Zapraszamy do zapoznania się z projektem dostępnym pod tym linkiem. Projekt można pobrać w formie archiwum ZIP lub skorzystać z oprogramowania Git.

Aby pobrać projekt za pomocą Git, należy skopiować link SSH lub HTTPS, a następnie w terminalu lokalnego komputera wykonać polecenie:

- Dla SSH: `git clone git@github.com:AGH-Code-Industry/sfi-warsztaty.git docelowy-folder`
- Dla HTTPS: `git clone https://github.com/AGH-Code-Industry/sfiwarsztaty.git docelowy-folder`

Do pracy z projektem w Unity należy zainstalować Unity Hub i wybrać wersję **2022.3.22f1**. W przypadku braku tej wersji w sekcji "Installs" można ją wybrać jako docelową.

Możliwe, że projekt nie będzie zawierał zintegrowanego środowiska programistycznego.

Aby to zrobić, należy przejść do zakładki **Edit->Preferences->External Tools** w Unity i wybrać preferowane środowisko.

Jeśli preferujesz narzędzia takie jak **Rider**, **Visual Studio** lub **Visual Studio Code** to możesz je skonfigurować jako domyślne w Unity. Zalecamy korzystanie z Visual Studio Code ze względu na jego niewielkie rozmiary.

Konfiguracja Visual Studio Code ›

Aby zapewnić prawidłowe działanie Visual Studio Code z Unity, zaleca się zainstalowanie następujących dodatków w sekcji "Extensions":

- Unity
- Unity Code Snippets
- C#

Tworzenie Mapy

Na początku należy stworzyć podstawową mapę, po której będzie się poruszał gracz. W tym celu konieczne jest utworzenie tilemapy, czyli zestawu 2D obrazków, które posłużą jako pędzle do malowania mapy na płótnie zwanej sceną.

Aby to zrobić, należy przejść do okna Inspektora, a następnie kliknąć prawym przyciskiem myszy i wybrać kolejno: **2D Object -> Tilemap -> Rectangular**. W ten sposób zostanie utworzona pierwsza tilemapa. Następnie należy stworzyć dla niej paletę obrazków.

Jeśli nie posiadasz okna Tile Palette, można je otworzyć poprzez przejście do zakładki **Window** a następnie wybranie **2D -> Tile Palette**.

Aby stworzyć nową paletę, należy wejść do okna Tile Palette, wybrać **No Valid Palette -> Create New Palette**. Należy ustawić własną nazwę, a resztę ustawień pozostawić domyślnie. Następnie należy zapisać paletę w folderze **Tilemaps -> industrial-zone**.

Po utworzeniu palety, należy ją wypełnić przeciągając obrazy 2D do okna Tile Palette. Obrazy znajdują się w folderze **Sprites -> industrial-zone -> 1 Tiles**. Należy zaznaczyć wszystkie obrazy i przeciągnąć je do okna palety. Następnie należy zapisać je w tym samym folderze, w którym znajduje się tilemapa (**Tilemaps -> industrial-zone**).

Po przygotowaniu palety, można zacząć tworzyć własną mapę. Należy wybrać odpowiedni obraz w oknie Tile Palette, a następnie można nim malować w oknie Scene.

Postać

Po stworzeniu mapy, nadszedł czas na utworzenie postaci. Aby to zrobić, należy przejść do folderu **Sprites -> cyberpunk-characters -> 3 Cyborg -> Cyborg_idle** i przeciągnąć obraz postaci do okna **Hierarchy**.

Tym samym utworzono obiekt gracza, jednakże aktualnie jest on pustym obiektem, nieposiadającym żadnych funkcji.

Następnie dodajemy do niego komponent **Rigidbody 2D**. Po uruchomieniu gry, obiekt gracza powinien zacząć spadać z przyspieszeniem.

Aby umożliwić graczowi zatrzymywanie się na wcześniej utworzonej mapie, konieczne jest dodanie kolidorów do obiektu gracza oraz tilemapy. W obiekcie gracza należy dodać komponent **Capsule Collider 2D**, natomiast w obiekcie tilemapy należy dodać komponent **Tilemap Collider 2D**.

Skrypt Poruszania się

Aby umożliwić postaci poruszanie się, należy stworzyć odpowiedni skrypt obsługujący różne stany gracza. W tym celu należy zapoznać się ze skryptem **Script -> PlayerStateManager.cs**, który jest odpowiedzialny za zmianę stanów gracza, takich jak **Idle, Run, Jump, ...**. W tym skrypcie została już zaimplementowana logika stanu **Idle**. Twoim zadaniem będzie stworzenie stanu **Run**, aby gracz mógł się poruszać.

