

Joseph Hocking

UNITY W AKCJI

Helion 

Tytuł oryginału: Unity in Action: Multiplatform Game Development in C#

Tłumaczenie: Robert Górczyński

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-283-3519-6

Original edition copyright © 2015 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2017 by HELION SA
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
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)

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	11
Wprowadzenie	13
Podziękowania	15
O książce	17
CZĘŚĆ I. PIERWSZE KROKI	21
<i>Rozdział 1. Poznajemy Unity</i>	23
1.1. Dlaczego Unity jest doskonałym wyborem?	24
1.1.1. Mocne i słabe strony Unity	24
1.1.2. Wady, których należy być świadomym	27
1.1.3. Przykłady gier utworzonych za pomocą Unity	28
1.2. Jak używać Unity?	30
1.2.1. Widoki Scene i Game oraz pasek narzędzi	31
1.2.2. Użycie klawiatury i myszy	33
1.2.3. Panele Hierarchy i Inspector	34
1.2.4. Panele Project i Console	35
1.3. Rozpoczęcie programowania w Unity	35
1.3.1. Uruchamianie kodu w Unity — komponent skryptu	37
1.3.2. Użycie MonoDevelop, czyli niezależnego od platformy środowiska IDE	38
1.3.3. Wyświetlanie informacji w konsoli	40
1.4. Podsumowanie	42
<i>Rozdział 2. Budowa demo pokazującego przestrzeń 3D</i>	43
2.1. Zanim przystąpisz do pracy...	44
2.1.1. Planowanie projektu	44
2.1.2. Poznajemy układ współrzędnych 3D	45
2.2. Rozpoczęcie projektu — umieszczenie obiektów na scenie	47
2.2.1. Sceneria — podłoże, ściany zewnętrzne i wewnętrzne	48
2.2.2. Światła i kamera	50
2.2.3. Punkt widzenia i kolizje gracza	52
2.3. Poruszanie obiektami — skrypt pozwalający na zastosowanie transformacji	53
2.3.1. Diagram pokazujący sposób programowania ruchu	53
2.3.2. Utworzenie kodu źródłowego implementującego ruch	54
2.3.3. Współrzędne lokalne kontra globalne	56
2.4. Skrypt MouseLook pozwalający na rozglądanie się	57
2.4.1. Rotacja pozioma na podstawie ruchu myszy	58
2.4.2. Rotacja pionowa z ograniczeniami	59
2.4.3. Jednoczesna rotacja w poziomie i w pionie	61

2.5.	Komponent danych wejściowych klawiatury — kontrolki pierwszej osoby	64
2.5.1.	<i>Reakcja na naciśnięcie klawisza</i>	64
2.5.2.	<i>Ustawienie współczynnika ruchu niezależnie od szybkości komputera</i>	65
2.5.3.	<i>Komponent Character Controller i wykrywanie kolizji</i>	66
2.5.4.	<i>Dostosowanie komponentów do ruchu po ziemi, a nie do lotu</i>	68
2.6.	Podsumowanie	70
Rozdział 3. Dodanie nieprzyjaciół i pocisków do gry 3D		71
3.1.	Oddawanie strzałów i raycasting	72
3.1.1.	<i>Czym jest raycasting?</i>	73
3.1.2.	<i>Użycie metody ScreenPointToRay()</i>	73
3.1.3.	<i>Dodanie graficznych wskaźników celownika i miejsca trafienia pocisku</i>	76
3.2.	Skrypt obsługujący reaktywnych nieprzyjaciół	79
3.2.1.	<i>Ustalenie, co zostało trafione</i>	79
3.2.2.	<i>Poinformowanie celu o jego trafieniu</i>	80
3.3.	Podstawy sztucznej inteligencji	82
3.3.1.	<i>Diagram pokazujący sposób działania podstawowej sztucznej inteligencji</i>	82
3.3.2.	<i>Dostrzeżenie przeszkód za pomocą promienia światła</i>	83
3.3.3.	<i>Monitorowanie stanu postaci</i>	85
3.4.	Tworzenie prefabrykatu nieprzyjaciela	87
3.4.1.	<i>Czym jest prefabrykat?</i>	87
3.4.2.	<i>Utworzenie prefabrykatu nieprzyjaciela</i>	87
3.4.3.	<i>Utworzenie obiektu na podstawie niewidzialnego obiektu wraz z dołączonym skryptem SceneController</i>	88
3.5.	Oddawanie strzałów przez tworzenie obiektów	90
3.5.1.	<i>Utworzenie prefabrykatu pocisku</i>	90
3.5.2.	<i>Strzelanie pociskiem i trafianie w cel</i>	92
3.5.3.	<i>Uszkodzenie gracza</i>	95
3.6.	Podsumowanie	96
Rozdział 4. Przygotowanie grafiki dla gry		97
4.1.	Poznajemy zasoby graficzne	97
4.2.	Utworzenie prostej scenerii 3D — whiteboxing	100
4.2.1.	<i>Poznajemy whiteboxing</i>	101
4.2.2.	<i>Przygotowanie planu podłoża dla poziomu</i>	102
4.2.3.	<i>Ułożenie kształtów na podstawie planu</i>	102
4.3.	Teksturowanie sceny za pomocą obrazów 2D	103
4.3.1.	<i>Wybór formatu pliku</i>	104
4.3.2.	<i>Import pliku graficznego</i>	106
4.3.3.	<i>Użycie obrazu tekstury</i>	107
4.4.	Generowanie nieba za pomocą tekstury	109
4.4.1.	<i>Czym jest skybox?</i>	109
4.4.2.	<i>Utworzenie nowego materiału skyboxu</i>	110
4.5.	Praca z niestandardowymi modelami 3D	112
4.5.1.	<i>Który format pliku wybrać?</i>	113
4.5.2.	<i>Eksport i import modelu</i>	114

4.6.	Tworzenie efektów specjalnych za pomocą systemu cząstek	116
4.6.1.	<i>Dostosowanie parametrów w domyślnym efekcie systemu cząstek</i>	117
4.6.2.	<i>Zastosowanie nowej tekstury dla ognia</i>	118
4.6.3.	<i>Dolączenie efektu systemu cząstek do obiektu 3D</i>	120
4.7.	Podsumowanie	121

CZĘŚĆ II. KOMFORTOWA PRACA 123

Rozdział 5. Utworzenie gry 2D za pomocą funkcji oferowanych przez Unity 125

5.1.	Przygotowanie środowiska dla grafiki 2D	126
5.1.1.	<i>Przygotowanie projektu</i>	127
5.1.2.	<i>Wyświetlanie obrazów 2D (sprite'ów)</i>	129
5.1.3.	<i>Przełączenie kamery do trybu 2D</i>	130
5.2.	Utworzenie obiektu karty i zdefiniowanie jego reakcji na kliknięcie	133
5.2.1.	<i>Utworzenie obiektu sprite'ów</i>	133
5.2.2.	<i>Kod obsługujący dane wejściowe myszy</i>	133
5.2.3.	<i>Odkrycie karty po jej kliknięciu</i>	134
5.3.	Wyświetlanie różnych obrazów kart	135
5.3.1.	<i>Programowe wczytywanie obrazów</i>	135
5.3.2.	<i>Ustawienie obrazu na podstawie niewidocznego obiektu wraz ze skrypcem SceneController</i>	136
5.3.3.	<i>Utworzenie siatki kart</i>	138
5.3.4.	<i>Tasowanie kart</i>	140
5.4.	Utworzenie i ocena dopasowania	141
5.4.1.	<i>Przechowywanie i porównywanie odkrytych kart</i>	142
5.4.2.	<i>Ukrycie niedopasowanych kart</i>	143
5.4.3.	<i>Wyświetlenie punktacji</i>	144
5.5.	Przycisk zerujący grę	146
5.5.1.	<i>Programowanie komponentu UIButton za pomocą metody SendMessage()</i>	146
5.5.2.	<i>Wywołanie LoadLevel z poziomu skryptu SceneController</i>	149
5.6.	Podsumowanie	150

