Workshop 3: Grundläggande Animation, Ljudhantering och Coroutines i Unity

Syfte

Syftet med denna workshop är att introducera och ge praktisk erfarenhet av:

1. **Implementera Enkel 2D Animation:** Skapa och styra grundläggande 2D-animationer (Idle, Run, Jump, Death, Respawn) med Unitys Animator Controller, driven av enkla parametrar från kod.
2. **Integrera Grundläggande Audio:** Lära sig grunderna i att spela upp ljudeffekter (SFX) och bakgrundsmusik (BGM) och skapa en enkel, central `AudioManager` (Singleton) som direkt hanterar `AudioSource`-komponenter. **Bakgrundsmusik spelas automatiskt.**
3. **Hantera Tid och Sekvenser med Coroutines:** Förstå och implementera **Coroutines** som ett kraftfullt verktyg för att:
   - Skapa sekvenser av handlingar.
   - Exekvera logik över flera bildrutor.
   - Införa fördröjningar och vänta på händelser (t.ex. för respawn-timing, tidsbegränsade effekter som invincibility, upprepade handlingar som attack cooldowns).
4. **Kombinera Tekniker:** Tillämpa dessa nya koncept (animation, ljud, coroutines) för att förbättra befintliga mekaniker och lägga till mer "liv", feedback och polish i det 2D-plattformsspel vi utvecklat, inklusive dödsanimationen, dödsskriket, och en respawn-sekvens med fördröjning och odödlighet.

Förutsättningar

- Unity **6.0** (eller senare) installerat.
- Genomfört **Workshop 1 och 2**, eller har motsvarande kunskaper om:
  - Grundläggande Unity-gränssnitt och C#-programmering.
  - Fysik (`Rigidbody 2D` , `Collider 2D`), Prefabs.
  - Input System, Cinemachine, grundläggande UI (Canvas, TextMeshPro).
  - SOLID-principerna (grundläggande förståelse).
  - Designmönster: Singleton, Strategy, Object Pooling, Builder.
  - Raycasting för AI/detektion.
- Projektet från Workshop 2 är tillgängligt.
- **Nya Resurser (Behövs för denna workshop):**
  - **Spritesheets eller bildsekvenser:** För spelar-animationer (Idle, Run, Jump, Death, Respawn). *(Exempel: player_anim_idle.png, player_anim_run.png, etc.)*
  - **Ljudfiler (WAV/MP3/OGG):**
    - Bakgrundsmusik (BGM): `music.mp3` (eller liknande). **Krävs för automatisk uppspelning.**
    - SFX: `deathscream.mp3` , `jump.mp3` , `respawn.mp3` .

        - *Valfria SFX:* Landningsljud, Skjutljud (fiende), Träffljud (projektil), Powerup-upplockningsljud.
    - *(Exempelresurser kan tillhandahållas eller hämtas från gratis källor som OpenGameArt, Kenney Assets, freesound.org).*

Kontext

Efter Workshop 2 har vi en fungerande spelgrund. Spelet saknar dock liv – det är tyst och statiskt. Denna workshop introducerar animation, ljud och **coroutines** för att åtgärda detta.

Vi börjar med spelar-animationer (Idle, Run). Därefter sätter vi upp en `AudioManager` som automatiskt spelar bakgrundsmusik och hanterar ljudeffekter. Sedan integrerar vi ljud och fler animationer (Jump, Death, Respawn).

Kärnan i workshopen är att kombinera dessa element med **Coroutines** för att hantera tid och sekvenser. Vi använder Coroutines för att skapa en fördröjning i respawn-processen, hantera tidsbegränsade effekter (invincibility med blinkande), och strukturera upprepade handlingar (fiendens attack-cooldown). *(Unity erbjuder även enklare metoder som `Invoke` och `InvokeRepeating` för tidsstyrning, men vi fokuserar på den mer flexibla Coroutine-metoden i denna workshop).*

Målet är en mer engagerande upplevelse genom kombinerad visuell och auditiv feedback med korrekt timing, exemplifierat i döds- och respawn-sekvensen.

Animation i Unity (Mecanim / Animator Controller)

Unity erbjuder flera sätt att skapa och kontrollera animationer. För 2D-spel, och särskilt när man animerar karaktärer baserade på en sekvens av bilder (sprites), är det vanligaste och mest väletablerade systemet **Mecanim**, som använder en **Animator Controller**.

Detta system låter oss:

1. **Skapa Animation Clips:** Definiera själva animationen (t.ex. en sekvens av sprites för en "Run"-cykel) i fönstret `Animation`.
2. **Använda en Animator Controller:** Skapa en state machine (ett flödesschema) i fönstret `Animator` där varje "state" (tillstånd) representerar en animation (t.ex. Idle, Run, Jump).
3. **Definiera Transitions (Övergångar):** Bestämma hur och när spelet ska växla mellan olika animation states baserat på villkor.
4. **Använda Parameters (Parametrar):** Skapa variabler (t.ex. `Speed` (Float), `IsGrounded` (Bool), `Jump` (Trigger)) i Animator Controller som vi kan styra från våra C#-skript för att driva övergångarna.
5. **Lägga till en Animator Component:** Placera en `Animator`-komponent på det GameObject som ska animeras och koppla den till vår skapade Animator Controller-fil.

I denna workshop kommer vi att fokusera på detta Animator Controller-baserade tillvägagångssätt för att skapa och styra spelarens grundläggande animationer (Idle, Run, Jump, Death, Respawn) med hjälp av **spritesheets** (bildsekvenser).

För mer information, se Unitys dokumentation:
https://docs.unity3d.com/Manual/AnimationOverview.html

Ljud i Unity

Precis som med animation är ljud avgörande för att skapa en levande och engagerande spelupplevelse. Unity har ett inbyggt system för att hantera och spela upp ljud. De centrala delarna i detta system är:

1. **`AudioClip`:** Representerar själva ljudfilen (t.ex. en `.mp3`, `.wav` eller `.ogg` fil) som importerats till ditt projekt.
2. **`AudioSource`:** Fungerar som en ljudkälla (en virtuell högtalare) placerad i spelvärlden. Denna komponent är ansvarig för att faktiskt *spela upp* ett `AudioClip`.
3. **`AudioListener`:** Agerar som en virtuell mikrofon som fångar upp ljud från `AudioSource`-komponenter i scenen. Den finns oftast på huvudkameran (`Main Camera`).

I denna workshop kommer vi att implementera ett grundläggande system (`AudioManager`) för att hantera BGM och SFX genom att styra `AudioSource`-komponenter från kod.

För mer information, se Unitys dokumentation:
https://docs.unity3d.com/Manual/AudioOverview.html

Tidshantering och Sekvenser med Coroutines

När vi utvecklar spel behöver vi ofta exekvera logik som inte sker omedelbart, utan sträcker sig över tid eller sker efter en fördröjning. Standardmetoder i `Update` eller `FixedUpdate` körs varje bildruta, men för sekvenser, väntetider eller effekter som varar en viss tid är **Coroutines** ett kraftfullt verktyg i Unity.

En Coroutine är i grunden en speciell typ av funktion (som returnerar `IEnumerator`) som har förmågan att **pausa sin exekvering** och återuppta den vid ett senare tillfälle, utan att blockera resten av spelets uppdateringsloop. Detta görs med hjälp av `yield return`-instruktioner.

Coroutines låter oss:

1. **Introducera Fördröjningar:** Pausa exekveringen i ett visst antal sekunder (`yield return new WaitForSeconds(delayTime)`).
2. **Vänta på Nästa Bildruta:** Pausa till nästa `Update`-cykel (`yield return null`).
3. **Vänta på Fysikuppdatering:** Pausa till nästa `FixedUpdate`-cykel (`yield return new WaitForFixedUpdate()`).
4. **Vänta på Andra Coroutines:** Starta en annan coroutine och vänta tills den är klar (`yield return StartCoroutine(AnotherCoroutine())`).
5. **Skapa Sekvenser:** Utföra en serie handlingar i ordning, med pauser emellan.
6. **Hantera Logik över Tid:** Implementera beteenden som gradvis förändras eller upprepas under en tidsperiod (t.ex. en blinkande effekt, en attack-cooldown).

Man startar en coroutine från en annan metod (som `Start`, `Update`, eller en eventhanterare) med `StartCoroutine(MyCoroutine())`. I denna workshop kommer vi att använda coroutines för att hantera respawn-

fördröjning, spelarens odödlighet och fiendens attack-cooldown.

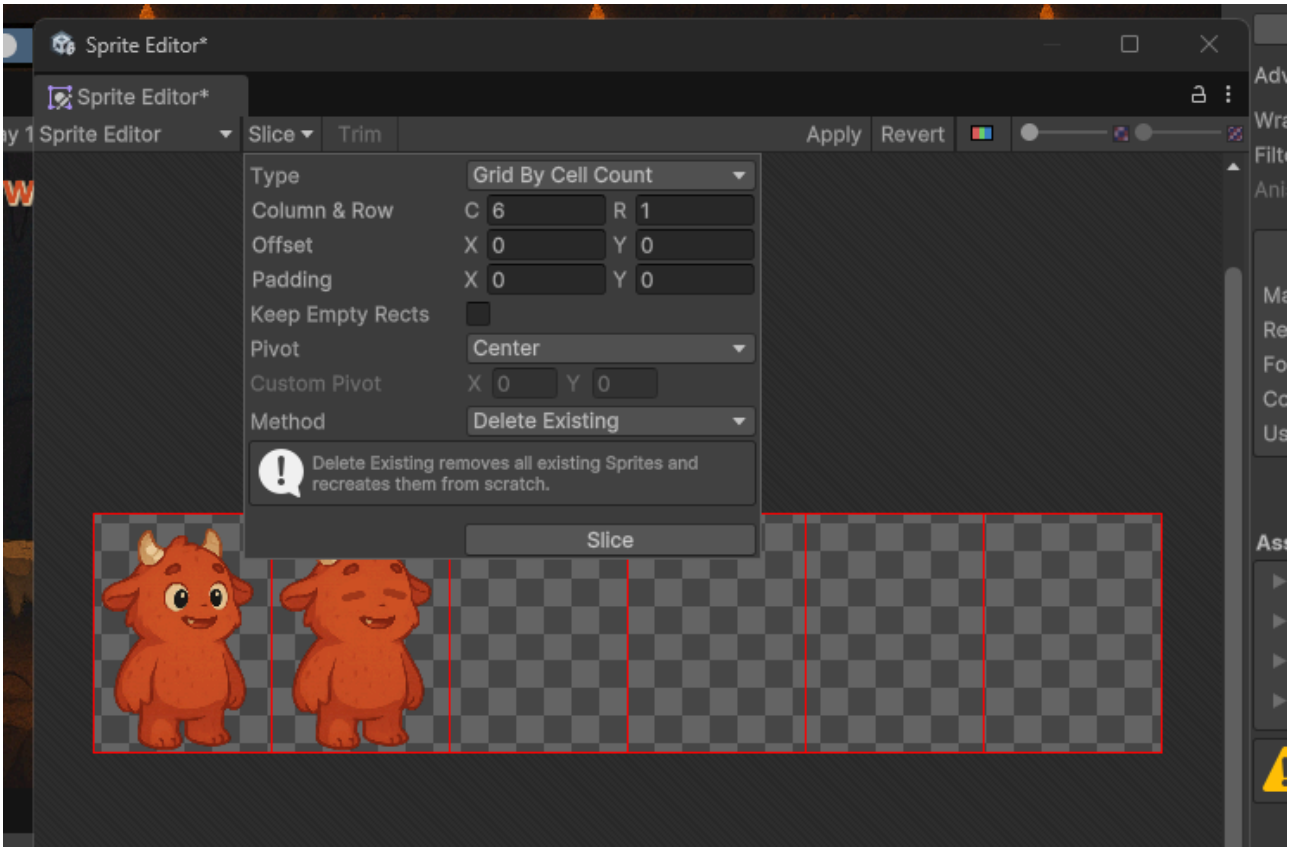
(För extremt enkla, fasta fördröjningar eller repetitioner finns även `Invoke` och `InvokeRepeating`, men de erbjuder mindre flexibilitet än coroutines och täcks inte praktiskt här).

