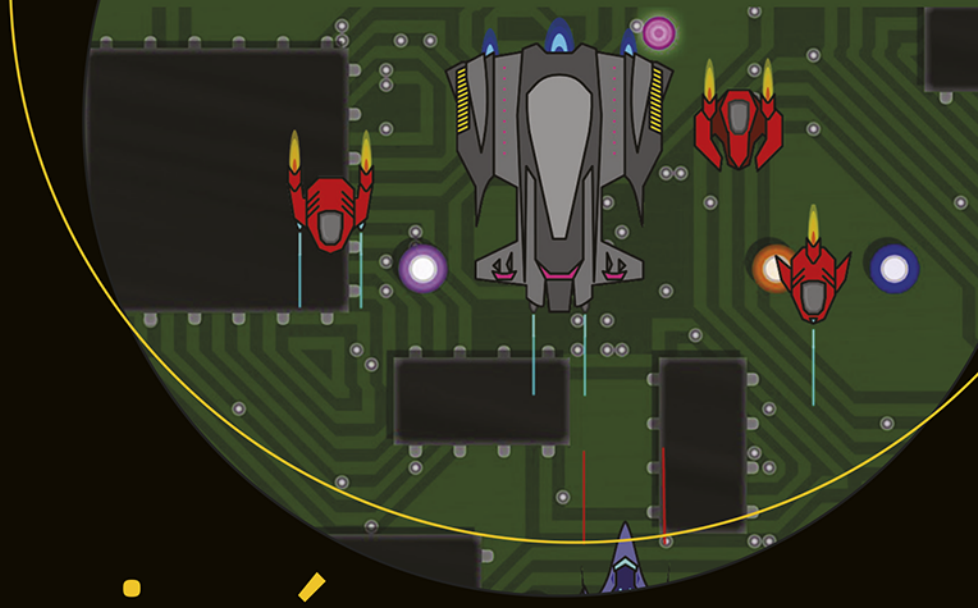


Jak pisać świetne gry 2D w Unity

Niezależne programowanie w języku C#

—

Jared Halpern

Helion

apress

Tytuł oryginału: Developing 2D Games with Unity: Independent Game Programming with C#

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-8235-0

First published in English under the title Developing 2D Games with Unity: Independent Game Programming with C# by Jared Halpern, edition: 1

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/japg2d.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/japg2d
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 3.

Podstawy

Po zapoznaniu się z edytorem Unity pora zacząć tworzyć grę. W tym rozdziale dowiesz się, jak konstruuje się jej obiekty i pisze kod. Poznasz wzorce programistyczne stosowane w programie Unity oraz kilka wysokopoziomowych zasad tworzenia gier. Nauczysz się też sterować postacią gracza i odtwarzać animacje.

Obiekt GameObject

Gra tworzona za pomocą programu Unity składa się ze scen, a każda scena jest zbudowana z obiektów *GameObject*. Wszystko, z czym będziesz miał do czynienia podczas tworzenia gry, tj. skrypty, zderzacze i innego typu elementy, są obiektami *GameObject*. Możesz sobie wyobrazić, że taki obiekt składa się z wielu części, z których każda implementuje określoną funkcjonalność. Jak się dowiedziałeś w rozdziale 2., obiekt może zawierać również inne obiekty *GameObject* i tworzyć z nimi relacje typu „rodzic-dziecko".

Teraz utworzysz swój pierwszy obiekt *GameObject*, a potem dowiesz się, dlaczego stanowi on fundamentalny element konstrukcyjny gry w programie Unity.

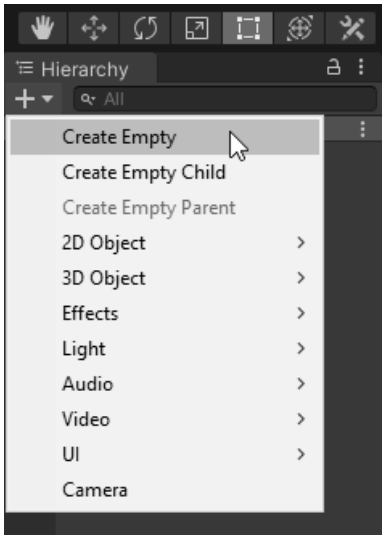
Kliknij symbol znaku dodawania znajdujący się w lewym górnym rogu panelu *Hierarchy* (hierarchia) i wybierz polecenie *Create Empty* (utwórz pusty obiekt), jak na rysunku 3.1. W panelu pojawi się nowy obiekt.

Obiekt *GameObject* można tworzyć również na kilka innych sposobów, m.in. klikając prawym przyciskiem myszy w dowolnym miejscu panelu *Hierarchy* i wybierając z podręcznego menu polecenie *Create Empty*.

Kliknij prawym przyciskiem myszy nowo utworzony obiekt, wybierz polecenie *Rename* (zmień nazwę) i wpisz nazwę **`PlayerObject`**. Ten obiekt będzie zawierał całą logikę obsługi postaci odważnego gracza w Twojej grze RPG.

Utwórz drugi obiekt i nazwij go **`EnemyObject`**. Ten obiekt będzie zawierał logikę przeciwnika, z którym będzie mierzył się gracz.

Ucząc się tworzyć grę za pomocą programu Unity, poznasz kilka informatycznych pojęć, które ułatwią Ci pracę i dzięki którym staniesz się lepszym programistą.



Rysunek 3.1. *Jeden ze sposobów tworzenia obiektu GameObject w panelu Hierarchy*

Wzorzec projektowy Jednostka-Komponent

W programowaniu istnieje zasada **separacji zakresów** (ang. *separation of concerns*), zgodnie z którą program należy dzielić na moduły realizujące określone funkcjonalności. Każdy moduł jest odpowiedzialny za określony zakres funkcjonalności, całkowicie oddzielony od innych zakresów. W dziedzinie tworzenia gier zasada ta jest dość luźno interpretowana, ponieważ zakresy mogą być bardzo szerokie i obejmować na przykład wyświetlanie grafiki na ekranie, jak również bardziej specjalistyczne, związane na przykład z wyliczaniem wspólnej powierzchni nakładających się na siebie trójkątów na płaszczyźnie.

Jednym z podstawowych celów separowania zakresów podczas tworzenia oprogramowania jest zapobieganie niepotrzebnemu powielaniu kodu i implementowaniu podobnych do siebie funkcjonalności. Na przykład kod wyświetlający obraz na ekranie powinien być tylko jeden. Gra może się składać z dziesiątków lub setek scen wymagających wyświetlania grafiki, ale programista powinien odpowiedni kod napisać tylko raz i wielokrotnie wykorzystywać go wszędzie tam, gdzie jest potrzebny.

Program Unity jest przystosowany do zasady separacji zakresów. Gry projektuje się wykorzystując bardzo popularny wzorzec **Jednostka-Komponent** (ang. *entity-component*), preferujący komponowanie funkcjonalności zamiast ich dziedziczenia. Zgodnie z nim należy tworzyć obiekty (jednostki) łącząc ze sobą instancje klas implementujących określone funkcjonalności. Jednostki uzyskują dostęp do funkcjonalności za pomocą instancji klas składowych. Umiejętnie stosując ten wzorzec tworzy się krótszy, bardziej zrozumiały i łatwiejszy w utrzymaniu kod.

Wzorzec Jednostka-Komponent różni się od popularnej techniki programistycznej, w której obiekty dziedziczą funkcjonalności klas nadrzędnych. Technika ta ma tę wadę, że skutkuje powstaniem rozbudowanych hierarchicznych struktur, w których wprowadzenie pojedynczej, małej zmiany w klasie nadrzędnej może wywoływać lawinowe, nieprzewidywane efekty w klasach podrzędnych.

W programie Unity jednostką we wzorcu Jednostka-Komponent jest obiekt *GameObject*. Każdy element składowy sceny jest takim obiektem. Jednak obiekt sam w sobie nie oferuje żadnej funkcjonalności. To jest zadanie komponentów. Dodając komponenty do obiektu, implementuje się żądane funkcjonalności. Dodawanie funkcjonalności do jednostki jest bardzo proste, ponieważ polega na dodawaniu komponentów do obiektu. Komponenty można traktować jako osobne moduły kodu wyspecjalizowane w określonych, oddzielonych od siebie zakresach.

Aby lepiej zrozumieć projektowanie gry z wykorzystaniem wzorca Jednostka-Komponent, przeanalizujmy poniższy diagram. Poszczególne kolumny zawierają komponenty realizujące określone funkcjonalności, a w wierszach są umieszczone jednostki.

	Wyświetlanie grafiki	Wykrywanie kolizji	Integracja cech fizycznych	Odtwarzanie dźwięku
Gracz	X	X	X	X
Przeciwnik	X	X	X	X
Dzida	X	X	X	
Drzewo	X	X		
Wieśniak	X	X		X

Jak widać, obiekty gracza i przeciwnika potrzebują wszystkich czterech komponentów. Dzida wymaga większości funkcjonalności, w szczególności cech fizycznych, gdy się nią rzuca. Nie potrzebuje jednak dźwięku. Drzewo nie wymaga cech fizycznych ani dźwięku, tylko wyświetlania grafiki i wykrywania kolizji, aby rzucane w nie przedmioty nie przelatywały na wskroś. Wieśniak wymaga grafiki i wykrywania kolizji, ale nie cech fizycznych, ponieważ tylko przemieszcza się po scenie. Dźwięk może mu być potrzebny do akustycznej interakcji z graczem.

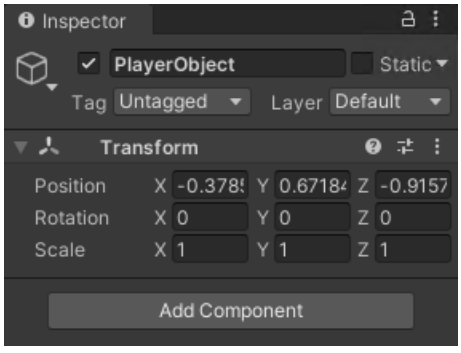
Wzorzec Jednostka-Komponent nie jest pozbawiony ograniczeń, ujawniających się szczególnie w dużych projektach po wielu latach stosowania. Prawdopodobnie w przyszłości zostanie zastąpiony wzorcem bardziej zorientowanym na dane.

Teraz wykorzystajmy nowo nabytą wiedzę.

Komponenty — bloki konstrukcyjne

Zaznacz w panelu *Hierarchy* obiekt *PlayerObject* i zwróć uwagę na wartości, które pojawią się w panelu *Inspector* (inspektor), pokazanym na rysunku 3.2.

Komponentem wspólnym dla wszystkich obiektów *GameObject* jest *Transform* (przekształcenie), który określa położenie, orientację i skalę obiektu umieszczonego na scenie. Tego komponentu będziesz używał do przemieszczania postaci gracza.



Rysunek 3.2. *Komponent Transform*

Duszki

Jeżeli jesteś początkującym twórcą gier, możesz zapytać, co to jest **duszek**? W kontekście gry komputerowej, jest to po prostu dwuwymiarowy obraz. Jeżeli grałeś w *Super Mario Brothers* na konsoli Nintendo (rysunek 3.3), w *Stardew Valley* (rysunek 3.4), *Celeste*, *Thimbleweed Park* lub *Terraria*, miałeś do czynienia z duszkami.

Rysunek 3.3. *Duszek Mario, bohaterski hydraulik z gry Super Mario Brothers (Nintendo)*

Rysunek 3.4. *Kurczaki, kaczki, strach na wróble, warzywa, drzewa i wszystkie inne obrazki w grze Stardew Valley są duszkami*

Efekty animacyjne w dwuwymiarowych grach uzyskuje się za pomocą technik analogicznych do stosowanych w kreskówkach. Podobnie jak komórki (ramki) w filmie rysunkowym, tak i duszki trzeba najpierw narysować i zapisać na dysku. Szybkie wyświetlanie sekwencji duszków wywołuje wrażenie ruchu, na przykład marszu, walki, skoków lub nieuchronnej śmierci.

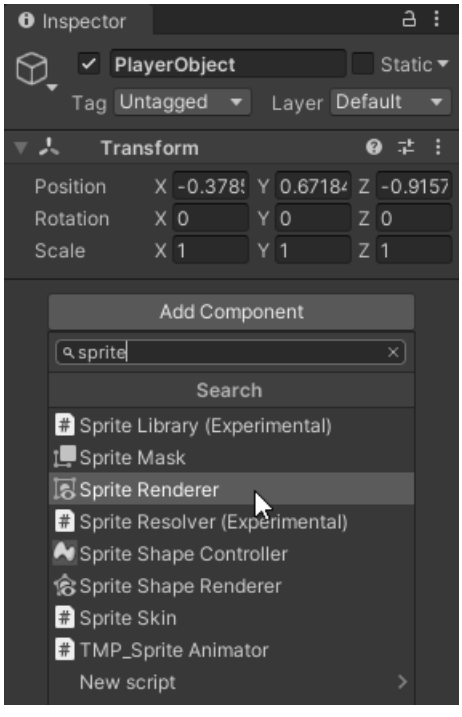
Aby na ekranie pojawiła się postać gracza, trzeba wyświetlić jej obraz za pomocą komponentu *Sprite Renderer* (odtwarzacz duszka). Dodaj teraz ten komponent do obiektu *PlayerObject*. Możesz to zrobić na kilka sposobów, ale na początek użyj w panelu *Inspector* przycisku *Add Component* (dodaj komponent). Następnie w polu wyszukiwania wpisz **`sprite`** i wybierz pozycję *Sprite Renderer*, jak na rysunku 3.5. Inny sposób polega na użyciu polecenia menu *GameObject/2D Object/Sprites* dodającego obiekt zawierający już komponent *Sprite Renderer*.

W ten sam sposób dodaj komponent *Sprite Renderer* do obiektu *EnemyObject*.