Rozdział 6. Umieszczenie interfejsu użytkownika 2D w grze 3D 151

6.1.	Zanim przystąpisz do tworzenia kodu...	153
6.1.1.	<i>Tryb bezpośredniego graficznego interfejsu użytkownika czy zaawansowany interfejs 2D?</i>	153
6.1.2.	<i>Planowanie układu</i>	154
6.1.3.	<i>Import obrazów interfejsu użytkownika</i>	155
6.2.	Konfiguracja graficznego interfejsu użytkownika	156
6.2.1.	<i>Utworzenie płótna dla interfejsu</i>	156
6.2.2.	<i>Przyciski, obrazy i etykiety tekstowe</i>	157
6.2.3.	<i>Kontrolowanie położenia elementów interfejsu użytkownika</i>	160
6.3.	Programowanie interaktywności interfejsu użytkownika	161
6.3.1.	<i>Programowanie niewidocznego komponentu UIController</i>	162
6.3.2.	<i>Utworzenie wyskakującego okna</i>	164
6.3.3.	<i>Ustawienie wartości za pomocą suwaka i pola tekstowego</i>	167

6.4.	Uaktualnianie gry przez udzielanie odpowiedzi na zdarzenia	170
6.4.1.	<i>Integracja systemu zdarzeń</i>	170
6.4.2.	<i>Rozglaszanie i nasłuchiwanie zdarzeń pochodzących ze sceny</i>	171
6.4.3.	<i>Rozglaszanie i nasłuchiwanie zdarzeń pochodzących z okna HUD</i>	172
6.5.	Podsumowanie	174

Rozdział 7. Utworzenie gry 3D z widokiem z perspektywy trzeciej osoby — animacja i ruch postaci gracza 175

7.1.	Dostosowanie widoku kamery do perspektywy trzeciej osoby	177
7.1.1.	<i>Import postaci używanej w grze</i>	178
7.1.2.	<i>Dodanie cieni na scenie</i>	179
7.1.3.	<i>Orbitowanie kamery wokół postaci gracza</i>	181
7.2.	Programowanie kontrolek ruchu względem kamery	184
7.2.1.	<i>Rotacja postaci gracza w celu ustawienia jej zgodnie z kierunkiem ruchu</i>	184
7.2.2.	<i>Poruszanie się do przodu we wskazanym kierunku</i>	187
7.3.	Implementacja akcji skoku	188
7.3.1.	<i>Zastosowanie szybkości i przyspieszenia w pionie</i>	189
7.3.2.	<i>Modyfikacja wykrywania ziemi w celu obsługi krawędzi i zboczy</i>	190
7.4.	Konfiguracja animacji postaci gracza	195
7.4.1.	<i>Zdefiniowanie klipów animacji w zaimportowanym modelu</i>	197
7.4.2.	<i>Utworzenie kontrolera animacji</i>	199
7.4.3.	<i>Utworzenie kodu źródłowego używającego kontrolera animacji</i>	202
7.5.	Podsumowanie	204

Rozdział 8. Dodanie do gry interaktywnych urządzeń i przedmiotów 205

8.1.	Utworzenie drzwi oraz innych urządzeń	206
8.1.1.	<i>Drzwi otwierane i zamykane po naciśnięciu klawisza</i>	207
8.1.2.	<i>Sprawdzenie odległości i stanięcie przed otwierającymi się drzwiami</i>	208
8.1.3.	<i>Użycie monitora zmieniającego kolor</i>	210
8.2.	Interakcja z obiektami przez ich uderzanie	211
8.2.1.	<i>Kolizja z przeszkodą stosującą zasady fizyki</i>	212
8.2.2.	<i>Obsługa drzwi za pomocą naciśnięcia</i>	213
8.2.3.	<i>Zbieranie przedmiotów rozrzuconych na poziomie gry</i>	216
8.3.	Zarządzanie danymi magazynu i stanem gry	217
8.3.1.	<i>Przygotowanie menedżerów gracza i magazynu</i>	218
8.3.2.	<i>Programowanie menedżera gry</i>	219
8.3.3.	<i>Przechowywanie danych magazynu w obiekcie kolekcji — lista kontra słownik</i>	224
8.4.	Interfejs użytkownika dla magazynu przedmiotów	226
8.4.1.	<i>Wyswietlanie w interfejsie użytkownika zebranych przedmiotów</i>	227
8.4.2.	<i>Przygotowanie klucza niezbędnego do otwarcia drzwi</i>	229
8.4.3.	<i>Przywrócenie dobrej kondycji gracza przez użycie odpowiedniego pakietu</i>	231
8.5.	Podsumowanie	232

CZĘŚĆ III. MOCNE ZAKOŃCZENIE	233
Rozdział 9. Połączenie gry z internetem	235
9.1. Tworzenie sceny zewnętrznej	237
9.1.1. Wygenerowanie nieba za pomocą skyboxu	237
9.1.2. Konfiguracja atmosfery kontrolowanej przez kod	238
9.2. Pobieranie danych dotyczących prognozy pogody z internetu	241
9.2.1. Żądanie danych WWW za pomocą podprocedur	244
9.2.2. Przetwarzanie danych XML	248
9.2.3. Przetwarzanie danych w formacie JSON	251
9.2.4. Zmiana sceny na podstawie danych dotyczących prognozy pogody	252
9.3. Dodanie billboardu pochodzącego z sieci	254
9.3.1. Wczytywanie obrazów z internetu	254
9.3.2. Wyświetlenie obrazu na billboardzie	257
9.3.3. Buforowanie pobranego obrazu w celu jego wielokrotnego użycia	258
9.4. Przekazanie danych do serwera WWW	260
9.4.1. Monitorowanie aktualnej pogody — wykonywanie żądań typu POST	261
9.4.2. Kod PHP działający po stronie serwera WWW	263
9.5. Podsumowanie	263
Rozdział 10. Odtwarzanie dźwięku — muzyka i efekty dźwiękowe	265
10.1. Import efektów dźwiękowych	266
10.1.1. Obsługiwane formaty plików	266
10.1.2. Import plików muzycznych	268
10.2. Odtwarzanie efektów dźwiękowych	270
10.2.1. Opis mechanizmu działania systemu audio — klip muzyczny kontra źródło kontra odbiorca	270
10.2.2. Zdefiniowanie zapętlonego dźwięku	271
10.2.3. Wywoływanie efektów dźwiękowych z poziomu kodu	272
10.3. Interfejs audio	274
10.3.1. Konfiguracja centralnego menedżera AudioManager	274
10.3.2. Interfejs użytkownika zmiany poziomu głośności	276
10.3.3. Odtwarzanie dźwięków interfejsu użytkownika	280
10.4. Muzyka odtwarzana w tle	281
10.4.1. Odtwarzanie muzyki w pętli	282
10.4.2. Oddzielna kontrola poziomu głośności muzyki	285
10.4.3. Wyciszenie między ścieżkami muzycznymi	288
10.5. Podsumowanie	290
Rozdział 11. Zebranie wszystkiego w całość w pełnej grze	291
11.1. Przygotowanie akcji w grze RPG poprzez wykorzystanie innych projektów	293
11.1.1. Łączenie zasobów i kodu z wielu projektów	293
11.1.2. Programowanie kontrolerek typu „wskaż i kliknij” — obsługa ruchu i urządzeń	296
11.1.3. Zastąpienie starego interfejsu użytkownika nowym	302