För mer information, se Unitys dokumentation:
https://docs.unity3d.com/Manual/Coroutines.html

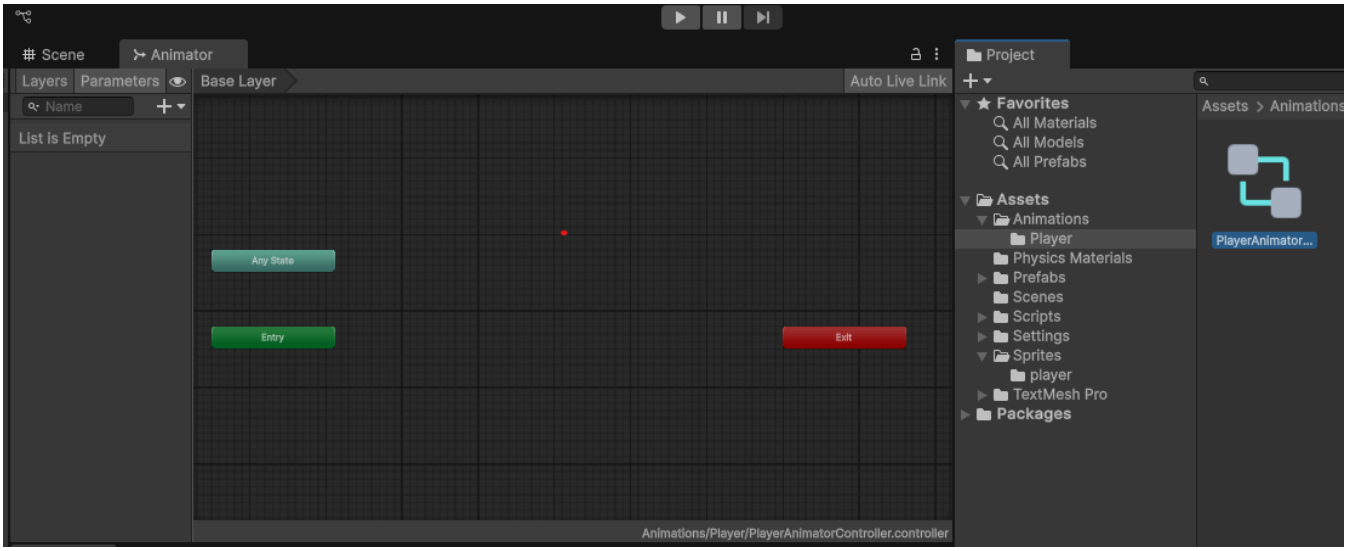
Del 1: Player Animation - Idle & Run

Mål: Skapa och styra spelarens Idle- och Run-animationer.

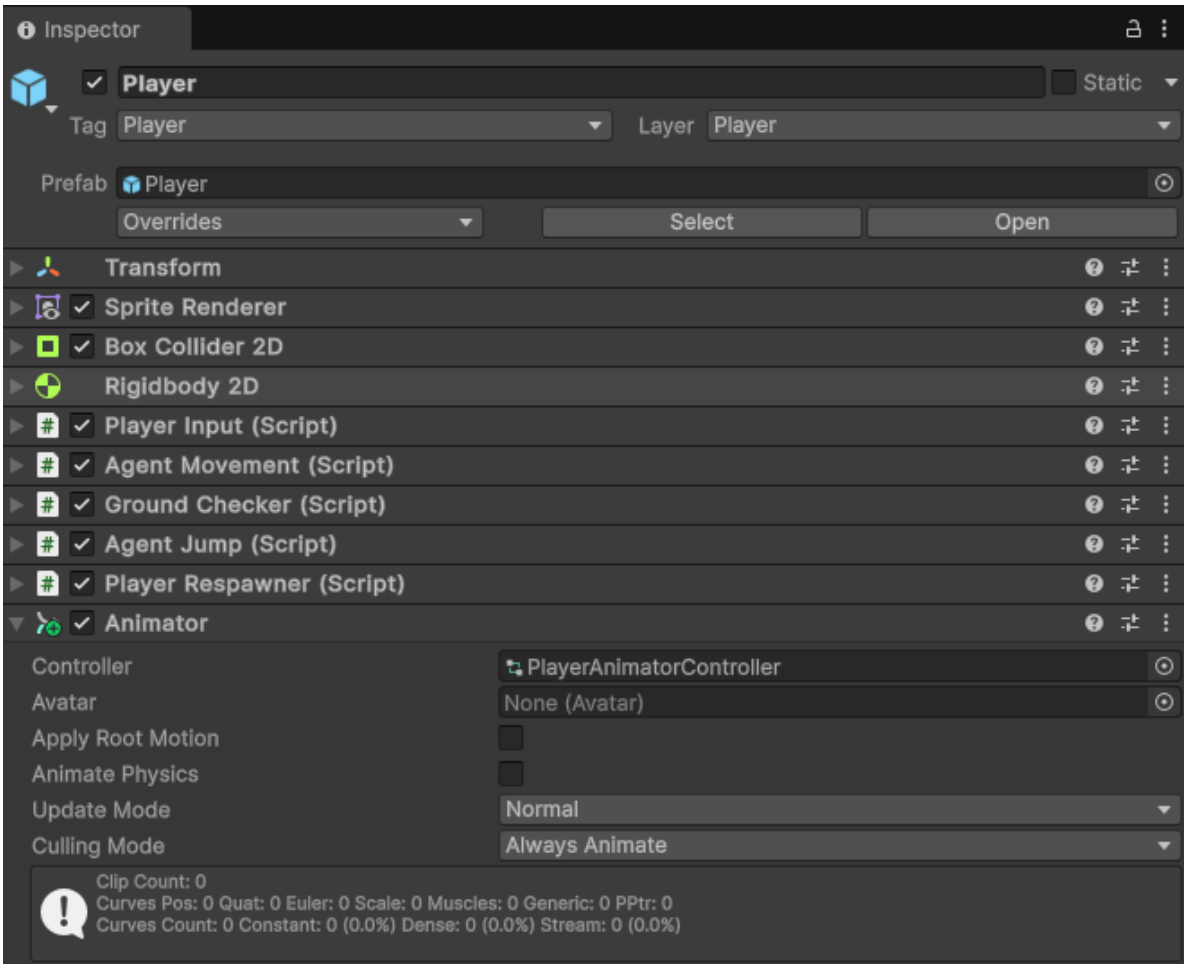
1. **Förbered Sprites:**
   - Importera dina spritesheets/bildsekvenser för Idle och Run (t.ex. `player_anim_idle.png`, `player_anim_run.png`) till ditt Unity-projekt (t.ex. i en mapp `Sprites/Player`).
     Det är inte ovanligt att alla sprites ligger i ett enda sprite sheet men här har vi delat upp det för att det ska bli lite tydligare.
   - Markera varje importerad bildfil. I Inspector:
     - Sätt `Texture Type` till `Sprite (2D and UI)`.
     - Sätt `Sprite Mode` till `Multiple`.
     - Klicka `Apply`.
     - Klicka `Sprite Editor`.
   - I Sprite Editor:
     - Klicka på `Slice`.
     - Sätt `Type` till `Grid By Cell Count`.
     - Ange rätt antal `Columns` och `Rows` för din specifika spritesheet (t.ex. `player_anim_idle.png` kanske är 6 kolumner, 1 rad; `player_anim_run.png` kanske är 6 kolumner, 1 rad).
     - Klicka `Slice` och sedan `Apply`. Stäng Sprite Editor. Nu ska du kunna expandera spritesheet-filen i Project-fönstret för att se de individuella bildrutorna.



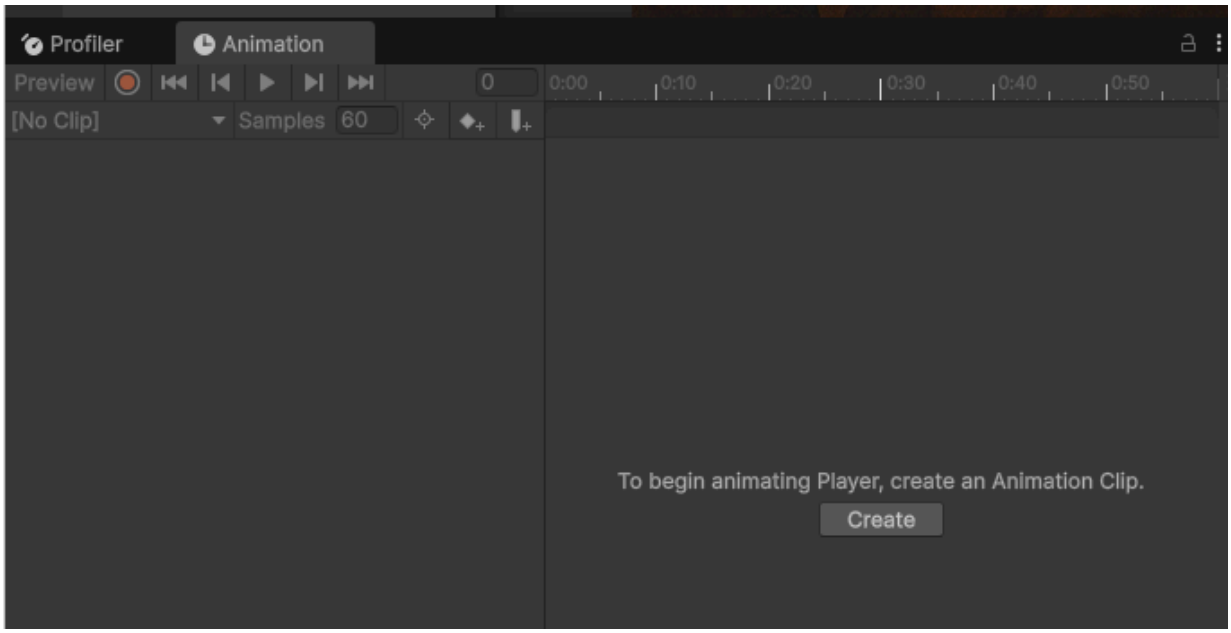
2. **Skapa Animator Controller:**
   - I Project-fönstret, skapa en mapp `Animations/Player` (om den inte finns).
   - Högerklicka i mappen -> `Create` -> `Animator Controller`. Döp den till `PlayerAnimatorController`.



3. **Lägg till Animator Component:**
   - Markera din `Player`-prefab.
   - I Inspector, klicka `Add Component`. Sök efter och lägg till `Animator`.
   - I `Animator`-komponenten, hitta fältet `Controller`. Dra din nyskapade `PlayerAnimatorController`-fil från Project-fönstret till detta fält.

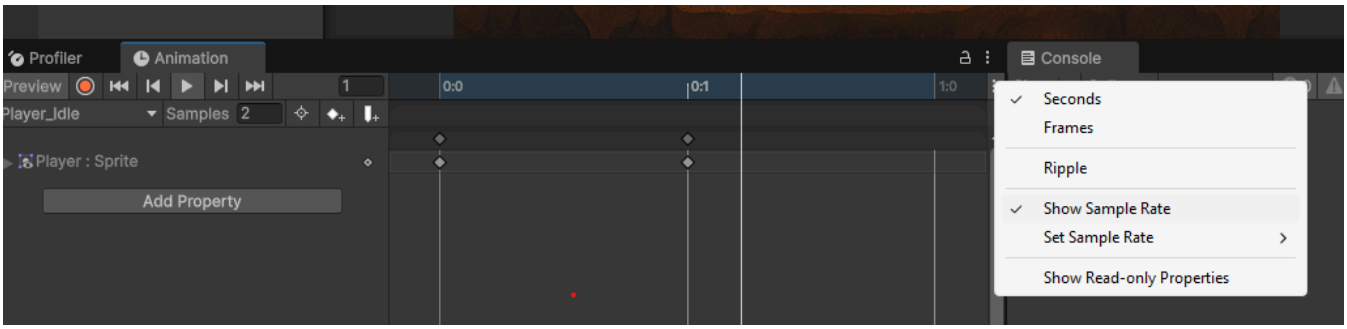


4. **Skapa Animation Clips:**
   - Markera `Player` -prefaben.
   - Öppna `Animation` -fönstret (`Window -> Animation -> Animation`).
   - Om du ser texten "To begin animating [Player], create an Animator and an Animation Clip.", klicka på `Create` -knappen.