Aby wprowadzić nowy stan, należy stworzyć klasę implementującą **IPlayerState** oraz zdefiniować obiekt tej klasy w **PlayerStateManager.cs**.

Tworzenie Stanu RUN ›

- Stwórz plik **PlayerRunState.cs**
- Zaimplementuj interfejs **IPlayerState** w klasie **PlayerRunState**
- Utwórz obiekt klasy **PlayerRunState** w metodzie **Awake** klasy **PlayerStateManager**

Mając zdefiniowany stan, należy określić warunki przełączania się między stanami. Najlepiej umieścić to w metodzie **Update** danego stanu poprzez ustawienie warunków zmiany stanu.

Przykładowy kod zmieniający stan z **Idle** na **Run** w pliku **PlayerIdleState.cs**:

```
public void Update(PlayerStateManager manager)
{
        if(manager.frameInput.Move ≠ Vector2.zero)
                manager.SwitchState(manager.RunState);
}
```

Twoim zadaniem jest napisanie skryptu w pliku **PlayerRunState.cs**, który będzie zmieniał stan na **Idle**.

Zmiana na stan Idle >

Zmienna **frameInput.Move** zawiera informacje o klawiszach wciśniętych przez gracza (klawisz A, D, strzałki w lewo i prawo). Jeśli wektor Move będzie równy zero, oznacza to, że gracz powinien zmienić stan na Idle.

Teraz, gdy postać może swobodnie przechodzić między stanami, brakuje tylko skryptu kontrolującego prędkość gracza. Aby to osiągnąć, należy utworzyć zmienną **speed** oraz pobrać komponent **Rigidbody2D** w klasie **PlayerStateManager**. Obie zmienne powinny być publiczne, aby skrypty stanów miały do nich dostęp.

W klasie **PlayerRunState** w funkcji **FixUpdate** należy zawrzeć logikę zmieniającą prędkość gracza.

Zmiana prędkości >

- **Rigidbody2D** zawiera zmienną **velocity**, która kontroluje prędkość
- **frameInput.Move** zawiera kierunek poruszania się gracza
- **speed** to prędkość zdefiniowana przez gracza
- Należy pamiętać, aby zmieniać tylko składową **x**, a składową **y** pozostawić bez zmian

Aby zapewnić płynność ruchu gracza, w komponencie **Rigidbody2D** należy zaznaczyć opcję **Freeze Rotation Z**.

Może się zdarzyć, że po puszczeniu klawisza, postać nadal będzie się poruszać. Aby temu zapobiec, należy wprowadzić dwie zmiany:

- Zmodyfikować zakres **frameInput.Move** za pomocą następującego kodu:

```
frameInput.Move.x = Mathf.Abs(frameInput.Move.x) < 0.1f ? 0 :
Mathf.Sign(frameInput.Move.x);
```

- W metodzie **Start** stanu **Idle** należy zerować prędkość:

```
manager.rig.velocity = Vector2.zero;
```

Animacja

Po zaimplementowaniu poruszania się gracza, kolejnym krokiem jest dodanie do niego animacji. Aby to zrobić, należy przejść do obiektu gracza i dodać mu komponent **Animator**. Następnie potrzebny będzie kontroler animacji, który należy utworzyć. Stwórz folder **Animations -> Cyborg**, a następnie klikając prawym przyciskiem myszy wybierz **Create -> Animator Controller**. Przypisz ten kontroler do **Animatora** gracza.

Mając **Animatora**, konieczne jest stworzenie animacji. W tym celu użyteczne będą dwa okna: **Animator** oraz **Animation**.

Okna Animator oraz Animation >

Okna Animator i Animation znajdują się w zakładce **Window->Animation->Animation** oraz **Window->Animation->Animator**.

Teraz stworzymy animację **Idle**. Wejdź w zakładkę **Animation** i utwórz animację o nazwie **Idle** w folderze **Animations->Cyborg**.

Aby stworzyć animację, potrzebne będą obrazy. Przejdź do **Sprites->cyberpunk-characters->3 Cyborg**, a następnie rozwiń **Cyborg_idle**. Zaznacz wszystkie 4 obrazy i przeciągnij je do okna **Animation**.