11.2.	Opracowanie nadrzędnej struktury gry	309
11.2.1.	<i>Kontrolowanie przepływu misji i wielu poziomów gry</i>	309
11.2.2.	<i>Ukończenie poziomu przez dotarcie do końca</i>	313
11.2.3.	<i>Przegrana gracza po złapaniu go przez nieprzyjaciela</i>	315
11.3.	Obsługa postępów gracza w całej grze	317
11.3.1.	<i>Zapisanie i wczytanie informacji o postępach gracza w grze</i>	317
11.3.2.	<i>Ukończenie całej gry przez przejście trzech poziomów</i>	321
11.4.	Podsumowanie	323
Rozdział 12. Wdrożenie gry w urządzeniach graczy		325
12.1.	Platformy tradycyjnych komputerów — Windows, macOS i Linux	327
12.1.1.	<i>Kompilacja aplikacji</i>	328
12.1.2.	<i>Dostosowanie ustawień Player Settings — nazwa i ikona gry</i>	328
12.1.3.	<i>Kompilacja zależna od platformy</i>	330
12.2.	Kompilacja aplikacji internetowej	331
12.2.1.	<i>Odtwarzacz Unity kontra HTML5 i WebGL</i>	332
12.2.2.	<i>Utworzenie pliku Unity i testowej strony internetowej</i>	332
12.2.3.	<i>Komunikacja z kodem JavaScript w przeglądarce WWW</i>	332
12.3.	Kompilacja na platformy mobilne — iOS i Android	335
12.3.1.	<i>Konfiguracja narzędzi kompilacji</i>	336
12.3.2.	<i>Kompresja tekstur</i>	339
12.3.3.	<i>Opracowywanie wtyczek</i>	341
12.4.	Podsumowanie	349
Zakończenie		351
Dodatek A. Nawigacja po scenie i skróty klawiszowe		355
A.1.	Nawigacja po scenie za pomocą myszy	355
A.2.	Najczęściej używane skróty klawiszowe	356
Dodatek B. Narzędzia zewnętrzne używane wraz z Unity		359
B.1.	Narzędzia programistyczne	359
B.1.1.	<i>Visual Studio</i>	359
B.1.2.	<i>Xcode</i>	360
B.1.3.	<i>Android SDK</i>	360
B.1.4.	<i>SVN, Git lub Mercurial</i>	360
B.2.	Aplikacje do grafiki 3D	360
B.2.1.	<i>Maya</i>	361
B.2.2.	<i>3ds Max</i>	361
B.2.3.	<i>Blender</i>	361
B.3.	Edytory grafiki 2D	361
B.3.1.	<i>Photoshop</i>	361
B.3.2.	<i>GIMP</i>	362
B.3.3.	<i>TexturePacker</i>	362
B.4.	Oprogramowanie audio	362
B.4.1.	<i>Pro Tools</i>	362
B.4.2.	<i>Audacity</i>	362

<i>Dodatek C. Modelowanie ławki w programie Blender</i>	363
C.1. Tworzenie geometrii siatki	364
C.2. Mapowanie tekstury w modelu	367
<i>Dodatek D. Zasoby dostępne w internecie</i>	371
D.1. Dodatkowe samouczki	371
D.2. Biblioteki kodu	372
<i>Skorowidz</i>	375

Budowa demo pokazującego przestrzeń 3D

W tym rozdziale:

- Poznanie układu współrzędnych 3D.
- Umieszczanie gracza na scenie.
- Tworzenie skryptu przesuwającego obiekty.
- Implementowanie kontrolki FPS.

Rozdział 1. zakończyliśmy tradycyjnym komunikatem: „Witaj, świecie!”, wyświetlanym po rozpoczęciu nauki nowego narzędzia programistycznego. Możemy więc już przejść do bardziej zaawansowanego projektu Unity, zawierającego grafikę i definiującego pewien poziom interaktywności. Umieścisz obiekty na scenie oraz utworzysz kod pozwalający graczowi na poruszanie się po niej. Przygotowana scena (patrz rysunek 2.1) będzie pod względem koncepcji podobna do sceny z kultowej gry *Doom*, ale bez żadnych potworów. Edytor graficzny w Unity umożliwia nowym użytkownikom szybkie rozpoczęcie tworzenia prototypu 3D bez konieczności wcześniejszego przygotowania dużej ilości kodu odpowiedzialnego na przykład za inicjalizację widoku 3D lub zdefiniowanie pętli generowania danych.

Kuszące może być natychmiastowe rozpoczęcie tworzenia sceny w Unity, zwłaszcza że mamy do czynienia z tak prostym (pod względem koncepcji!) projektem. Jednak zawsze dobrze jest zatrzymać się na chwilę na początku pracy i zaplanować to, co ma zostać zrobione. Takie podejście jest szczególnie ważne teraz, gdy proces tworzenia projektu w Unity jest dla Ciebie nowością.



Rysunek 2.1. Kadr z budowanego tutaj demo (przypominającego grę Doom, ale bez potworów)

2.1. Zanim przystąpisz do pracy...

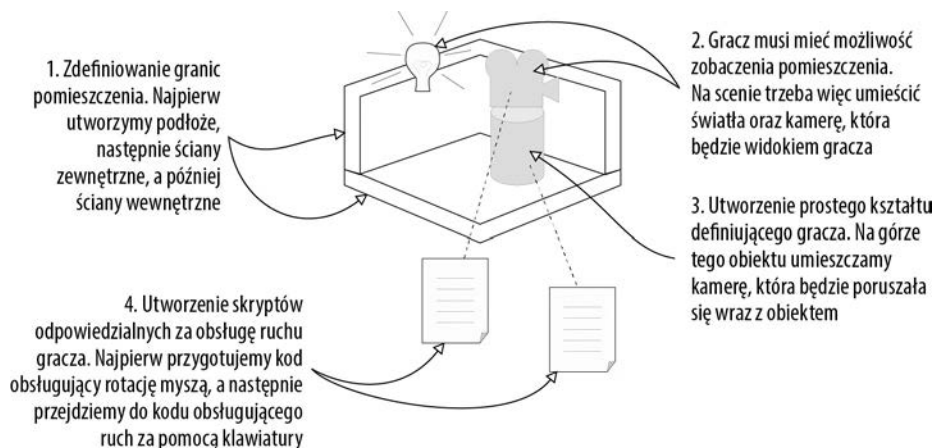
Unity znacznie ułatwia rozpoczęcie pracy początkującym programistom, choć konieczne jest zwrócenie uwagi na kilka kwestii przed przystąpieniem do budowy kompletnej sceny. Nawet podczas pracy z tak elastycznym narzędziem jak Unity niezbędne jest odkrycie celu, który będziesz próbował osiągnąć. Trzeba więc poznać sposób działania współrzędnych 3D, gdyż w przeciwnym razie możesz napotkać przeszkody już na etapie umieszczania obiektu na scenie.