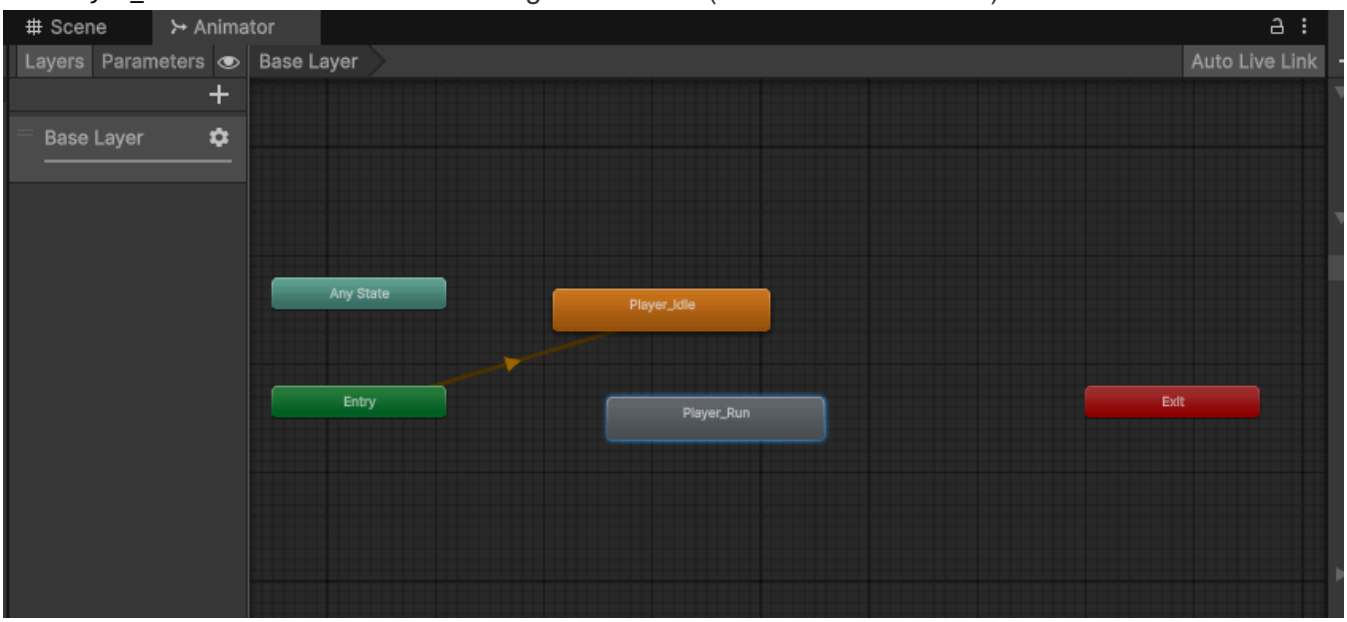
   - Spara animationen som `Player_Idle` i mappen `Animations/Player` .
   - Markera `Player_Idle` i Project-fönstret. Öppna `Animation` -fönstret igen (om det stängdes).

- Dra de individuella Idle-spritesen (det ska bara vara **två stycken**, en med öppna ögon och en med stängda ögon) från din slicade `player_anim_idle.png` (expandera den i Project-fönstret) till tidslinjen i `Animation` -fönstret.
- Justera `Samples` uppe till höger i `Animation` -fönstret (t.ex. **2** funkar ganska bra för denna animation) för att kontrollera uppspelningshastigheten. Värdet anger bilder per sekund. Tryck Play i `Animation` -fönstret för att förhandsgranska. Om du inte ser `Samples` -fältet klicka på de tre prickarna och välj **"Show Sample Rate"**.

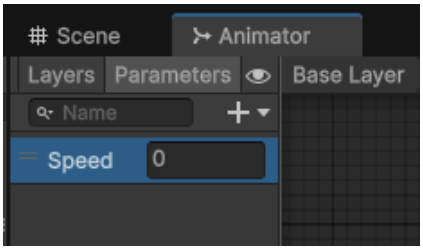


- I `Animation` -fönstret, klicka på dropdown-menyn där det står `Player_Idle` och välj `[Create New Clip...]`.
- Spara den nya animationen som `Player_Run` i mappen `Animations/Player`.
- Dra in Run-spritesen från din slicade `player_anim_run.png` till tidslinjen.
- Justera `Samples` för Run-animationen (t.ex. 12, 16 eller 24). Testa dig fram till en bra hastighet. OBS: animation är rätt ful så det kommer aldrig bli riktigt bra 😀

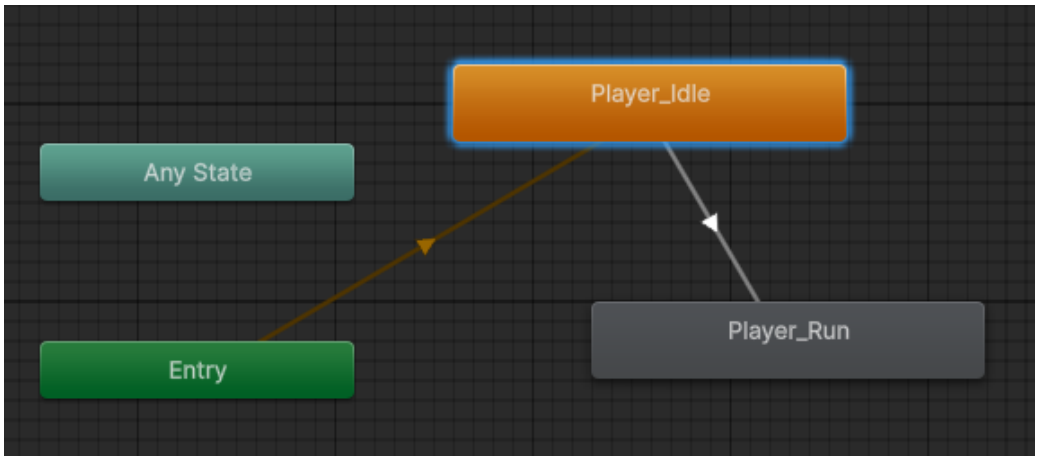
5. **Konfigurera Animator Controller:**
   - Dubbelklicka på `PlayerAnimatorController` -filen i Project-fönstret för att öppna den i `Animator` -fönstret.
   - Du bör se `Player_Idle` som en orange ruta (detta är standardläget, "Entry" pekar på den). Du bör också se `Player_Run`. Dra runt rutorna för att organisera dem (t.ex. Idle ovanför Run).



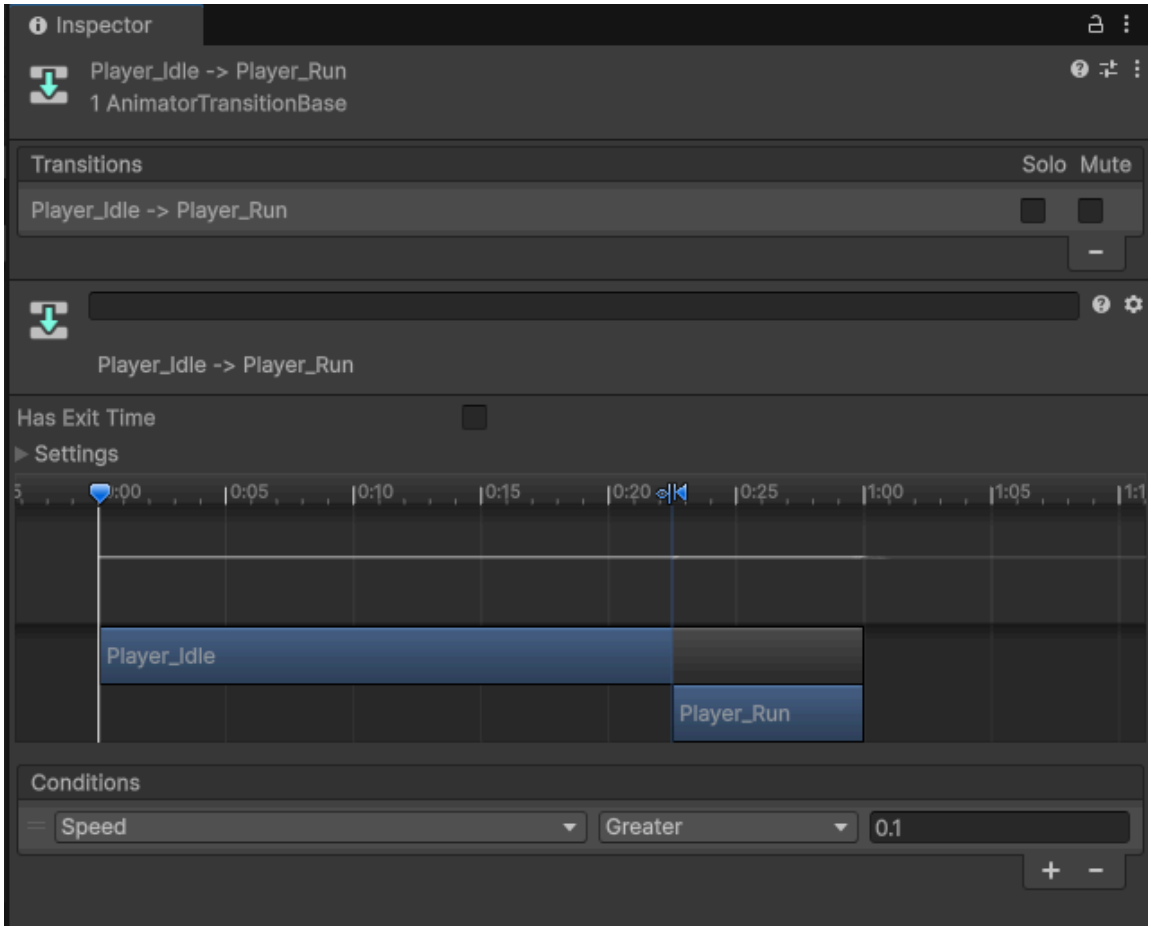
   - **Parameter:** Gå till `Parameters` -fliken (oftast till vänster i Animator-fönstret). Klicka på `+` och välj `Float`. Döp parametern till `Speed`.



- **Transitions (Övergångar):**
  - **Idle -> Run:** Högerklicka på `Player_Idle` -rutan -> `Make Transition` -> klicka på `Player_Run` -rutan. En pil skapas.



  - Markera pilen. I Inspector:
    - Avkryssa `Has Exit Time`.
    - Expandera `Settings`. Sätt `Transition Duration (s)` till `0`.
    - Under `Conditions`, klicka `+`. Listan ska nu visa `Speed`. Ändra `Greater` till `0.1`.



- **Run -> Idle:** Högerklicka på `Player_Run` -rutan -> `Make Transition` -> klicka på `Player_Idle` -rutan.
- Markera den nya pilen. I Inspector:
  - Avkryssa `Has Exit Time` .
  - Sätt `Transition Duration (s)` till `0` .
  - Under `Conditions` , klicka `+` . Ändra `Greater` till `Less` och värdet till `0.1` .

6. **Uppdatera `AgentMovement.cs` :** Lägg till referens till `Animator` , uppdatera `Speed` -parametern i `UpdateAnimator()` baserat på `Mathf.Abs(rb.linearVelocity.x)` .

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
[RequireComponent(typeof(IInput))]
[RequireComponent(typeof(Animator))] //nytt
public class AgentMovement : MonoBehaviour
{
    [Header("Movement Settings")]
    [SerializeField]
    [Tooltip("Hastigheten spelaren rör sig horisontellt med.")]
    private float moveSpeed = 7f;

    private Rigidbody2D rb;
    private IInput inputSource;
    private Animator animator; //nytt

    private bool isFacingRight = true;

    void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        inputSource = GetComponent<IInput>();
        animator = GetComponent<Animator>(); //nytt

        if (rb == null) Debug.LogError("Rigidbody2D saknas!", this);
        if (inputSource == null) Debug.LogError("PlayerInput saknas!", this);
        if (animator == null) Debug.LogError("animator saknas!", this); //nytt
    }

    //lägger till
    void Update() {
        UpdateAnimator();
    }

    void FixedUpdate()
    {
        float moveHorizontal = inputSource.MoveInput.x;
        Vector2 targetVelocity = new Vector2(moveHorizontal * moveSpeed, rb.linearVelocity.y);
        rb.linearVelocity = targetVelocity;
        HandleSpriteFlip(moveHorizontal);
    }

    //Ny metod
    private void UpdateAnimator()
    {
        if (animator != null && rb != null)
        {
            //  Förtydligande om Mathf.Abs: Vi använder Mathf.Abs() för att få absolutvärdet (magnitude
```

```
            float horizontalVelocity = Mathf.Abs(rb.linearVelocity.x);
            animator.SetFloat("Speed", horizontalVelocity);
        }
    }

    private void HandleSpriteFlip(float horizontalInput)
    {
        if (horizontalInput > 0.01f && !isFacingRight)
        {
            Flip();
        }
        else if (horizontalInput < -0.01f && isFacingRight)
        {
            Flip();
        }
    }

    private void Flip()
    {
        isFacingRight = !isFacingRight;