Po włączeniu przycisku **Play** w zakładce **Animation**, obraz gracza powinien zacząć się animować. Jednakże może to być zbyt szybkie, dlatego należy zaznaczyć 4 obrazy w oknie **Animation** i rozciągnąć je do 30 milisekund.

Na podstawie animacji **Idle**, stwórz animację **Run**.

Animacja Run >

> Aby utworzyć kolejną animację, wejdź do okna **Animation**, rozwiń aktualnie odtwarzaną animację i wybierz **Create New Clip**. Obrazy do animacji Run znajdują się w **Sprites->cyberpunk-characters->3 Cyborg-Cyborg_run**.

W oknie **Animator** powinieneś teraz widzieć dwie animacje: **Idle** oraz **Run**. Należy zapamiętać nazwy tych animacji, ponieważ będą potrzebne w kodzie.

Aby gracz zmieniał animację wraz ze zmianą stanu, w metodzie **Start** stanu należy umieścić kod:

```
manager.animator.Play("nazwa-animacji")
```

animator to komponent **Animator**, który został pobrany w **PlayerStateManager**.

Metoda **Start** stanu **Idle** będzie wyglądała następująco:

```
public void Start(PlayerStateManager manager)
{
        manager.rig.velocity = Vector2.zero;
        manager.animator.Play("Idle");
}
```

Na podstawie stanu **Idle** należy zmienić animację dla stanu **Run**.

Stan Jump oraz DoubleJump

Nadeszła pora na zaimplementowanie stanów **Jump** oraz **DoubleJump** dla postaci gracza.

Aby móc wejść w stan **Jump**, gracza należy poinformować o kolizji z podłożem. W tym celu stworzymy zmienną **grounded** typu **bool** w klasie **PlayerStateManager**.

Następnie użyjemy funkcji Physics2D.CapsuleCast do sprawdzenia kolizji. Kod będzie wyglądał następująco:

```
private void CheckCollisions()
{
```

```
        // zapobiega kolizji w własnym koliderem
    Physics2D.queriesStartInColliders = false;

    // używamy kopi kolidera gracza skierowanego o 0.05
    // jednostki w dół by zmierzyć kolizję
        grounded = Physics2D.CapsuleCast(col.bounds.center,
        col.size, col.direction, 0, Vector2.down, 0.05f);
}
```

Gdzie **col** jest komponentem **CapsuleCollider2D** pobranym w metodzie **Awake**.

Powyższa funkcja powinna być umieszczona w metodzie **FixedUpdate** klasy **PlayerStateManager**, ponieważ operuje na fizyce gry.

Następnie, aby możliwe było wykrywanie naciśnięcia klawisza **spacji** oraz informacja o kontakcie z podłożem, dodajemy nową zmienną do struktury **FrameInput**: `public bool Jump;`.

Możemy teraz uaktualnić zmienną **frameInput** w metodzie **Update** gracza:

```
frameInput = new FrameInput
{
        Jump = Input.GetButtonDown("Jump"),
        Move = new Vector2(Input.GetAxisRaw("Horizontal"),
        Input.GetAxisRaw("Vertical"))
};
```

Po ustawieniu odpowiednich warunków przejścia między stanami, stwórz stan **Jump**. W metodzie **Start** stanu **Jump** należy użyć funkcji **AddForce** na komponencie **Rigidbody2D**, aby wyrzucić gracza w górę.

Tworzenie Stanu Jump >

- Stwórz plik **PlayerJumpState.cs**
- Zaimplementuj w klasie **PlayerJumpState** interfejs **IPlayerState**
- Utwórz obiekt klasy **PlayerJumpState** w metodzie **Awake** klasy **PlayerStateManager**

Wymaga to zdefiniowania zmiennej `float jumpForce` w klasie **PlayerStateManager** oraz dodanie odpowiedniego kodu w stanie **PlayerJumpState**.

Następnie należy dodać warunek do stanów **Idle** oraz **Run** umożliwiający przejście do stanu **Jump** po naciśnięciu klawisza **spacji**.

Ponieważ metoda **Update** wykonuje się od razu po metodzie **Start**, wartość zmiennej **grounded** może nie zostać zmieniona na **false**. Aby temu zapobiec, należy stworzyć timer, który opóźni działanie warunków przejścia do stanów **Idle** oraz **Run**.

