Workshop 4: PowerUps med Visitor Pattern, Scene Management och Nivåavslutning

Syfte

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

1. **Implementera PowerUps med Visitor Pattern:** Designa och implementera olika typer av PowerUps (t.ex. Speed Boost, Invincibility) med hjälp av **Visitor Pattern**. Detta designmönster möjliggör ett flexibelt och utbyggbart system för att lägga till nya effekter på spelaren (eller andra spelobjekt) utan att modifiera deras kärnklasser.
2. **Hantera Scener med Scene Management:** Förstå och använda Unitys `SceneManagement` API för att:
    - Ladda och ladda om den aktuella spelnivån (t.ex. vid omstart).
    - Potentiellt ladda nya scener (t.ex. en "Du Vann"-skärm eller huvudmeny).
3. **Skapa en Nivåavslutningsmekanism:** Implementera logik för att slutföra en nivå baserat på specifika villkor:
    - Samla ett fördefinierat antal föremål (specifikt **3 "gems"**).
    - Utlösa en händelse när målet är uppnått (t.ex. visa ett meddelande, spela ett ljud, ladda en ny scen).
4. **Integrera och Kombinera Tekniker:** Sammanföra PowerUp-systemet, insamlingsobjekten (gems) och scenhanteringen för att skapa en mer komplett spel-loop med tydliga mål och belöningar.

Förutsättningar

- Unity **6.0** (eller senare) installerat.
- Genomfört **Workshop 1, 2 och 3**, 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.
  - **Grundläggande Animation (Animator Controller), Ljudhantering (AudioManager, AudioSource) och Coroutines (för timing och sekvenser).** (Från Workshop 3)
- Projektet från **Workshop 3** är tillgängligt (med animationer, ljud och respawn-logik).
- **Nya Resurser (Behövs för denna workshop):**
  - **Sprites/Grafik:**
    - Minst en typ av "Gem" (juvel/ädelsten) som kan samlas in.
    - Visuella representationer för minst två olika PowerUps (t.ex. en ikon för Speed Boost, en för Invincibility).
  - **(Valfritt) Ljudfiler (WAV/MP3/OGG):**
    - Ljud för att plocka upp en Gem (`gem_pickup.wav`).
    - Ljud för att plocka upp en PowerUp (`powerup_pickup.wav`).
    - Ljud eller musik för när nivån är avklarad (`level_complete.mp3`).
  - **(Valfritt) UI-element:**
    - En `TextMeshPro` -komponent för att visa antalet insamlade Gems.
    - En enkel panel eller text som visar "Du Vann!" eller liknande.
  - *(Exempelresurser kan tillhandahållas eller hämtas från gratis källor som OpenGameArt, Kenney Assets, freesound.org).*

Kontext

Efter Workshop 3 har vi en spelare med liv, animationer, ljud och grundläggande interaktioner i en spelvärld. Spelet har dock inget tydligt mål

utöver att överleva och saknar mekaniker som temporärt förändrar spelupplevelsen. Denna workshop introducerar **PowerUps**, ett **mål (samla Gems)** och **Scene Management** för att skapa en mer komplett och engagerande spel-loop.

Vi börjar med att implementera **PowerUps**. Istället för att lägga all logik direkt i spelaren använder vi **Visitor Pattern**. Detta gör systemet robust och lätt att utöka – nya PowerUps eller nya objekt som kan påverkas av PowerUps kan läggas till med minimala ändringar i befintlig kod.

Därefter skapar vi **insamlingsobjekt (Gems)** och logiken för att räkna dem. Vi behöver ett sätt att veta när spelaren har samlat de **nödvändiga 3 Gemsen**.

Slutligen knyter vi ihop detta med **Scene Management**. När spelaren samlat alla Gems ska något hända – vi kan ladda om nivån, visa en vinstskärm (som kan vara en egen scen), eller gå vidare till nästa nivå (om en sådan fanns). Vi utforskar hur man använder Unitys inbyggda verktyg för att hantera dessa övergångar.

Målet är att deltagarna ska förstå hur man implementerar vanliga spelmekaniker som PowerUps och nivåmål på ett strukturerat sätt (med Visitor Pattern) och hur man använder Scene Management för att definiera spelets flöde och slutförande.

Del 1: Refactoring PlayerRespawner

Att separera ansvarsområden är en bra refaktorering enligt Single Responsibility Principle (SRP). Vi delar upp PlayerRespawner.cs i två skript: PlayerDeathHandler.cs (som hanterar död och initierar respawn) och PlayerInvincibility.cs (som hanterar odödlighetsperioden). Vi kommer senare att lägga till en PowerUp-klass som kommer att använda PlayerInvincibility.cs för att ge spelaren en tillfällig odödlighetseffekt.

PlayerDeathHandler.cs

Detta skript fokuserar på att upptäcka dödsorsaker, hantera dödssekvensen (inaktivering, effekter, fördröjning) och sedan trigga respawn-positionering

och be PlayerInvincibility att starta.

```
using System.Collections;
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D), typeof(Collider2D),
typeof(Animator))]
[RequireComponent(typeof(IInput), typeof(AgentMovement),
typeof(PlayerInvincibility))] // Se till att Invincibility finns
public class PlayerDeathHandler : 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;

    // Komponentreferenser
    private Rigidbody2D rb;
    private Collider2D playerCollider;
    private Animator animator;
    private IInput playerInput;
    private AgentMovement playerMovement;
    private PlayerInvincibility playerInvincibility; // Referens till
det nya skriptet

    // 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

    void Awake()
    {
        // Hämta komponenter
```

```
        rb = GetComponent<Rigidbody2D>();
        playerCollider = GetComponent<Collider2D>();
        animator = GetComponent<Animator>();
        playerInput = GetComponent<IInput>();
        playerMovement = GetComponent<AgentMovement>();
        playerInvincibility = GetComponent<PlayerInvincibility>(); //
Hämta Invincibility-komponenten

        // Validera
        if (rb == null || playerCollider == null || animator == null ||
playerInput == null || playerMovement == null || playerInvincibility ==
null)
            Debug.LogError("PlayerDeathHandler: En eller flera
nödvändiga komponenter saknas!", this);

        if (startPosition == null)
        {
            Debug.LogError("PlayerDeathHandler: Start Position är INTE
tilldelad!", this);
            enabled = false; // Inaktivera om startposition saknas
            return;
        }

        if (deathScreamClip == null)
Debug.LogWarning("PlayerDeathHandler: Death Scream Clip ej tilldelat.",
this);
        if (respawnSoundClip == null)
Debug.LogWarning("PlayerDeathHandler: Respawn Sound Clip ej tilldelat.",
this);

        // Hämta KillZone Layer
        killZoneLayer = LayerMask.NameToLayer("KillZone");
        if (killZoneLayer == -1)
        {
            Debug.LogError("Physics Layer 'KillZone' hittades inte!",
this);
            // Fundera på om skriptet ska inaktiveras här också
        }
    }

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

    private void OnCollisionEnter2D(Collision2D collision)
```

```
    {
        HandlePotentialDeathSource(collision.gameObject);
    }

    // Kollar om objektet vi kolliderade med ska döda spelaren
    private void HandlePotentialDeathSource(GameObject sourceObject)
    {
        // Gör inget om vi redan är döda, inaktiva, eller odödliga (via
PlayerInvincibility)
        if (!enabled || isDead || (playerInvincibility != null &&
playerInvincibility.IsInvincible)) return;

        bool shouldDie = false;
        if (sourceObject.layer == killZoneLayer) shouldDie = true;
        else if (sourceObject.CompareTag("Enemy")) shouldDie = true;
        else if (sourceObject.CompareTag("Projectile")) shouldDie =
true;
        // Lägg till fler dödsorsaker här vid behov

        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.");

        // 1. Inaktivera spelarkontroll och fysik
        rb.velocity = Vector2.zero; // Stoppa rörelse omedelbart
        rb.simulated = false;       // Stäng av fysiksimulering
        playerCollider.enabled = false; // Stäng av kollideraren
        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
    }

    // Initierar själva respawn-processen
    private void InitiateRespawn()
    {
        if (!this.enabled || startPosition == null) return; // Om
scriptet inaktiverats eller startpos saknas
        Debug.Log("Initiating Player Respawn...");

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

        // 2. Återställ fysik/kolliderare (kontroller/movement
återaktiveras av Invincibility)
        rb.simulated = true;
        playerCollider.enabled = true;
        // Se till att hastigheten är noll vid respawn
        rb.velocity = Vector2.zero;
        rb.angularVelocity = 0f;

        // 3. Starta respawn-effekter (Animation + Ljud)
        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.");

        // 4. Starta odödlighets-perioden via PlayerInvincibility
        // Notera: playerInvincibility ansvarar nu för att återaktivera
input/movement
        playerInvincibility?.StartInvincibility(); // Använd ?. för
säkerhets skull

        // 5. Markera att spelaren inte längre är i dödsprocessen (men
kan vara odödlig)
        // Notera: isDead återställs helt när odödligheten är över (i
PlayerInvincibility),
        // men vi sätter den till false här för att tillåta en ny
dödssekvens *efter* respawn,
        // *om* odödligheten avbryts eller är väldigt kort.
        // En bättre lösning kan vara att PlayerInvincibility signalerar
tillbaka när den är klar.
        // För nu: Sätt isDead till false när respawn initieras.
Kontrollen i HandlePotentialDeathSource
        // kommer fortfarande att blockera om IsInvincible är true.
        isDead = false;
        Debug.Log("Player respawned. Invincibility triggered.");
    }

    // Städning vid inaktivering/förstöring
    void OnDisable()
    {
        // Stoppa coroutines för att undvika problem
         if (activeDeathSequence != null)
         {
             StopCoroutine(activeDeathSequence);
             activeDeathSequence = null;
             Debug.LogWarning("PlayerDeathHandler disabled during death
sequence. Attempting reset.");

             // Försök återställa state om vi avbröt mitt i
dödssekvensen
             // Detta kanske inte är perfekt, men bättre än att lämna
spelaren inaktiv
             if (rb != null) rb.simulated = true;
             if (playerCollider != null) playerCollider.enabled = true;
             if (playerMovement != null) playerMovement.enabled = true;
             MonoBehaviour inputComponent = playerInput as
MonoBehaviour;
```

```
            if (inputComponent != null) inputComponent.enabled = true;
            isDead = false;
        }
    }

    // Public metod för att externt trigga död (t.ex. från en UI-knapp
eller annan logik)
    public void TriggerDeath()
    {
        if (!enabled || isDead || (playerInvincibility != null &&
playerInvincibility.IsInvincible)) return;

        if (activeDeathSequence == null)
        {
            Debug.Log("External death trigger received. Starting Death
Sequence...");
            activeDeathSequence =
StartCoroutine(DeathSequenceRoutine());
        }
    }
}
```