        Vector3 localScale = transform.localScale;
        localScale.x *= -1f;

        transform.localScale = localScale;
    }
}
```

7. **Testa:** Kör spelet. Verifiera att spelaren växlar mellan Player_Idle och Player_Run baserat på horisontell rörelse och att sprite flip fungerar. Observera att spelaren just nu kommer att fortsätta spela Run-animationen i luften om den hoppar medan den springer - detta åtgärdar vi i Del 3.
8. **Enemy-prefaben:** Eftersom våran fiende o spelare delar mycket logik just nu innebär det att vi får lite varningar. För att slippa varningar och animera fienderna så se sitt att Enemy-prefaben har en Animator-komponent och den har en referens till PlayerAnimationController. Då bör fienderna animeras på samma sätt som spelaren. **I ett riktigt spel hade det förmodligen varit egna animation controllers för fienderna.**

Del 2: Audio Fundamentals & Enkel AudioManager (Förenklad Volymhantering)

Mål: Skapa ett system för att spela upp ljud och som automatiskt spelar bakgrundsmusik och hanterar ljudefekter.

1. **Förbered Ljudfiler:**
    - Importera dina ljudfiler (t.ex. `music.mp3`, `jump.mp3`, `deathscream.mp3`) till ditt Unity-projekt, förslagsvis i en mapp som `Audio/Music` och `Audio/SFX`.

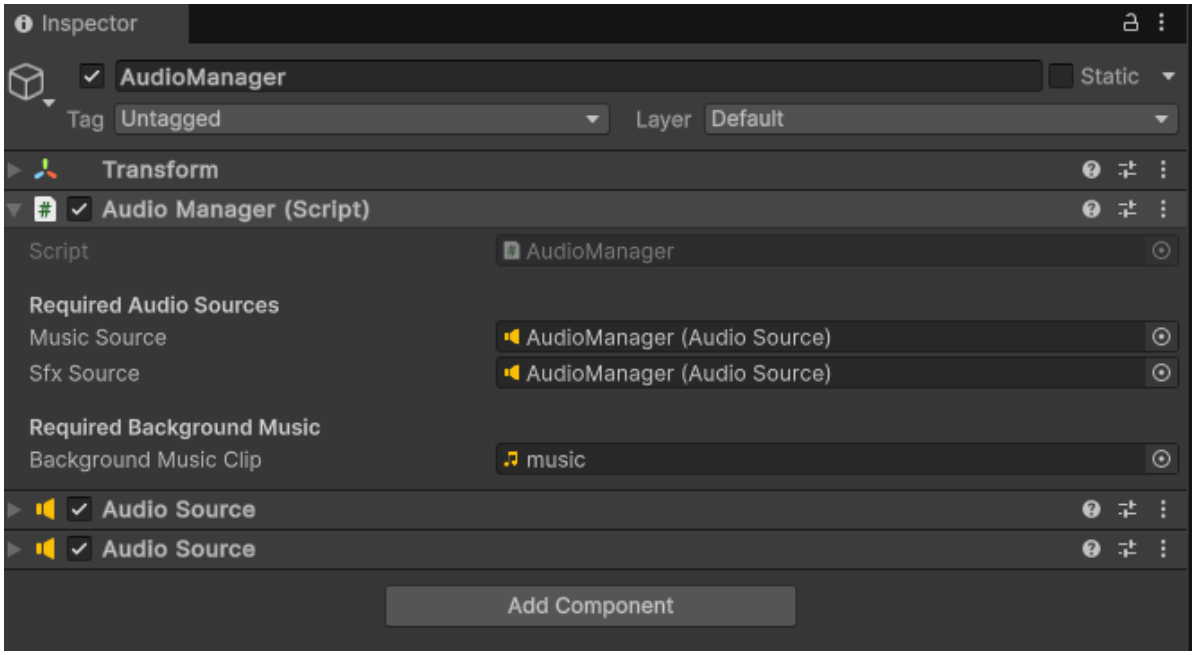
2. **Skapa `AudioManager.cs`:** Singleton, kräver manuell tilldelning av `musicSource`, `sfxSource`, `backgroundMusicClip`. Startar BGM i `Start()`.

```
using UnityEngine;

public class AudioManager : MonoBehaviour
{
    // Singleton instans
    public static AudioManager Instance { get; private set; }

    [Header("Required Audio Sources")]
    [Tooltip("AudioSource dedikerad för BGM. MÅSTE TILLDELAS. Ställ volym direkt på denna komponent.")]
    [SerializeField] private AudioSource musicSource;
    [Tooltip("AudioSource dedikerad för SFX. MÅSTE TILLDELAS. Ställ volym direkt på denna komponent.")]
    [SerializeField] private AudioSource sfxSource;

    [Header("Required Background Music")]
    [Tooltip("Musikfilen för BGM. MÅSTE TILLDELAS.")]
    [SerializeField] private AudioClip backgroundMusicClip;

    void Awake()
    {
        // Singleton-mönster implementation
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
            return;
        }

        // Validera att alla nödvändiga referenser har tilldelats i Inspector.
        bool valid = true;
        if (musicSource == null) { Debug.LogError("AudioManager: Music Source MÅSTE tilldelas i Inspect
        if (sfxSource == null) { Debug.LogError("AudioManager: SFX Source MÅSTE tilldelas i Inspector!'
        if (backgroundMusicClip == null) Debug.LogWarning("AudioManager: Background Music Clip är inte

        if (!valid) {
            Debug.LogError("AudioManager inaktiveras på grund av saknade referenser.", this);
            enabled = false;
            return;
        }

        // Konfigurera de tilldelade AudioSource-komponenterna.
        ConfigureAudioSources();
    }

    void Start()
```

```
{
    // Försök starta bakgrundsmusiken automatiskt.
    PlayBackgroundMusic();
}

// Grundläggande konfiguration för våra Audio Sources.
private void ConfigureAudioSources()
{
    if(musicSource != null)
    {
        // Musiken ska loopa, men inte starta automatiskt (vi startar den i Start).
        musicSource.loop = true;
        musicSource.playOnAwake = false;
    }

    if(sfxSource != null)
    {
        // Ljudeffekter ska inte loopa och inte starta automatiskt.
        sfxSource.loop = false;
        sfxSource.playOnAwake = false;
    }
}

// Metod för att spela ett engångsljud (SFX).
// volumeScale multiplicerar volymen som är inställd på sfxSource komponenten.
public void PlaySoundOneShot(AudioClip clip, float volumeScale = 1.0f)
{
     if (!enabled || sfxSource == null || clip == null) return;

     sfxSource.PlayOneShot(clip, volumeScale);
}

// Metod för att starta eller byta bakgrundsmusik.
private void PlayBackgroundMusic()
{
     if (!enabled || musicSource == null || backgroundMusicClip == null) return;

     if (musicSource.isPlaying) musicSource.Stop();

     musicSource.clip = backgroundMusicClip;

     musicSource.Play();
     Debug.Log($"AudioManager: Playing background music '{backgroundMusicClip.name}' at volume {mus
}

// Metod för att stoppa musiken externt vid behov.
public void StopMusic() {
    if (musicSource != null && musicSource.isPlaying)
```