2.1.1. Planowanie projektu

Zanim rozpoczniesz programowanie czegokolwiek, zawsze warto się zatrzymać i zadać sobie pytanie: „Co zamierzam tutaj zbudować?”. Projektowanie gier to niezwykle obszerny temat, któremu poświęcono wiele naprawdę grubych książek. Na szczęście w naszym przypadku wystarczy jedynie przygotowanie w głowie planu tego prostego demo, co umożliwi wykonanie tego niezbyt skomplikowanego projektu treningowego. Takie projekty początkowe i tak nie będą zbyt złożone pod względem układu, aby nie odciągały Twojej uwagi od poznawania koncepcji programistycznych. Zagadnieniami projektowymi na wyższym poziomie możesz (i powinieneś) zajmować się dopiero po opanowaniu podstaw projektowania gier wideo.

W prezentowanym tutaj projekcie budujemy prostą scenę **FPS** (ang. *first-person shooter*). Scena zawiera pomieszczenie, po którym będzie poruszał się gracz. Z kolei gracz będzie widział świat z perspektywy postaci, którą kieruje w grze. Do sterowania nią zostaną wykorzystane mysz i klawiatura. Wszystkie interesujące złożone aspekty kompletnej gry możemy teraz odłożyć na bok, aby skoncentrować się na podstawowej mechanice, czyli poruszaniu się w przestrzeni 3D. Na rysunku 2.2 pokazałem mapę drogową dla omawianego tutaj projektu. Jest to praktycznie ułożona przeze mnie w głowie lista rzeczy do zrobienia.

1. Przygotowanie pomieszczenia: utworzenie podłoża oraz ścian zewnętrznych i wewnętrznych.
2. Umieszczenie na scenie świateł i kamery.
3. Utworzenie obiektu gracza i umieszczenie na nim kamery.
4. Utworzenie skryptów odpowiedzialnych za obsługę ruchu: rotacji za pomocą myszy, przesuwania za pomocą klawiatury.



Rysunek 2.2. Mapa drogową dla naszego demo 3D

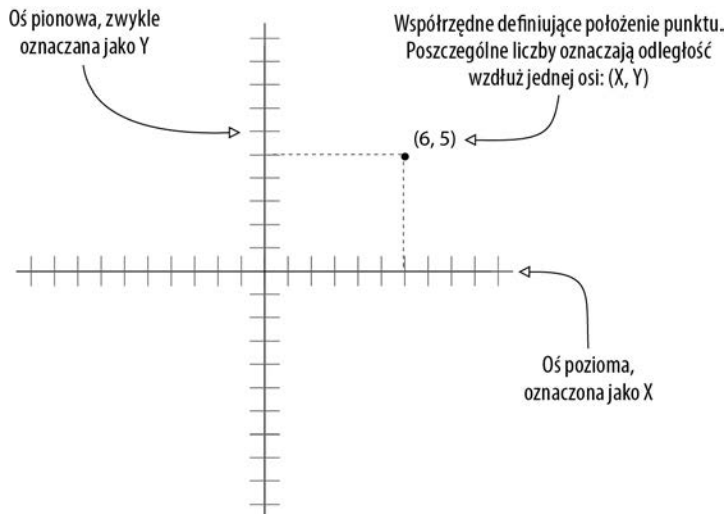
Nie przerażaj się zawartością przedstawionej powyżej mapy drogowej! Wprawdzie wygląda na to, że czeka nas dużo pracy, ale na szczęście Unity bardzo nam wszystko ułatwi. Kolejne punkty rozdziału dotyczące skryptów odpowiedzialnych za obsługę ruchu są dość obszerne, ale tylko dlatego, że przeanalizujemy poszczególne wiersze skryptów, aby dokładnie poznać i zrozumieć stojące za nimi koncepcje. Ten projekt prezentuje demo prostej gry typu FPS w celu zachowania minimalnych wymagań dotyczących grafiki używanej w grze. Ponieważ nie możesz zobaczyć siebie w grze, nie ma żadnych przeciwwskazań, aby postać gracza była przedstawiona jako cylindryczny kształt z kamerą umieszczoną na górze. Teraz musisz jedynie poznać sposób działania współrzędnych 3D, a umieszczenie obiektów na scenie za pomocą edytora graficznego będzie łatwizną.

2.1.2. Poznajemy układ współrzędnych 3D

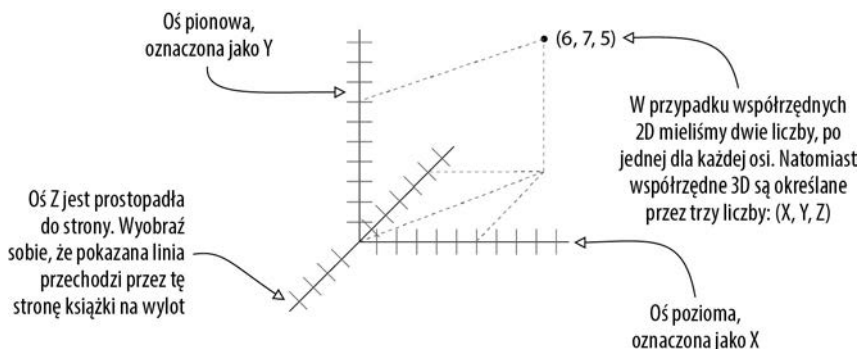
Przypomnij sobie prosty plan, od którego ułożenia rozpoczęliśmy pracę. Trzeba zwrócić uwagę na trzy elementy: pomieszczenie, widok i kontrolki. W przypadku tych wszystkich elementów trzeba wiedzieć, jak ich położenie i ruch będą przedstawiane w trójwymiarowych symulacjach komputerowych. Jeżeli dopiero rozpoczynasz pracę z grafiką 3D, pewne koncepcje mogą być Ci jeszcze nieznanne.

Tak naprawdę wszystko sprowadza się do liczb określających punkty w przestrzeni oraz do sposobu korelacji tych liczb z przestrzenią poprzez osie współrzędnych. Na lekcjach matematyki w szkole poznałeś i stosowałeś pokazane na rysunku 2.3 osie X i Y przeznaczone do przypisywania współrzędnych punktom w tak zwanym układzie współrzędnych kartezjańskich.