PlayerInvincibility.cs

Detta skript hanterar endast odödlighetsperioden, inklusive blinkning och återaktivering av kontroller när perioden är slut.

```
using System.Collections;
using UnityEngine;

[RequireComponent(typeof(SpriteRenderer))]
[RequireComponent(typeof(IInput), typeof(AgentMovement),
typeof(Animator))] // Behöver dessa för att återaktivera
public class PlayerInvincibility : MonoBehaviour
{
    [Header("Invincibility Settings")]
    [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 SpriteRenderer spriteRenderer;
```

```
    private IInput playerInput;
    private AgentMovement playerMovement;
    private Animator animator;

    // State Variabler
    private Coroutine invincibilityCoroutine = null; // Referens till
pågående odödlighets-coroutine
    public bool IsInvincible => invincibilityCoroutine != null; //
Public property för att kolla state

    void Awake()
    {
        // Hämta komponenter
        spriteRenderer = GetComponent<SpriteRenderer>();
        playerInput = GetComponent<IInput>();
        playerMovement = GetComponent<AgentMovement>();
        animator = GetComponent<Animator>(); // Behövs för att
återställa triggers

        // Validera
        if (spriteRenderer == null || playerInput == null ||
playerMovement == null || animator == null)
            Debug.LogError("PlayerInvincibility: En eller flera
nödvändiga komponenter saknas!", this);
    }

    /// <summary>
    /// Startar odödlighetsperioden med standardduration.
    /// </summary>
    public void StartInvincibility()
    {
        StartInvincibility(invincibilityDuration, true);
    }

    /// <summary>
    /// Startar odödlighetsperioden med en specifik duration.
    /// Stoppar eventuell tidigare pågående odödlighet.
    /// </summary>
    /// <param name="duration">Hur länge odödligheten ska vara.</param>
    public void StartInvincibility(float duration, bool
playerControlActive)
    {
        if (duration <= 0) return; // Gör inget om duration är noll
eller negativ

        StopInvincibility(); // Stoppa ev. tidigare odödlighet
```

```
        Debug.Log($"Starting Invincibility for {duration} seconds.");
        invincibilityCoroutine =
StartCoroutine(InvincibilityRoutine(duration, playerControlActive));
    }

    // Coroutine för odödlighet och blinkning
    private IEnumerator InvincibilityRoutine(float duration, bool
playerControlActive)
    {
        float endTime = Time.time + duration;
        bool visible = false; // Starta osynlig för första blinkningen

        // Se till att input/movement är avstängt i början (ifall de
inte redan är det)
        SetPlayerControlActive(playerControlActive);

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

        // Slut på odödligheten
        // Använd en finally-block för att garantera att detta körs,
även om coroutinen avbryts
        try
        {
            // Vänta sista blink-intervallet om det behövs för att
säkerställa att tiden passerat helt
            // Detta är en extra säkerhetsåtgärd, kan tas bort om
blinkIntervallet är litet.
            if (Time.time < endTime)
            {
                yield return new WaitForSeconds(endTime - Time.time);
            }
        }
        finally
        {
            Debug.Log("Invincibility Ended.");
            if (spriteRenderer != null) spriteRenderer.enabled = true;
// Se till att alltid vara synlig efteråt
```

```
            // Återaktivera kontroller och rörelse
            SetPlayerControlActive(true);

            // Återställ animator triggers som kan ha fastnat
            if (animator != null)
            {
                animator.ResetTrigger("Die");
                animator.ResetTrigger("Respawn");
                // Lägg till fler trigger-återställningar här om det
behövs
            }

            invincibilityCoroutine = null; // Markera att coroutinen är
klar
            Debug.Log("Player is vulnerable again and controls are
enabled.");
        }
    }

    /// <summary>
    /// Stoppar den pågående odödlighets-coroutinen i förtid och
återställer spelaren.
    /// </summary>
    public void StopInvincibility()
    {
        if (invincibilityCoroutine != null)
        {
            Debug.Log("Stopping active invincibility coroutine.");
            StopCoroutine(invincibilityCoroutine);
            invincibilityCoroutine = null; // Viktigt att nollställa
direkt

            // Omedelbar återställning vid avbrott
            if (spriteRenderer != null) spriteRenderer.enabled = true;
// Säkerställ synlighet
            SetPlayerControlActive(true); // Återaktivera kontroller
direkt
                                          // Överväg om animator
triggers också ska återställas här
            if (animator != null)
            {
                animator.ResetTrigger("Die");
                animator.ResetTrigger("Respawn");
            }
            Debug.Log("Invincibility stopped prematurely. Player
vulnerable, controls enabled.");
```

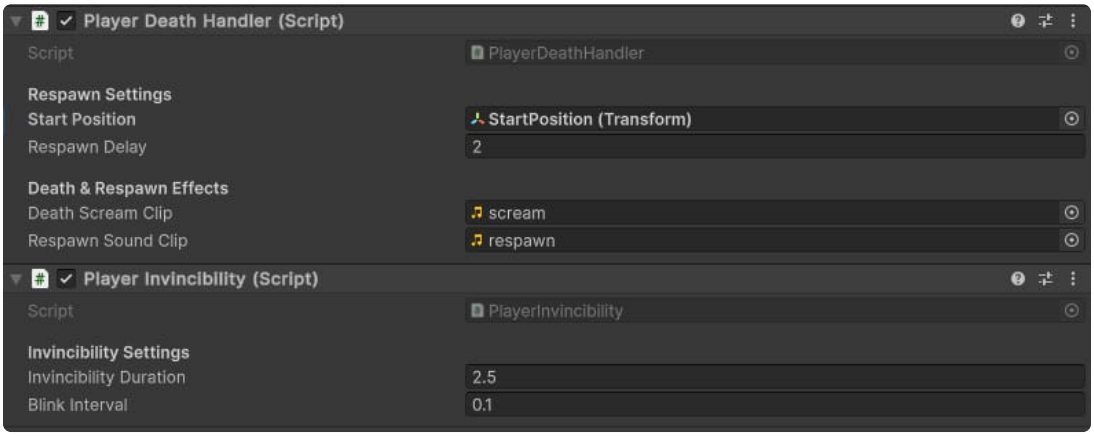
```
        }
    }

    // Hjälpmetod för att aktivera/inaktivera spelarkontroller
    private void SetPlayerControlActive(bool isActive)
    {
        if (playerMovement != null) playerMovement.enabled = isActive;
        MonoBehaviour inputComponent = playerInput as MonoBehaviour;
        if (inputComponent != null) inputComponent.enabled = isActive;
    }

    // Städning vid inaktivering/förstöring
    void OnDisable()
    {
        // Om skriptet inaktiveras, stoppa odödligheten och försök
återställa
        StopInvincibility();
    }
}
```