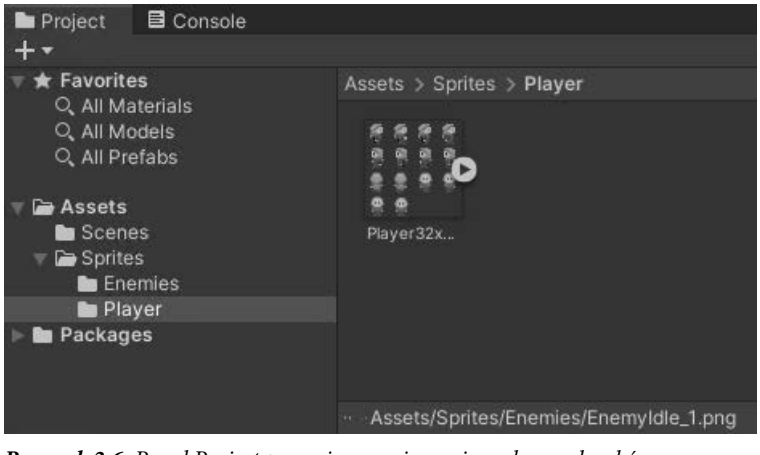
Dobrym nawykiem jest zapisywanie sceny, więc zrób to teraz. Naciśnij klawisze *Ctrl+S* (PC) lub *Cmd/⌘+S* (Mac), utwórz nowy folder o nazwie *Scenes* i zapisz scenę pod nazwą *LevelOne*. W tym folderze zapiszesz również inne sceny, które utworzysz.

Teraz w panelu *Project* utwórz podfolder *Sprites*. Jak się domyślasz, będziesz w nim zapisywał wszystkie duszki. Następnie utwórz podfoldery *Player* i *Enemies*. W eksploratorze plików otwórz folder, w którym umieściłeś rozpakowane pliki załączone do tej książki, przejdź do podfolderu *r03\Spritesheets*, zaznacz pliki *EnemyIdle_1.png*, *EnemyWalk_1.png* oraz *Player32x32.png* i przeciągnij je do panelu *Project* w programie Unity. Gdy je tam umieścisz, przenieś je z folderu *Sprites* do odpowiednich podfolderów. Panel *Project* powinien wyglądać jak na rysunku 3.6.

W panelu *Project* zaznacz arkusz duszków i zwróć uwagę na wartości, które pojawią się w panelu *Inspector*. Teraz zmienisz ustawienia importu zasobów i użyjesz edytora duszków do podzielenia arkusza na osobne duszki.



Rysunek 3.5. *Dodanie komponentu Renderer Component do obiektu PlayerObject*



Rysunek 3.6. *Panel Project po umieszczeniu w nim arkusza duszków; duszki przeciwnika znajdują się w podfolderze Enemies*

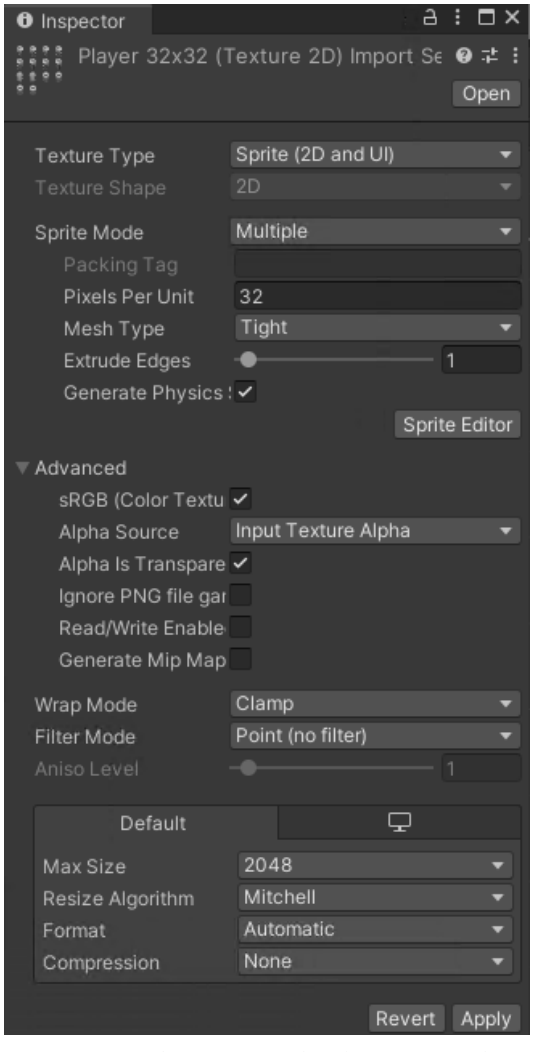
Z listy *Texture Type* (typ tekstury) wybierz pozycję *Sprite (2D and UI)*, a z listy *Sprite Mode* (tryb duszka) pozycję *Multiple* (wielokrotny). W ten sposób określisz, że arkusz zawiera wiele duszków.

W polu *Pixels Per Unit* (piksele na jednostkę) wpisz liczbę **32**. Tym ustawieniem zajmiemy się przy omawianiu kamer.

Z listy *Filter Mode* (tryb filtru) wybierz pozycję *Point (no filter)* (punkt, bez filtru). Dzięki temu tekstura duszka będzie w powiększeniu wyświetlana blokowo, idealnie w pikselowej grafice gry.

Na zakładce *Default* (domyślne) w dolnej części panelu wybierz z listy *Compression* (kompresja) pozycję *None* (brak).

Sprawdź, czy ustawienia w panelu *Inspector* są takie, jak pokazane na rysunku 3.7.

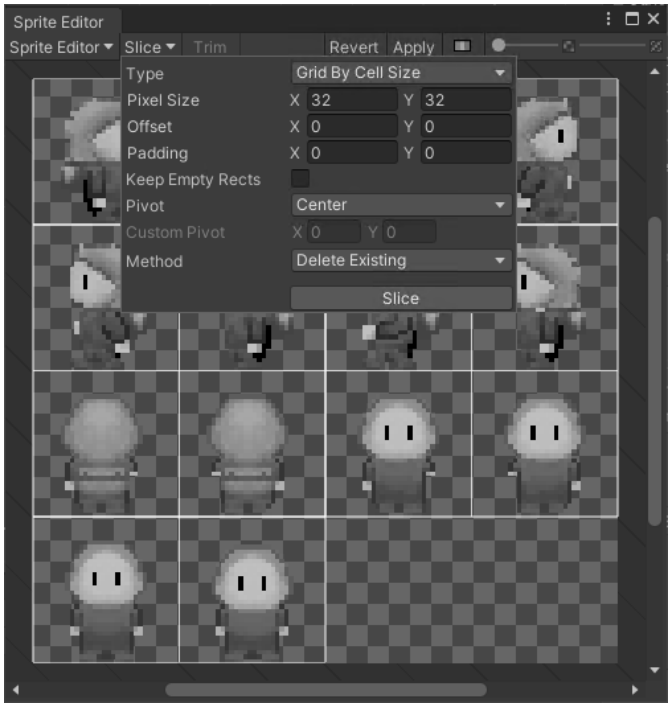


Rysunek 3.7. *Właściwości zasobu Player32x32 widoczne w panelu Inspector*

Kliknij przycisk *Apply*, aby zatwierdzić zmiany, a następnie przycisk *Sprite Editor*.

Teraz pora podzielić arkusz na osobne duszki. Za pomocą wbudowanego edytora duszków bardzo wygodnie pracuje się z arkuszami zawierającymi wiele duszków.
Można je m.in. dzielić na osobne duszki.

Rozwiń menu *Slice* (podziel) widoczne w lewym górnym rogu panelu, a następnie z listy *Type* (typ) wybierz pozycję *Grid By Cell Size* (siatka zgodna z wielkością komórki), aby określić wymiary fragmentów. W wierszu *Pixel Size* (wielkość w pikselach) wpisz w polach *X* i *Y* wartości **32** i kliknij przycisk *Slice*. Powinna pojawić się siatka, jak na rysunku 3.8, rozdzielająca poszczególne duszki.



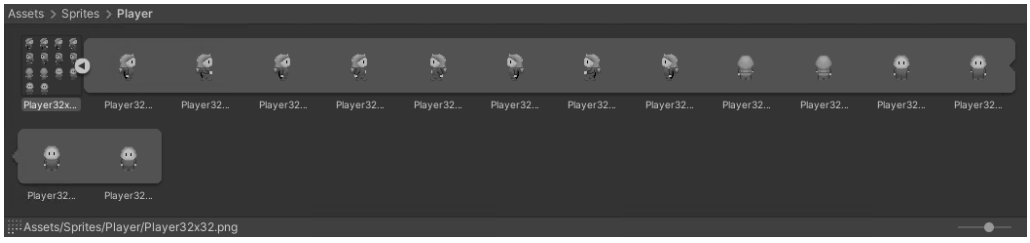
Rysunek 3.8. *Podział zaimportowanego arkusza na duszki o zadanej wielkości*

Kliknij przycisk *Apply*, aby zatwierdzić podział, a następnie zamknij edytor.

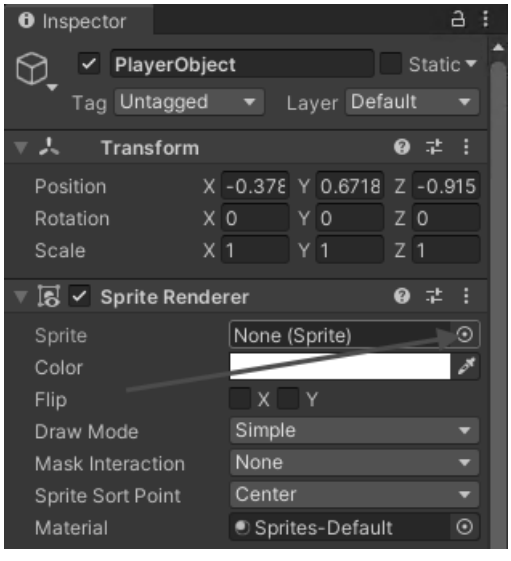
W tym przypadku wprowadziłeś dokładne wymiary duszków, ponieważ znałeś je zawczasu. Gdy będziesz tworzył inne gry, będziesz korzystał z arkuszy zawierających duszki o różnych wymiarach i dobranie właściwych będzie wymagało nieco więcej zachodu. Edytor oferuje funkcję automatycznego wykrywania wymiarów duszków w zaimportowanym arkuszu. Aby jej użyć, należy rozwinąć menu *Slice* i z listy *Type* wybrać pozycję *Automatic* (automatyczny). Ta technika może dawać różne rezultaty w zależności od arkusza, ale stanowi dobry punkt wyjścia.

Czemu ma służyć dzielenie arkusza duszków? Kliknij mały symbol trójkąta umieszczony na prawej krawędzi arkusza. Pojawią się duszki wyodrębnione z arkusza, jak na rysunku 3.9. Wykorzystasz je do przygotowania animacji.

Zaprzęgnij teraz duszki do pracy. Zaznacz obiekt *PlayerObject*. W panelu *Inspector* po prawej stronie listy *Sprite* widoczne jest niewielkie kółko, jak na rysunku 3.10.
Gdy je klikniesz, pojawi się okno umożliwiające wybranie duszka, pokazane na rysunku 3.11.



Rysunek 3.9. *Duszki wyodrębnione z arkusza*



Rysunek 3.10. *Przycisk otwierający okno umożliwiające wybranie duszka*

Kliknij dwukrotnie jednego z duszków, który zastąpi obiekt *PlayerObject* w widoku sceny.

Po zdefiniowaniu duszka gracza, zaimportuj arkusz duszków przeciwnika. Zaznacz arkusz *EnemyIdle_1* i w panelu *Inspector* określ ustawienia, takie same jak dla obiektu *PlayerObject*, tj.:

Texture Type (typ tekstury): *Sprite (2D and UI)* (duszek (2D i interfejs użytkownika))

Sprite Mode (tryb duszka): *Multiple* (wielokrotny)

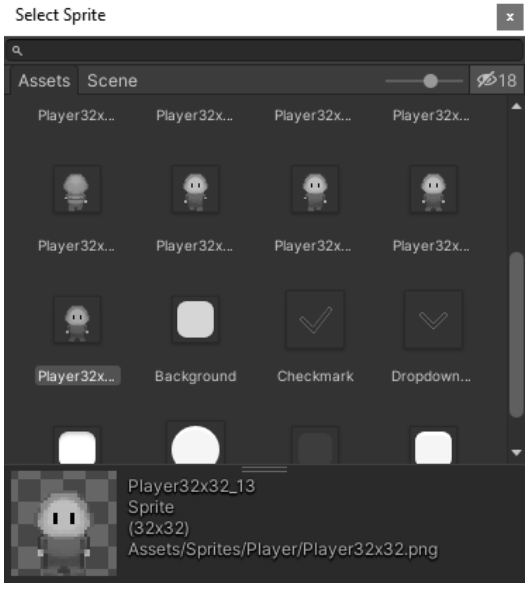
Pixels Per Unit (piksele na jednostkę): *32*

Filter Mode (tryb filtru): *Point (no filter)* (punkt, bez filtru)

Compression (kompresja): *None* (brak)

Kliknij przycisk *Apply*.

Kliknij przycisk *Sprite Editor* i podziel arkusz na duszki o wymiarach 32×32 piksele. Upewnij się, że pojawiła się właściwa biała siatka, kliknij przycisk *Apply* i zamknij edytor. W ten sam sposób podziel arkusz *EnemyWalk_1*.