Dwie osie dają dwuwymiarowy układ współrzędnych, w którym wszystkie punkty znajdują się na tej samej płaszczyźnie. Z kolei trzy osie są wykorzystywane do zdefiniowania przestrzeni trójwymiarowej. Ponieważ oś X biegnie poziomo wzdłuż strony, a oś Y biegnie pionowo wzdłuż strony, wyobraź sobie teraz, że trzecia oś przechodzi przez tę stronę na wylot i jest prostopadła do osi X i Y. Na rysunku 2.4 pokazałem



Rysunek 2.3.
Współrzędne na osi X i Y definiują punkt 2D



Rysunek 2.4. Współrzędne dla osi X, Y i Z definiują punkt 3D

osie X, Y i Z tworzące trójwymiarowy układ współrzędnych. Wszystkie elementy znajdujące się w pewnym miejscu na scenie — miejsce umieszczenia gracza, położenie ściany itd. — będą miały współrzędne XYZ.

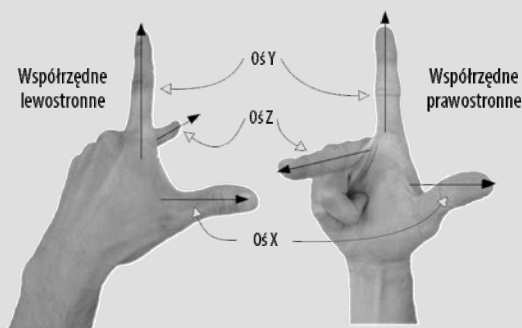
W widoku *Scene* w Unity możesz zobaczyć wyświetlone trzy osie definiujące układ współrzędnych 3D. Z kolei w panelu *Inspector* masz możliwość wpisania trzech liczb definiujących położenie danego obiektu w przestrzeni 3D. Nie tylko będziesz tworzyć kod źródłowy umieszczający obiekty za pomocą trzech współrzędnych, ale również możesz zdefiniować ruch jako odległość do pokonania wzdłuż każdej z trzech osi.

Skoro już wiesz, jak ma wyglądać budowany tutaj projekt, i poznałeś współrzędne używane do umieszczania obiektów w przestrzeni trójwymiarowej, możemy przystąpić do rozpoczęcia pracy nad projektem.

Współrzędne lewostronne kontra prawostronne

Kierunek poszczególnych osi jest dowolny, a współrzędne nadal będą działały, niezależnie od kierunku wskazywanego przez daną oś. Po prostu musisz zachować spójność podczas pracy z danym narzędziem przeznaczonym do grafiki trójwymiarowej (programem do animacji, aplikacją do tworzenia gier itd.).

Jednak w większości przypadków wartości osi X zwiększają się w prawą stronę, a osi Y rosną w górę. Różnice między poszczególnymi narzędziami wiążą się z osią Z i kierunkiem, w którym zwiększają się jej wartości: w stronę kartki i od kartki. Te dwa kierunki są nazywane (odpowiednio) lewostronnym i prawostronnym, jak pokazałem na poniższym rysunku. Jeżeli umieścisz kciuk wzdłuż osi X, a palec wskazujący wzdłuż osi Y, wtedy palec środkowy znajduje się wzdłuż osi Z.



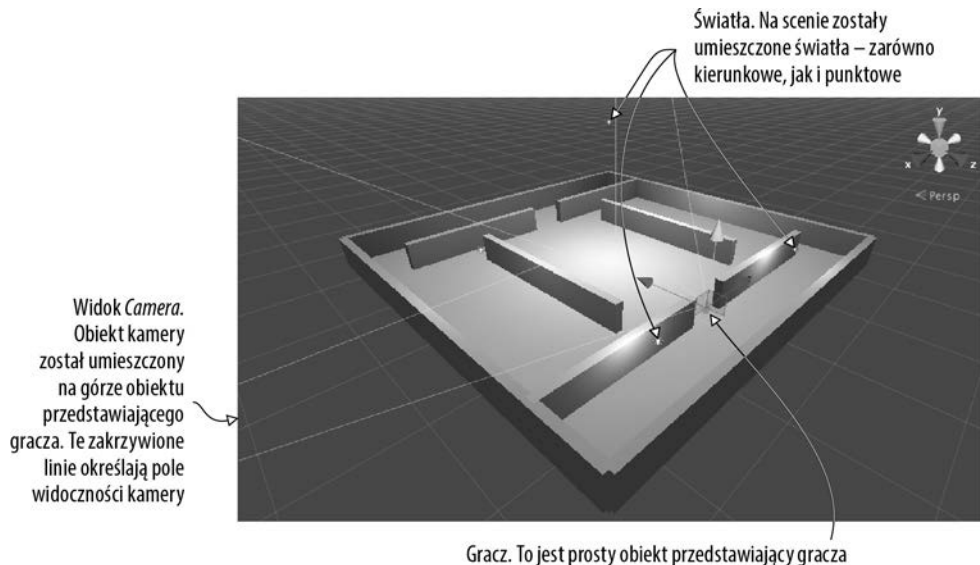
Kierunek wskazywany przez oś Z zależy od używanej dłoni

Podobnie jak wiele aplikacji przeznaczonych do pracy z grafiką 3D, także Unity używa lewostronnego układu współrzędnych. Mamy również sporą grupę narzędzi korzystających z prawostronnego układu współrzędnych, na przykład z OpenGL, więc nie bądź zdziwiony, gdy spotkasz się z innym kierunkiem współrzędnych.

2.2. Rozpoczęcie projektu — umieszczenie obiektów na scenie

W porządku, przystępujemy do utworzenia i umieszczenia obiektów na scenie. Pracę zaczniemy od przygotowania wszystkich elementów statycznych, którymi w budowanym tutaj projekcie są podłóżo i ściany. Następnie zajmiemy się ułożeniem świateł i kamery. Dopiero na końcu utworzymy obiekt przedstawiający gracza, do którego dołączymy skrypty pozwalające graczowi na poruszanie się po scenie. Na rysunku 2.5 pokazałem wygląd edytora graficznego Unity po zakończeniu tego etapu pracy nad projektem.

W rozdziale 1. dowiedziałeś się, jak utworzyć nowy projekt w Unity. Teraz masz okazję wykorzystać tę wiedzę w praktyce. Przypominam, że należy wybrać opcję menu *File/New Project*, a następnie w wyświetlonym oknie dialogowym podać nazwę dla projektu. Po utworzeniu projektu natychmiast zapisz aktualnie pustą scenę domyślną, ponieważ początkowo projekt nie zawiera żadnego pliku sceny. Najpierw scena będzie pusta, a pierwsze obiekty do utworzenia są bardzo oczywiste.