Steg för Implementering

1. **Ta bort Gammal Kod:** Om du har ett `PlayerRespawner.cs` eller liknande skript från Workshop 3, ta bort det från ditt Player GameObject.
2. **Skapa Nya Skript:** Skapa två nya C# skript i ditt Unity-projekt: `PlayerDeathHandler.cs` och `PlayerInvincibility.cs`.
3. **Kopiera Koden:** Kopiera koden som tillhandahållits tidigare i dokumentet och klistra in den i respektive fil (`PlayerDeathHandler.cs` och `PlayerInvincibility.cs`).
4. **Lägg Till Komponenter:** Dra båda de nya skripten (`PlayerDeathHandler` och `PlayerInvincibility`) till ditt Player GameObject i hierarkin. `RequireComponent` -attributen kommer att varna om nödvändiga komponenter (som `Rigidbody2D` , `Animator` , etc.) saknas. Lägg till eventuella saknade komponenter via `Add Component` -knappen i Inspektorn.
5. **Konfigurera i Inspektorn:**
    - **`PlayerDeathHandler` :**
        - Dra ditt `StartPosition` GameObject (en tom transform som markerar var spelaren ska återuppstå) från Hierarkin till

fältet `Start Position` i Inspektorn.
        - Dra dina ljudfiler för dödsskrik och respawn-ljud till fälten `Death Scream Clip` och `Respawn Sound Clip` (om du har sådana).
        - Justera `Respawn Delay` (tiden mellan död och respawn) till önskat värde (t.ex. `1.0` sekund).
    - **`PlayerInvincibility`:**
        - Justera `Default Invincibility Duration` (tiden spelaren är odödlig efter respawn) till önskat värde (t.ex. `1.5` sekunder).
        - Justera `Blink Interval` (hur snabbt spelaren blinkar) till önskat värde (t.ex. `0.1` sekunder).
6. **Kontrollera Layers och Tags:**
    - Gå till `Edit -> Project Settings -> Tags and Layers`. Under `Layers`, se till att du har ett lager som heter exakt `"KillZone"` (skiftlägeskänsligt). Lägg till det om det saknas.
    - Se till att de GameObjects i din scen som ska döda spelaren (t.ex. spikar, fallgropar, lava) har sitt **Layer** satt till `"KillZone"` i Inspektorn.
    - Se till att dina fiende-prefabs/GameObjects har **Tag** satt till `"Enemy"`.
    - Se till att dina projektil-prefabs/GameObjects har **Tag** satt till `"Projectile"`.
    - (Om du använder andra tags eller lager för död, anpassa `HandlePotentialDeathSource`-metoden i `PlayerDeathHandler.cs`.)
7. **Verifiera Referenser:** Dubbelkolla i Inspektorn för ditt Player GameObject att alla nödvändiga komponenter (`Rigidbody2D`, `Collider2D`, `Animator`, `SpriteRenderer`, din `IInput`-implementation, `AgentMovement`, `PlayerDeathHandler`, `PlayerInvincibility`) finns och är aktiverade (checkboxen ikryssad).
8. **Testa:** Kör spelet (`Play`-knappen) och testa noggrant att:
    - Spelaren dör korrekt när den kolliderar med ett objekt i `"KillZone"`-lagret eller ett objekt med taggen `"Enemy"` eller `"Projectile"`.
    - Korrekt dödsanimation och ljud spelas.
    - Det finns en märkbar fördröjning (`Respawn Delay`) innan respawn inträffar.

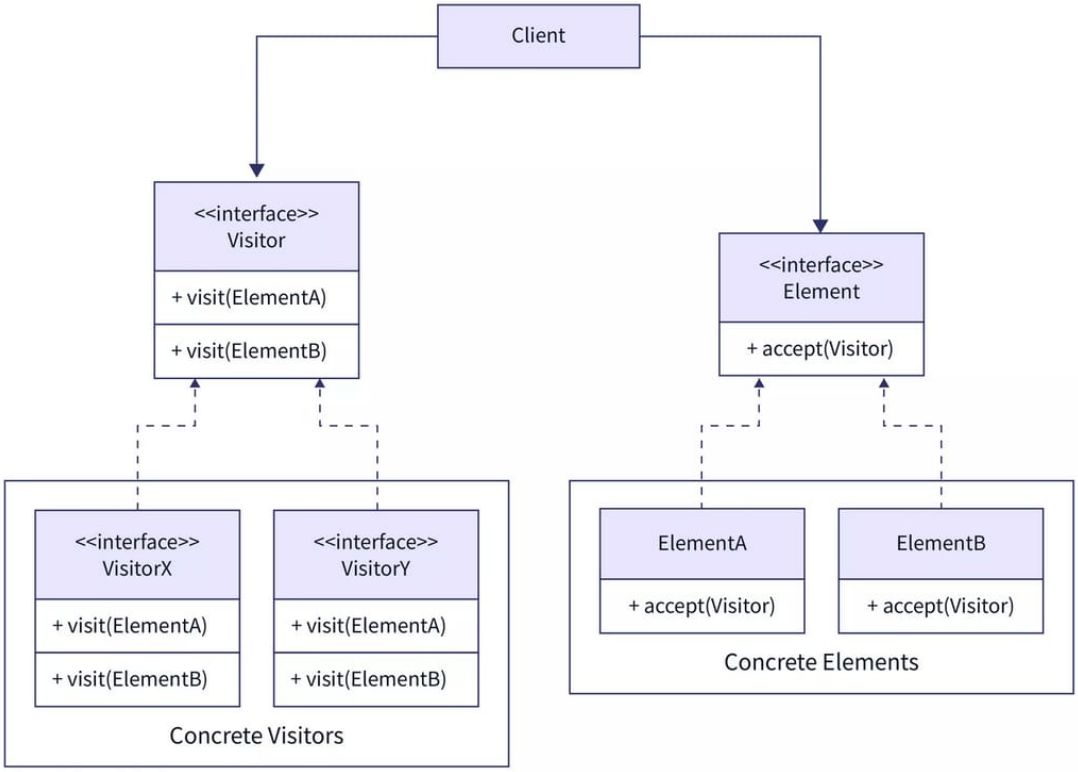
- Spelaren återuppstår vid den `StartPosition` du angett.
- Korrekt respawn-animation och ljud spelas.
- Spelaren blinkar och är immun mot att dö igen under den tid som anges i `Default Invincibility Duration`.
- Spelaren slutar blinka, blir solid och kan dö igen efter att odödlighetsperioden är slut.
- Spelarens kontroller (`IInput` , `AgentMovement`) fungerar som de ska efter att odödligheten upphört.



Nu har du framgångsrikt separerat logiken för död och odödlighet i två dedikerade komponenter. `PlayerDeathHandler` hanterar detektering av död och initiering av respawn, medan `PlayerInvincibility` sköter den temporära odödligheten och återaktiveringen av spelaren. Detta skapar en mer modulär och underhållbar kodbas, vilket är en utmärkt grund för att bygga vidare med PowerUps i nästa del av workshopen.

Del 2: Implementera PowerUps med Visitor Pattern

Nu när vi har en stabil grund för spelarens död och odödlighet, ska vi introducera **PowerUps**. Målet är att skapa ett system där vi enkelt kan lägga till nya typer av PowerUps (t.ex. snabbhet, hopphöjd, vapenuppgradering) utan att behöva modifiera spelarens kärnklasser varje gång. För detta använder vi **Visitor Pattern**.



Varför Visitor Pattern?

- **Flexibilitet:** Nya PowerUp-effekter kan läggas till genom att skapa nya "Visitor"-klasser, utan att ändra på de klasser som *tar emot* effekten (t.ex. spelaren). Detta följer **Open/Closed Principle (OCP)** - öppen för utökning, stängd för modifiering.
- **Separation of Concerns:** Logiken för en specifik PowerUp-effekt samlas i dess Visitor-klass, istället för att spridas ut i spelarens klass.
- **Hanterar Olika Typer:** Mönstret gör det enklare att applicera effekter på olika typer av spelobjekt (t.ex. både spelare och fiender) om så önskas i framtiden.

Här har du lite videomaterial och wiki-sidan för Visitor pattern som kan vara bra att studera för att förstå mönstret bättre.

- Visitor Pattern Video
- Visitor Pattern Wiki

Översikt av Implementeringen:

1. **Interfaces:** Vi definierar `IVisitable` (för objekt som kan ta emot effekter, som spelaren) och `IPowerUpVisitor` (för de specifika PowerUp-effekterna).

2. **Spelarens Anpassning:** Vi skapar (eller anpassar) en `PlayerStats`-komponent på spelaren som implementerar `IVisitable` och innehåller de värden som PowerUps ska påverka (t.ex. rörelsehastighet) samt referenser till andra relevanta komponenter (som `PlayerInvincibility`).
3. **Concrete Visitors:** Vi skapar specifika klasser för varje PowerUp-effekt (t.ex. `SpeedBoostVisitor`, `InvincibilityVisitor`) som implementerar `IPowerUpVisitor`.
4. **PowerUp-Objekt:** Vi skapar en generell `PowerUpEffect`-komponent för de objekt som spelaren plockar upp i världen. Denna komponent kommer att skapa och applicera rätt `Visitor` när den plockas upp.