Posiadając tą zmienną możemy stworzyć następujący kod w stanie **PlayerJumpState**:

```
if (manager.grounded)
{
        manager.rg.AddForce(Vector2.up * manager.jumpForce,
        ForceMode2D.Impulse);
}
```

Gdy gracz przechodzi do stanu **Jump**, chcemy, aby mógł on zostać wyrzucony w górę tylko wtedy, gdy dotyka podłoża.

Po zaimplementowaniu działającego stanu **Jump**, pozostało nam zdefiniować, kiedy gracz powinien przejść do tego stanu.

W stanach **Idle** oraz **Run**, dodajemy warunek do metody **Update**:

```
if(manager.grounded && manager.frameInput.Jump)
        manager.SwitchState(manager.JumpState);
```

Następnie, w stanie **Jump**, należy dodać warunki przejścia do stanów **Idle** oraz **Run**.

Jednakże, metoda **Update** wykonuje się od razu po metodzie **Start**, przez co zmienna **grounded** może nie zmienić wartości na **false**, co skutkuje natychmiastowym przejściem do stanu **Run** lub **Idle** mimo, że gracz jest w powietrzu. Aby temu zaradzić, należy stworzyć timer, który opóźni działanie warunków przejścia do stanów **Idle** oraz **Run**.

Przykładowy kod na początku metody **Update** w klasie **PlayerJumpState** może wyglądać tak:

```
_timer += Time.deltaTime;
if (_timer < _jumpTime) return;
```

Pamiętaj, aby zresetować timer w metodzie **Start**: `_timer = 0;`.

Po wykonaniu tych kroków, stan **Jump** powinien działać poprawnie. Jednakże, nie jesteśmy w stanie kontrolować postaci podczas skoku. Aby to zmienić, skopiujemy skrypt dotyczący zmiany prędkości z stanu **Run** i wkleimy go w metodę **Update** stanu **Jump**.

Mając to zrobione, przechodzenie do stanu **Jump** będzie działać poprawnie, a gracz będzie mógł skakać tylko wtedy, gdy dotyka podłoża.

Aby skok działał poprawnie, należy ustawić odpowiednie parametry dla komponentu **Rigidbody 2D** oraz dla maszyny stanów gracza (**PlayerStateMachine**).

Dla komponentu **Rigidbody 2D**

- Use Auto Mass = true
- Gravity Scale = 10
- Collision Detection = Continuous
- Sleeping mode = Never Sleep

Dla komponentu **PlayerStateMachine**

- Jump Force = 80
- Speed = 20

Po zdefiniowaniu tych parametrów, należy stworzyć animację dla stanu skoku w oknie **Animation**. Obrazy do animacji skoku znajdują się w folderze **Sprites->cyberpunk-characters->3 Cyborg->Cyborg_jump**. Po stworzeniu animacji, należy uruchomić ją w metodzie **Start** stanu **Jump**.

Następnie przejdziemy do implementacji stanu **DoubleJump**. Ten stan będzie podobny do stanu **Jump**, więc należy wykonać wszystkie kroki, które zostały wykonane przy tworzeniu stanu **Jump**.

Wyjątkiem jest metoda **Start**, w której nie musimy już sprawdzać, czy gracz dotknął podłoża.

Po dodaniu stanu **DoubleJump**, pozostaje tylko dodać warunek do stanu **Jump**, który przejdzie do stanu **DoubleJump** po naciśnięciu klawisza **spacji** po skoku.

Po zakończeniu implementacji stanu **DoubleJump**, można zauważyć, że po podwójnym wciśnięciu klawisza **spacji**, gracz zaczyna animację **DoubleJump**, która wykonuje się w nieskończoność. Aby temu zapobiec, należy wejść do pliku animacji, który jest prawdopodobnie utworzony w folderze **Animations->Cyborg**, a następnie odznaczyć zmienną **Loop Time**.

Aby jeszcze ulepszyć tę animację, można na jej końcu dodać obrazek z animacji **Jump**.

Wiele kodu powtarza się w stanach, takich jak warunki zmiany stanów oraz zmiana prędkości gracza. Aby kod był bardziej czytelny, można te części kodu wydzielić do metod w **PlayerStateManager**, a następnie za pomocą **managera** wywoływać je w skryptach stanowych. To sprawi, że kod będzie czytelniejszy i oszczędzi czasu podczas dodawania nowych stanów.

Pułapki

Jeśli nie stworzyłeś własnej mapy, w **Scenes->SampleMap** znajdziesz przykładową mapę z jedną pułapką (*Hammer*).