```
        {
            musicSource.Stop();
            Debug.Log("AudioManager: Background music stopped.");
        }
    }
}
```

3. **Konfigurera i Scenen :**
   - Skapa ett tomt GameObject i din scen. Döp det till `AudioManager`.
   - Markera `AudioManager`-objektet. Dra `AudioManager.cs`-scriptet till Inspector.
   - Lägg till **två** `Audio Source`-komponenter på `AudioManager`-objektet.
   - Dra den **första** **`AudioSource`-komponenten** till fältet `Music Source` i `AudioManager (Script)`.
   - Dra den **andra** **`AudioSource`-komponenten** till fältet `SFX Source`.
   - Dra din **BGM-fil** (t.ex. `music.mp3`) till fältet `Background Music Clip` i `AudioManager (Script)`.



   - Markera **var och en** av de två `AudioSource`-komponenterna och se till att `Play On Awake` är **avkryssat**.
   - **Ställ in Volymen Direkt på Komponenterna:**
     - Markera den **första** **`AudioSource`** (den du drog till `Music Source`). Hitta fältet `Volume` i dess Inspector-inställningar. **Sätt detta till ett lågt värde till att börja med (t.ex. 0.1 eller 0.2)**.
     - Markera den **andra** **`AudioSource`** (den du drog till `SFX Source`). Hitta fältet `Volume` i dess Inspector. Du kan lämna detta på `1` eller justera efter behov för dina ljudeffekter (t.ex. 0.8).
   - **Testa:** Kör spelet. Bakgrundsmusiken bör starta automatiskt med den volym du ställde in på `Music Source`-komponenten. Om ingen musik hörs, kontrollera Console-fönstret för eventuella felmeddelanden. Justera volymen direkt på respektive `AudioSource`-komponent i Inspector vid behov.
     - **OBS:** När du har testat funktionaliteten, vänligen sänk volymen så att du inte stör dina kurskamrater i salen med ljudet! 😊

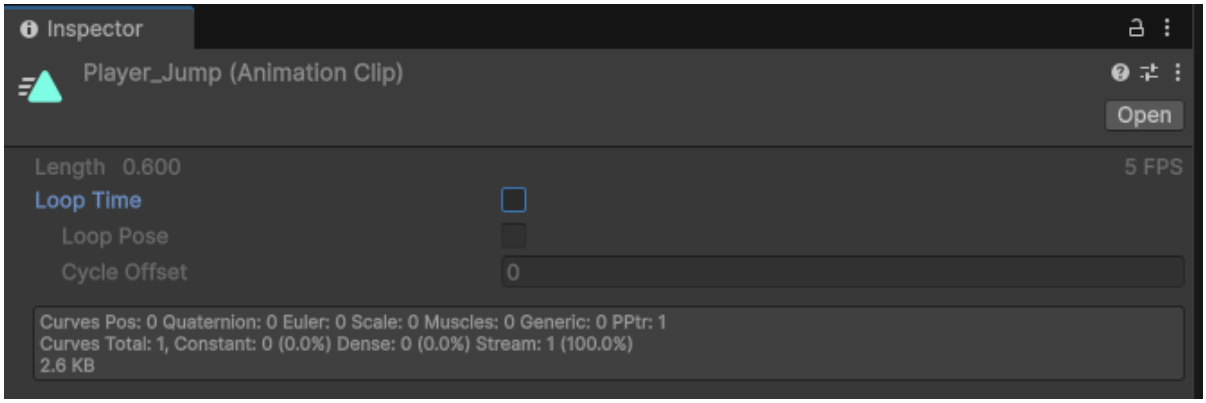
Del 3: Hoppanimation och Ljud

Mål: Lägga till animation och ljud för spelarens hopp och landning, samt korrigera animationen så att spelaren inte springer i luften.

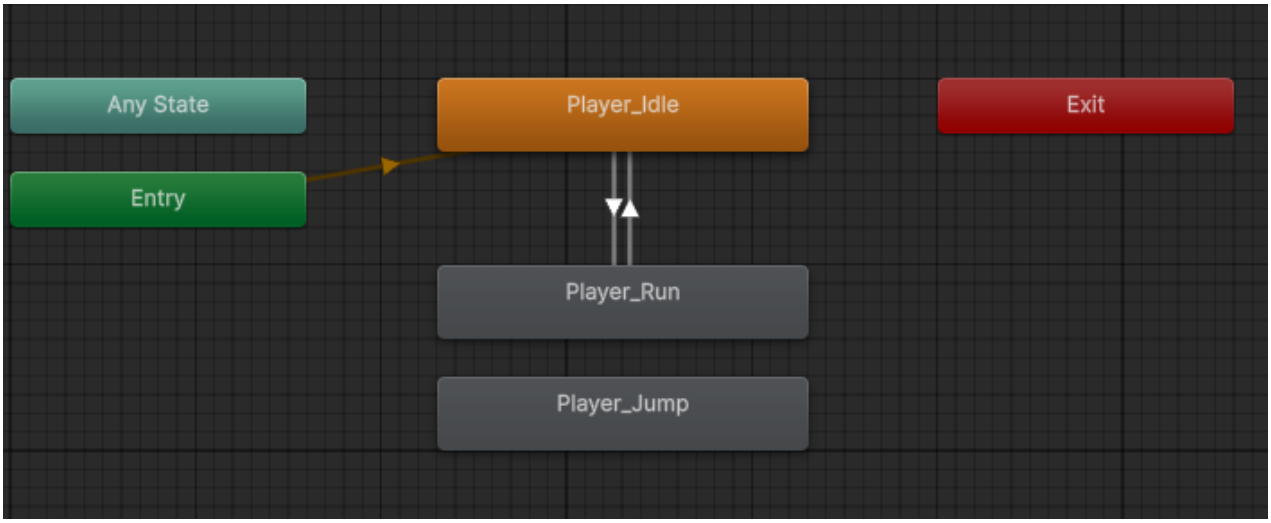
Utöver `Float`-parametern (som vi använde för `Speed` tidigare) behöver vi nu två till typer för att styra hopp- och landningsanimationerna:

- **Bool:** Representerar ett sant/falskt tillstånd. Vi använder detta för `IsGrounded` för att veta om spelaren är på marken eller i luften.
- **Trigger:** Representerar en engångshändelse. Vi använder detta för `Jump` för att starta hoppanimationen precis när spelaren trycker på hoppknappen. Triggern återställs automatiskt.

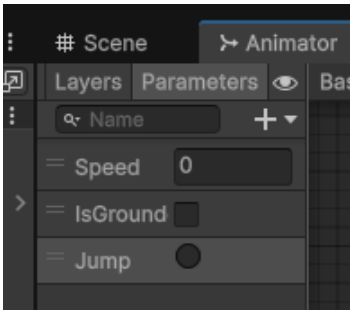
1. **Skapa Animation Clip:** `Player_Jump`.
   - Markera din `Player`-prefab.
   - Öppna `Animation`-fönstret (`Window -> Animation -> Animation`).
   - Klicka på dropdown-menyn där det står `Player_Run` (eller `Player_Idle`) och välj `[Create New Clip...]`.
   - Spara den nya animationen som `Player_Jump` i mappen `Animations/Player`.
   - Dra in dina Jump-sprites (ofta bara en eller två bilder, t.ex. en i luften och kanske en förberedande) till tidslinjen.
     `Player_Jump`-tidslinjen med hopp-sprites.
   - Justera `Samples` efter behov (om det är en sekvens).
   - **VIKTIGT:** Markera `Player_Jump.anim`-filen i Project-fönstret. I Inspector, se till att `Loop Time` **INTE** är ikryssad. Hoppanimationen ska bara spelas en gång per hopp.



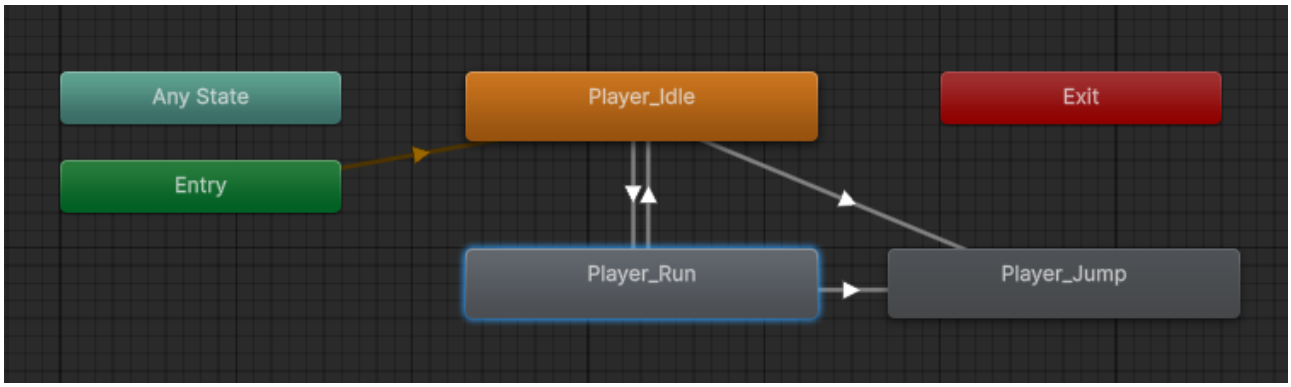
2. **Uppdatera Animator Controller (Jump State & Parameters):**
   - Dubbelklicka på `PlayerAnimatorController` för att öppna `Animator`-fönstret.
   - **States:** Dra in den nyskapade `Player_Jump`-animationen från Project-fönstret till Animator-grafen. Placera den logiskt (t.ex. ovanför Idle/Run).



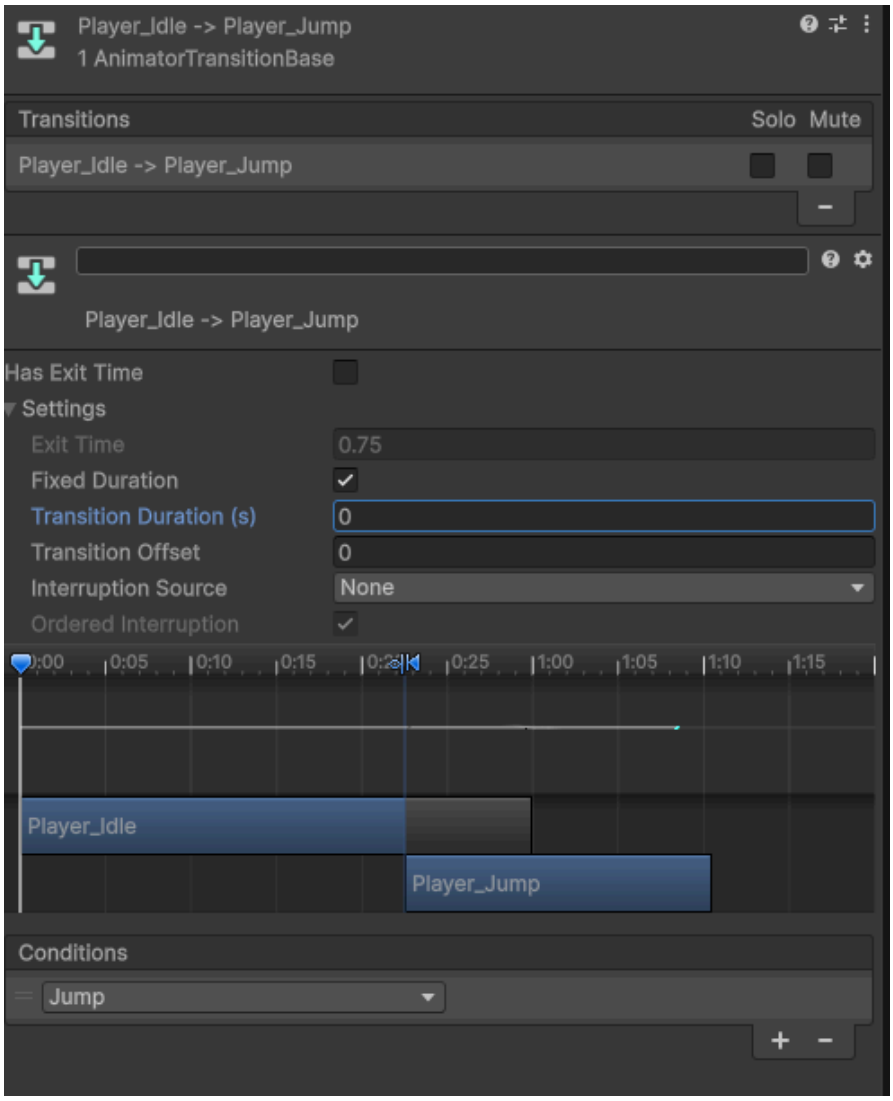
- **Parameters (Nu lägger vi till `IsGrounded` och `Jump`):**
  - Gå till `Parameters`-fliken.
  - Klicka `+` -> `Bool`. Döp den till `IsGrounded`.
  - Klicka `+` -> `Trigger`. Döp den till `Jump`.



3. **Konfigurera Transitions:**
   - **Idle/Run -> Jump (Vid Hopp-Input):**
     - Högerklicka på `Player_Idle` -> `Make Transition` -> Klicka på `Player_Jump`.
     - Högerklicka på `Player_Run` -> `Make Transition` -> Klicka på `Player_Jump`.



     - För bägge nya pilarna:
       - Avkryssa `Has Exit Time`.
       - Sätt `Transition Duration (s)` till `0`.
       - Under `Conditions`, klicka `+`. Välj `Jump`-triggern. *(Detta triggas av kod när spelaren trycker på hoppknappen).*



- **Jump -> Idle/Run (Vid Landning):**
  - Högerklicka på `Player_Jump` -> `Make Transition` -> Klicka på `Player_Idle`. Markera pilen:
    - **Kryssa i** **`Has Exit Time`**. Sätt `Exit Time` till ca `0.75` (justera efter din animation).
    - Sätt `Transition Duration (s)` till `0.1`.
    - Under `Conditions`, klicka `+`: `IsGrounded` `true`.
    - Klicka `+` igen: `Speed` `Less` `0.1`. *(Gå till Idle om landat och står still).*
  - Högerklicka på `Player_Jump` -> `Make Transition` -> Klicka på `Player_Run`. Markera pilen:
    - **Kryssa i** **`Has Exit Time`**. Sätt `Exit Time` till samma värde (t.ex. `0.75`).
    - Sätt `Transition Duration (s)` till `0.1`.
    - Under `Conditions`, klicka `+`: `IsGrounded` `true`.
    - Klicka `+` igen: `Speed` `Greater` `0.1`. *(Gå till Run om landat och rör sig).*

> **Brasklapp:** Det här är lite rörigt och jag hoppas jag har fått allt rätt men det vara så att jag har gjort några tankevurpor i logiken. Även olika tider bör tas med en nypa salt. Prova och testa lite olika värden tills ni är hyffsat nöjda. Efter ni har uppdaterat skripten nedan.

4. **Uppdatera** **`GroundChecker.cs`** **(Nu lägger vi till Animator-koppling):** Sätter `IsGrounded` -parametern i Animator, spelar `landingSoundClip` via `AudioManager.Instance.PlaySoundOneShot()` vid landning.

```
using UnityEngine;

public class GroundChecker : MonoBehaviour
{
    [Header("Ground Check Settings")]
    [SerializeField]
    [Tooltip("Ett tomt GameObject placerat vid spelarens fötter, varifrån markkontrollen utgår.")]
    private Transform groundCheckPoint;

    [SerializeField]
    [Tooltip("Vilka physics layers som räknas som 'mark'.")]
    private LayerMask groundLayer;

    [SerializeField]
    [Tooltip("Radien på cirkeln som används för att kontrollera markkontakt.")]
    private float groundCheckRadius = 0.15f;

    // NYTT
    [Header("Animation & Audio")]
    [Tooltip("Valfritt: Animator på detta eller förälderobjektet som ska styras. Om tom, försöker hitta
    [SerializeField] private Animator parentAnimator; // Referens till Animatorn
    [Tooltip("Ljud som spelas vid landning (frivilligt). Dra in ljudfil här.")]
    [SerializeField] private AudioClip landingSoundClip;


    public bool IsGrounded { get; private set; }
    private bool wasGroundedLastFrame;

    void Awake()
    {
        if (groundCheckPoint == null)
        {
            Debug.LogError("Ground Check Point är INTE tilldelad i GroundChecker! Markkontroll fungerar
        }
        if (groundLayer == 0 || groundLayer == -1)
        {
            Debug.LogWarning("Ingen specifik Ground Layer är vald i GroundChecker.", this);
        }

        //NYTT: Försök hitta Animator automatiskt
        if (parentAnimator == null)
        {
            parentAnimator = GetComponentInParent<Animator>(); // Försöker hitta Animator-komponenten.

            if (parentAnimator == null) Debug.LogWarning("Ingen Animator tilldelad eller hittades i för
        }
```

```
    if (landingSoundClip == null) Debug.LogWarning("Landing Sound Clip ej tilldelat i GroundChecker
}

void Update()
{
    UpdateAnimator(); // Uppdatera Animatorn varje frame
}

void FixedUpdate() // Markkontroll fortfarande i FixedUpdate
{
    wasGroundedLastFrame = IsGrounded; // Spara föregående status

    if (groundCheckPoint != null)
    {
        IsGrounded = Physics2D.OverlapCircle(
            groundCheckPoint.position,
            groundCheckRadius,
            groundLayer
        );
    }
    else
    {
        IsGrounded = false;
    }

    HandleLanding(); //NYTT  Kolla om vi precis landat
}

// NYTT
private void UpdateAnimator()
{
    // Säkerhetskoll innan vi sätter parametern
    if (parentAnimator != null && parentAnimator.isActiveAndEnabled && parentAnimator.runtimeAnimat
    {
        // Sätt IsGrounded-parametern i Animator Controller
        parentAnimator.SetBool("IsGrounded", IsGrounded);
    }
}