Steg 1: Definiera Interfaces

Skapa två nya C# filer/skript: `IVisitable.cs` och `IPowerUpVisitor.cs`.

IVisitable.cs:

Definierar att ett objekt kan "acceptera" en besökare (en PowerUp-effekt).

```
public interface IVisitable
{
    void Accept(IPowerUpVisitor visitor);
}
```

IPowerUpVisitor.cs:

Definierar "besökaren". Varje metod representerar en typ av objekt som besökaren kan påverka. Just nu fokuserar vi bara på PlayerStats.

```
public interface IPowerUpVisitor
{
    void Visit(PlayerStats playerStats);
}
```

Steg 2: Anpassa Spelaren (PlayerStats)**

Vi behöver en komponent på spelaren som håller reda på dess statusvärden och som kan acceptera PowerUps. Skapa ett nytt C# skript PlayerStats.cs och

lägg till det på ditt Player GameObject. Om du redan har en liknande klass, anpassa den.

PlayerStats.cs:

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(AgentMovement), typeof(PlayerInvincibility))]
public class PlayerStats : MonoBehaviour, IVisitable // Implementera
IVisitable
{
    [Header("Movement Stats")]
    [Tooltip("Basrörelsehastighet för spelaren.")]
    [SerializeField] private float baseMoveSpeed = 5f;

    private AgentMovement agentMovement;
    private PlayerInvincibility playerInvincibility;

    // Temporära effekt-variabler
    private Coroutine activeSpeedBoostCoroutine;
    private float currentMoveSpeed;

    public float CurrentMoveSpeed => currentMoveSpeed;

    void Awake()
    {
        agentMovement = GetComponent<AgentMovement>();
        playerInvincibility = GetComponent<PlayerInvincibility>();

        if (agentMovement == null || playerInvincibility == null)
        {
            Debug.LogError($"{nameof(PlayerStats)}: Saknar nödvändig
komponent (AgentMovement eller PlayerInvincibility)!", this);
            enabled = false;
            return;
        }

        currentMoveSpeed = baseMoveSpeed;
        ApplySpeedToMovement(currentMoveSpeed);
    }

    /// <summary>
    /// Implementering av IVisitable. Accepterar en PowerUp-besökare.
```

```
    /// Detta är "ingången" för Visitor-mönstret.
    /// </summary>
    public void Accept(IPowerUpVisitor visitor)
    {
        // Dubbel dispatch: Anropar rätt Visit-metod på besökaren,
        // och skickar med sig själv (denna PlayerStats-instans) som
argument.
        visitor.Visit(this);
        Debug.Log($"{gameObject.name} accepted visitor:
{visitor.GetType().Name}");
    }

    /// <summary>
    /// Metod som anropas av SpeedBoostVisitor för att applicera
hastighetsökning.
    /// </summary>
    public void ApplySpeedBoost(float multiplier, float duration)
    {
        if (duration <= 0 || multiplier <= 0) return;

        // Stoppa eventuell tidigare hastighetsboost för att undvika
konflikter
        if (activeSpeedBoostCoroutine != null)
        {
            StopCoroutine(activeSpeedBoostCoroutine);
            // Återställ till bashastighet innan ny boost appliceras?
Eller stacka?
            // För enkelhetens skull, återställ och starta om.
            currentMoveSpeed = baseMoveSpeed;
        }

        activeSpeedBoostCoroutine =
StartCoroutine(SpeedBoostRoutine(multiplier, duration));
    }

    /// <summary>
    /// Coroutine som hanterar den temporära hastighetsökningen.
    /// </summary>
    private IEnumerator SpeedBoostRoutine(float multiplier, float
duration)
    {
        currentMoveSpeed = baseMoveSpeed * multiplier;
        ApplySpeedToMovement(currentMoveSpeed);
        Debug.Log($"Speed Boost Applied! Speed: {currentMoveSpeed} for
{duration}s");
```

```
        yield return new WaitForSeconds(duration);

        Debug.Log("Speed Boost Ended.");
        currentMoveSpeed = baseMoveSpeed; // Återställ till bashastighet
        ApplySpeedToMovement(currentMoveSpeed);
        activeSpeedBoostCoroutine = null; // Markera att coroutine är
klar
    }

    /// <summary>
    /// Metod som anropas av InvincibilityVisitor.
    /// </summary>
    public void ApplyInvincibility(float duration)
    {
        if (playerInvincibility != null)
        {
            playerInvincibility.StartInvincibility(duration, true);
            Debug.Log($"Invincibility Applied via Visitor for {duration}
s");
        }
        else
        {
            Debug.LogWarning($"{nameof(PlayerStats)}: Försökte applicera
odödlighet, men PlayerInvincibility-komponent saknas!", this);
        }
    }

    /// <summary>
    /// Hjälpmetod för att uppdatera AgentMovement-komponenten med
aktuell hastighet.
    /// </summary>
    private void ApplySpeedToMovement(float speed)
    {
        if (agentMovement != null)
        {
            agentMovement.moveSpeed = speed;
        }
    }

    // Säkerställ att hastigheten återställs om objektet förstörs eller
inaktiveras
    void OnDisable()
    {
        if (activeSpeedBoostCoroutine != null)
        {
            StopCoroutine(activeSpeedBoostCoroutine);
```

```
            activeSpeedBoostCoroutine = null;
            currentMoveSpeed = baseMoveSpeed;
            Debug.Log("PlayerStats disabled, Speed Boost coroutine
stopped.");
        }
    }
}
```

Viktigt: Anpassa ApplySpeedToMovement-metoden så att den fungerar med din specifika AgentMovement-komponent (t.ex. om den heter SetSpeed, UpdateMaxSpeed eller liknande). Se också till att AgentMovement använder det värde som PlayerStats sätter.

Steg 3: Skapa Concrete Visitors (PowerUp-Effekterna)

Skapa nya C# skript för varje PowerUp-typ.

SpeedBoostVisitor.cs

Denna klass kommer att applicera en hastighetsökning på spelaren. Den tar emot en hastighetsmultiplikator och en varaktighet.

```
using UnityEngine;

/// <summary>
/// Visitor som applicerar en temporär hastighetsökning på PlayerStats.
/// </summary>
public class SpeedBoostVisitor : IPowerUpVisitor
{
    private readonly float speedMultiplier;
    private readonly float duration;

    public SpeedBoostVisitor(float multiplier, float dur)
    {
        speedMultiplier = multiplier;
        duration = dur;
    }

    /// <summary>
    /// Besöker PlayerStats och anropar dess ApplySpeedBoost-metod.
    /// </summary>
    public void Visit(PlayerStats playerStats)
    {
```

```
        playerStats.ApplySpeedBoost(speedMultiplier, duration);
    }

    // Tom implementation för andra Visit-metoder (om de fanns i
interfacet)
    // public void Visit(EnemyStats enemyStats) { /* Gör inget för
fiender */ }
}
```

InvincibilityVisitor.cs

Denna klass kommer att applicera en temporär odödlighet på spelaren. Den tar emot en varaktighet.

```
using UnityEngine;

/// <summary>
/// Visitor som applicerar temporär odödlighet via PlayerStats (som
anropar PlayerInvincibility).
/// </summary>
public class InvincibilityVisitor : IPowerUpVisitor
{
    private readonly float duration;

    public InvincibilityVisitor(float dur)
    {
        duration = dur;
    }

    /// <summary>
    /// Besöker PlayerStats och anropar dess ApplyInvincibility-metod.
    /// </summary>
    public void Visit(PlayerStats playerStats)
    {
        playerStats.ApplyInvincibility(duration);
    }
}
```

Steg 4: Skapa PowerUp-Objektet för Upplockning

Skapa ett abstrakt basskript PowerUpEffect.cs och sedan konkreta skript för varje PowerUp-typ som kan plockas upp.

PowerUpEffect.cs (Abstrakt Basklass):

```
using UnityEngine;

[RequireComponent(typeof(Collider2D))]
public abstract class PowerUpEffect : MonoBehaviour
{
    [Header("PowerUp Settings")]
    [Tooltip("Hur länge effekten ska vara (om den är tidsbegränsad).")]
    [SerializeField] protected float duration = 5f;
    [Tooltip("Ljud som spelas när PowerUp plockas upp.")]
    [SerializeField] protected AudioClip pickupSound;