Twoim zadaniem będzie teraz stworzenie dwóch nowych pułapek oraz zaimplementowanie logiki zadającej obrażenia graczowi.

Pierwsza pułapka - Piła

Obrazy do tego obiektu znajdziesz w **Sprites->Traps->Saw**. Przeciągnij obraz **off** na scenę. Do utworzonego obiektu dodaj następujące komponenty:

- **Circle Collider 2D**, dostosuj w nim odpowiedni promień,
- **Animator**, w którym stworzysz animację z obrazów **Sprites->Traps->Saw->On**.

Po uruchomieniu gry, **Piła** powinna zacząć się kręcić i móc kolidować z graczem.

Teraz należy napisać logikę dla **Piły**. Będzie ona przemieszczała się między wybranymi punktami na mapie. Gdy dojdzie do ostatniego punktu, zaczyna swoją podróż od nowa, wracając do początkowego punktu.

Logika Piły >

- Stwórz plik **Saw.cs**
- Zdefiniuj w nim publiczną tablicę typu **Transform**,
- W metodzie **Update** użyj metody Vector3.MoveTowards do przemieszczania się między punktami,
- Jako trzeci argument użyj własnej zmiennej **speed**,
- Zmienną **speed** pomnóż przez **Time.deltaTime**, aby zapewnić, że piła będzie miała taką samą szybkość w każdej ilości klatek,
- Gdy pozycja piły będzie równa docelowej pozycji, zwiększ counter w tablicy i zacznij przemieszczać się do nowego punktu.

Po skończeniu konfigurowania obiektu, przeciągnij go do folderu **Prefabs**. Gdy to zrobisz, zostanie on skopiowany i pojawi się jako obiekt prefabrykowany w tym folderze. Od teraz będziesz mógł używać tego obiektu, aby tworzyć setki tego typu pułapek.

Druga pułapka - Skrzynka na Strzały

Zadaniem skrzynki będzie strzelanie strzałami, które po kontakcie z graczem zabiorą mu życie.

Obrazy znajdziesz w folderze **Sprites->Traps->Arrows**.

Potrzebujemy stworzyć dwa skrypty:

- **Arrow.cs** -> logika latania strzały
- **ArrowBox.cs** -> logika strzelania strzał

Wpierw musisz stworzyć **Prefaba Strzał**, aby móc go użyć podczas tworzenia **ArrowBox**.

Strzała >

- Stwórz plik **Arrow.cs**
- Dodaj **Rigidbody2D**
- Dodaj **BoxCollider2D**
- Dodaj zmienną **speed**
- W metodzie **FixUpdate** zmień velocity na pokrywające się z **speed**

Skrzynka na Strzały >

- Stwórz plik **ArrowBox.cs**
- Dodaj zmienną **timeToShot** typu **float** oraz **arrow** typu **GameObject**,
- Stwórz **timer**, a następnie co określony czas generuj strzały za pomocą metody Instantiate,
- Kierunek strzał będzie definiowany poprzez rotację skrzynki.

Ukończone obiekty zapisz jako prefaby w folderze **prefabs**.

Na koniec rozmieść po mapie pułapki.

Podążanie Kamery

Teraz czas na ustawienie kamery. Zajmie nam to chwilę. Wystarczy, że w oknie **Hierarchy** przeciągniesz obiekt kamery na obiekt cyborga tak, aby był on pod nim. Kamera będąc dzieckiem Cyborga będzie automatycznie zmieniać pozycję wraz z graczem.

Pozycja Z Kamery >

Nie zapomnij, by pozycja Z była oddalona od reszty obiektów. Może się okazać, że kamera nie będzie nic widzieć, gdyż wszystkie obiekty znajdują się za jej polem widzenia. Ustaw pozycję Z na wartość mniejszą niż -1, aby temu zaradzić.

Tło Gry

1. Prawym przyciskiem kliknij na okno **Hierarchy**.
2. Wybierz opcję **UI -> Image**. Spowoduje to stworzenie obiektu **Canvas** oraz obiektu **Image** w nim.
3. Przejdź do obiektu **Canvas**.
4. Znajdź zmienną **Sort Order** i zmień jej wartość na **-1**. Ta zmiana pozwoli na wyświetlanie tła pod innymi elementami gry.
5. Przejdź do obiektu **Image** i przypisz do niego odpowiedni obraz. Możesz znaleźć go w ścieżce: **Sprites -> industrial-zone -> 2 Background/Background**.