//NYTT: Metod för att hantera landning (ljud)
private void HandleLanding()
{
    // Om vi landade precis denna frame...
    if (IsGrounded && !wasGroundedLastFrame)
    {
        PlayLandingSound(); // ...spela landningsljud.
```

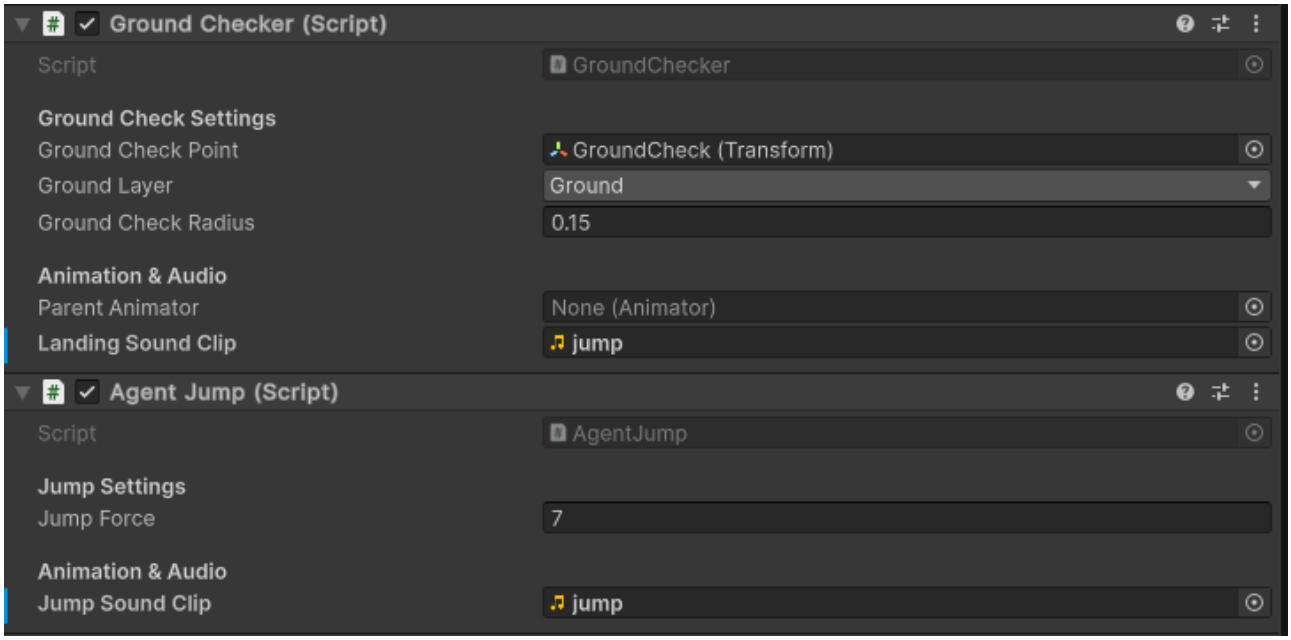
```
        }
    }

    //  NYTT
    private void PlayLandingSound()
    {
        if (landingSoundClip != null && AudioManager.Instance != null)
        {
            // Spela ljud via AudioManager, med en liten volymjustering (80%)
            AudioManager.Instance.PlaySoundOneShot(landingSoundClip, 0.8f);
        }
        else if (landingSoundClip != null && AudioManager.Instance == null)
        {
            Debug.LogWarning("LandingSoundClip finns, men AudioManager.Instance hittades inte!", this);
        }
    }

    void OnDrawGizmosSelected()
    {
        if (groundCheckPoint != null)
        {
            Gizmos.color = Color.yellow;
            Gizmos.DrawWireSphere(groundCheckPoint.position, groundCheckRadius);
        }
    }
}
```

5. **Uppdatera `AgentJump.cs`:** Modifiera `AgentJump.cs` för att trigga `Jump`-parametern i Animator och spela `jumpSoundClip` via `AudioManager`.

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
[RequireComponent(typeof(IInput))]
[RequireComponent(typeof(GroundChecker))]
public class AgentJump : MonoBehaviour
{
    [Header("Jump Settings")]
    [SerializeField]
    [Tooltip("Kraften som appliceras när spelaren hoppar.")]
    private float jumpForce = 15f;

    [Header("Animation & Audio")] // nytt
    [SerializeField] private AudioClip jumpSoundClip; // nytt - Ljudfil för hopp
    private Animator animator; // nytt - Referens till Animator

    // Ursprungliga variabler
    private Rigidbody2D rb;
    private IInput inputSource;
    private GroundChecker groundChecker;
    private bool jump;

    void Awake()
    {
        // Ursprunglig hämtning
        rb = GetComponent<Rigidbody2D>();
        inputSource = GetComponent<IInput>();
        groundChecker = GetComponent<GroundChecker>();
        animator = GetComponent<Animator>(); // nytt - Hämta Animator

        // Ursprunglig validering + nytt
        if (rb == null) Debug.LogError("Rigidbody2D saknas!", this);
        if (inputSource == null) Debug.LogError("IInput (PlayerInput) saknas!", this);
        if (groundChecker == null) Debug.LogError("GroundChecker saknas!", this);
        if (animator == null) Debug.LogWarning("Animator saknas på AgentJump.", this); // nytt - Varnir
        if (jumpSoundClip == null) Debug.LogWarning("Jump Sound Clip ej tilldelat.", this); // nytt - \
    }

    private void Update() // Läs input i Update
    {
        // Ursprunglig kontroll för hopp-input
        if (inputSource.JumpTriggered && groundChecker.IsGrounded)
        {
            if (animator != null) animator.SetTrigger("Jump"); // nytt - Trigga animation
            if (AudioManager.Instance != null && jumpSoundClip != null) // nytt - Spela ljud
                AudioManager.Instance.PlaySoundOneShot(jumpSoundClip); // nytt - Spela ljud

            jump = true; // Sätt ursprunglig flagga för FixedUpdate
```

```
        }
    }

    void FixedUpdate() // Applicera fysik i FixedUpdate
    {
        // Ursprunglig logik
        if (jump)
            PerformJump();
    }

    private void PerformJump()
    {
        // Ursprunglig logik
        rb.linearVelocity = new Vector2(rb.linearVelocity.x, 0f);
        rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);

        jump = false; // Återställ flaggan här (som i originalet)
    }
}
```

6. **Konfigurera & Testa:**
   - Markera din `Player`-prefab.
   - I `AgentJump`-komponenten:
     - Dra eventuellt spelarens `Animator`-komponent till fältet `Animator` (om den inte sitter på samma GameObject).
     - Dra din hopp-ljudfil (t.ex. `jump.mp3`) till fältet `Jump Sound Clip`.
   - I `GroundChecker`-komponenten:
     - Dra eventuellt spelarens `Animator`-komponent till fältet `Parent Animator` (om den sitter på en förälder och inte hittades automatiskt).
     - Dra din landnings-ljudfil (om du har en, t.ex. `landing.mp3`) till fältet `Landing Sound Clip`.

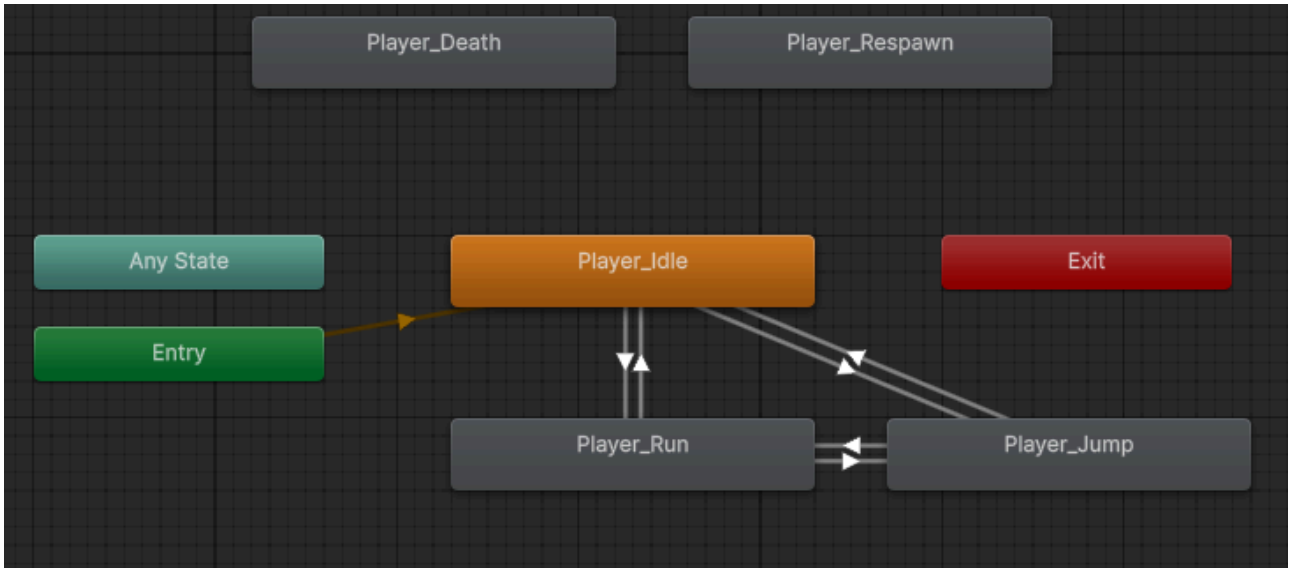


   - Kör spelet. Testa att hoppa och springa.

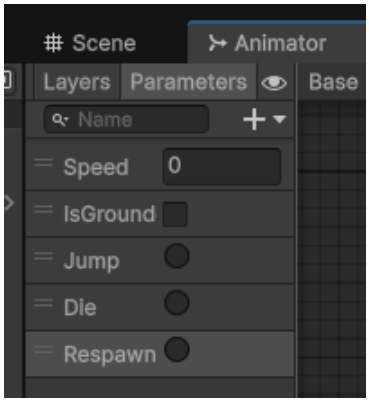
Del 4: Döds- och Respawn-sekvens (Coroutines + Animation/Ljud)

Mål: Implementera en komplett döds- och respawn-sekvens: trigga dödsanimation och ljud, använda en **Coroutine** för att skapa en fördröjning innan spelaren flyttas, och sedan använda en annan **Coroutine** för att hantera en tidsbegränsad odödlighetsperiod med visuell feedback (blinkande) efter respawn.

1. **Skapa Animation Clips & Uppdatera Animator:**
   - **Skapa Clips:**
     - Skapa ett nytt animation clip med namnet `Player_Death` (liknande hur du skapade Idle/Run/Jump). Dra in dina döds-sprites. Se till att `Loop Time` **INTE** är ikryssat för denna animation.
     - Skapa ett till clip med namnet `Player_Respawn`. Dra in dina respawn-sprites. Se till att `Loop Time` **INTE** är ikryssat.
   - **Uppdatera Animator Controller (`PlayerAnimatorController`):**
     - **States:** Dra in `Player_Death` och `Player_Respawn` från Project-fönstret till Animator-grafen, om dom inte redan ligger där. Placera dem logiskt, t.ex. `Player_Death` separat och `Player_Respawn` nära `Player_Idle`.

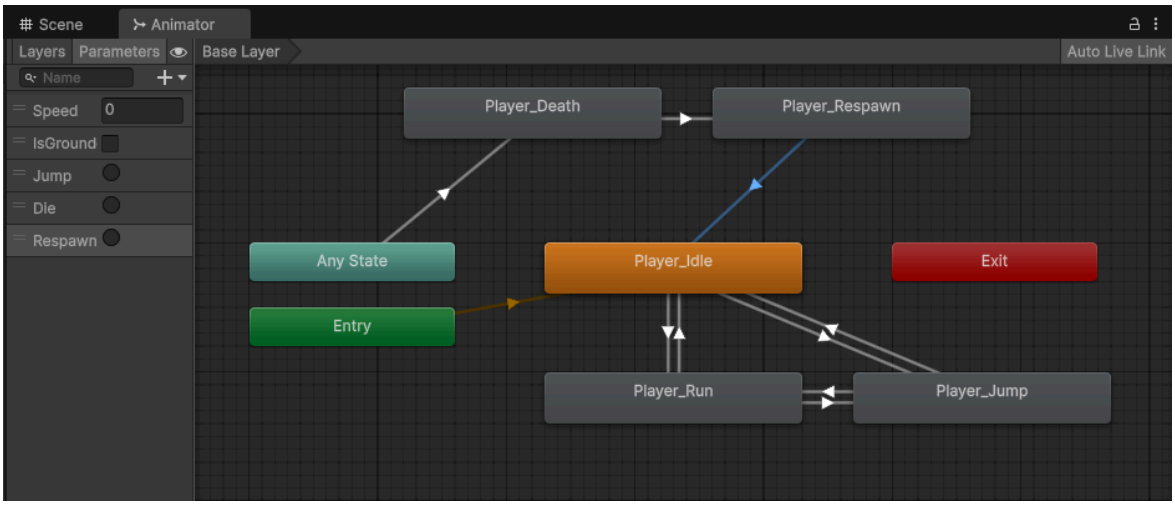


     - **Parameters:**
       - Klicka `+` -> `Trigger`. Döp den till `Die`.
       - Klicka `+` -> `Trigger`. Döp den till `Respawn`.