    // Abstrakt metod som subklasser måste implementera för att skapa
rätt Visitor
    protected abstract IPowerUpVisitor CreateVisitor();

    protected virtual void Awake()
    {
        // Säkerställ att Collidern är en Trigger
        Collider2D col = GetComponent<Collider2D>();
        if (col != null && !col.isTrigger)
        {
            Debug.LogWarning($"Collider på {gameObject.name} var inte
satt till Trigger. Sätter den nu.", this);
            col.isTrigger = true;
        }
    }

    protected virtual void OnTriggerEnter2D(Collider2D other)
    {
        // Kolla om det är spelaren som plockar upp (via Tag eller
Layer)
        if (other.CompareTag("Player")) // Antag att spelaren har taggen
"Player"
        {
            // Försök hämta IVisitable-komponenten från spelaren (vår
PlayerStats)
            IVisitable visitable = other.GetComponent<IVisitable>();

            if (visitable != null)
            {
                // Skapa den specifika Visitor-instansen
                IPowerUpVisitor visitor = CreateVisitor();
```

```
                // Applicera effekten genom att anropa Accept på
spelaren
                visitable.Accept(visitor);

                // Spela ljud (om AudioManager finns och ljudklipp är
satt)
                if (pickupSound != null && AudioManager.Instance !=
null)
                {
                    AudioManager.Instance.PlaySoundOneShot(pickupSound);
                }
                else if (pickupSound != null)
                {
                    Debug.LogWarning("AudioManager not found for pickup
sound.");
                }

                // Förstör PowerUp-objektet efter upplockning
                Destroy(gameObject);
            }
            else
            {
                Debug.LogWarning($"Spelaren ({other.name}) saknar en
IVisitable-komponent (t.ex. PlayerStats). Kan inte applicera PowerUp.",
other);
            }
        }
    }
}
```

SpeedBoostPowerUpItem.cs (Konkret PowerUp-Objekt):

```
using UnityEngine;

public class SpeedBoostPowerUpItem : PowerUpEffect
{
    [Header("Speed Boost Specifics")]
    [Tooltip("Multiplikator för hastigheten (t.ex. 1.5 = 50% ökning).")]
    [SerializeField] private float speedMultiplier = 1.5f;

    /// <summary>
    /// Skapar och returnerar en SpeedBoostVisitor med värden från
inspektorn.
    /// </summary>
```

```
    protected override IPowerUpVisitor CreateVisitor()
    {
        // Använder 'duration' från basklassen PowerUpEffect
        return new SpeedBoostVisitor(speedMultiplier, duration);
    }
}
```

InvincibilityPowerUpItem.cs (Konkret PowerUp-Objekt):

```
// InvincibilityPowerUpItem.cs
using UnityEngine;

public class InvincibilityPowerUpItem : PowerUpEffect
{
    // Inga specifika inställningar förutom duration från basklassen
just nu.
    // Man kan lägga till t.ex. visuella effekter här om man vill.

    /// <summary>
    /// Skapar och returnerar en InvincibilityVisitor med värden från
inspektorn.
    /// </summary>
    protected override IPowerUpVisitor CreateVisitor()
    {
        // Använder 'duration' från basklassen PowerUpEffect
        return new InvincibilityVisitor(duration);
    }
}
```

Steg 5: Skapa Prefabs och Placera i Scenen

1. **Skapa Tomma GameObjects:** Skapa två nya tomma GameObjects i din scen.
2. **Lägg Till Komponenter:**
    - I mappen /assets/ finns det flera sprites som kan användas för att representera PowerUps. Du kan använda dessa eller skapa egna. Importera de nya sprites i Unity och sätt dem till "Sprite (2D and UI)" i inspektorn.
    - På orben, lägg till SpriteRenderer, Collider2D (t.ex. CircleCollider2D, kom ihåg att sätta Is Trigger till true), och skriptet SpeedBoostPowerUpItem.

    - På stjärnan, lägg till SpriteRenderer, Collider2D (Is Trigger = true), och skriptet InvincibilityPowerUpItem.
    - Diamanten ska vi implementera senare, så vi hoppar över den för nu.

1. **Konfigurera i Inspektorn:**
    - På SpeedBoostPowerUpItem-objektet, justera Duration och Speed Multiplier. Tilldela eventuellt ett Pickup Sound. I /assets/ finns det en ljudfil som heter "item pick up" som kan användas.

- På InvincibilityPowerUpItem-objektet, justera Duration. Tilldela samma Pickup Sound.

2. Skapa Prefabs: Dra båda dessa konfigurerade GameObjects från hierarkin till din Projekt-panel (t.ex. i en Prefabs/PowerUps-mapp) för att skapa Prefabs av dem. Du kan nu ta bort dem från hierarkin om du vill, eller placera ut kopior av prefabsen i din level.
3. **Spelar-Tag:** Säkerställ att ditt Player GameObject har taggen satt till "Player" i Inspektorn.
4. **PlayerStats:** Gå till ditt Player GameObject och se till att PlayerStats-komponenten finns där. Konfigurera Base Move Speed i Inspektorn på PlayerStats. Kontrollera att AgentMovement och PlayerInvincibility också finns på spelaren.

Steg 6: Testning

1. Placera ut instanser av dina PowerUp-prefabs i din spelscen.
2. Kör spelet.
3. Testa Speed Boost: Gå över Speed Boost-objektet.
    - Verifiera att objektet försvinner.
    - Verifiera att pickup-ljudet spelas (om konfigurerat).
    - Verifiera att spelaren rör sig märkbart snabbare. Använd Debug.Log i PlayerStats eller titta på värden i Inspektorn (om du gör currentMoveSpeed synlig) för att bekräfta.
    - Verifiera att spelarens hastighet återgår till det normala efter den angivna Duration.
4. **Testa Invincibility:** Gå över Invincibility-objektet.
    - Verifiera att objektet försvinner.
    - Verifiera att pickup-ljudet spelas.
    - Verifiera att spelaren börjar blinka (via PlayerInvincibility-skriptet).
    - Försök att få spelaren att kollidera med en fiende eller KillZone medan den blinkar. Spelaren ska inte dö. Notera att om vi faller av banan så dör inte spelaren om vi har invincibility. På grund av detta kan det vara värt att implementera en hård "fall av banan"-effekt på spelaren. Vilket hade dödat spelaren om de faller av banan under invincibility-tiden.
    - Verifiera att spelaren slutar blinka och blir sårbar igen efter den angivna Duration.

Du har nu implementerat ett flexibelt PowerUp-system med Visitor Pattern! För att lägga till en ny PowerUp (t.ex. "Double Jump") behöver du bara:

1. Skapa en ny DoubleJumpVisitor-klass.
2. Lägga till en ApplyDoubleJump(float duration)-metod i PlayerStats och uppdatera IPowerUpVisitor och PlayerStats.Accept om det behövs (eller om ApplyDoubleJump anropas direkt från Visit).
3. Skapa ett DoubleJumpPowerUpItem-skript som ärver från PowerUpEffect och implementerar CreateVisitor för att returnera en DoubleJumpVisitor.
4. Skapa en prefab för det nya PowerUp-objektet.

Inga ändringar behövs i PlayerDeathHandler, befintliga PowerUp-items, eller

kärnlogiken i PlayerStats (förutom att lägga till den nya effekthanteringsmetoden).

Del 3: Implementera hård "fall av banan"-effekt

Nu ska vi implementera en hård "fall av banan"-effekt för spelaren. Detta innebär att om spelaren faller utanför banan (t.ex. i en dödszon eller avgränsad yta), så ska spelaren dö direkt, oavsett om de är odödliga eller inte. Detta kan vara användbart för att förhindra att spelaren kan utnyttja odödligheten genom att hoppa ner i en dödszon eller liknande.

Skapa en mekanism på egen hand så att spelaren dör ifall spelaren faller av banan. Skapa en ny metod i PlayerDeathHandler som hanterar detta, och se till att den anropas när spelaren faller utanför spelområdet. I mitt fall skapade jag följande method på PlayerDeathHandler-scriptet för att säkerställa att spelaren ska dö ifall spelaren faller av banan.

PlayerDeathHandler.cs (Lägg till följande kod):

```
    public void Update()
    {
        CheckIfFallenOffMap();
    }

    private void CheckIfFallenOffMap()
    {
        // Döda spelaren om den faller under en viss y-nivå
        if (transform.position.y < -6f && !isDead && activeDeathSequence
== null)
        {
            Debug.Log("Player fell below threshold. Triggering death.");
            activeDeathSequence =
StartCoroutine(DeathSequenceRoutine());
        }
    }
```

Del 4: Insamlingsobjekt (Gems) och Nivåavslutning

Nu ska vi lägga till ett mål i spelet: samla 3 Gems. När spelaren har samlat alla Gems ska nivån startas om (eller så kan man ladda en "Du Vann"-scen om man vill).

Steg 1: Skapa Gem-Objektet och Skriptet

1. **Skapa GemCollectible.cs Skript:** Skapa ett nytt C# skript med namnet GemCollectible.cs.
    - Namnge det GameObjektet till "GemCollectible" för att hålla det enkelt och tydligt. Tidigare kallad Gem.

GemCollectible.cs:

```
using UnityEngine;

[RequireComponent(typeof(Collider2D))]
public class GemCollectible : MonoBehaviour
{
    [Header("Effects")]
    [Tooltip("Ljud som spelas när en Gem plockas upp.")]
    [SerializeField] private AudioClip pickupSound;

    private void Awake()
    {
        Collider2D col = GetComponent<Collider2D>();
        if (col != null && !col.isTrigger)
        {
            Debug.LogWarning($"Collider på {gameObject.name} var inte
satt till Trigger. Sätter den nu.", this);
            col.isTrigger = true;
        }
    }

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

```
            Debug.Log("Gem collected by Player!");

            if (LevelManager.Instance != null)
            {
                LevelManager.Instance.CollectGem();
            }
            else
            {
                Debug.LogError("LevelManager.Instance not found! Kan
inte registrera Gem-insamling.", this);
            }

            if (pickupSound != null && AudioManager.Instance != null)
            {
                AudioManager.Instance.PlaySoundOneShot(pickupSound);
            }
            else if (pickupSound != null)
            {
                Debug.LogWarning("AudioManager not found for Gem pickup
sound.");
            }

            Destroy(gameObject);
        }
    }
}
```

1. Konfigurera Gem-Objektet:

- Lägg till GemCollectible.cs-skriptet på ditt GemCollectible GameObject.
- Om du inte redan gjort det. Gör en Prefab av GemCollectible-objektet genom att dra det från Hierarkin till din Projekt-panel (t.ex. i en Prefabs/Collectibles-mapp). Du kan nu ta bort originalet från scenen.

Steg 2: Skapa LevelManager

Denna manager kommer att hålla reda på Gems och hantera nivåavslutningen.

1. Skapa LevelManager.cs Skript:
    - Skapa ett nytt C# skript med namnet LevelManager.cs.
    - Kopiera och klistra in följande kod:

LevelManager.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement; // Viktigt för scenhantering!
using System.Collections;

public class LevelManager : MonoBehaviour
{
    private static LevelManager _instance;
    public static LevelManager Instance
    {
        get
        {
            if (_instance == null)
            {
                Debug.LogError("LevelManager is NULL. Se till att det
finns ett LevelManager-objekt i scenen.");
            }
            return _instance;
        }
    }

    [Header("Level Goal")]
    [Tooltip("Antal Gems som krävs för att klara nivån.")]
    [SerializeField] private int targetGemCount = 3;

    [Header("Level Completion")]
    [Tooltip("Ljud/Musik som spelas när nivån är klar.")]
    [SerializeField] private AudioClip levelCompleteSound;
    [Tooltip("Fördröjning i sekunder innan nivån laddas om efter
avslut.")]
    [SerializeField] private float levelReloadDelay = 2.0f;

    // --- State ---
    private int currentGemCount = 0;
    private bool levelCompleted = false; // Förhindrar att slutförandet
triggas flera gånger

    // --- Events (för UI-uppdatering) ---
    public event System.Action<int, int> OnGemCountChanged; // Skickar
(current, target)

    private void Awake()
    {
        if (_instance != null && _instance != this)
```

```
        {
            Debug.LogWarning("Flera instanser av LevelManager hittades.
Förstör denna nya instans.");
            Destroy(gameObject);
        }
        else
        {
            _instance = this;
            DontDestroyOnLoad(gameObject);
        }

        // Validering
        if (targetGemCount <= 0)
        {
            Debug.LogWarning("Target Gem Count är satt till 0 eller
mindre. Nivån kan aldrig slutföras.", this);
            targetGemCount = 1; // Sätt ett minimumvärde
        }
    }

    private void Start()
    {
        // Nollställ räknare vid start och meddela UI
        currentGemCount = 0;
        levelCompleted = false;
        // Meddela UI initialt
        OnGemCountChanged?.Invoke(currentGemCount, targetGemCount);
        Debug.Log($"Level started. Collect {targetGemCount} gems.");
    }

    public void CollectGem()
    {
        if (levelCompleted) return;

        currentGemCount++;
        Debug.Log($"Gem collected! Total: {currentGemCount} /
{targetGemCount}");

        UIManager.Instance.UpdateGemCounter(currentGemCount,
targetGemCount);

        CheckForLevelCompletion();
    }

    private void CheckForLevelCompletion()
    {
```

```
        if (currentGemCount >= targetGemCount && !levelCompleted)
        {
            levelCompleted = true;
            Debug.Log("Level Complete!");
            StartCoroutine(LevelCompleteSequence());
        }
    }

    private IEnumerator LevelCompleteSequence()
    {
        if (levelCompleteSound != null && AudioManager.Instance != null)
        {
            AudioManager.Instance.PlaySoundOneShot(levelCompleteSound);
        }
        else if (levelCompleteSound != null)
        {
            Debug.LogWarning("AudioManager not found for level complete
sound.");
        }

        UIManager.Instance?.ShowWinMessage();

        yield return new WaitForSeconds(levelReloadDelay);

        Debug.Log("Reloading current scene...");
        Scene currentScene = SceneManager.GetActiveScene();
        SceneManager.LoadScene(currentScene.buildIndex);
    }

    // Lägg till en getter för att UI ska kunna hämta target count
    public int GetTargetGemCount() => targetGemCount;
}
```

1. Skapa LevelManager GameObject:
    - Skapa ett tomt GameObject i din scen (Create Empty).
    - Byt namn på det till LevelManager.
    - Dra LevelManager.cs-skriptet till detta GameObject.
    - Konfigurera Target Gem Count i Inspektorn till 3.
    - (Valfritt) Dra en ljudfil till Level Complete Sound.
    - Justera Level Reload Delay om du vill ha en annan fördröjning.

Steg 3: Uppdatera UI för att Visa Gem-Räknare

1. Skapa Text-Element:
    - På din Canvas, skapa ett nytt UI -> Text - TextMeshPro-objekt.
    - Placera det där du vill ha Gem-räknaren (t.ex. i övre högra hörnet).
    - Byt namn på objektet till GemCounterText.
2. Uppdatera UIManager.cs:
    - Öppna ditt UIManager.cs skript (eller skapa ett om det inte finns).
    - Uppdatera till följande:

UIManager.cs

```
using UnityEngine;
using TMPro;
using System;

public class UIManager : MonoBehaviour
{
    private static UIManager _instance;
    public static UIManager Instance
    {
        get
        {
            if (_instance == null)
```

```
            {
                _instance = FindFirstObjectByType<UIManager>();

                if (_instance == null)
                {
                    Debug.LogError("UIManager instance not found in the
scene! Make sure a GameObject with the UIManager script exists.");
                }
            }
            return _instance;
        }
    }

    [Header("UI Elements")]
    [Tooltip("Textfält för att visa antal dödsfall.")]
    [SerializeField] private TextMeshProUGUI deathsText;
    [Tooltip("Textfält för att visa Gem-räknaren.")]
    [SerializeField] private TextMeshProUGUI gemCounterText;
    [Tooltip("Panel eller text som visas när man vinner.")]
    [SerializeField] private GameObject winMessagePanel;

    private int deathCount = 0;


    void Awake()
    {
        if (_instance == null)
        {
            _instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else if (_instance != this)
        {
            Debug.LogWarning("Duplicate UIManager found, destroying this
one.");
            Destroy(gameObject);
            return;
        }

        if (deathsText == null)
            Debug.LogError("Deaths Text är INTE tilldelad i UIManager!",
this);
        if (gemCounterText == null)
            Debug.LogError("Gem Counter Text är INTE tilldelad i
UIManager!", this);
        if (winMessagePanel == null)
```

```
            Debug.LogWarning("Win Message Panel är INTE tilldelad i
UIManager.", this);
        else
            winMessagePanel.SetActive(false);
    }

    void Start()
    {
        deathCount = 0;
        UpdateDeathUI();
    }

    private void OnEnable()
    {
        if (LevelManager.Instance != null)
        {
            LevelManager.Instance.OnGemCountChanged += UpdateGemCounter;