To wszystko, co musisz zrobić, aby dodać tło do swojej gry.

Zadawanie obrażeń

Aby zadawać graczowi obrażenia, musimy odróżnić zwykłe obiekty od pułapek. W Unity istnieje możliwość przypisania tagów do obiektów, co ułatwia identyfikację ich typu.

1. Przejdź do dowolnego obiektu w hierarchii.
2. W oknie Inspector znajdź właściwość **Tag** na samej górze.
3. Kliknij bieżący tag (zwykle **Untagged**) i wybierz opcję **Add Tag**.
4. Dodaj nowy tag o nazwie "Trap" do listy.
5. Następnie przejdź do wszystkich pułapek, które mają zadawać obrażenia graczowi, i przypisz im ten tag.

W skrypcie **PlayerStateManager** utwórz zmienną `life`, która będzie przechowywać aktualną ilość życia gracza. Następnie, na końcu klasy, dodaj następujący kod:

```
private void OnCollisionEnter2D(Collision2D other)
{
    if (other.transform.CompareTag("Trap"))
    {
        life -= 1;
    }
}
```

Gdy wartość zmiennej `life` spadnie do zera, oznacza to, że gracz umarł.

Twoim zadaniem jest teraz stworzenie stanu **Death**. Obrazy do animacji znajdziesz w folderze **Sprites->cyberpunk-characters->3 Cyborg->Cyborg_death**.

Poza tym, aby zasygnalizować graczowi otrzymane obrażenia, zmienimy kolor gracza na czerwony na krótki okres czasu. Do tego celu stworzymy coroutine w klasie **PlayerStateManager**:

```
IEnumerator ShowDamage()
{
        // zmiana kolor gracza na zerwony
    render.color = Color.red;
    // czeka 0.2 sekundy
    yield return new WaitForSeconds(0.2f);
    // zmiena kolor gracza na domyślny
    render.color = Color.white;
}
```

Następnie będziemy wywoływać tę metodę podczas zadawania obrażeń. Możemy to zrobić za pomocą metody `StartCoroutine("ShowDamage")`.

Teraz, gdy gracz zostanie trafiony, zmieni się jego kolor na czerwony na krótki czas, co zasygnalizuje mu otrzymane obrażenia.

Pamiętaj, żeby usunąć obiekt gracza po jego śmierci, możesz użyć metody `Destroy(gameObject)`.

UI - Serca Gracza

Aby stworzyć interfejs użytkownika (UI) z sercami gracza, będziemy korzystać z elementów UI w Unity, takich jak Canvas i Image.

1. Najpierw stwórz nowy canvas, klikając prawym przyciskiem myszy na oknie **Hierarchy**, wybierz **UI -> Canvas**. To stworzy pierwsze płótno UI.
2. Następnie kliknij prawym przyciskiem myszy na obiekcie Canvas i wybierz **Create Empty**. Nazwij ten nowy obiekt "life". To będzie kontenerem dla serc gracza.
3. Dodaj do obiektu "life" komponent **Horizontal Layout Group**. Dzięki temu jego dzieci (czyli serca) będą automatycznie ustawiać się w linii poziomej.

4. Dodaj trzy obiekty typu Image do obiektu "life". Zauważ, że automatycznie ustawiają się one w linii obok siebie. Wejdź w każdy z tych obiektów i przypisz im grafikę **Sprites->Heart**.
5. Teraz musimy przenieść obiekt "life" do lewego górnego rogu. W komponencie **Rect Transform** obiektu "life" znajdź kwadrat symbolizujący jego pozycję i przesuń go w lewy górny róg. Możesz to zrobić klikając na kwadrat i przeciągając go, lub wybierając odpowiednią opcję z rozwijanej listy.

Mamy już stworzone UI serc gracza, teraz potrzebujemy logiki.

Potrzebujemy wiedzieć, kiedy gracz otrzymuje obrażenia. W tym celu użyjemy eventów, czyli zdarzeń, do których inne skrypty mogą się podłączyć i wywołać swoje funkcje w odpowiedzi na to zdarzenie. W naszym przypadku zdarzeniem będzie otrzymanie obrażeń przez gracza.

Mamy właśnie stworzone UI serc, teraz tylko brakuje nam logiki.