Rysunek 3.11. *Wybierz duszka, który będzie reprezentował gracza w widoku sceny, gdy gra nie będzie uruchomiona*

Animacje

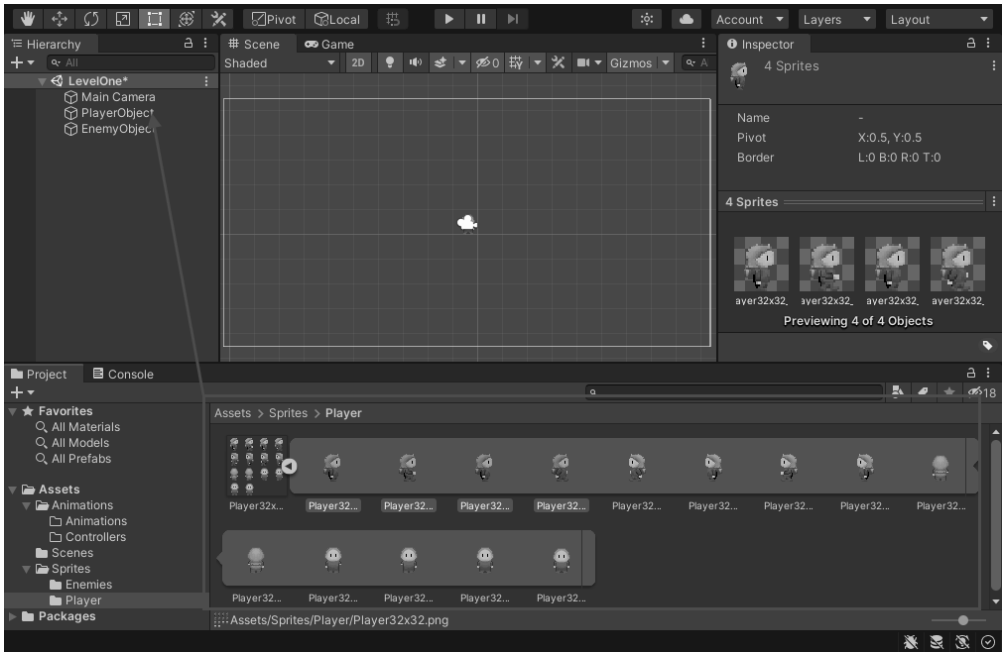
Teraz utwórz folder, w którym będziesz umieszczał tworzone animacje. Wiesz już, jak to zrobić, prawda? W panelu *Project* kliknij prawym przyciskiem myszy folder *Assets* i w podręcznym menu wybierz polecenie *Create/Folder*. Możesz również kliknąć symbol znaku dodawania widoczny w lewym górnym rogu panelu. Nadaj folderowi nazwę *Animations*. Następnie zaznacz go i utwórz dwa podfoldery o nazwach *Animations* (animacje) i *Controllers* (kontrolery).

W panelu *Project* rozwiń arkusz duszków *Player32x32*, klikając symbol trójkąta znajdujący się na prawej krawędzi. Zaznacz pierwszego duszka, który powinien być zwrócony w prawą stronę (na wschód). Trzymając naciśnięty klawisz *Shift*, zaznacz trzy kolejne duszki i przeciągnij je do obiektu *PlayerObject*, jak na rysunku 3.12.

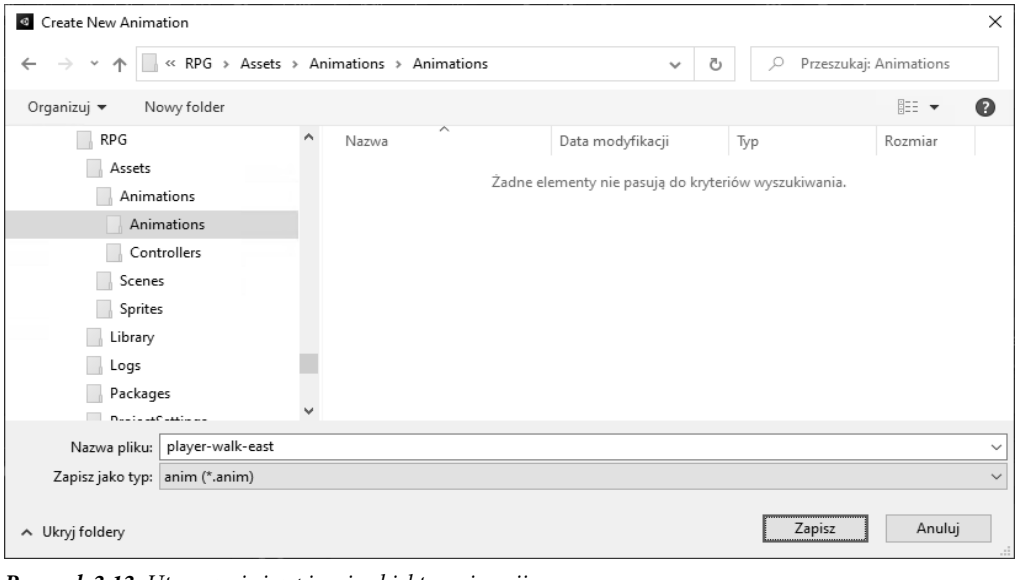
Pojawi się okno *Create New Animation* (utwórz nową animację), jak na rysunku 3.13. Przejdź do folderu *Animations*, który utworzyłeś wcześniej, i zapisz animację pod nazwą *player-walk-east*.

Zaznacz obiekt *PlayerObject* i zwróć uwagę, że w panelu *Inspector* pojawią się dwa nowe komponenty: *Sprite Renderer* i *Animator* (animator), jak na rysunku 3.14. Zadaniem pierwszego jest wyświetlanie duszka, natomiast drugiego odtwarzanie animacji za pomocą kontrolera.

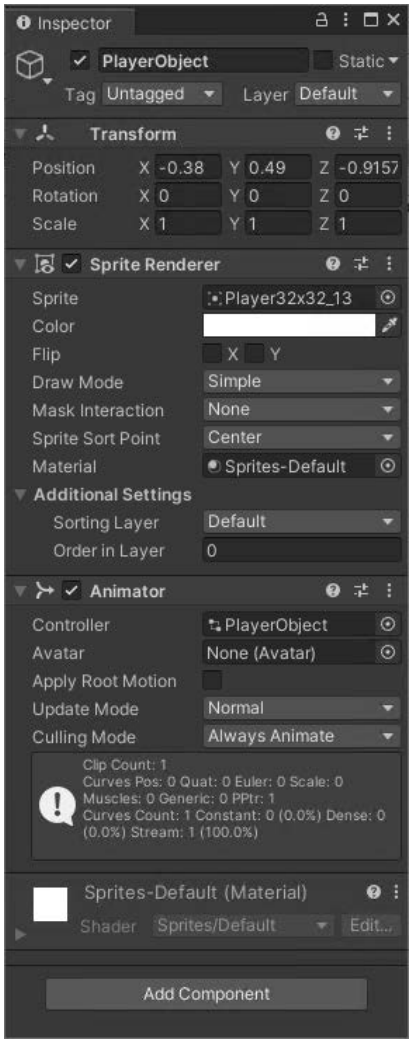
Edytor Unity jest inteligentny i „wie”, że dodając animację, będziesz chciał w jakiś sposób ją odtwarzać i kontrolować. Dlatego automatycznie utworzył komponent *Animator* i przypisał mu kontroler o nazwie *PlayerObject*. Komponent ten możesz też dodać ręcznie, klikając przycisk *Add Component* z panelu *Inspector* i wpisując w polu wyszukiwania słowo **`Animator`**.



Rysunek 3.12. *Tworzenie animacji poprzez przeciągnięcie duszków do obiektu PlayerObject*

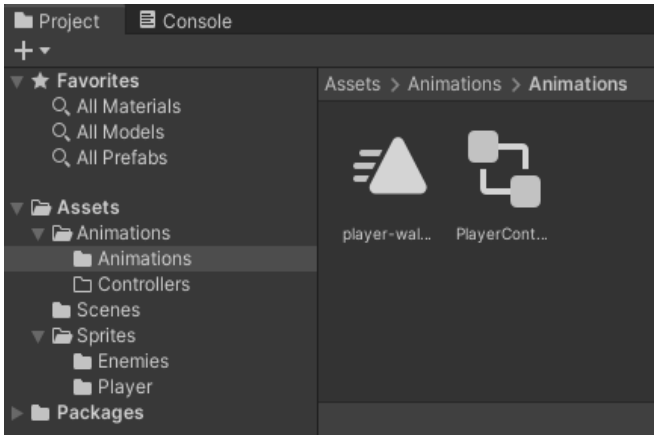


Rysunek 3.13. *Utworzenie i zapisanie obiektu animacji*



Rysunek 3.14. *Nowe, automatycznie dodane komponenty: Sprite Renderer i Animator*

W folderze, w którym zapisałeś animację *player-walk-east*, pojawił się kontroler animacji o domyślnej nazwie *PlayerObject*, jak na rysunku 3.15. Nazwa ta jest nieco myląca, ponieważ tak samo nazywa się obiekt *GameObject* reprezentujący gracza. Dlatego nadaj kontrolerowi inną, bardziej odpowiednią nazwę. W tym celu kliknij kontroler prawym przyciskiem myszy, wybierz z menu polecenie *Rename* i wpisz nazwę **`PlayerController`**. Następnie przeciągnij kontroler do utworzonego wcześniej folderu *Controllers*.

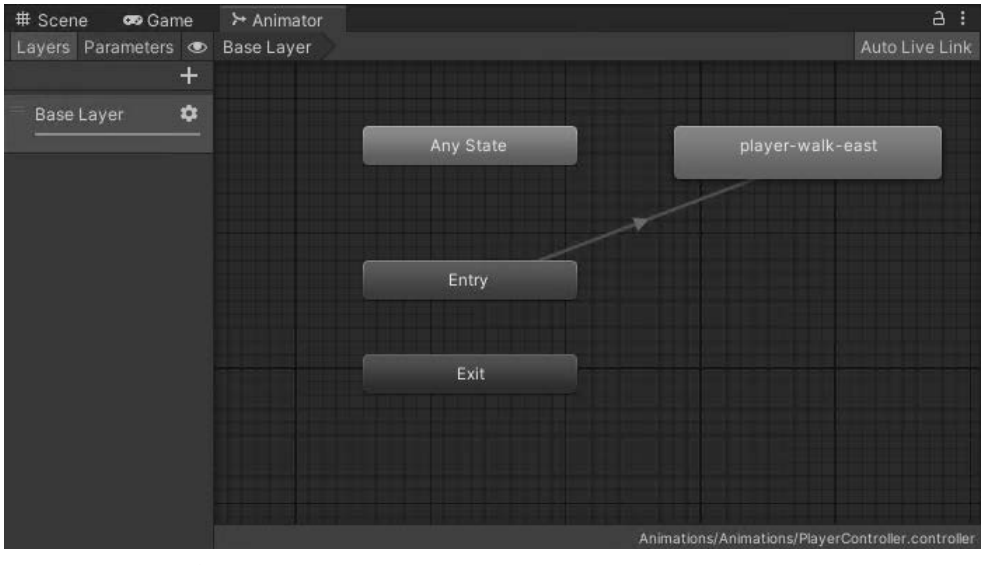


Rysunek 3.15. *Automatycznie utworzony kontroler animacji PlayerObject, umieszczony obok pierwszego obiektu animacji player-walk-east*

Kliknij dwukrotnie obiekt *PlayerController*, aby otworzyć panel *Animator*.

Maszyna stanów

Kontroler animacji definiuje zestaw reguł zwany **maszyną stanów**. Maszyna ta decyduje o tym, które klipy animacyjne mają być odtwarzane w zależności od stanu obiektu gracza. Przykładami takich stanów może być marsz, atak, bezczynność, jedzenie i śmierć. Stany podzielimy dodatkowo według kierunków ruchu, ponieważ obiekt gracza, znajdując się w poszczególnych stanach, może przemieszczać się na północ, południe, wchód lub zachód. Panel *Animator* zawiera wizualną reprezentację stanów, podobną do diagramu przepływów, pokazaną na rysunku 3.16.



Rysunek 3.16. *Panel Animator*

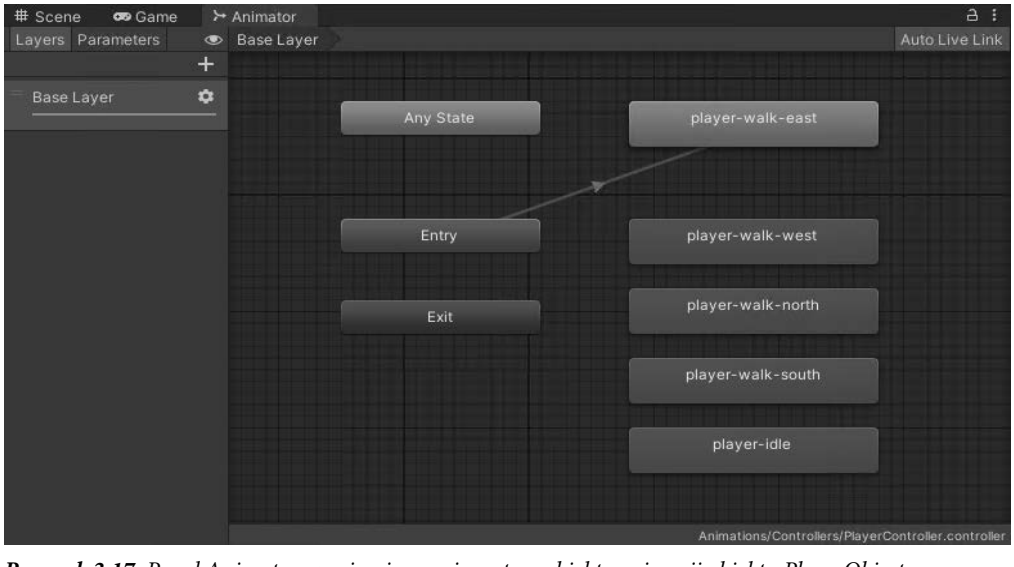
Kontroler można traktować jak *mózg* sterujący animacją. Każdy stan maszyny jest reprezentowany jako obiekt, który zawiera klip animacyjny odtwarzany w danym stanie. Kontroler definiuje również szczegóły przechodzenia obiektu pomiędzy stanami.