            UpdateGemCounter(0,
LevelManager.Instance.GetTargetGemCount());
            Debug.Log("UIManager prenumererar på
LevelManager.OnGemCountChanged.");
        }
        else
        {
            Debug.LogWarning("Kunde inte prenumerera på LevelManager
events vid OnEnable (LevelManager ej redo?). Försöker igen senare om
LevelManager startar.");
            if (gemCounterText != null) gemCounterText.text = "Gems: - /
-";
        }
    }

    private void OnDisable()
    {
        if (LevelManager.Instance != null)
        {
            LevelManager.Instance.OnGemCountChanged -= UpdateGemCounter;
            Debug.Log("UIManager avprenumererar från
LevelManager.OnGemCountChanged.");
        }
    }

    public void IncrementDeaths()
    {
        deathCount++;
```

```
        UpdateDeathUI();
        Debug.Log("Death count updated by UIManager: " + deathCount);
    }

    private void UpdateDeathUI()
    {
        if (deathsText != null)
        {
            deathsText.text = "Deaths: " + deathCount;
        }
    }

    public void UpdateGemCounter(int currentGems, int targetGems)
    {
        if (gemCounterText != null)
        {
            gemCounterText.text = $"Gems: {currentGems} / {targetGems}";
        }
    }

    public void ShowWinMessage()
    {
        if (winMessagePanel != null)
        {
            winMessagePanel.SetActive(true);
            Debug.Log("Showing Win Message Panel.");
        }
        else
        {
            Debug.LogWarning("Win Message Panel not set, cannot show
message.");
        }
    }
}
```

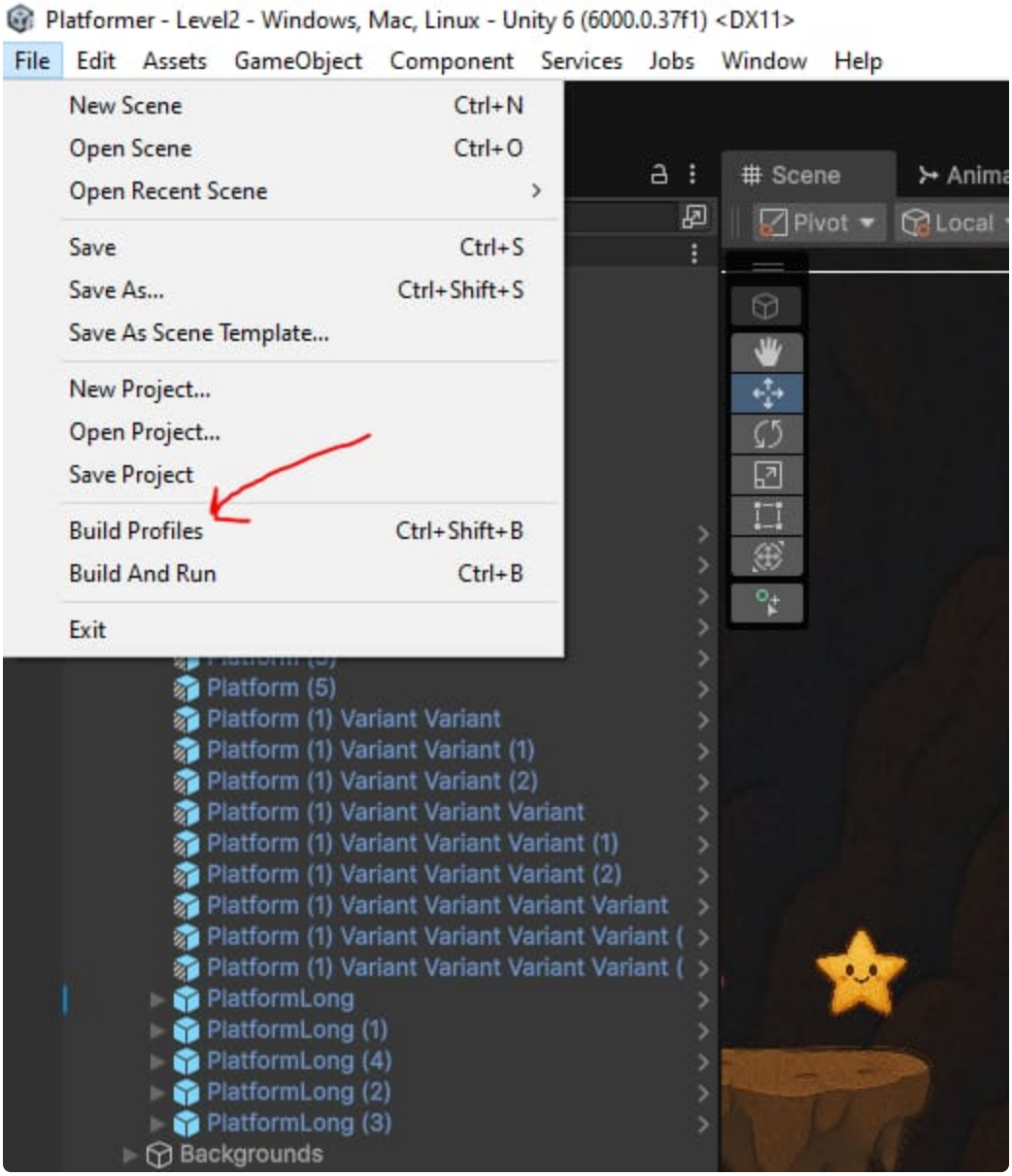
3. **Konfigurera UIManager i Inspektorn:**

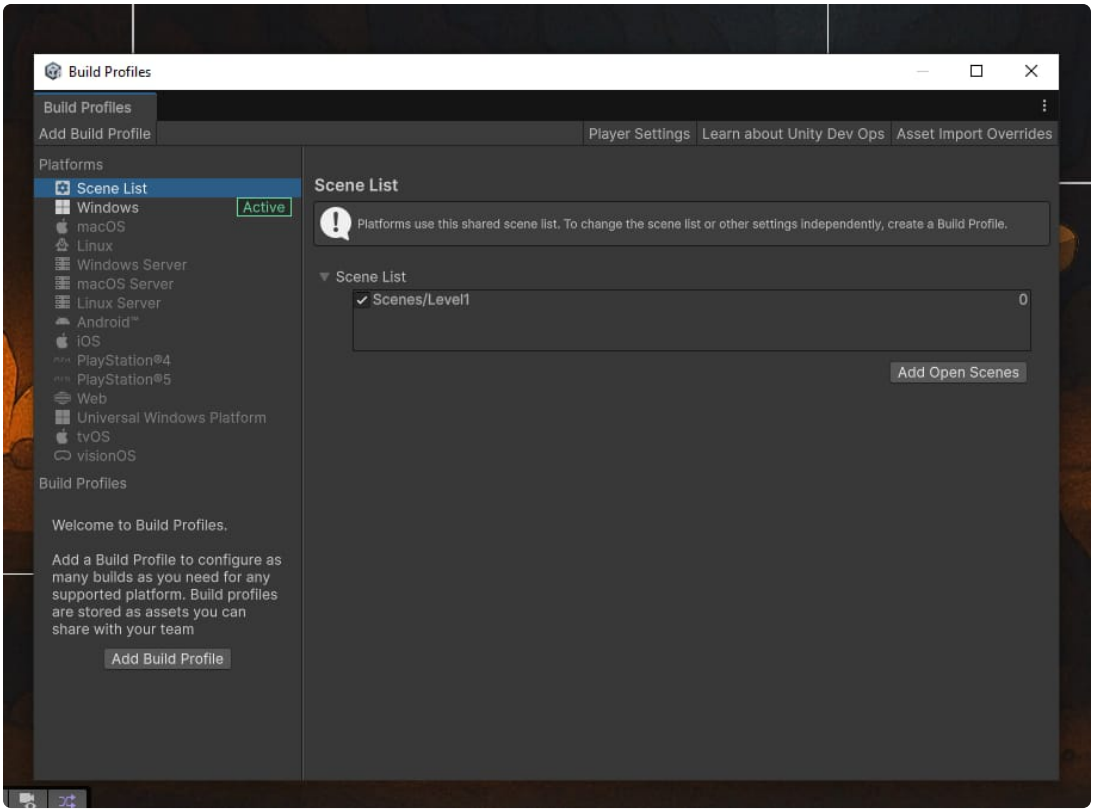
- Se till att ditt UIManager-skript sitter på ett GameObject i scenen (ofta på själva Canvas-objektet eller ett dedikerat UIManager-GameObject).
- Dra ditt GemCounterText (TextMeshPro-objektet) från Hierarkin till Gem Counter Text-fältet i Inspektorn på ditt UIManager-objekt.
- (Valfritt) Skapa en enkel panel (UI -> Panel) eller text (UI -> Text - TextMeshPro) som säger "Du Vann!". Dra detta objekt till Win Message Panel-fältet på UIManager. Se till att detta panel/text-objekt är

inaktiverat (checkbox avmarkerad) i Hierarkin från början.

Steg 4: Se till att Scenen finns i Build Settings

1. För att SceneManager.LoadScene() ska fungera korrekt (även för att ladda om den aktuella scenen) måste scenen finnas med i projektets Build Settings.
2. Spara din nuvarande scen (File -> Save Scene).
3. Gå till File -> Build Settings....
    - Om listan under "Scenes In Build" är tom, eller om din nuvarande scen inte finns där:
4. Klicka på knappen Add Open Scenes.
5. Din nuvarande scen bör nu visas i listan med ett index (t.ex. 0).
6. Stäng Build Settings-fönstret.


Platformer - Level2 - Windows, Mac, Linux - Unity 6 (6000.0.37f1) <DX11>
File
Edit
Assets
GameObject
Component
Services
Jobs
Window
Help
New Scene Ctrl+N
Open Scene Ctrl+O
Open Recent Scene
Save Ctrl+S
Save As... Ctrl+Shift+S
Save As Scene Template...
New Project...
Open Project...
Save Project
Build Profiles Ctrl+Shift+B
Build And Run Ctrl+B
Exit
Scene
Pivot
Local
Platform (5)
Platform (1) Variant Variant
Platform (1) Variant Variant (1)
Platform (1) Variant Variant (2)
Platform (1) Variant Variant Variant
Platform (1) Variant Variant Variant (1)
Platform (1) Variant Variant Variant (2)
Platform (1) Variant Variant Variant Variant
PlatformLong
PlatformLong (1)
PlatformLong (4)
PlatformLong (2)
PlatformLong (3)
Backgrounds



Steg 5: Placera ut Gems och Testa

Placera Gems: Dra din GemCollectible-prefab från Projekt-panelen in i din scen. Placera ut minst 3 Gems på olika platser i din nivå.

Kör Spelet: Tryck på Play.

Verifiera att följande fungerar:

- Startar Gem-räknaren i UI:t korrekt (t.ex. "Gems: 0 / 3")?
- När du rör vid en Gem med spelaren:
    - Försvinner Gem-objektet?
    - Spelas pickup-ljudet (om du konfigurerat det)?
    - Ökar räknaren i UI:t (t.ex. "Gems: 1 / 3")?
    - Skrivs loggmeddelanden ut i Console-fönstret?
- När du samlar den tredje Gem:en:
    - Skrivs "Level Complete!" ut i Console?
    - Spelas level complete-ljudet (om konfigurerat)?
    - Visas ditt "Du Vann!"-meddelande (om konfigurerat)?
    - Efter den inställda fördröjningen (Level Reload Delay), startar nivån om från början? (Spelaren är tillbaka vid startpositionen, alla Gems är tillbaka, räknaren är nollställd).

Du har nu implementerat en komplett spel-loop med ett tydligt mål (samla Gems) och en mekanism för att avsluta (och starta om) nivån med hjälp av Unitys SceneManagement. LevelManager agerar central koordinator för nivåns tillstånd, medan GemCollectible hanterar själva insamlingen och UIManager presenterar informationen för spelaren.

Del 5: Skapa en ny scen och ladda den

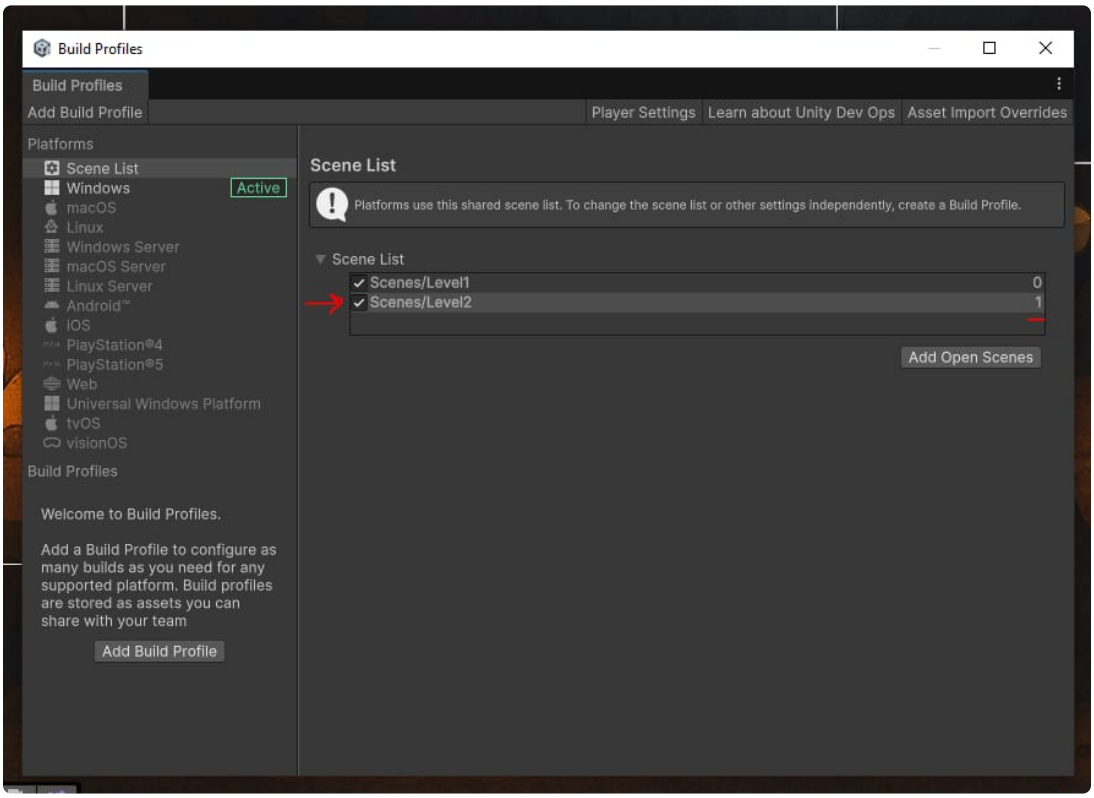
Nu ska vi skapa en ny scen och ladda den när spelaren har samlat alla Gems. Vi ska göra så att en ny nivå laddas genom att öka indexet för den aktuella scenen. För att göra detta behöver vi juster ai LevelCompleteSequence-metoden i LevelManager.cs.

LevelManager.cs (uppdatering)