Potrzebujemy wiedzieć kiedy gracz otrzymał obrażenia, w tym celu posłużymy się eventami. Są to zdarzenia do których każdy skrypt może się podpiąć a następnie wywołać swój kod jeżeli takowe zdarzenie nadejdzie. Tym zdarzeniem będą obrażenia gracza.

Wejdź w skrypt **PlayerStateManager** i dodaj na samej górze poniższy kod:

```
// Action posłuży nam do obsługi eventu
public static event Action Damage;
```

Następnie dodaj wywołanie tego eventu do metody **OnCollisionEnter2D**:

```
Damage?.Invoke();
```

Teraz musimy się podłączyć do tego eventu, aby nasze serca UI mogły odpowiednio reagować. Stwórzmy więc nowy skrypt **LifeView.cs** i przypiszmy go do obiektu "life" w naszym stworzonym canvasie.

Rozpocznijmy od utworzenia metody w klasie **LifeView**, która będzie wywoływana po otrzymaniu obrażeń przez gracza:

```
private void OnPlayerGetDamage()
{

}
```

Następnie podpinamy tę metodę w metodzie **Start**:

```
PlayerStateManager.Damage += OnPlayerGetDamage;
```

Teraz każde wywołanie eventu **Damage** będzie powodowało uruchomienie metody **OnPlayerGetDamage**.

Przejdźmy teraz do obsługi serc. Stwórzmy publiczną tablicę obiektów typu **Image** w klasie **LifeView**. Ta tablica posłuży nam do przechowywania referencji do wszystkich obiektów serc. Po zdefiniowaniu tablicy w Inspektorze obiektu **life**, będziesz mógł przypisać do niej wszystkie serca.

```
using UnityEngine.UI;

public class LifeView : MonoBehaviour
{
    public Image[] hearts;

    // Reszta kodu ...
}
```

Następnie, po przypisaniu obiektów serc do tablicy, musimy wyłączać kolejne serca po otrzymaniu obrażeń przez gracza. Stwórzmy do tego skrypt, który będzie wyłączał kolejne serca:

```
private void OnPlayerGetDamage()
{
    foreach (Image heart in hearts)
    {
        if (heart.enabled)
        {
            heart.enabled = false;
            break; // Wyłączamy tylko jedno serce na raz
```

```
        }
    }
}
```

Teraz, po wejściu gracza w pułapki, powinieneś zauważyć, jak kolejne serca znikają.

Na koniec dodajmy do klasy **LifeView** kod, który odpowie za odpięcie się od eventu po usunięciu obiektu:

```
private void OnDestroy()
{
    PlayerStateManager.Damage -= OnPlayerGetDamage;
}
```

Ten fragment kodu służy do odpięcia się od eventu po usunięciu obiektu. Jest to ważne, ponieważ zapobiegnie to problemom związanych z odwoływaniem się do obiektu, który już nie istnieje. Dzięki temu event zostanie poprawnie odpięty, gdy obiekt zostanie zniszczony, co minimalizuje ryzyko błędów w działaniu programu.

UI - Money

Rozpocznijmy od stworzenia obiektu **Money** w scenie gry. Przeciągnij obraz pieniędzy z folderu **Sprites->industrial-zone->4 Animated objects->Money** do okna **Hierarchy**. Następnie dodaj do niego animację oraz komponent **BoxCollider2D**. Upewnij się, że opcja **Is Trigger** jest ustawiona na **true**. Włączenie tej opcji pozwoli na wykrycie kolizji gracza z pieniędzmi. Dodaj również tag **Player** do obiektu gracza, abyśmy mogli go później zidentyfikować.

> **Dodawanie Tagu** >
>
> Wejdź w jakikolwiek obiekt a następnie w oknie **Inspector** u samej góry ujrzysz właściwość **Tag untagged**. kliknij ją a następnie kliknij **Add Tag**.

Następnie utwórz skrypt **Money.cs** i dodaj w nim eventa **Collected**:

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

```
public class Money : MonoBehaviour
{
    public static event Action Collected;

    // Dodaj kod obsługujący kolizję z graczem
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            // Dodaj wywołanie eventa Collected
            Collected?.Invoke();

            // Zniszcz obiekt pieniędzy
            Destroy(gameObject);
        }
    }
}
```

Teraz, gdy gracz wejdzie w kolizję z pieniędzmi, zostanie wywołany event **Collected**, a obiekt pieniędzy zostanie zniszczony.

