("_Color", shieldColor);
                }
            }
        }
    }
    public void takeDamage(long damage)
    {
        if (damage > 0)
        {
            health -= damage;
        } else
        {
            --damage;
        }
    }
}
```

```
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.UI;
public class SetVolume : MonoBehaviour
{
    [SerializeField] private AudioMixer audioMixer = null;
    [SerializeField] private Canvas menu = null;
    [SerializeField] private string volume = "";
    private Slider slider;
    private bool lowering = false;
    private bool increasing = false;
    void Start()
    {
        slider = GetComponent<Slider>();
        if (PlayerPrefs.HasKey(volume))
        {
            slider.value = PlayerPrefs.GetFloat(volume);
        } else
        {
            slider.value = 1;
        }
    }
    void Update()
    {
        if (increasing)
        {
            slider.value += 0.005f;
        } else if (lowering)
        {
            slider.value -= 0.005f;
        }
        audioMixer.SetFloat(volume, Mathf.Log10(slider.value) * 20);
    }
    public void controllerAdjust(bool i, bool l)
    {
        if (menu)
        {
            if (menu.enabled)
            {
                increasing = i;
                lowering = l;
            }
        } else
        {
            increasing = i;
            lowering = l;
        }
    }
    public void controllerCancel()
    {
        increasing = false;
        lowering = false;
    }
    public void setVolume()
    {
        PlayerPrefs.SetFloat(volume, slider.value);
        PlayerPrefs.Save();
    }
}
```

```
using UnityEngine;
public class Rotator : MonoBehaviour
{
    [Tooltip("How fast this object rotates on the Y axis.")] [SerializeField] private float y = 0;
    void Update()
    {
        transform.Rotate(new Vector3(0, y * Time.deltaTime, 0));
    }
}
```

```
using UnityEngine;
public class Mover : MonoBehaviour
{
    public float speed = 0;
    [Tooltip("Where the object should move towards, if it's not using transform.forward.")] [SerializeField]
private Vector2 movement = Vector2.zero;
    [SerializeField] private bool useForwardMovement = true;
    void Update()
    {
        if (useForwardMovement)
        {
            transform.position += transform.forward * speed * Time.deltaTime;
        } else
        {
```

```
            transform.position += new Vector3(movement.x, movement.y, 0) * speed * Time.deltaTime;
        }
        transform.position = new Vector3(transform.position.x, transform.position.y, 0);
    }
}
```

```
using UnityEngine;
public class LevelHolder : MonoBehaviour
{
    public static LevelHolder instance;
    [SerializeField] private int level = 1;
    [SerializeField] private int maxLevels = 13;
    void Awake()
    {
        PlayerPrefs.SetInt("StandardLevel", level);
        PlayerPrefs.SetInt("MaxCampaignLevels", maxLevels);
        PlayerPrefs.Save();
        Destroy(gameObject);
    }
}
```

```
using UnityEngine;
public class DestroyOnClear : MonoBehaviour
{
    void Update()
    {
        if (transform.childCount <= 0) Destroy(gameObject);
    }
}
```

```
using UnityEngine;
public class DestroyByBoundary : MonoBehaviour
{
    void OnTriggerExit(Collider other)
    {
        if (CompareTag("Boundary") && !other.CompareTag("Player")) Destroy(other.gameObject);
    }
}
```

```
using UnityEngine;
public class DestroyAfterTime : MonoBehaviour
{
    [SerializeField] private float lifetime = 0;
    void Awake()
    {
        Destroy(gameObject, lifetime);
    }
}
```

```
using UnityEngine;
public class ControllerSoundAdjuster : MonoBehaviour
{
    [SerializeField] private SetVolume volumeSlider = null;

    private Controls input;

    private void Awake()
    {
        input = new Controls();
    }
    void OnEnable()
    {
        input.Enable();
        input.Sound.LowerSound.performed += context => volumeSlider.controllerAdjust(false, true);
        input.Sound.IncreaseSound.performed += context => volumeSlider.controllerAdjust(true, false);
        input.Sound.LowerSound.canceled += context => volumeSlider.controllerCancel();
        input.Sound.IncreaseSound.canceled += context => volumeSlider.controllerCancel();
    }
    void OnDisable()
    {
        input.Disable();
        input.Sound.LowerSound.performed -= context => volumeSlider.controllerAdjust(false, true);
        input.Sound.IncreaseSound.performed -= context => volumeSlider.controllerAdjust(true, false);
        input.Sound.LowerSound.canceled -= context => volumeSlider.controllerCancel();
        input.Sound.IncreaseSound.canceled -= context => volumeSlider.controllerCancel();
    }
}
```