Rysunek 2.5. Wyświetlona w edytorze Unity scena wraz z podłożem, ścianami, światłami, kamerą i graczem

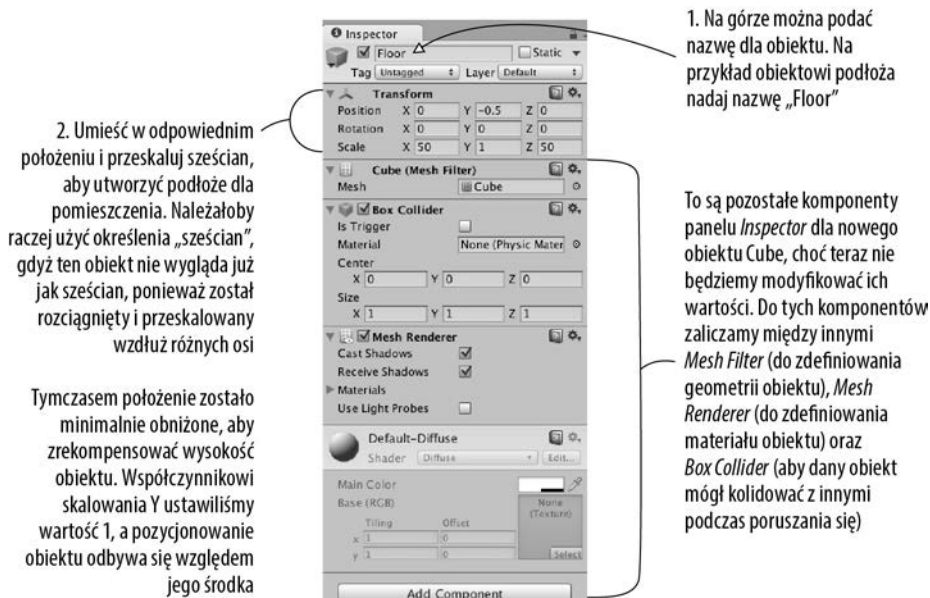
2.2.1. Sceneria – podłoże, ściany zewnętrzne i wewnętrzne

Kliknij menu *GameObject* na górze ekranu, a następnie przejdź do podmenu *3D Object*, aby zobaczyć listę dostępnych opcji. Wybierz *Cube* w celu utworzenia na scenie nowego obiektu sześcianu; później będziemy używać jeszcze innych kształtów, takich jak *Sphere* i *Capsule*. Dostosuj położenie nowego obiektu i przeskaluj go, a także nadaj mu nazwę, aby w ten sposób utworzyć podłoże. Na rysunku 2.6 pokazałem wartości, jakie dla podłoża powinny być zdefiniowane w panelu *Inspector* (na początku mieliśmy sześcian, który został później spłaszczony).

UWAGA Liczby określające położenie mogą być wyrażone w dowolnych jednostkach, o ile będziesz je konsekwentnie stosować na scenie. Najczęściej wybieraną jednostką (także przeze mnie) jest metr, czasami używana jest stopa, spotkałem się także z sytuacjami, w których stosowane były cale!

Te same kroki trzeba powtórzyć w celu utworzenia ścian zewnętrznych pomieszczenia. Możesz budować nowy sześcian za każdym razem lub też kopiować i wklejać istniejące obiekty, używając do tego standardowych skrótów. Przesuń, obróć i przeskaluj ściany w taki sposób, aby utworzyły obrzeża dla podłoża, jak pokazałem na rysunku 2.5. Poeksperymentuj z różnymi wartościami, na przykład 1, 4 i 50 dla skalowania, lub wykorzystaj omówione w punkcie 1.2.2 narzędzia przeznaczone do przeprowadzania transformacji. (Nie zapominaj, że matematycznym określeniem przesuwania i rotacji w przestrzeni 3D jest „transformacja”).

WSKAZÓWKA Przypomnij sobie również o kontrolkach nawigacyjnych, za pomocą których można oglądać scenę pod różnymi kątami oraz na odmiennych



Rysunek 2.6. Panel Inspector z wartościami dla podłoża

poziomach przybliżenia i oddalenia. Jeżeli kiedykolwiek zgubisz się na scenie, naciśnięcie klawisza *F* spowoduje wyzerowanie widoku i wyświetlenie obecnie wybranego obiektu.

Dokładne wartości potrzebne do wykonania transformacji ścian będą zależały od tego, jak przeprowadzisz rotację i skalowanie sześcianów, a także od sposobu połączenia obiektów ze sobą w panelu *Hierarchy*. Na przykład na rysunku 2.7 możesz zobaczyć, że ściany są elementami potomnymi obiektu głównego. Dlatego też lista wyświetlana przez ten panel wygląda na zorganizowaną. Jeżeli chcesz skopiować dokładne wartości użyte przeze mnie, pobierz przykładowy projekt z materiałów przygotowanych dla tej książki i sprawdź wartości, które zastosowałem podczas tworzenia ścian.



Rysunek 2.7. Panel Hierarchy pokazuje ściany i podłoże zorganizowane jako elementy potomne obiektu głównego

WSKAZÓWKA Przeciągnięcie obiektów na siebie w panelu *Hierarchy* powoduje zdefiniowanie połączeń między nimi. Obiekt, który ma dołączone do siebie inne obiekty, jest określany mianem **obektu nadrzędnego**. Z kolei obiekt dołączony do innego obiektu jest nazywany **obiektem potomnym**. Podczas przesuwania (bądź też rotacji lub skalowania) obiektu nadrzędnego jego obiekty potomne również ulegają transformacji.

WSKAZÓWKA Puste obiekty gry można wykorzystać do organizacji sceny w pokazany sposób. Dzięki połączeniu widocznych obiektów z obiektem głównym

dotyczący ich fragment listy wyświetlanej w panelu *Hierarchy* można zwinąć. Pamiętaj jednak o tym, że przed dołączeniem jakichkolwiek obiektów potomnych do obiektu głównego pusty obiekt główny należy umieścić w położeniu (0, 0, 0), aby uniknąć później wszelkich dziwactw podczas jego transformacji.

Co to jest GameObject?

Wszystkie obiekty na scenie są egzemplarzami klasy `GameObject`, podobnie jak wszystkie komponenty skryptów dziedziczą po klasie `MonoBehaviour`. Było to znacznie bardziej widoczne w przypadku pustego obiektu o nazwie `GameObject`. Jednak nazwa obiektu nie ma tutaj żadnego znaczenia; równie dobrze moglibyśmy nadać mu nazwę taką jak `Floor`, `Camera` lub `Player`.

`GameObject` to tak naprawdę kontener przeznaczony do przechowywania wielu komponentów. Głównym celem istnienia `GameObject` jest dostarczanie elementu, do którego `MonoBehaviour` może coś dołączyć. Dokładna postać obiektu na scenie zależy od tego, jakie komponenty zostały dodane do `GameObject`. Na przykład obiekt `Cube` ma komponent `Cube`, obiekt `Sphere` ma komponent `Sphere` itd.

Po utworzeniu ścian zewnętrznych powinieneś przygotować także ściany wewnętrzne. Rozmieść je dowolnie, wedle własnego upodobnia. Idea polega na przygotowaniu pewnych korytarzy i przeszkód, które trzeba będzie pokonywać po opracowaniu kodu odpowiedzialnego za poruszanie graczem.

W ten sposób przygotowaliśmy pomieszczenie na scenie, ale bez jakiegokolwiek światła gracz nie będzie w stanie niczego zobaczyć. Dlatego kolejnym zadaniem jest dodanie światła.

2.2.2. Światła i kamera

Scena 3D jest oświetlana najczęściej za pomocą światła kierunkowego oraz serii światła punktowych. Najpierw rozpocznij od ustawienia światła kierunkowego. Scena prawdopodobnie domyślnie ma już tego rodzaju światło, ale jeśli nie, utwórz je przez przejście do menu *GameObject/Light* i wybranie opcji *Directional Light*.