Po zakończeniu konfiguracji obiektu pieniędzy, przeciągnij go do folderu **Prefabs**. Unity automatycznie utworzy kopię prefabrykatu, co umożliwi łatwe umieszczanie go w scenie gry.

Następnie, w obiekcie **Canvas**, stwórz obiekt **Money** na wzór obiektu **Life**. Dodaj do niego komponenty **TextMeshPro** oraz **Image** jako dzieci.

Stwórz skrypt **MoneyView.cs** i zdefiniuj w nim dwie zmienne: jedną dla komponentu TextMeshProUGUI (_text) oraz drugą dla przechowywanej ilości pieniędzy (_money).

```
using UnityEngine;
using TMPro;

public class MoneyView : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI _text;
    private int _money = 0;
```

```
    // Reszta kodu ...
}
```

Następnie podłącz event **Collected** w metodzie **Start** oraz wykonaj odpowiednie operacje, aby zmienić UI w zależności od zebranych pieniędzy.

```
using UnityEngine;
using TMPro;

public class MoneyView : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI _text;
    private int _money = 0;

    private void Start()
    {
        Money.Collected += OnMoneyCollected;
    }

    private void OnDestroy()
    {
        Money.Collected -= OnMoneyCollected;
    }

    private void OnMoneyCollected()
    {
        // Inkrementuj ilość zebranych pieniędzy
        _money++;

        // Zaktualizuj tekst na interfejsie użytkownika
        _text.text = _money;
    }
}
```

Teraz każde zebranie pieniędzy spowoduje zwiększenie licznika i aktualizację tekstu na interfejsie użytkownika. Upewnij się również, że skrypt **MoneyView.cs** jest przypisany do obiektu **Money** w scenie gry.

UI - Koniec Gry

Rozpocznijmy od dodania eventu informującego o śmierci gracza w skrypcie **PlayerStateManager**, a następnie wywołaj ten event, gdy życie gracza osiągnie wartość zero.

```
using UnityEngine;
using System;

public class PlayerStateManager : MonoBehaviour
{
    public static event Action PlayerDied;

    private int _life = 3; // Przykładowa liczba życia gracza

    // Metoda zmniejszająca życie gracza i wywołująca event
PlayerDied, gdy życie wynosi 0
    private void TakeDamage()
    {
        _life--;
        if (_life ≤ 0)
        {
            PlayerDied?.Invoke();
        }
    }

    // Reszta kodu ...
}
```

Teraz stwórzmy obiekt **Card** na podstawie tworzenia obiektu pieniędzy, znajdującego się pod ścieżką **Sprites->industrial-zone->4 Animated objects->Card**.

Następnie utwórz nowy obiekt **Canvas**, aby panel końca gry był wyświetlany nad widokiem gracza. Wewnątrz tego Canvasa stwórz obiekt **Panel** oraz dodaj do niego tekst "Game Over" oraz przycisk "Play Again".

Następnie stwórz skrypt **EndGameView.cs** i dodaj go do obiektu **Canvas**. W skrypcie zdefiniuj zmienną dla obiektu panelu, która będzie odpowiedzialna za

wyświetlanie panelu Game Over. Dodaj dwa eventy: jeden związany ze śmiercią gracza, drugi z zebraniem karty. Następnie przypisz funkcje do tych eventów, które będą wyświetlać panel Game Over.

```
using UnityEngine;
using UnityEngine.UI;

public class EndGameView : MonoBehaviour
{
    public GameObject gameOverPanel;

    private void Start()
    {
        PlayerStateManager.PlayerDied += ShowGameOverPanel;
    }

    private void OnDestroy()
    {
        PlayerStateManager.PlayerDied -= ShowGameOverPanel;
    }

    private void ShowGameOverPanel()
    {
        gameOverPanel.SetActive(true);
    }
}
```

Na koniec przypisz do przycisku "Play Again" funkcję `ReloadGame()`:

```
using UnityEngine.SceneManagement;

public class EndGameView : MonoBehaviour
{
    // Reszta kodu ...

    public void ReloadGame()
    {
        Scene scene = SceneManager.GetActiveScene();
        SceneManager.LoadScene(scene.name);
    }
```

```
}
```

Teraz, gdy gracz umrze lub zbierze kartę, panel Game Over zostanie wyświetlony, a przycisk "Play Again" umożliwi mu ponowne rozpoczęcie gry. Upewnij się, że skrypt **EndGameView.cs** jest przypisany do obiektu **Canvas** w scenie gry.