Jak widać w panelu *Animator*, w naszym przykładzie kontroler składa się z następujących stanów: *Any State* (dowolny stan), *Entry* (wejście), *Exit* (wyjście) i dodanego przed chwilą *player-walk-east*. Stan *Any State* jest wykorzystywany podczas przechodzenia pomiędzy innymi stanami.

Jeżeli stan *Exit* nie jest widoczny, przesuń zawartość panelu. W tym celu naciśnij i przytrzymaj klawisz *Alt* (PC) lub *Option* (Mac) i przesuń tło panelu. Możesz również pomniejszyć widok, kręcąc kółkiem myszy. W każdej chwili możesz dowolnie przesuwać obiekty animacyjne i rozmieszczać je w najbardziej czytelny dla siebie sposób.

Dodajmy pozostałe obiekty animacji. Wróć do folderu *Player*, zaznacz cztery kolejne duszki i tak samo jak poprzednio przeciągnij je do obiektu *PlayerObject*. W wyświetlonym oknie przejdź do folderu *Animations/Animations* i zapisz animację pod nazwą *player-walk-west*. Nowa animacja pojawi się w panelu *Animator*. W ten sam sposób utwórz kolejne animacje pozostałych duszków. Zwróć uwagę, że animacje duszków idących na północ i południe oraz stojącego bezczynnie składają się tylko z dwóch ramek. Animacje zapisz pod nazwami *player-walk-north*, *player-walk-south* i *player-idle*.

W tym momencie panel *Animator*, zawierający wszystkie obiekty animacji, powinien wyglądać jak na rysunku 3.17. Pięć obiektów umieszczonych po prawej stronie reprezentuje różne stany duszka, każdy z nich zawiera ponadto referencje do klipów animacyjnych.



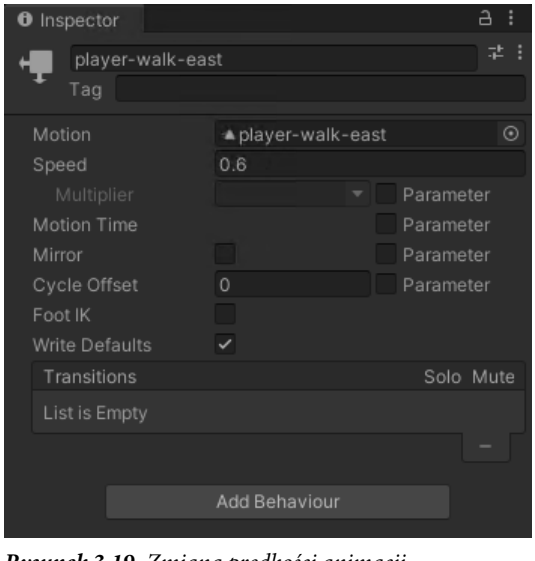
Rysunek 3.17. *Panel Animator, zawierający m.in. cztery obiekty animacji obiektu PlayerObject*

Zrobiliśmy wszystko, co trzeba, ale na ekranie wciąż nie widać żadnych animacji. Pozostał jeszcze jeden krok. W panelu *Hierarchy* zaznacz obiekt *Main Camera* (główna kamera), a w panelu *Inspector* wpisz w polu *Size* (wielkość) wartość **1**. Jest to tymczasowa zmiana, niezbędna do odtworzenia animacji duszka. Kamerami zajmiemy się w dalszej części książki.

Kliknij przycisk odtwarzania na pasku narzędzi. Jeżeli wszystko wykonałeś poprawnie, powinieneś zobaczyć nieustraszonego duszka gracza, biegnącego szaleńczo w miejscu, jak na rysunku 3.18.

Rysunek 3.18. *Skosztuj pikselowych owoców swej pracy*

Spowolnij nieco pędzącego duszka. W tym celu w panelu *Animator* zaznacz obiekt *player-walk-east* i w panelu *Inspector* wpisz w polu *Speed* (prędkość) wartość **`0.6`**, jak na rysunku 3.19.



Rysunek 3.19. *Zmiana prędkości animacji*

Ponownie kliknij przycisk odtwarzania i zauważ, że teraz duszek porusza się w rozsądnym tempie. Możesz ustawić inną prędkość, która według Ciebie będzie wyglądała naturalnie.

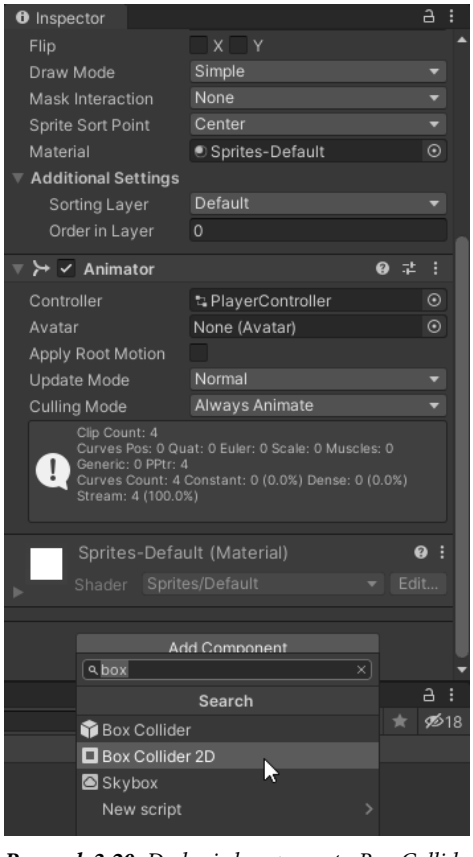
Zatrzymaj animację, klikając ponownie przycisk odtwarzania.

Teraz utwórz animacje wykorzystując arkusze *EnemyIdle_1* i *EnemyWalk_1*. Każdy z nich składa się z pięciu duszków. Zapisz animacje w podfolderze *Animations/Animations*, odpowiednio pod nazwami *enemy-idle-1* i *enemy-walk-1*. Zmień nazwę kontrolera z *EnemyObject* na *EnemyController* i przenieś go do podfolderu *Animations/Controllers*.

Zderzacze

Teraz poznaj *zderzacze*. Te komponenty są dodawane do obiektów *GameObject* i wykorzystywane przez silnik Unity do wykrywania kolizji dwóch obiektów. Zderzacz może mieć dowolny kształt, ale zazwyczaj pokrywa się mniej więcej z obrysem obiektu, z którym jest skojarzony. Ze względów obliczeniowych nie jest zalecane nadawanie zderzaczowi kształtu ściśle dopasowanego do obrysu. Zazwyczaj nie jest to nawet potrzebne, ponieważ przybliżony kształt zupełnie wystarczy do wykrywania kolizji, a różnica dla gracza jest niezauważalna. Ponadto zderzacz o przybliżonym kształcie, zwany „prymitywnym zderzaczem", w mniejszym stopniu obciąża procesor. Program Unity oferuje dwa dwuwymiarowe prymitywne zderzacze: *Box Collider 2D* i *Circle Collider 2D*.

Zaznacz obiekt *PlayerObject*, a następnie w panelu *Inspector* kliknij przycisk *Add Component*. W polu wyszukiwania wpisz frazę **box** i wybierz pozycję *Box Collider 2D*, jak na rysunku 3.20.



Rysunek 3.20. *Dodanie komponentu Box Collider 2D do obiektu PlayerObject*

Ponieważ chcemy wykrywać kolizje gracza z przeciwnikiem, w ten sam sposób dodaj komponent *Box Collider 2D* do obiektu *EnemyObject*.

Komponent Rigidbody

Komponent *Rigidbody* jest wykorzystywany przez silnik Unity do interakcji z obiektem *GameObject*. Za pośrednictwem tego komponentu wywierana jest np. siła grawitacji na dany obiekt. Siły można też wywierać za pomocą skryptów. Załóżmy, że mamy obiekt o nazwie *Samochód*, który zawiera obiekt *Rigidbody*. Aby wprawić samochód w ruch w określonym kierunku, trzeba przyłożyć do niego siłę, klikając np. przyciski *Gaz* i *Turbo*.

Zaznacz obiekt *PlayerObject*, w panelu *Inspector* kliknij przycisk *Add Component*, w polu wyszukiwania wpisz frazę **`rigid`** i dodaj komponent *Rigidbody 2D*. Z listy *Body Type* (typ ciała) wybierz pozycję *Dynamic* (dynamiczne). Dzięki temu obiekt będzie wchodził w interakcje i kolidował z innymi obiektami. W polach *Linear Drag* (przeciąganie liniowe), *Angular Drag* (przeciąganie kątowe) i *Gravity Scale* (skala grawitacji) wpisz wartości **0**, a w polu *Set Mass* (ustawienie masy) wartość **1**.

Drugą pozycją listy *Body Type* jest ***Kinematic*** (kinetyczne). Tego rodzaju komponent powoduje, że na obiekt nie oddziałują zewnętrzne siły, na przykład grawitacji. Taki obiekt można przemieszczać, ale wyłącznie za pomocą komponentów *Transform* i *Script*. Jest to inne podejście niż opisane wyżej przykładanie siły. Na liście *Body Type* jest jeszcze trzecia pozycja: ***Static*** (statyczne) oznaczająca, że obiekt nie przemieszcza się w ogóle.

Zaznacz obiekt *EnemyObject* i do niego również dodaj komponent *Rigidbody 2D* typu *Dynamic*.

Po dodaniu komponentów *Rigidbody 2D* do obiektów gracza i przeciwnika zaczyna na nie oddziaływać siła grawitacji. Ponieważ w tej grze jest stosowana perspektywa góra-dół, wyłącz grawitację, aby duszek gracza nie uciekł poza krawędzie ekranu. W tym celu kliknij polecenie menu *Edit/Project Settings/Physics 2D* (edycja/ustawienia projektu/fizyczne 2D) i w wierszu *Gravity* (grawitacja) w polu *Y* wpisz wartość **0**.

Znaczniki i warstwy

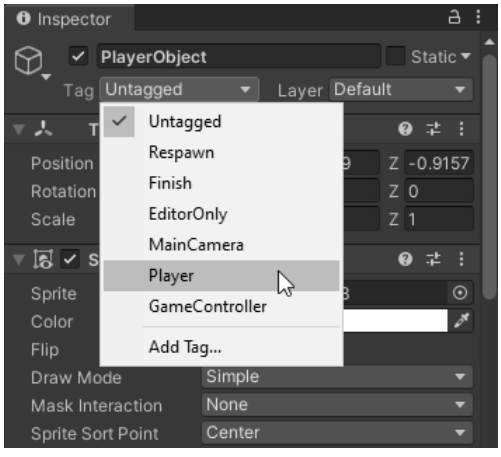
Znaczniki

Znaczniki przypisuje się obiektom *GameObject*, aby można się było łatwo do nich odwoływać i porównywać ze sobą po uruchomieniu gry.

Zaznacz obiekt *PlayerObject*, a następnie w górnej części panelu *Inspector* wybierz z listy *Tag* (znacznik) pozycję *Player*, jak na rysunku 3.21.

Znacznik *Player* jest domyślnie dostępny w każdej scenie. Można również definiować własne znaczniki, jeżeli zajdzie taka potrzeba.

Zdefiniuj nowy znacznik *Enemy* i przypisz go obiektowi *EnemyObject*. W miarę rozwijania gry będziesz przypisywał znaczniki również innym obiektom.

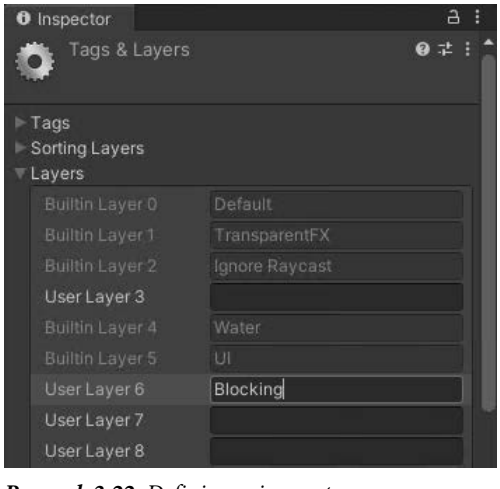


Rysunek 3.21. *Przypisanie znacznika Player do obiektu PlayerObject*

Warstwy

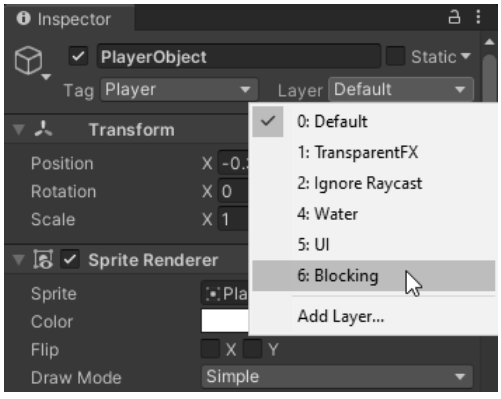
Warstwy wykorzystuje się do definiowania kolekcji obiektów *GameObject*. Kolekcje stosuje się podczas wykrywania kolizji do określania warstw, które muszą „wiedzieć" o swoim istnieniu i komunikować się ze sobą. W skrypcie implementuje się operacje wykonywane w przypadku kolizji obiektów.

W naszej grze będzie potrzebna nowa warstwa użytkownika (*User Layer*) o nazwie *Blocking* (blokująca). Aby ją utworzyć, wybierz w panelu *Inspector* z listy *Layer* pozycję *Add Layer* (dodaj warstwę) i w polu *User Layer 6* wpisz nazwę **`Blocking`**, jak na rysunku 3.22.



Rysunek 3.22. *Definiowanie warstw*

Następnie zaznacz obiekt *PlayerObject*, w panelu *Inspector* ponownie kliknij listę *Layer* i wybierz utworzoną przed chwilą warstwę *Blocking*, jak na rysunku 3.23. Tę samą operację wykonaj na obiekcie *EnemyObject*.