Rodzaje światła

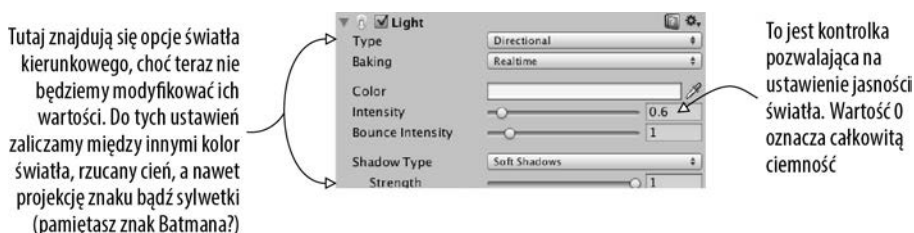
Istnieje możliwość utworzenia wielu różnych typów światła zdefiniowanych na podstawie tego, jak i gdzie emitują one promienie światła. Do dyspozycji mamy trzy rodzaje światła: punktowe, reflektor i kierunkowe.

Światło punktowe (ang. *point light*) to taki rodzaj światła, w którym wszystkie promienie wywodzą się z pojedynczego punktu i są emitowane we wszystkich kierunkach, podobnie jak w przypadku żarówki używanej w rzeczywistości. Z bliskiej odległości to światło jest jaśniejsze, ponieważ promienie światła tworzą silniejszą wiązkę.

Światło reflektora (ang. *spot light*) to taki rodzaj światła, w którym wszystkie promienie wywodzą się z pojedynczego punktu, ale są skoncentrowane na pewnym punkcie. Takiego światła nie będziemy używać w bieżącym projekcie, ale jest ono powszechnie stosowane do pokazania pewnych określonych fragmentów poziomu gry.

Światło kierunkowe (ang. *directional light*) to taki rodzaj światła, w którym wszystkie promienie są równoległe i rozchodzą się równo, oświetlając w ten sam sposób wszystko to, co znajduje się na scenie. Takie światło można porównać do światła słonecznego.

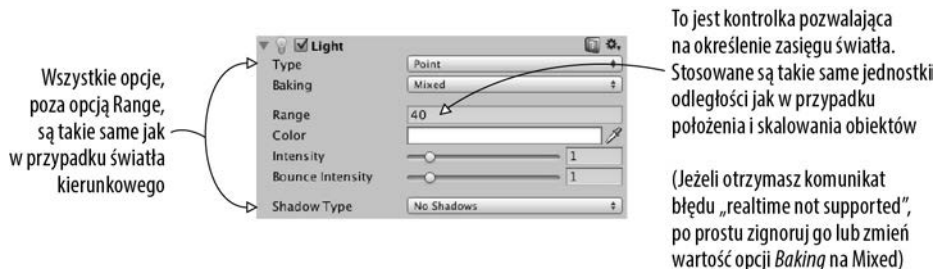
Ponieważ inaczej niż w przypadku rotacji położenie światła kierunkowego nie wpływa na kierowane przez niego promienie światła, z technicznego punktu widzenia można je umieścić w dowolnym miejscu na scenie. Zalecam umieszczenie tego światła ponad pomieszczeniem, aby intuicyjnie było odbierane jako słońce, a ponadto nie przeszkadzało podczas pracy nad pozostałą częścią sceny. Obróć to światło i zwróć uwagę na to, jaki ta zmiana ma wpływ na pomieszczenie. Proponuję przeprowadzić niewielką rotację wzdłuż osi X i Y, co pozwoli na osiągnięcie dobrego efektu. Ustawienia intensywności (ang. *Intensity*) znajdują się w panelu *Inspector* (patrz rysunek 2.8). Jak sama nazwa wskazuje, te ustawienia decydują o jasności światła. Jeżeli na scenie znajduje się tylko jedno światło, powinno być bardziej intensywne. Jednak w omawianym tutaj projekcie dodamy więcej światła, więc kierunkowe nie musi być zbyt jasne — wystarczy wartość 0.6 dla opcji *Intensity*.



Rysunek 2.8. Ustawienia światła kierunkowego wyświetlane w panelu *Inspector*

Światło punktowe tworzymy za pomocą tego samego menu: *GameObject/Light*. Na scenie umieść kilka tego rodzaju światła, aby mieć pewność, że wszystkie ściany zostały oświetlone. Nie przesadzaj z liczbą tych światła, ponieważ jeżeli w grze umieścić zbyt wiele światła, jej wydajność ulegnie znacznemu pogorszeniu. W zupełności wystarczające powinno być ustawienie po jednym świetle punktowym w poszczególnych rogach; sugeruję przy tym podniesienie miejsca umieszczenia światła ponad krawędź ściany. Dobrze jest umieścić jedno wysoko ponad sceną (na przykład wartość współrzędnej osi Y dla tego światła może wynosić 18) i tym samym zapewnić pewne zróżnicowanie światła w pomieszczeniu. Zwróć uwagę na to, że światło punktowe ma opcję *Range* w panelu *Inspector*, jak pokazałem na rysunku 2.9. Wymieniona opcja kontroluje odległość, na jaką dociera światło. Warto pamiętać, że światło kierunkowe świeci równo na całej scenie, a punktowe jest jaśniejsze, gdy obiekt znajduje się bliżej. Światło punktowe znajdujące się bliżej podłoża powinno mieć zasięg wynoszący 18, a światło umieszczone wysoko nad sceną powinno mieć zasięg 40, aby objąć całe pomieszczenie.

Ostatni rodzaj obiektu niezbędny do tego, aby gracz mógł zobaczyć scenę, to kamera. Skoro „pusta” scena jest domyślnie dostarczana wraz z kamerą główną, wykorzystamy ją. Jeżeli kiedykolwiek zajdzie potrzeba utworzenia nowej kamery (na przykład w przypadku podziału widoku w grach przeznaczonych dla wielu graczy), to pamiętaj, że wspomniane już wcześniej menu *GameObject* oprócz opcji takich jak *Cube* i *Light* ma również opcję *Camera*. Kamerę umieścimy na górze obiektu przedstawiającego gracza, aby otrzymać widok na wysokości oczu gracza.

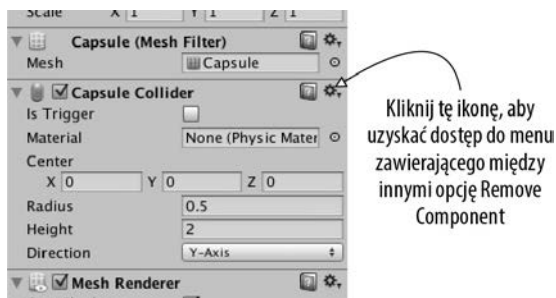


Rysunek 2.9. Ustawienia światła punktowego wyświetlane w panelu Inspector

2.2.3. Punkt widzenia i kolizje gracza

Na potrzeby tego projektu gracz zostanie przedstawiony za pomocą prostego kształtu. W menu *GameObject/3D Object* wybierz opcję *Capsule*. Unity utworzy kształt cylindra wraz z zaokrąglonymi końcami; ten prosty kształt wykorzystamy do przedstawienia gracza w budowanej tutaj grze. Temu obiektowi przypisz wartość 1.1 dla współrzędnej Y (połowa wysokości obiektu i trochę, tak aby uniknąć nakładania się na podłoże). Obiektem gracza można poruszać dowolnie wzdłuż osi X i Z, o ile gracz pozostanie w pomieszczeniu i nie będzie dotykał żadnych ścian. Obiektowi nadaj nazwę *Player*.