```
using UnityEngine;
public class ControllerMusicAdjuster : MonoBehaviour
{
    [SerializeField] private SetVolume volumeSlider = null;
    private Controls input;
    void Awake()
    {
        input = new Controls();
    }
    void OnEnable()
    {
        input.Enable();
        input.Sound.LowerMusic.performed += context => volumeSlider.controllerAdjust(false, true);
        input.Sound.IncreaseMusic.performed += context => volumeSlider.controllerAdjust(true, false);
        input.Sound.LowerMusic.canceled += context => volumeSlider.controllerCancel();
        input.Sound.IncreaseMusic.canceled += context => volumeSlider.controllerCancel();
    }
    void OnDisable()
    {
        input.Disable();
        input.Sound.LowerMusic.performed -= context => volumeSlider.controllerAdjust(false, true);
        input.Sound.IncreaseMusic.performed -= context => volumeSlider.controllerAdjust(true, false);
        input.Sound.LowerMusic.canceled -= context => volumeSlider.controllerCancel();
        input.Sound.IncreaseMusic.canceled -= context => volumeSlider.controllerCancel();
    }
}
```

```
using UnityEngine;
using UnityEngine.UI;
public class ButtonHover : MonoBehaviour
{
    [SerializeField] private Color32 normalColor = new Color32(200, 200, 200, 255);
    [SerializeField] private Color32 hoverColor = new Color32(255, 255, 255, 255);
    [SerializeField] private Text[] textsToShow = new Text[0];
    private Text text;
    void Start()
    {
        text = GetComponent<Text>();
        text.color = normalColor;
        if (textsToShow.Length > 0)
        {
            foreach (Text t in textsToShow) if (t) t.enabled = false;
        }
    }
    public void OnMouseEnter()
    {
        text.color = hoverColor;
        if (textsToShow.Length > 0)
        {
            foreach (Text t in textsToShow) if (t) t.enabled = true;
        }
    }
    public void OnMouseExit()
    {
        text.color = normalColor;
        if (textsToShow.Length > 0)
        {
            foreach (Text t in textsToShow) if (t) t.enabled = false;
        }
    }
}
```

```
using UnityEngine;
public class BackgroundScroll : MonoBehaviour
{
    [SerializeField] private float speed = -1;
    [SerializeField] private float z = 36.86f;
    private Vector3 initialPosition;
    void Start()
    {
        initialPosition = transform.position;
    }
    void Update()
    {
        transform.position = initialPosition + Vector3.up * Mathf.Repeat(Time.time * speed, z);
        transform.position = new Vector3(0, transform.position.y, 0);
    }
}
```

```
using UnityEngine;
public class ReinforcedPlating : MonoBehaviour
{
    private EnemyHealth enemyHealth;
    void Start()
    {
        enemyHealth = GetComponent<EnemyHealth>();
        if (!GameController.instance.isCampaignLevel && GameController.instance.difficulty >= 4) main();
    }
    void main()
    {
        ++enemyHealth.health;
        enemyHealth.spawnEnemiesOnDeath = true;
        enabled = false;
    }
}
```

```
using System.Collections;
using UnityEngine;
public class Blink : MonoBehaviour
{
    void Start()
    {
        if (!GameController.instance.isCampaignLevel && GameController.instance.difficulty >= 4)
StartCoroutine(main());
    }

    IEnumerator main()
    {
        while (true)
        {
            yield return new WaitForSeconds(Random.Range(0.5f, 1));
            foreach (Transform obj in transform)
            {
                if (obj.GetComponent<Renderer>()) obj.GetComponent<Renderer>().enabled = false;
            }
            yield return new WaitForSeconds(Random.Range(0.75f, 1.5f));
            foreach (Transform obj in transform)
            {
                if (obj.GetComponent<Renderer>()) obj.GetComponent<Renderer>().enabled = true;
            }
        }
    }
}
```