Rysunek 3.23. *Wybieranie warstwy Blocking*

Później zdefiniujesz warunki uniemożliwiające niektórym obiektom w warstwie *Blocking* przechodzenie przez inne obiekty. Na przykład obiekt gracza będzie znajdował się w tej samej warstwie, co ściany, drzewa i przeciwnicy. Przeciwnicy nie mogą przechodzić przez gracza, a gracz nie może przechodzić przez ściany, drzewa i przeciwników.

Warstwy sortujące

Przyjrzyjmy się teraz innego rodzaju warstwie — *sortującej*. Różni się ona od zwykłej warstwy tym, że można w niej określić kolejność, w jakiej silnik Unity ma rysować dwuwymiarowe duszki na ekranie. Ponieważ warstwa sortująca jest związana z grafiką, w ustawieniach komponentu *Sprite Renderer* (odtwarzacz duszka) znajduje się rozwijana lista *Sorting Layer* (warstwa sortująca).

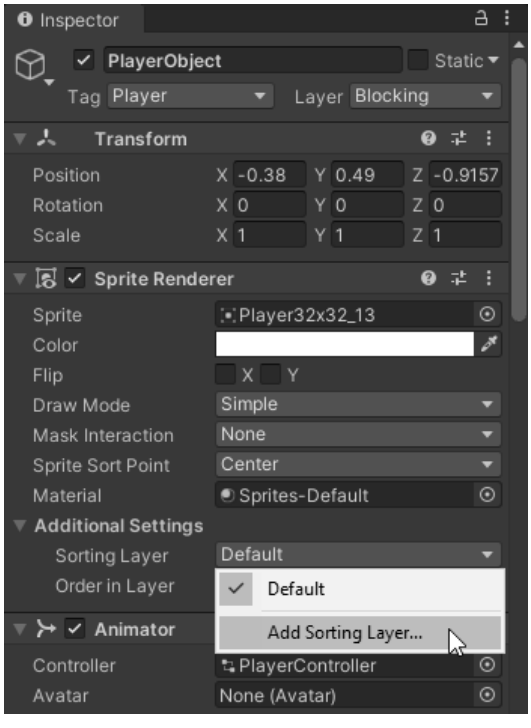
Aby zrozumieć, na czym polega kolejność wyświetlania duszków, spójrz na rysunek 3.24, przedstawiający grę *Thimbleweed Park* typu „wskaż i kliknij”. Rysunek przedstawia dwie postacie graczy w pomieszczeniu, w którym znajdują się różne meble, m.in. szafka na dokumenty i stół. Postać agentki Ray jest umieszczona przed szafką. Ten efekt został osiągnięty poprzez wyświetlenie najpierw szafki, a następnie duszka agentki.

Rysunek 3.24. *Widok gry Thimbleweed Park przedstawiający postacie znajdujące się na tle obiektów*

W powyższej grze nie jest wykorzystywany silnik Unity, tylko własny silnik twórcy. Jednak w obu przypadkach potrzebny jest jakiś algorytm określający kolejność wyświetlania pikseli.

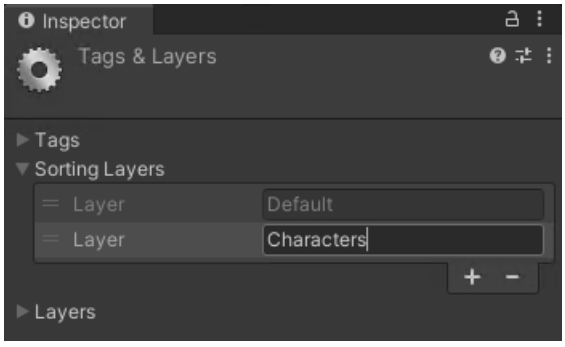
W naszej grze jest stosowana tzw. **perspektywa ortogonalna**, czyli góra-dół. Więcej szczegółów poznasz przy opisie kamer, natomiast teraz wystarczy, abyś wiedział, że trzeba silnikowi Unity w jakiś sposób wskazać, że musi najpierw wyświetlić piksele ziemi, a na ich tle postacie gracza i przeciwników, aby wywołać wrażenie, że chodzą po ziemi.

Teraz dodaj warstwę sortującą o nazwie *Characters* (postacie), którą wykorzystasz do wyświetlenia duszków gracza i przeciwników. Kliknij obiekt *PlayerObject* i w panelu *Inspector* w sekcji komponentu *Sprite Renderer* wybierz z listy *Sorting Layer* pozycję *Add Sorting Layer*, jak na rysunku 3.25. Warstwa, którą utworzysz, będzie dostępna w całej grze, mimo że zdefiniujesz ją w ustawieniach obiektu *PlayerObject*.

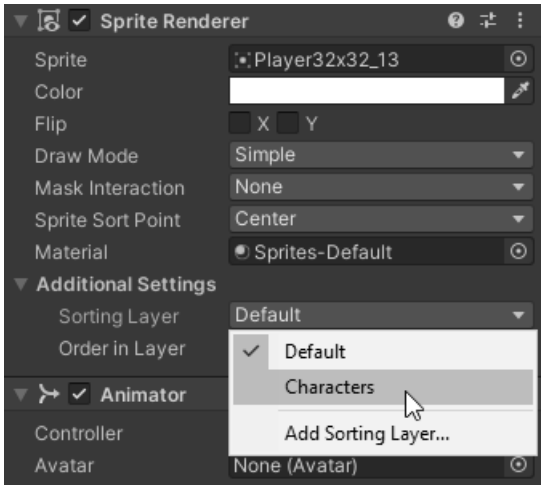


Rysunek 3.25. *Tworzenie warstwy sortującej*

W panelu, który się pojawi, utwórz nową warstwę o nazwie *Characters*, jak na rysunku 3.26. Następnie ponownie kliknij obiekt *PlayerObject* i w panelu *Inspector* wybierz z listy *Sorting Layer* nowo utworzoną warstwę *Characters*, jak na rysunku 3.27.



Rysunek 3.26. *Tworzenie warstwy sortującej o nazwie Characters*



Rysunek 3.27. *Przypisanie warstwy sortującej Characters obiektowi PlayerObject*

Zaznacz obiekt *EnemyObject* i przypisz mu warstwę *Characters*, ponieważ duszki przeciwników również muszą być wyświetlane na tle innych obiektów, na przykład ziemi.

Prefabrykaty

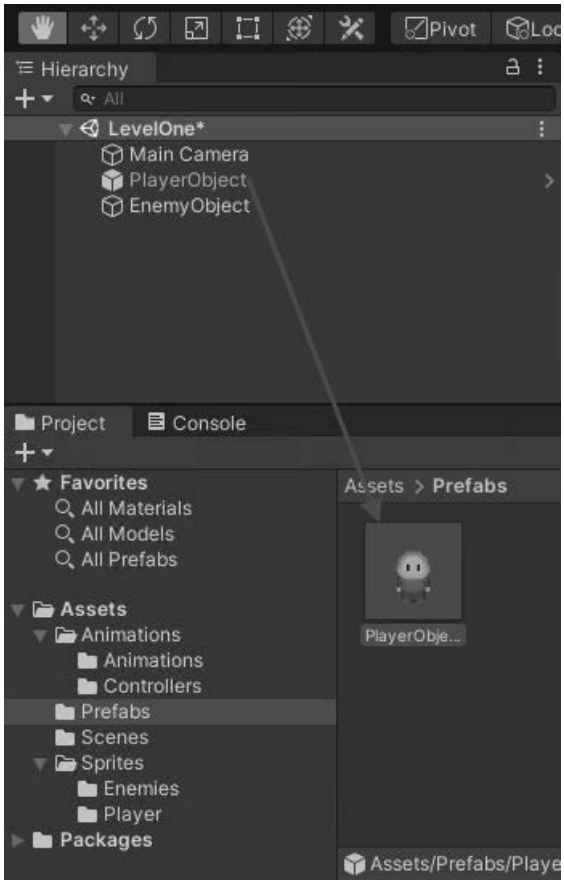
W edytorze Unity tworzy się obiekty *GameObject* złożone z komponentów, a z obiektów buduje się tzw. **prefabrykaty**. Prefabrykaty można traktować jako szablony, za pomocą których tworzy się instancje, czyli kopie gotowych obiektów. Prefabrykat ma tę cenną cechę, że edytując go można za jednym razem zmienić wszystkie obiekty. Oprócz tego można również zmieniać pojedyncze, wybrane obiekty, a pozostałe pozostawiać takie same, jak oryginał.

Załóżmy, że mamy scenę przedstawiającą gracza w karczmie, w której znajduje się wiele różnych obiektów, na przykład krzesła, stoły i kufle piwa. Gdybyś utworzył poszczególne obiekty *GameObject* osobno, musiałbyś edytować każdy z nich niezależnie od innych. W takim wypadku, chcąc zmienić zaledwie jedną właściwość każdego stołu, na przykład kolor drewna z ciemnego na jasny, musiałbyś kolejno zaznaczać poszczególne obiekty stołów i w każdym

zmieniać odpowiednią właściwość. Natomiast gdyby stół był instancją prefabrykatu, wystarczyłoby zmienić jego właściwość tylko raz, a następnie kliknąć przycisk aplikujący wprowadzoną zmianę we wszystkich instancjach tego prefabrykatu.

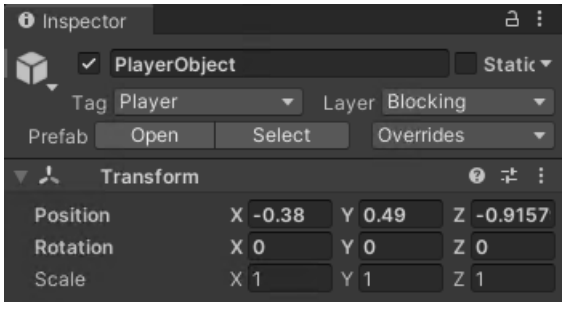
Prefabrykaty będziesz nieustannie wykorzystywał w całym procesie tworzenia gry.

Tworzenie prefabrykatu jest bardzo proste. Najpierw w panelu *Project* utwórz w folderze *Assets* podfolder *Prefabs* (prefabrykaty). Następnie w panelu *Hierarchy* zaznacz obiekt *PlayerObject* i przeciągnij go do folderu *Prefabs*. Uzyskasz widok jak na rysunku 3.28.



Rysunek 3.28. *Tworzenie prefabrykatu poprzez przeciągnięcie obiektu GameObject do folderu Prefabs*

Zwróć uwagę, że w panelu *Hierarchy* napis *PlayerObject* zmienił kolor na niebieski, informujący, że obiekt ten bazuje na prefabrykacie. Oznacza to również, że gdy wprowadzisz zmiany w prefabrykacie i będziesz chciał je zastosować we wszystkich jego instalacjach, będziesz musiał w panelu *Project* zaznaczyć obiekt *GameObject*, a następnie w panelu *Inspector* kliknąć przycisk *Overrides* (nadpisanie), pokazany na rysunku 3.29.



Rysunek 3.29. *Aby zastosować zmiany wprowadzone w prefabrykacie we wszystkich jego instancjach, należy kliknąć przycisk Overrides*

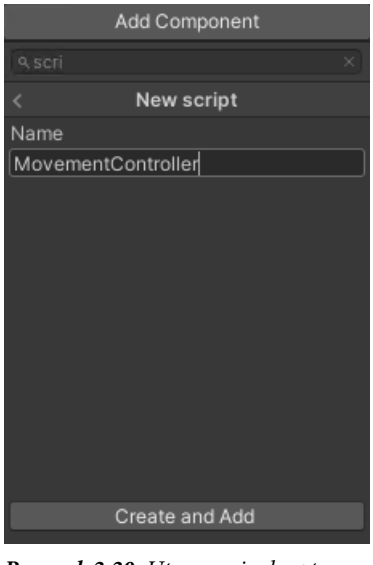
Teraz możesz bezpiecznie usunąć obiekt *PlayerObject* z panelu *Hierarchy*, ponieważ masz prefabrykat *PlayerObject*, którego możesz w każdej chwili użyć do odtworzenia obiektu *PlayerObject*. Aby zmienić wszystkie instancje tego prefabrykatu, po prostu przeciągnij go do panelu *Hierarchy*, wprowadź zmiany i kliknij przycisk *Overrides*.

Obiekt *EnemyObject* też przeciągnij do panelu *Prefabs*, a następnie usuń go z panelu *Hierarchy*.

Skrypt — logika komponentu

Pora wprawić obiekty *PlayerObject* i *EnemyObject* w ruch! Zaznacz prefabrykat *PlayerObject* i przeciągnij go do panelu *Hierarchy*. Zwróć uwagę, że w panelu *Inspector* pojawią się właściwości obiektu *PlayerObject*.

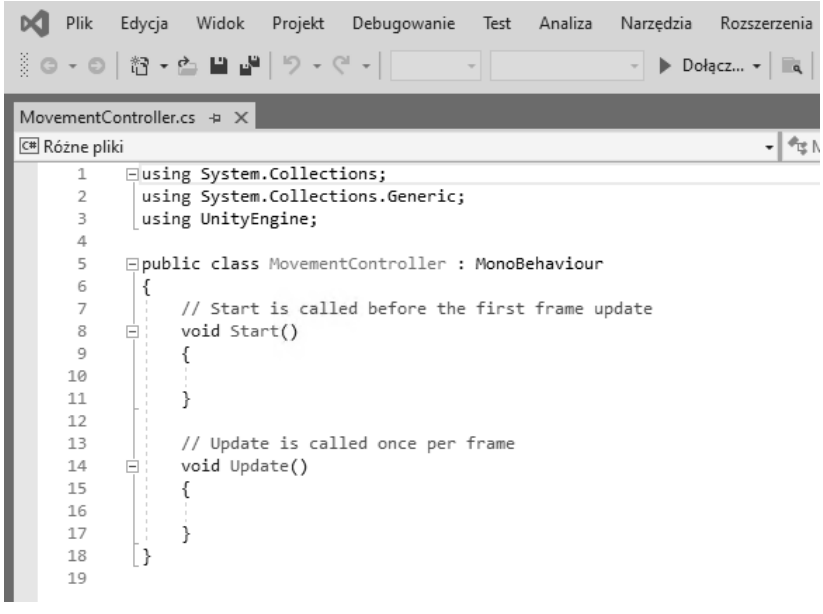
Przewiń zawartość panelu *Inspector* w dół, kliknij przycisk *Add Component*, w polu wyszukiwania wpisz frazę **`script`**, wybierz pozycję *New Script* (nowy skrypt) i nadaj nowemu skryptowi nazwę *MovementController*, jak na rysunku 3.30.