     - **Transitions:**

- **`Any State -> Player_Death`:** Högerklicka på `Any State` (den turkosa rutan) -> `Make Transition` -> Klicka på `Player_Death`. Markera pilen:
  Förklaring: `Any State` är en speciell nod som låter en transition ske från *vilken annan state som helst* när villkoret uppfylls. Detta är perfekt för dödsanimationen, som ska kunna triggas oavsett om spelaren är Idle, Run, eller Jump.
  - Sätt `Transition Duration (s)` till `0`. (Omedelbar övergång till dödsanimation).
  - Avkryssa `Has Exit Time`. (Vi vill inte vänta på att någon annan animation ska spelas klart).
  - Under `Conditions`, klicka `+`, välj `Die`.
- **`Player_Death -> Player_Respawn`:** Högerklicka på `Player_Death` -> `Make Transition` -> Klicka på `Player_Respawn`. Markera pilen:
  - Sätt `Transition Duration (s)` till `0`.
  - Avkryssa `Has Exit Time`.
  - Under `Conditions`, klicka `+`, välj `Respawn`.
- **`Player_Respawn -> Player_Idle`:** Högerklicka på `Player_Respawn` -> `Make Transition` -> Klicka på `Player_Idle`. Markera pilen:
  - **Kryssa i `Has Exit Time`**. Sätt `Exit Time` till `1`. (`Exit Time` 1 betyder att hela `Player_Respawn`-animationen måste spelas klart innan övergången till Idle sker). Justera vid behov om din respawn-animation är kortare eller längre och du vill tajma övergången annorlunda.
  - Sätt `Transition Duration (s)` till `0` (eller en mycket kort tid, t.ex. 0.1).
  - Inga Conditions behövs (övergår automatiskt när animationen är klar).



2. **Modifiera `PlayerRespawner.cs` (Kärnlogiken - Använder nu Coroutine för delay och invincibility):**

```
using System.Collections;
using UnityEngine;

public class PlayerRespawner : MonoBehaviour
{
    [Header("Respawn Settings")]
    [SerializeField]
    [Tooltip("Ett tomt GameObject som markerar position och rotation där spelaren ska återuppstå.")]
    private Transform startPosition;
    [Tooltip("Hur många sekunder spelet väntar efter död innan respawn initieras.")]
    [SerializeField] private float respawnDelay = 1.0f;

    [Header("Death & Respawn Effects")]
    [Tooltip("Ljud som spelas när spelaren dör.")]
    [SerializeField] private AudioClip deathScreamClip;
    [Tooltip("Ljud som spelas när spelaren återuppstår.")]
    [SerializeField] private AudioClip respawnSoundClip;

    [Header("Invincibility")]
    [Tooltip("Hur många sekunder spelaren är odödlig efter respawn.")]
    [SerializeField] private float invincibilityDuration = 1.5f;
    [Tooltip("Hur snabbt spelaren blinkar under odödlighet (intervall i sek).")]
    [SerializeField] private float blinkInterval = 0.1f;

    // Komponentreferenser
    private Rigidbody2D rb;
    private Collider2D playerCollider;
    private Animator animator;
    private SpriteRenderer spriteRenderer;
    private IInput playerInput;
    private AgentMovement playerMovement;

    // State Variabler
    private int killZoneLayer;
    private bool isDead = false; // Håller koll på om spelaren är i döds/respawn-processen
    private Coroutine activeDeathSequence = null; // Referens till pågående döds-coroutine
    private Coroutine invincibilityCoroutine = null; // Referens till pågående odödlighets-coroutine

    void Awake()
    {
        // Hämta komponenter
        rb = GetComponent<Rigidbody2D>();
        playerCollider = GetComponent<Collider2D>();
        animator = GetComponent<Animator>();
        spriteRenderer = GetComponent<SpriteRenderer>();
        playerInput = GetComponent<IInput>();
        playerMovement = GetComponent<AgentMovement>();
```

```
    // Validera
    if (rb == null || playerCollider == null || animator == null || spriteRenderer == null || playe
        Debug.LogError("PlayerRespawner: En eller flera nödvändiga komponenter saknas!", this);

    if (startPosition == null)
    {
        Debug.LogError("PlayerRespawner: Start Position är INTE tilldelad!", this);
        enabled = false;
        return;
    }

    if (deathScreamClip == null) Debug.LogWarning("PlayerRespawner: Death Scream Clip ej tilldelat.
    if (respawnSoundClip == null) Debug.LogWarning("PlayerRespawner: Respawn Sound Clip ej tilldela

    killZoneLayer = LayerMask.NameToLayer("KillZone");
    if (killZoneLayer == -1)
    {
        Debug.LogError("Physics Layer 'KillZone' hittades inte!", this);
    }
}

private void OnTriggerEnter2D(Collider2D other)
{
    HandlePotentialDeathSource(other.gameObject);
}

private void OnCollisionEnter2D(Collision2D collision)
{
    HandlePotentialDeathSource(collision.gameObject);
}

private void HandlePotentialDeathSource(GameObject sourceObject)
{
    // Gör inget om vi redan är döda, inaktiva eller odödliga
    if (!enabled || isDead || invincibilityCoroutine != null) return;

    bool shouldDie = false;
    if (sourceObject.layer == killZoneLayer) shouldDie = true;
    else if (sourceObject.CompareTag("Enemy")) shouldDie = true;
    else if (sourceObject.CompareTag("Projectile")) shouldDie = true;

    if (shouldDie && activeDeathSequence == null) // Starta bara om ingen sekvens redan pågår
    {
        Debug.Log($"Player hit by {sourceObject.name}. Starting Death Sequence...");
        activeDeathSequence = StartCoroutine(DeathSequenceRoutine());
    }
}
```

```
// Coroutine för dödssekvensen
private IEnumerator DeathSequenceRoutine()
{
    if (isDead) yield break; // Säkerhetscheck
    isDead = true;
    Debug.Log("Player Death Sequence Started.");

    StopInvincibility(); // Stoppa ev. tidigare odödlighet

    // 1. Inaktivera spelarkontroll och fysik
    rb.linearVelocity = Vector2.zero;
    rb.simulated = false;
    playerCollider.enabled = false;
    if (playerMovement != null) playerMovement.enabled = false;
    MonoBehaviour inputComponent = playerInput as MonoBehaviour; // Cast för att kunna inaktivera
    if (inputComponent != null) inputComponent.enabled = false;

    // 2. Rapportera dödsfall (om UIManager finns)
    if (UIManager.Instance != null) UIManager.Instance.IncrementDeaths();
    else Debug.LogWarning("UIManager not found for reporting death.");

    // 3. Spela dödseffekter
    if (animator != null) animator.SetTrigger("Die");
    if (deathScreamClip != null && AudioManager.Instance != null)
        AudioManager.Instance.PlaySoundOneShot(deathScreamClip);
    else if(deathScreamClip != null) Debug.LogWarning("AudioManager not found for death scream.");

    // 4. Vänta för respawn-fördröjning
    Debug.Log($"Waiting {respawnDelay} seconds before respawn...");
    yield return new WaitForSeconds(respawnDelay);

    // Fortsätt med Respawn
    Debug.Log("Respawn delay finished. Initiating respawn...");
    InitiateRespawn();

    activeDeathSequence = null; // Markera att denna coroutine är klar
}

private void InitiateRespawn()
{
    if (!this.enabled) return; // Om scriptet inaktiverats under väntan
    Debug.Log("Initiating Player Respawn...");

    // 1. Flytta spelaren
    transform.position = startPosition.position;
    transform.rotation = startPosition.rotation;

    // 2. Starta respawn-effekter
```

```
    if (animator != null) animator.SetTrigger("Respawn");
    if (respawnSoundClip != null && AudioManager.Instance != null)
        AudioManager.Instance.PlaySoundOneShot(respawnSoundClip);
    else if(respawnSoundClip != null) Debug.LogWarning("AudioManager not found for respawn sound.");

    // 3. Återaktivera fysik/kolliderare (kontroller återaktiveras efter odödlighet)
    rb.simulated = true;
    playerCollider.enabled = true;

    // 4. Starta odödlighets-coroutine
    if (invincibilityCoroutine != null) StopCoroutine(invincibilityCoroutine); // Stoppa ev. gammal
    invincibilityCoroutine = StartCoroutine(InvincibilityRoutine(invincibilityDuration));
}

private IEnumerator InvincibilityRoutine(float duration)
{
    Debug.Log($"Invincibility Started for {duration} seconds.");
    float endTime = Time.time + duration;
    bool visible = false; // Starta osynlig för första blinkningen

    while (Time.time < endTime)
    {
        if(spriteRenderer != null) spriteRenderer.enabled = visible; // Växla synlighet
        visible = !visible;
        yield return new WaitForSeconds(blinkInterval); // Pausa för blink-intervall
    }

    Debug.Log("Invincibility Ended.");
    if(spriteRenderer != null) spriteRenderer.enabled = true; // Se till att vara synlig

    // Återaktivera kontroller
    if (playerMovement != null) playerMovement.enabled = true;
    MonoBehaviour inputComponent = playerInput as MonoBehaviour;
    if (inputComponent != null) inputComponent.enabled = true;

    // Återställ animator triggers
    if (animator != null)
    {
        animator.ResetTrigger("Die");
        animator.ResetTrigger("Respawn");
    }

    // Markera att spelaren kan dö igen
    isDead = false;
    invincibilityCoroutine = null;
    Debug.Log("Player is vulnerable again.");
}
```

```
    // Metod för att stoppa odödlighet i förtid
    private void StopInvincibility()
    {
        if (invincibilityCoroutine != null)
        {
            Debug.Log("Stopping active invincibility coroutine.");
            StopCoroutine(invincibilityCoroutine);
            invincibilityCoroutine = null;
            if(spriteRenderer != null) spriteRenderer.enabled = true; // Säkerställ synlighet
        }
    }

    // Städning vid inaktivering/förstöring
    void OnDisable()
    {
        // Stoppa coroutines för att undvika problem
        if (activeDeathSequence != null)
        {
            StopCoroutine(activeDeathSequence);
            activeDeathSequence = null;
        }
        StopInvincibility();

        // Återställ state om inaktiverad medan "död"
        if (isDead)
        {
            Debug.LogWarning("PlayerRespawner disabled while 'isDead'. Resetting player state.");
            if (rb != null) rb.simulated = true;
            if (playerCollider != null) playerCollider.enabled = true;
            if (spriteRenderer != null) spriteRenderer.enabled = true;
            if (playerMovement != null) playerMovement.enabled = true;
            MonoBehaviour inputComponent = playerInput as MonoBehaviour;
            if (inputComponent != null) inputComponent.enabled = true;
            isDead = false;
        }
    }
}
```

Projektilskriptet

Om ditt `Projectile.cs` tidigare anropade `player.Respawn()` direkt vid träff måste detta tas bort.
Det korrekta flödet är nu:

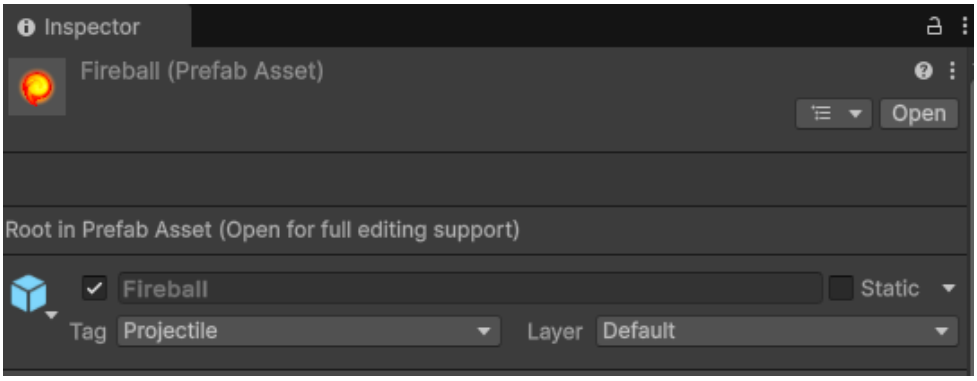
- Kollisionen mellan projektil och spelare ska endast aktivera `HandlePotentialDeathSource` i `PlayerRespawner` (via `OnTriggerEnter2D` eller `OnCollisionEnter2D`)
- Detta fungerar under förutsättning att projektilen har rätt tagg (`"Projectile"`) och/eller lager
- `PlayerRespawner` hanterar sedan hela döds- och respawnprocessen automatiskt