```
    private IEnumerator LevelCompleteSequence()
    {
        if (levelCompleteSound != null && AudioManager.Instance != null)
        {
            AudioManager.Instance.PlaySoundOneShot(levelCompleteSound);
        }
        else if (levelCompleteSound != null)
        {
            Debug.LogWarning("AudioManager not found for level complete
sound.");
        }

        UIManager.Instance?.ShowWinMessage();

        yield return new WaitForSeconds(levelReloadDelay);

        Debug.Log("Reloading current scene...");
        Scene currentScene = SceneManager.GetActiveScene();
        SceneManager.LoadScene(currentScene.buildIndex + 1); // Ladda
nästa scen genom att öka indexet med 1
    }
```

- Vi behöver lägga till en ny scen i vår Build Settings. Skapa en ny scen och spara den som "Level2". I Build Settings ska den nu vara med i

listan. Se till att den ligger efter din första scen (t.ex. index 1 om din första scen har index 0). Se bilden nedanför.

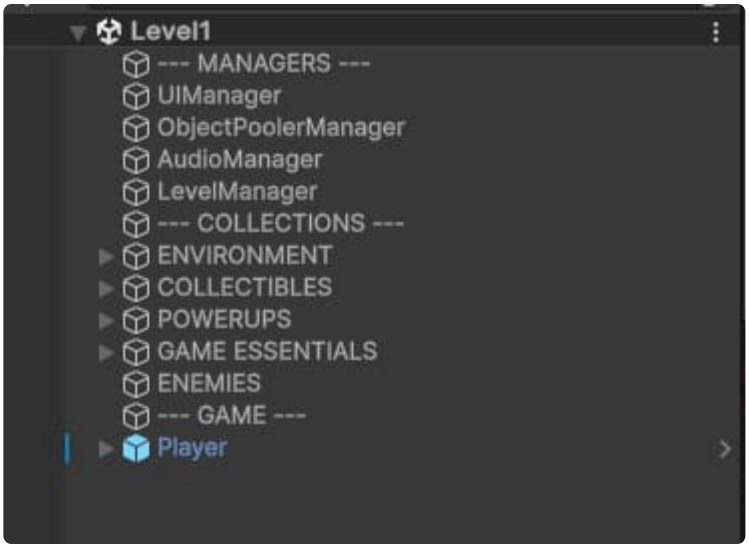


- Om allt fungerar som det ska nu - så borde Level 2 spelas efter att vi har samlat alla Gems i Level 1.

Del 6: Städa upp och avsluta

Nu har vi implementerat ett komplett PowerUp-system med Visitor Pattern, en nivåavslutningsmekanism och en ny scenladdning. Du kan nu städa upp din kod och projektstruktur. Här är några förslag:

- Rensa upp oanvända skript och prefabs.
- Städa i Hierarkin och Projekt-panelen. Nedanför kan vi se hur Hierachy kan se ut. Se till att du har en struktur som är lätt att navigera i. Här har jag organiserat mina GameObjects med hjälp av tomma GameObjects som jag döpt till "PowerUps" och "Collectibles" osv. Se bilden nedan



för exempel.

- Se till att alla GameObjects har rätt namn och taggar.