Rysunek 3.30. *Utworzenie skryptu o nazwie MovementController*

Nowy skrypt zostanie utworzony w głównym folderze *Assets*. W panelu *Project* utwórz nowy podfolder o nazwie *Scripts* i przeciągnij do niego skrypt *MovementController*. Kliknij skrypt dwukrotnie, aby otworzyć go w programie Microsoft Visual Studio.

Nadszedł czas, aby napisać pierwszy skrypt. Skrypty w programie Unity tworzy się w języku C#. Po otwarciu skryptu *MovementController* w programie Microsoft Visual Studio powinieneś uzyskać widok jak na rysunku 3.31.



Rysunek 3.31. *Skrypt MovementController otwarty w programie Microsoft Visual Studio*

■ **Uwaga** Jeszcze do niedawna skrypty można było pisać w dwóch językach: C# i odmianie JavaScript, zwanej „UnityScript". Od wersji Unity 2017.2 beta rozpoczął się proces wycofywania UnityScript. W zakątkach internetu możesz jeszcze znaleźć skrypty napisane w tym języku. Niemniej jednak, od tej chwili powinieneś używać tylko C#. Informacje o powodach wycofania języka UnityScript znajdziesz w blogu pod adresem *https://blogs.unity3d.com*.

Przeanalizujmy strukturę typowego skryptu w programie Unity. Wszystkie widoczne niżej wiersze należy wpisywać dokładnie tak, jak wyglądają. Języki programowania mają ściśle określoną składnię i nie można pomijać średników czy końców wierszy, nie można też wpisywać niepotrzebnych liter i cyfr. Wiersze rozpoczynające się od znaków // są komentarzami mającymi znaczenie wyłącznie informacyjne i można je pominąć. Komentarze umieszcza się, wpisując znaki // przed jednoliniowym komentarzem lub /* na początku i */ na końcu komentarza.

```
// 1
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
// 2
public class MovementController : MonoBehaviour
{
    // 3
    void Start()
    {

    }

    // 4
    void Update()
    {

    }
}
```

Przeanalizujmy poszczególne fragmenty powyższego kodu.

```
// 1
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

Przestrzenie nazw służą do porządkowana i kontrolowania zakresów działania klas oraz do zapobiegania konfliktom, co ułatwia pracę programiście. Instrukcja `using` wskazuje przestrzeń platformy .NET i zwalnia programistę z wpisywania pełnych nazw klas we wszystkich miejscach, w których są stosowane ich metody. Na przykład wskazując przestrzeń `System` za pomocą poniższego wiersza:

```
using System;
```

można zamiast skomplikowanej instrukcji:

```
System.Console.WriteLine("Najlepsza gra RPG na świecie!");
```

wpisać jej krótszą wersję:

```
Console.WriteLine("Najlepsza gra RPG na świecie!");
```

Jest to możliwe, ponieważ deklaracja `using System;` oznacza, że w bieżącym kodzie jest stosowana przestrzeń nazw `System`.

Przestrzenie nazw można zagnieżdżać. Dzięki temu można odwoływać się do przestrzeni umieszczonych wewnątrz innych przestrzeni, na przykład `Collections` wewnątrz `System`. W tym celu należy wpisać następujący wiersz:

```
using System.Collections;
```

Przestrzeń `UnityEngine` zawiera wiele klas właściwych dla programu Unity. Niektóre z nich, np. `MonoBehaviour`, `GameObject`, `Rigidbody2D` i `BoxCollider2D` zostały już wykorzystane na scenie w tej grze. Deklarując przestrzeń `UnityEngine`, można jej klasy wykorzystywać w skrypcie.

```
// 2
public class MovementController : MonoBehaviour
```

Klasa, która ma być przypisana obiektowi *GameObject* za pomocą komponentu, musi dziedziczyć cechy klasy `MonoBehaviour`. Tak zdefiniowana nowa klasa posiada m.in. metody

`Awake()`, `Start()`, `Update()`, `LateUpdate()` i `OnCollisionEnter()`, wywoływane przez funkcję obsługującą zdarzenia.

```
// 3
void Start()
```

Jedną z metod nadrzędnej klasy `MonoBehaviour` jest `Start()`. Funkcja obsługi zdarzeń będzie opisana później, natomiast teraz można się domyśleć na postawie nazwy, że metoda `Start()` jest jedną z pierwszych wywoływanych po uruchomieniu skryptu. Jest ona wywoływana przed aktualizacją pierwszej ramki, gdy są spełnione następujące warunki:

1. Klasa dziedziczy cechy klasy `MonoBehaviour`. Nasza klasa `MovementController` spełnia ten warunek.
2. Skrypt musi być włączony w chwili inicjalizacji gry. Domyślnie wszystkie skrypty są włączone. Wyłączony skrypt może być źródłem błędów.

```
// 4
void Update()
```

Metoda `Update()` jest wywoływana jeden raz podczas wyświetlania ramki i w niej definiuje się działanie gry. W grze wyświetlającej 24 ramki na sekundę powyższa metoda jest wywoływana z taką samą częstością. Jednak interwały pomiędzy kolejnymi wywołaniami mogą być różne. Jeżeli kod ma być wykonywany w regularnych odstępach czasu, należy użyć metody `FixedUpdate()`.

Po zapoznaniu się z domyślnym skryptem zastąp go następującym kodem:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovementController : MonoBehaviour
{
    // 1
    public float movementSpeed = 3.0f;
    // 2
    Vector2 movement = new Vector2();
    // 3
    Rigidbody2D rb2D;
    private void Start()
    {
        // 4
        rb2D = GetComponent<Rigidbody2D>();
    }
    private void Update()
    {
        // Ta metoda na razie jest pusta.
    }
    // 5
    void FixedUpdate()
    {
        // 6
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");
        // 7
        movement.Normalize();
```

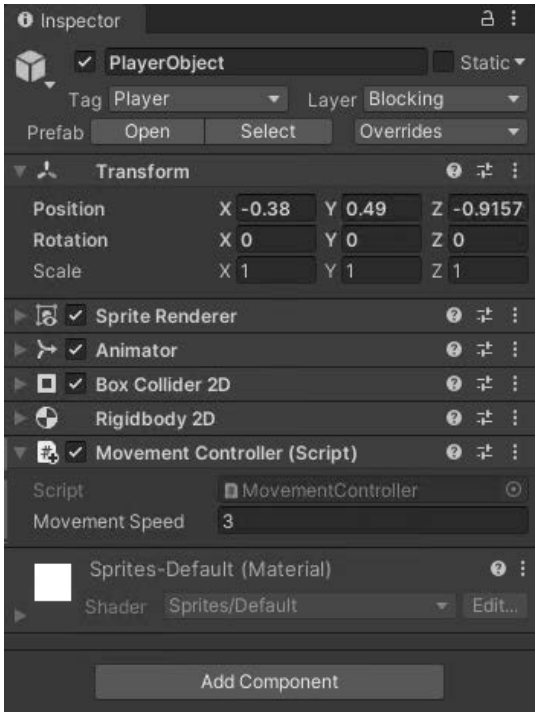
```
        // 8
        rb2D.velocity = movement * movementSpeed;
    }
}
```

Przeanalizujmy poszczególne fragmenty kodu.

```
// 1
public float movementSpeed = 3.0f;
```

Jest to deklaracja publicznej zmiennoprzecinkowej zmiennej, która będzie określać prędkość poruszania się postaci. Ponieważ jest to publiczna zmienna, pojawi się w panelu *Inspector* we właściwościach obiektu *GameObject*, do którego przypiszesz ten skrypt.

Rysunek 3.32 przedstawia panel *Inspector*, w którym z sekcji *Movement Controller (Script)* (kontroler ruchu (skrypt)) jest widoczna powyższa zmienna. Program Unity automatycznie zamienia pierwszą literę nazwy zmiennej na wielką i wprowadza spację przed wielką literą w środku nazwy. W efekcie nazwa *movementSpeed* jest wyświetlana jako *Movement Speed*.



Rysunek 3.32. *Publiczna zmienna movementSpeed wyświetlana z początkową wielką literą i spacją w środku*

```
// 2
Vector2 movement = new Vector2();
```

`Vector2` jest wbudowaną klasą zawierającą dwuwymiarowe wektory lub punkty. Wykorzystamy ją do przechowywania położenia obiektów gracza i przeciwnika w dwuwymiarowej przestrzeni oraz kierunków ich ruchów.

```
// 3
Rigidbody2D rb2D;
```

Deklaracja właściwości przechowującej referencję do instancji klasy `Rigidbody2D`:

```
// 4
rb2D = GetComponent<Rigidbody2D>();
```

Parametrem metody `GetComponent()` jest typ, a zwracanym wynikiem komponent przypisany bieżącemu obiektowi wskazanego typu. Metoda ta jest wywoływana w celu uzyskania referencji do komponentu `Rigidbody2D`, przypisanego obiektowi *PlayerObject* w edytorze Unity. Komponent ten będzie wykorzystywany do przemieszczania duszka gracza.

```
// 5
FixedUpdate()
```

Jak wspomniałem wcześniej, metoda `FixedUpdate()` jest wywoływana przez silnik Unity w stałych odstępach czasu. Różni się tym od metody `Update()`, wywoływanej jeden raz przy wyświetleniu ramki. Na słabszym urządzeniu ramki mogą być generowane wolniej i w efekcie metoda `Update()` może być wywoływana z mniejszą częstością.

```
// 6
movement.x = Input.GetAxisRaw("Horizontal");
movement.y = Input.GetAxisRaw("Vertical");
```

Za pomocą klasy `Input` można na różne sposoby odbierać sygnały od użytkownika. W tym przypadku jest użyta metoda `GetAxisRaw()`, której wyniki są przypisywane właściwościom `x` i `y` obiektu typu `Vector2`. Parametrem metody jest ciąg znaków określający oś poziomą (`"Horizontal"`) lub pionową (`"Vertical"`). Zwracanym wynikiem, odbieranym z menedżera wartości wejściowych, jest liczba –1, 0 lub 1. Wartość 1 oznacza naciśnięcie przez gracza klawisza ze strzałką w prawo lub literą *D* (w przypadku typowej konfiguracji klawiszy *W*, *A*, *S*, *D*), a wartość –1 naciśnięcie klawisza ze strzałką w lewo lub literą *A*. Wartość 0 oznacza, że gracz nie nacisnął żadnego klawisza. Układ klawiszy sterujących definiuje się za pomocą menedżera wartości wejściowych, którym zajmiemy się później.

```
// 7
movement.Normalize();
```

Powyższa metoda normalizuje wektor i przemieszcza duszka gracza z taką samą prędkością w poziomie, pionie i po przekątnych.

```
// 8
rb2D.velocity = movement * movementSpeed;
```

Mnożąc zmienną `movementSpeed` przez wektor `movement`, uzyskuje się prędkość obiektu *PlayerObject*, któremu przypisany jest komponent `Rigidbody2D`, oraz przemieszcza się go.

Wróć do edytora Unity i upewnij się, że w panelu *Hierarchy* znajduje się obiekt *PlayerObject*. Jeżeli go tam nie ma, przeciągnij prefabrykat *PlayerObject* z folderu *Prefabs* do panelu *Hierarchy*.

Musisz wykonać jeszcze jeden, bardzo ważny krok: przypisać skrypt do obiektu *PlayerObject*. W tym celu przeciągnij skrypt *MovementController* z folderu *Scripts* do obiektu *PlayerObject* w panelu *Hierarchy* lub do panelu *Inspector* po uprzednim zaznaczeniu obiektu *PlayerObject*.

W ten sposób przypisuje się skrypt do obiektu w edytorze Unity. Skrypt *MovementController* ma teraz dostęp do innych komponentów przypisanych do obiektu *PlayerObject*.

Kliknij przycisk odtwarzania gry. Zobaczysz postać maszerującego gracza, którego możesz przesuwać, naciskając klawisze *W*, *A*, *S*, *D* lub klawisze strzałek.

Gratulacje! Właśnie tchnąłeś życie w to, co kiedyś było tylko elektronicznymi impulsami.

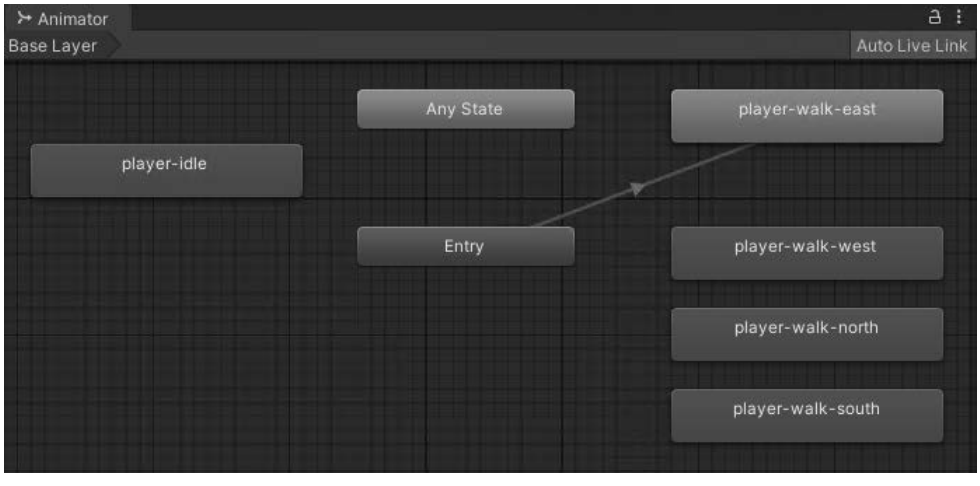
Stany i animacje

Więcej o maszynie stanów

Teraz, gdy wiesz już, jak przemieszcza się postacie na ekranie, zajmijmy się przełączaniem animacji w zależności od aktualnego stanu obiektu gracza.