Projektiltaggar

För att `HandlePotentialDeathSource` ska kunna identifiera projektiler via `sourceObject.CompareTag("Projectile")` måste:

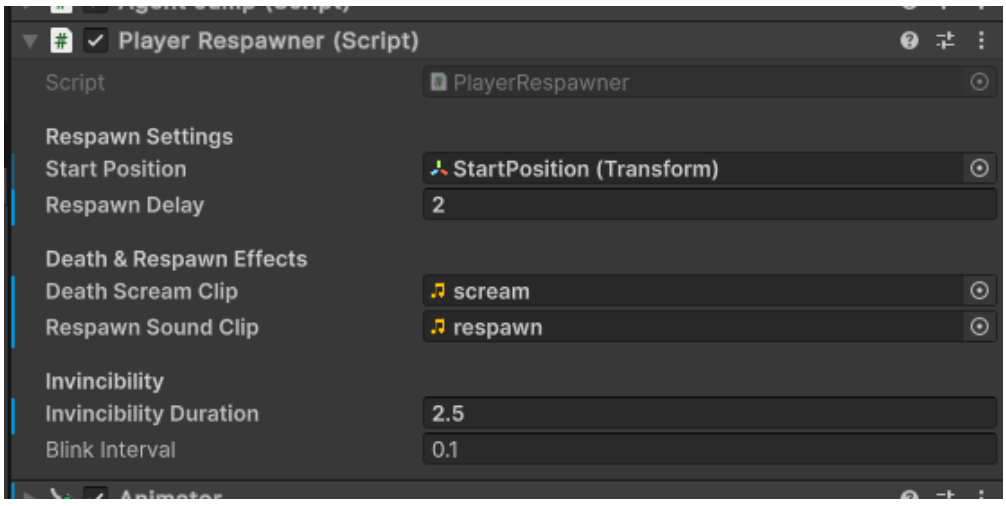
i. Taggen "Projectile" finnas i ditt projekt
   - Skapa taggen om den inte finns via: Edit → Project Settings → Tags and Layers → Tags

ii. Dina projektil-prefabs måste ha taggen "Projectile" tilldelad i Unity Editor



Kommentar om kodens storlek: `PlayerRespawner.cs` har nu växt betydligt och hanterar flera ansvarsområden: kollisionsdetektering, inaktivering/återaktivering av spelarkomponenter, animationstriggar, ljuduppspelning, tidsfördröjning, och odödlighet. För ett större projekt skulle man kunna överväga att refaktorisera detta. Exempelvis skulle odödlighetslogiken (coroutine, variabler, StopInvincibility) kunna brytas ut till en egen komponent (`PlayerInvincibility` eller liknande). Dödsdetekteringen skulle kunna hanteras av en mer generell `Health` -komponent som sedan signalerar till `PlayerRespawner` när spelaren ska dö. Detta skulle följa Single Responsibility Principle (SRP) bättre, men för denna workshop behåller vi det samlat för att tydligt visa hur coroutines används i sekvensen.

3. **Konfigurera & Testa:**
   - Markera din `Player` -prefab.
   - I `PlayerRespawner` -komponenten:
     - Kontrollera att `Start Position` -fältet har en Transform tilldelad (t.ex. ett tomt GameObject placerat där spelaren ska starta).
     - Dra din dödsljudfil (t.ex. `deathscream.mp3`) till fältet `Death Scream Clip` .
     - Dra din respawn-ljudfil (t.ex. `respawn.mp3`) till fältet `Respawn Sound Clip` .



     - Justera värdena för `Respawn Delay` (t.ex. 1.0-2.5s), `Invincibility Duration` (t.ex. 1.5-2.0s), och `Blink Interval` (t.ex. 0.1-0.15s) tills det känns bra i spelet.

- Se till att du har objekt i scenen som ligger på lagret "KillZone" (och att lagret är skapat) eller har taggen "Enemy" eller "Projectile" (och att taggarna är skapade och tilldelade).

Del 5: Enemy Attack Cooldown (Coroutine Exempel)

Mål: Använda en Coroutine för att implementera en tidsbaserad cooldown mellan fiendens attacker och spela ett attackljud när fienden attackerar.

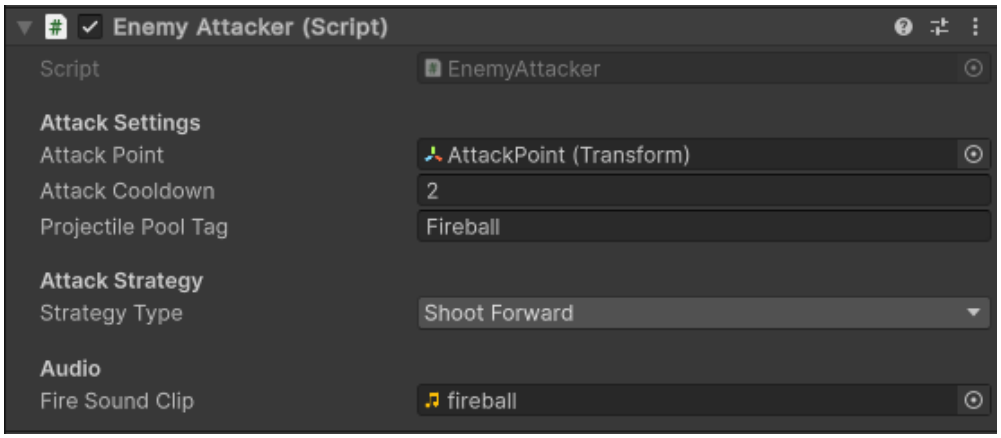
1. **Refaktorera `EnemyAttacker.cs`:** Använder nu `AttackLoopRoutine` (Coroutine) för att hantera attacklogiken och cooldown med `yield return new WaitForSeconds(attackCooldown)`. Spelar även `fireSoundClip` via `AudioManager`.

```
using System.Collections;
using UnityEngine;

public class EnemyAttacker : MonoBehaviour
{
    [Header("Attack Settings")]
    [Tooltip("Transform vars position används som startpunkt för projektilen.")]
    [SerializeField] private Transform attackPoint;
    [Tooltip("Tid i sekunder mellan varje attack.")]
    [SerializeField] private float attackCooldown = 2.0f;
    [Tooltip("Taggen som används i Object Pooler för att hämta rätt typ av projektil.")]
    [SerializeField] private string projectilePoolTag = "Fireball";

    [Header("Attack Strategy")]
    [Tooltip("Väljer hur fienden ska sikta.")]
    [SerializeField] private AttackStrategyType strategyType = AttackStrategyType.ShootForward;

    [Header("Audio")]
    [Tooltip("Ljud som spelas när fienden attackerar.")]
    [SerializeField] private AudioClip fireSoundClip;

    private IAttackStrategy currentStrategy;
    private ObjectPooler pooler;
    private Coroutine attackCoroutine;

    public enum AttackStrategyType { ShootForward, AimAtPlayer }

    void Start()
    {
        pooler = ObjectPooler.Instance;

        if (pooler == null) { Debug.LogError("EnemyAttacker: ObjectPooler.Instance hittades inte!", thi
        if (attackPoint == null) { Debug.LogError("EnemyAttacker: Attack Point är INTE tilldelad!", thi
        if (fireSoundClip == null) Debug.LogWarning("EnemyAttacker: Fire Sound Clip är inte tilldelat.'

        SetAttackStrategy(strategyType);
        if (currentStrategy == null) { Debug.LogError("EnemyAttacker: Kunde inte sätta attackstrategi!'

        // Starta attack-loopen om allt är ok.
        if (enabled)
        {
            attackCoroutine = StartCoroutine(AttackLoopRoutine());
        }
    }

    // Coroutine för attack-loop och cooldown.
    private IEnumerator AttackLoopRoutine()
    {
```

```
    // Slumpmässig startfördröjning för att undvika synkroniserade attacker.
    float initialDelay = Random.Range(0.1f, attackCooldown * 0.5f);

    yield return new WaitForSeconds(initialDelay);

    while (enabled)
    {
        Attack();
        // Pausar coroutinen för cooldown-tiden.
        yield return new WaitForSeconds(attackCooldown);
    }
}

// Utför själva attacken.
private void Attack()
{
    if (currentStrategy == null || pooler == null || attackPoint == null) return;

    if (AudioManager.Instance != null && fireSoundClip != null)
    {
        AudioManager.Instance.PlaySoundOneShot(fireSoundClip, 0.9f); // Spela med 90% volym
    }
    else if (fireSoundClip != null) Debug.LogWarning($"EnemyAttacker on {gameObject.name}: AudioMar

    // Beräkna riktning
    Vector2 direction = currentStrategy.CalculateDirection(transform, attackPoint);

    // Hämta projektil från pool
    GameObject projectileGO = pooler.SpawnFromPool(projectilePoolTag, attackPoint.position, Quaterr

    if (projectileGO != null)
    {
        Projectile projectile = projectileGO.GetComponent<Projectile>();
        if (projectile != null)
        {
            // Se till att din Projectile.Initialize tar emot och använder dessa.
            projectile.Initialize(direction, pooler);
        }
        else
        {
            Debug.LogError($"Spawned object '{projectilePoolTag}' lacks Projectile component!", pro
            pooler.ReturnToPool(projectilePoolTag, projectileGO); // Returnera trasigt objekt
        }
    }
}

public void SetAttackStrategy(AttackStrategyType type)
{
```

```
        switch (type)
        {
            case AttackStrategyType.ShootForward:
            default:
                currentStrategy = new ShootForwardStrategy(); //
                break;
        }
        strategyType = type;

    }

    // Körs när scriptet inaktiveras.
    void OnDisable()
    {
        // Stoppa coroutinen om den körs, för att undvika fel.
        if (attackCoroutine != null)
        {
            StopCoroutine(attackCoroutine);
            attackCoroutine = null;
        }
    }

    void OnDrawGizmosSelected()
    {
        if(attackPoint != null)
        {
            Gizmos.color = Color.red;
            Gizmos.DrawWireSphere(attackPoint.position, 0.3f);
        }
    }
}
```

2. **Konfigurera & Testa:**
   - Markera din fiende-prefab (den som har `EnemyAttacker.cs`).
   - I Inspector för `EnemyAttacker`-komponenten:
     - Dra ett `Transform`-objekt (oftast ett tomt barn-GameObject placerat framför fienden) till fältet `Attack Point`.
     - Kontrollera att `Attack Cooldown` har ett rimligt värde (t.ex. 2 sekunder).
     - Se till att `Projectile Pool Tag` matchar en existerande pool i din `ObjectPooler` (t.ex. "Fireball").
     - Dra en lämplig ljudfil (t.ex. `enemy_shoot.wav`) till fältet `Fire Sound Clip`.



- Kör spelet.
- Verifiera att fienden:
  - Väntar en kort, slumpmässig tid innan första attacken.
  - Sedan attackerar (skjuter en projektil från `Attack Point`) med jämna mellanrum, motsvarande `Attack Cooldown`.
  - Spelar `Fire Sound Clip` varje gång den attackerar.
  - Att projektilen spawnas korrekt och rör sig enligt `ShootForwardStrategy`.

Angående Bifogade Resurser

Precis som i tidigare workshops är bilderna och ljudfilerna som används här AI-genererade. Detta innebär att speciellt animationerna (som bygger på spritesheets) kanske inte är helt perfekta ur ett professionellt perspektiv. Trots detta visar det på den imponerande möjligheten att snabbt och enkelt generera grundläggande spelresurser med moderna verktyg.

Här är källorna för resurserna som användes:

- **Bilder (Spritesheets):** Genererade med AI-bildverktyg (t.ex. via ChatGPT/DALL-E). En exempelprompt kunde vara:
  `"look at this character. make a sprite sheet for a spawning animation. background should be transparent"`.
- **Ljudeffekter (SFX):** Skapade med en kombination av:
  - Freepik Tunes – Erbjuder ett begränsat antal gratis ljud per dag.
  - ElevenLabs SFX – Kan kräva prenumeration för full åtkomst.
- **Bakgrundsmusik (BGM):** Genererad med Suno AI. Exempelprompten som användes var:
  `"retro game, creepy, cave, background, 16-bit, slow, orchestral backing, electronic"`.

Det finns givetvis många fler ställen och sätt att hitta eller skapa spelresurser. Dela gärna med er till gruppen om ni hittar några bra verktyg eller källor! 😊