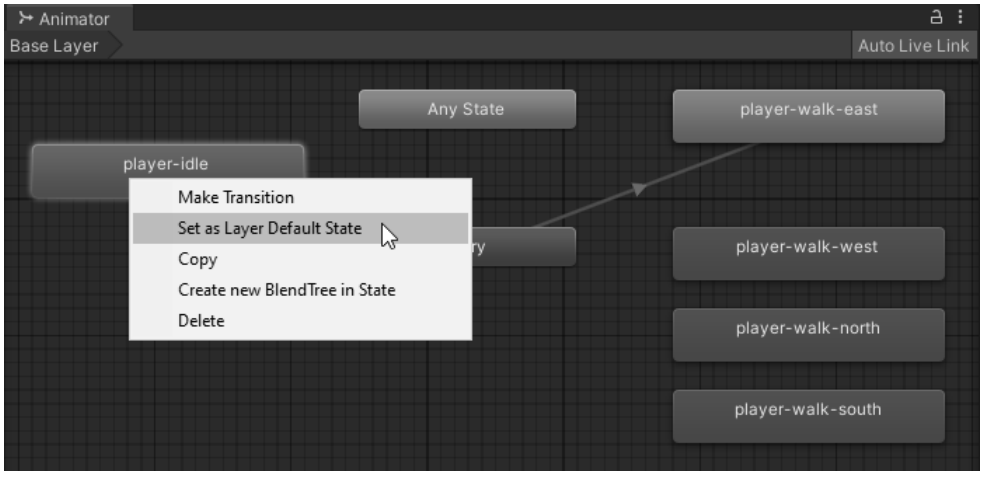
Przejdź do folderu *Animations/Controllers* i kliknij dwukrotnie obiekt *PlayerObject*. Pojawi się panel *Animator*, zawierający utworzoną wcześniej maszynę stanów. Jak wspomniałem wcześniej, maszyna reprezentuje różne stany obiektu gracza oraz skojarzone z nimi klipy animacyjne.

Rozmieść obiekty stanów tak jak na rysunku 3.33. Stan *player-idle* przesuń na bok, a stany *player-walk* umieść obok siebie. Nie musisz ich dokładnie wyrównywać, ponieważ znaczenie mają tylko strzałki pomiędzy nimi.



Rysunek 3.33. *Widok stanów w panelu Animator*

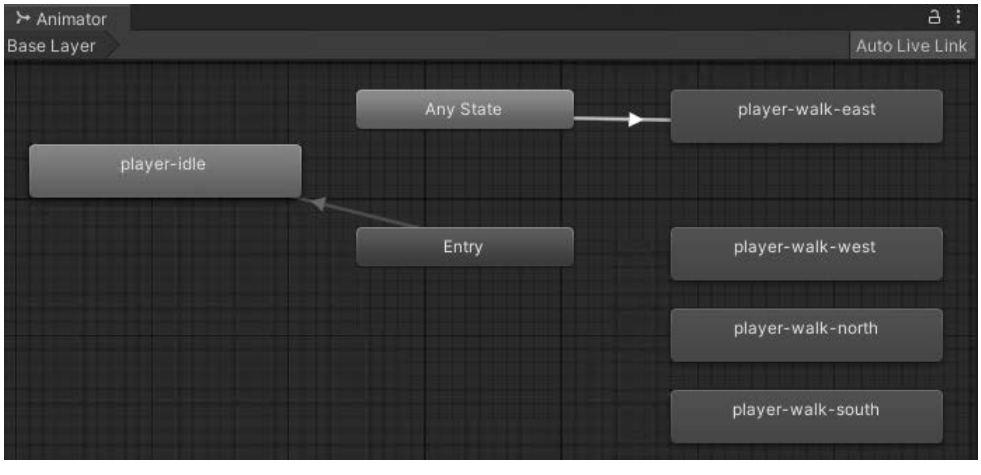
Stan *player-walk-east* jest wyróżniony kolorem pomarańczowym, oznaczającym, że jest to domyślny stan duszka. Kliknij prawym przyciskiem myszy stan *player-idle* i wybierz polecenie *Set as Layer Default State* (ustaw jako domyślny stan w warstwie), jak na rysunku 3.34. Kolor stanu zmieni się na pomarańczowy.



Rysunek 3.34. *Aby ustawić stan player-idle jako domyślny, kliknij go prawym przyciskiem myszy i wybierz polecenie Set as Layer Default State*

Stan *player-idle* powinien być domyślnym, ponieważ gdy gracz nie będzie naciskał żadnych klawiszy, postać powinna być zwrócona w kierunku na południe i nie powinna się poruszać. W ten sposób uzyska się efekt oczekiwania postaci na decyzję gracza.

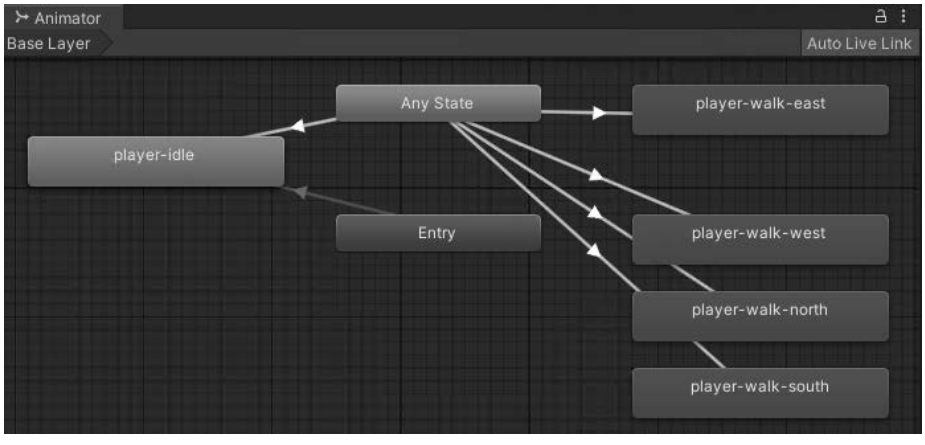
Teraz kliknij prawym przyciskiem myszy stan *Any State* (dowolny stan) i wybierz polecenie *Make Transition* (utwórz przejście). Pojawi się biała linia ze strzałką, podążającą za kursorem myszy. Kliknij stan *player-walk-east*. W ten sposób utworzysz przejście ze stanu *Any State* do *player-walk-east*. Powinieneś uzyskać widok jak na rysunku 3.35.



Rysunek 3.35. *Zdefiniowane przejście ze stanu Any State do player-walk-east*

Powtórz tę operację dla pozostałych stanów. Jak wspomniałem wcześniej, obiekt znajduje się w stanie *Any State*, gdy przechodzi do innego stanu. W sumie powinieneś zdefiniować pięć przejść oznaczonych białymi strzałkami wychodzącymi ze stanu *Any State* i kończącymi się

przy czterech stanach *player-walk-xxx* oraz przy stanie *player-idle*. Dodatkowo powinna być widoczna pomarańczowa strzałka wychodząca od stanu *Entry* (wejście) i kończąca się przy stanie *player-idle*, jak na rysunku 3.36.



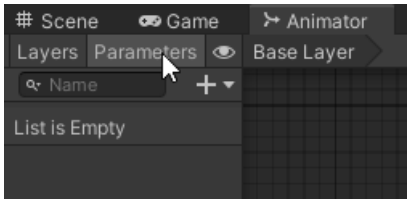
Rysunek 3.36. *Przejścia pomiędzy stanami*

Parametry animacji

Aby wykorzystać przejścia i stany, należy zdefiniować tzw. *parametry animacji*, czyli zmienne w kontrolerze animacji, które w skrypcie będą wykorzystane do sterowania maszyną stanów.

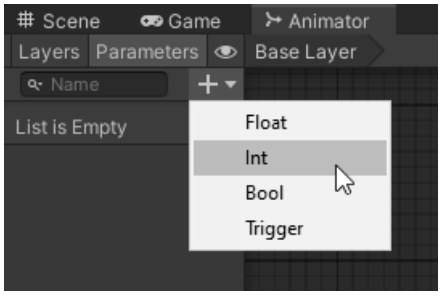
Parametr, który zdefiniujemy, wykorzystamy w przejściach i w skrypcie *MovementController* do sterowania obiektem *PlayerObject* oraz do przemieszczania go na ekranie.

Kliknij zakładkę *Parameters* widoczną w lewym górnym rogu panelu *Animator* (rysunek 3.37), następnie ikonę znaku dodawania i wybierz polecenie *Int* (liczba całkowita — rysunek 3.38). Nowemu parametrowi nadaj nazwę *AnimationState*, jak na rysunku 3.39.

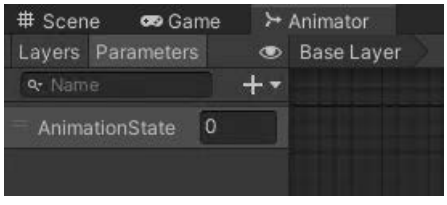


Rysunek 3.37. *Zakładka Parameters w panelu Animator*

Utworzony parametr wykorzystasz do zdefiniowania warunku każdego przejścia. Jeżeli w trakcie gry warunek zostanie spełniony, nastąpi przejście obiektu do określonego stanu i zostanie odtworzony odpowiedni klip animacyjny. Ponieważ komponent *Animator* jest przypisany do obiektu *PlayerObject*, klipy będą odtwarzane w miejscu, w którym na scenie będzie znajdował się komponent *Transform*. Za pomocą skryptu będziesz zmieniał wartość parametru *AnimationState*, aby warunek został spełniony i nastąpiło przejście pomiędzy stanami.



Rysunek 3.38. *Z rozwijanego menu wybierz polecenie Int*



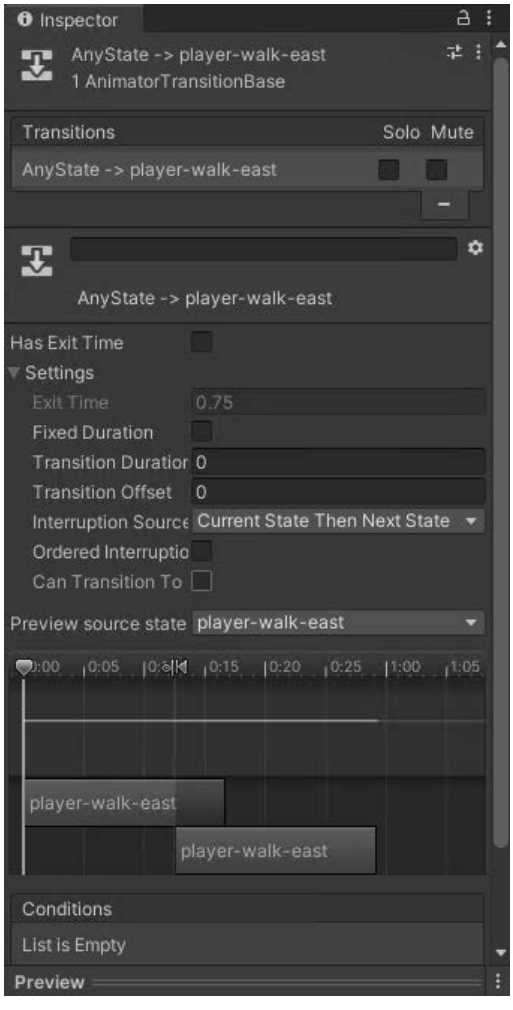
Rysunek 3.39. *Nadaj parametrowi nazwę AnimationState*

Zaznacz białą linię przejścia ze stanu *Any State* do *player-walk-east*. W panelu *Inspector* usuń zaznaczenie opcji *Exit Time* (czas wyjścia), *Fixed Duration* (stały czas trwania) i *Can Transition to Self* (możliwe przejście do tego samego stanu). W polu *Transition Duration (%)* (czas trwania przejścia) wpisz **0**, a z listy *Interruption Source* (źródło przerwań) wybierz pozycję *Current State Then Next State* (bieżący stan, potem następny stan). Usuń zaznaczenie opcji *Has Exit Time* (ma czas wyjścia), ponieważ animacja ma być przerywana, gdy gracz naciśnie inny klawisz. Gdyby ta opcja była zaznaczona, wtedy animacja musiałaby zostać odtworzona w części określonej w polu *Exit Time* (czas wyjścia) i dopiero potem mogłaby się rozpocząć następna animacja, co byłoby niepożądanym efektem.

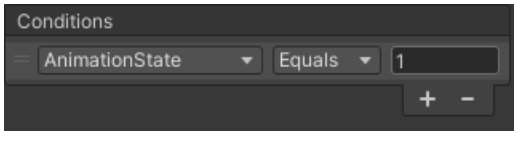
Panel powinien wyglądać jak na rysunku 3.40.

W dolnej części panelu widoczna jest sekcja *Conditions* (warunki). Kliknij widoczny w niej symbol plusa, z list wybierz pozycje *AnimationState* i *Equals* (jest równe), a w polu obok wpisz wartość **1**, jak na rysunku 3.41. Zdefiniowany w ten sposób warunek można wyrazić następująco: gdy parametr *AnimationState* będzie równy 1, przejdź do tego stanu i odtwórz animację. W ten sposób będziesz zmieniać stany za pomocą skryptu, który napiszesz.

■ **Uwaga** Łatwo można nieopatrznie pozostawić drugą listę warunku na pozycji *Greater* (większe niż), dlatego zwróć uwagę na to ustawienie. Jeżeli pozycja będzie inna niż *Equals*, wtedy przejście nie będzie poprawnie realizowane.



Rysunek 3.40. *Konfiguracja przejścia w panelu Inspector*



Rysunek 3.41. *Definiowanie warunku z wykorzystaniem parametru AnimationState*

Kolejnym zadaniem jest nadanie parametrowi *AnimationState* wartości 1 za pomocą skryptu. Wróć do programu Visual Studio i skryptu *MovementController*. Zastąp istniejący kod następującym:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class MovementController : MonoBehaviour
{
    public float movementSpeed = 3.0f;
    Vector2 movement = new Vector2();
// 1
    Animator animator;
// 2
    string animationState = "AnimationState";
    Rigidbody2D rb2D;
// 3
    enum CharStates
    {
        walkEast = 1,
        walkSouth = 2,
        walkWest = 3,
        walkNorth = 4,
        idleSouth = 5
    }

    private void Start()
    {
// 4
        animator = GetComponent<Animator>();
        rb2D = GetComponent<Rigidbody2D>();
    }

    private void Update()
    {
// 5
        UpdateState();
    }

    void FixedUpdate()
    {
// 6
        MoveCharacter();
    }

    private void MoveCharacter()
    {
        movement.x = Input.GetAxisRaw("Horizontal");
        movement.y = Input.GetAxisRaw("Vertical");
        movement.Normalize();
        rb2D.velocity = movement * movementSpeed;
    }

    private void UpdateState()
    {
// 7
        if (movement.x > 0)
        {
            animator.SetInteger(animationState, (int)CharStates.walkEast);
        }
        else if (movement.x < 0)
        {
            animator.SetInteger(animationState, (int)CharStates.walkWest);
        }
        else if (movement.y > 0)
```

```
        {
            animator.SetInteger(animationState, (int)CharStates.walkNorth);
        }
        else if (movement.y < 0)
        {
            animator.SetInteger(animationState, (int)CharStates.walkSouth);
        }
        else
        {
            animator.SetInteger(animationState, (int)CharStates.idleSouth);
        }
    }
}
```

Przeanalizujmy poszczególne fragmenty kodu.

```
// 1
Animator animator;
```

Definiujemy zmienną `animator`, w której później zapiszemy referencję do komponentu *Animator* obiektu *GameObject*, do którego jest przypisany ten skrypt.

```
// 2
string animationState = "AnimationState";
```

Wpisywanie tego samego ciągu znaków w wielu miejscach kodu (ang. *hard-coding*) jest częstym źródłem błędów, ponieważ w takim przypadku łatwo o pomyłkę. Aby tego uniknąć, należy ciąg wpisać tylko raz w deklaracji zmiennej i odwoływać się do niej wszędzie tam, gdzie jest on potrzebny.

```
// 3
enum CharStates
```

Instrukcja `enum` deklaruje zbiór wyliczanych stałych. Każda stała jest określonego typu, np. `int` (liczba całkowita). Odwołując się do elementu zbioru, uzyskuje się odpowiadającą mu wartość.

Tutaj deklarujemy zbiór `CharStates`, którego użyjesz do powiązania stanów duszka (marsz na wschód, na południe itp.) z liczbami całkowitymi. Liczby te wykorzystasz wkrótce w maszynie stanów.

```
// 4
animator = GetComponent<Animator>();
```

W tym wierszu zmiennej `animator` jest przypisywana referencja do komponentu *Animator* obiektu *GameObject*, do którego przypisany jest ten skrypt. Za pomocą tej zmiennej będziesz mógł szybko odwoływać się do komponentu, tj. nie będziesz musiał za każdym razem uzyskiwać referencji. Wywoływanie metody `GetComponent()` jest typowym sposobem uzyskiwania w skrypcie dostępu do komponentów, a nawet do innych skryptów.

```
// 5
UpdateState();
```

Powyższa, zdefiniowana w innym miejscu kodu, metoda aktualizuje maszynę stanów. Niezbędny algorytm został przeniesiony do osobnej metody, aby kod był bardziej czytelny. Im więcej kodu zawiera metoda, tym jest mniej zrozumiała, a mało czytelny kod jest trudniejszy w diagnozowaniu, testowaniu i utrzymaniu.

```
// 6
MoveCharacter();
```

Kod przemieszczający duszka również został przeniesiony do osobnej metody, aby kod był bardziej czytelny.

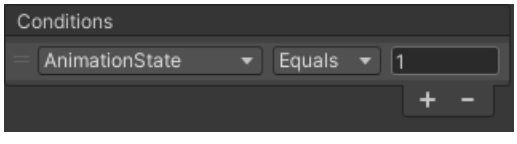
```
// 7
```

W serii instrukcji `if-else-if` oznaczonej tym komentarzem sprawdzany jest wynik zwrócony przez metodę `Input.GetAxisRaw()` (liczba –1, 0 lub 1) i odpowiednio przemieszczany jest duszek. Przeanalizujmy poniższy fragment:

```
if (movement.x > 0)
{
    animator.SetInteger(animationState, (int)CharStates.walkEast);
}
```

Jeżeli liczba opisująca przemieszczenie duszka wzdłuż osi *x* jest większa od 0, to znaczy, że gracz nacisnął klawisz strzałki w prawo.

Maszynie stanów trzeba przekazać informację, że obiekt musi przejść w stan *walk-east*. W tym celu wywoływana jest metoda `SetInteger()` przypisująca odpowiednią wartość zdefiniowanemu wcześniej parametrowi animacji, co powoduje przejście z jednego stanu do innego. Powyższa metoda ma dwa parametry: ciąg znaków i liczbę całkowitą. Pierwszy argument zawiera wartość zdefiniowanego wcześniej za pomocą edytora Unity parametru animacji o nazwie *AnimationState* (rysunek 3.42).



***Rysunek 3.42**. Wartość tego parametru jest nadawana za pomocą skryptu*

Nazwa tego parametru została dla wygody zapisana w skrypcie w zmiennej `animationState`, która jest umieszczona w pierwszym argumencie metody `SetInteger()`.

Drugim argumentem powyższej metody jest wartość przypisywana parametrowi. Ponieważ każdemu elementowi zbioru `CharStates` odpowiada liczba całkowita, wartością wyrażenia `CharStates.walkEast` jest liczba przypisana elementowi `walkEast`, tj. 1. Wymagane jest jawne rzutowanie (konwersja) typu wartości za pomocą prefiksu *(int)*. Wymóg rzutowania typu wynika z wewnętrznej implementacji języka C# i wykracza poza zakres tej książki.

Zapisz skrypt i wróć do programu Unity, aby zebrać wszystko razem. Kliknij białą strzałkę skierowaną do stanu *player-walk-south* i w sekcji *Conditions* (warunki) kliknij symbol plusa. Z rozwijanych list wybierz pozycje *AnimationState* i *Equals*, a w polu obok wpisz liczbę **2**, odpowiadającą elementowi `walkSouth` użytemu w skrypcie.

W analogiczny sposób zaznacz każdą ze strzałek skierowanych do stanów *player-walk-west*, *player-walk-north* i *player-idle*, dodaj warunek w panelu *Inspector* i w polu wpisz liczbę przypisaną odpowiedniemu elementowi `CharStates`:

```
enum CharStates
    {
        walkEast = 1,
```

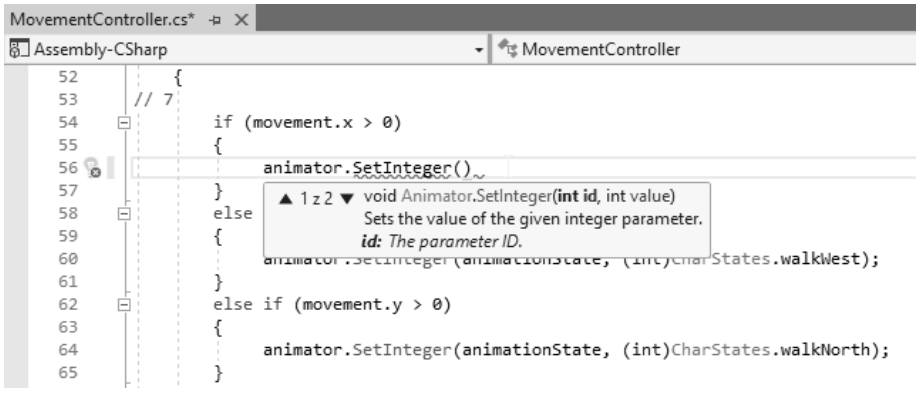
```
        walkSouth = 2,
        walkWest = 3,
        walkNorth = 4,
        idleSouth = 5
    }
```

Podczas konfigurowania przejść pamiętaj o usunięciu zaznaczeń opcji *Exit Time*, *Fixed Duration*, *Can Transition to Self* oraz o wpisaniu w polu *Transition Duration (%)* liczby 0.

Pozostała jeszcze jedna rzecz do zrobienia — ostatnia, obiecuję! W każdym stanie *player-walk-xxx* wpisz w polu *Speed* wartość **0.6**, a w stanie *player-idle* wartość **0.25**. Dzięki temu animacje będą odtwarzane w rozsądnym tempie.

Skonfigurowałeś dużą część animacji niezbędnych w Twojej grze. Kliknij przycisk odtwarzania i przesuwaj duszka za pomocą klawiszy strzałek lub *W*, *A*, *S*, *D*.

■ **Wskazówka** Jeżeli nie pamiętasz dokładnych parametrów metody, skorzystaj z podpowiedzi wyświetlanych w edytorze Visual Studio, pokazanych na rysunku 3.43. Aby automatycznie uzupełnić kod metody, naciśnij klawisz *Enter*.



Rysunek 3.43. *Program Visual Studio wyświetla podpowiedzi dotyczące nazw i typów parametrów metod*

Podsumowanie

W tym rozdziale posiadłeś ogromną podstawową wiedzę niezbędną do tworzenia gier za pomocą programu Unity. Poznałeś filozofię projektowania gier i zasady działania programu. Dowiedziałeś się, że gra składa się ze scen, a każda scena z obiektów *GameObject*. Poznałeś komponenty *Colliders* i *Rigidbody* oraz ich zastosowanie do wykrywania kolizji obiektów. Wiesz już, jak wyglądają interakcje silnika Unity z graczem. Potrafisz używać znaczników, czyli etykiet, do odwoływania się w skrypcie do obiektów *GameObject*, na przykład *PlayerObject*, w trakcie działania gry. Innym przydatnym pojęciem jest warstwa, wykorzystywana do grupowania obiektów. Algorytm gry implementuje się w warstwie za pomocą skryptów.

Jednym z najważniejszych pojęć opisanych w tym rozdziale jest prefabrykat, czyli gotowy szablon, wykorzystywany do tworzenia kopii zasobów. Jeżeli na przykład gra musi zawierać setki obiektów przeciwników, pojawiających się pojedynczo lub równocześnie (jeżeli zamierzasz zabić gracza), wtedy zamiast tworzyć każdy obiekt *GameObject* osobno, możesz przygotować prefabrykat i użyć go do utworzenia dowolnej liczby obiektów przeciwnika.

Rozpocząłeś naukę pisania skryptów, którą będziesz kontynuował w kolejnych rozdziałach książki. Pierwszy napisany skrypt służy do przemieszczania postaci gracza za pomocą komponentu *Transform* obiektu *PlayerObject*. W skrypcie wykorzystałeś parametr animacyjny, za pomocą którego możesz sterować maszyną stanów podczas przechodzenia obiektu między stanami i odtwarzać klipy animacyjne.

W tym rozdziale dowiedziałeś się mnóstwa rzeczy, a to dopiero początek!

Marzysz o grze idealnej? Więc siadaj i programuj!

Unity jest zaawansowanym środowiskiem do tworzenia gier. Zapewnia możliwość projektowania grywalnych, świetnie wyglądających i pasjonujących gier. Dzięki temu, że Unity stworzono specjalnie w tym celu, projektant może się skupić na tym, co najważniejsze: na świetnej fabule i emocjonujących zwrotach akcji. Żmudne tworzenie kodu obsługującego podstawowe funkcje interfejsu, postaci czy sceny zostało ograniczone do minimum. Z takimi możliwościami rozpoczęcie pracy z Unity i napisanie pierwszej gry przychodzi naturalnie.

Ta książka jest praktycznym wprowadzeniem do tworzenia gier 2D w Unity. Dokładnie wyjaśniono w niej filozofię działania tego środowiska i zasady projektowania gier korzystających z silnika Unity. Starannie opisano takie zagadnienia jak arkusze duszków, dzielone kafelki i mapa kafelków. Omówiono również powiązane z Unity kanały dystrybucyjne, dzięki którym odkrywanie, kupowanie i sprzedawanie gier jest bardzo proste. Poszczególne zagadnienia zostały wyjaśnione za pomocą fragmentów kodu C#. Co prawda nie jest to podręcznik programowania w tym języku, jednak dzięki analizie poszczególnych przykładów, a także samodzielnym próbom modyfikowania i rozwijania kodu możesz poprawić również swoje umiejętności programowania w C#, używając go w praktyczny i konkretny sposób.

W książce między innymi:

- gruntowne wprowadzenie do pracy w Unity
- projektowanie interfejsu użytkownika
- budowa postaci i świata gry
- tworzenie niezbędnych skryptów, klas, koprocedur
- algorytmy sztucznej inteligencji do zastosowania w grach

Jared Halpern — jest programistą. Od kilkunastu lat pisze aplikacje dla iPhone'a, między innymi gry oraz aplikacje fotograficzne, zakupowe, filmowe i graficzne. Interesuje się środowiskiem Unity, językiem Swift i kreatywnymi zastosowaniami techniki. Pasjonuje go potencjał gier w opowiadaniu historii i dostarczaniu wrażeń. Obecnie pracuje jako samodzielny programista.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8235-0
9 788328 382350
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 59,00 zł
Apress