W panelu *Inspector* zwróć uwagę na to, że obiekt przedstawiający gracza ma dołączony komponent *Capsule Collider*. To jest logiczna decyzja w przypadku obiektu *Capsule*. Na przykład obiekt *Cube* domyślnie otrzymuje komponent *Cube Collider*. Skoro obiekt *Capsule* reprezentuje w tej grze gracza, wymaga nieco innego zestawu komponentów niż większość obiektów. Usuń komponent *Capsule Collider*. W tym celu kliknij ikonę koła zębatego widoczną w prawym górnym rogu komponentu (patrz rysunek 2.10), a następnie z wyświetlonego menu wybierz opcję *Remove Component*. Ponieważ ten komponent jest przedstawiony w postaci zielonej siatki wokół obiektu, po usunięciu komponentu wspomniana siatka również zniknie.



Rysunek 2.10. Usunięcie komponentu za pomocą panelu Inspector

Zamiast *Capsule Collider* obiektowi gracza przypiszemy komponent *Character Controller*. Na dole panelu *Inspector* znajduje się przycisk o nazwie *Add Component*. Po jego kliknięciu zobaczysz menu komponentów, które można dodać do obiektu. W sekcji *Physics* wyświetlonego menu odszukaj opcję *Character Controller* i wybierz ją. Jak sama nazwa wskazuje, dodanie tego komponentu do obiektu powoduje, że będzie się on zachowywał jak postać.

Konieczne jest wykonanie jeszcze jednego kroku podczas przygotowywania obiektu gracza, czyli dołączenie kamery. Jak wcześniej wspomniałem, obiekty można przeciągać na siebie za pomocą panelu *Hierarchy*. Przeciągnij więc obiekt kamery na obiekt *Capsule* przedstawiający gracza. Następnie umieść kamerę w taki sposób, aby znajdowała się na wysokości oczu gracza — sugeruję położenie $(0, 0.5, 0)$. Jeżeli to konieczne, wyzeruj rotację kamery do wartości $(0, 0, 0)$.

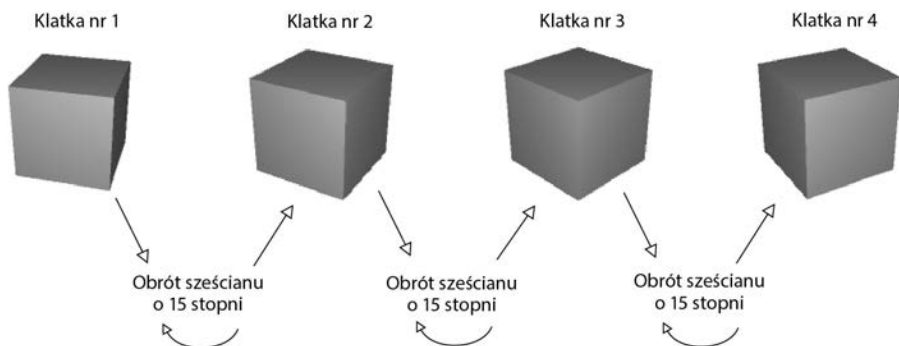
W ten sposób utworzyliśmy wszystkie obiekty niezbędne w budowanej scenie. Pozostało już tylko napisanie kodu źródłowego odpowiedzialnego za poruszanie obiektem gracza.

2.3. Poruszanie obiektami — skrypt pozwalający na zastosowanie transformacji

Aby umożliwić graczowi poruszanie się po scenie, konieczne jest dołączenie do obiektu gracza skryptów pozwalających na wykonywanie ruchu. Pamiętaj, że komponenty to dodawane do obiektów modułowe fragmenty funkcjonalności, a skrypt jest rodzajem komponentu. Ostatecznie te skrypty będą reagowały na dane wejściowe dostarczane przez klawiaturę i mysz. Zaczniemy jednak od obrotu gracza w miejscu. Dzięki temu zobaczysz, jak wygląda zastosowanie transformacji w kodzie źródłowym. Nie zapominaj o tym, że mamy trzy dostępne transformacje: translację, rotację i skalowanie. Obracanie obiektu oznacza wykorzystanie rotacji. Jednak o tym zadaniu trzeba wiedzieć nieco więcej niż zdawkowe: „wymaga rotacji”.

2.3.1. Diagram pokazujący sposób programowania ruchu

Animacja obiektu (na przykład jego obracanie) sprowadza się do niewielkiego przesunięcia go w trakcie każdej klatki, które następnie będą odtwarzane w nieskończoność. Transformacje mają zastosowanie natychmiast, w przeciwieństwie do ruchu widocznego na przestrzeni jakiegoś czasu. Jednak nieustanne używanie transformacji powoduje, że zauważamy ruch obiektu. Można to porównać do serii rysunków w kinetografie, które szybko przekartkowane dają złudzenie animacji. Sposób działania tego rodzaju rozwiązania pokazałem na rysunku 2.11.



Rysunek 2.11. Przedstawienie procesu ruchu — cykliczny proces transformacji między nieruchomymi rysunkami

Przypomnij sobie, że komponent skryptu ma metodę o nazwie `Update()`, wykonywaną w trakcie każdej klatki. Jeżeli chcesz obrócić sześcian, wystarczy w tej metodzie zdefiniować kod obracający sześcian o niewielką liczbę stopni. Tak przygotowany kod będzie działał w nieskończoność w trakcie każdej klatki. Opisane rozwiązanie wydaje się całkiem proste, prawda?

2.3.2. Utworzenie kodu źródłowego implementującego ruch

Przechodzimy teraz do implementacji omówionej powyżej koncepcji. Utwórz nowy skrypt w C# (pamiętaj o wykorzystaniu podmenu *Create* w menu *Assets*), nadaj mu nazwę *Spin* i umieść w nim kod przedstawiony na listingu 2.1. Nie zapomnij o zapisaniu pliku po zakończeniu wprowadzania kodu.

Listing 2.1. Skrypt zapewniający obracanie się obiektu

```
using UnityEngine;
using System.Collections;

public class Spin : MonoBehaviour {
    public float speed = 3.0f;

    void Update() {
        transform.Rotate(0, speed, 0);
    }
}
```

Deklaracja zmiennej publicznej określającej szybkość obracania się obiektu. Litera *f* umieszczona po liczbie nakazuje komputerowi potraktowanie tej liczby jako wartości zmiennoprzecinkowej typu float. Jeżeli pominiemy literę *f*, język C# domyślnie potraktuje liczbę zmiennoprzecinkową jako typu double, czyli podwójnej precyzji.

Wywołanie metody `Rotate()` umieszczamy tutaj, aby było wykonywane w trakcie każdej klatki.

W celu dodania komponentu skryptu do obiektu gracza należy go przeciągnąć z panelu *Project* i upuścić na obiekcie *Player* w panelu *Hierarchy*. Teraz po naciśnięciu przycisku *Start* zobaczysz, że widok się obraca. W ten sposób przygotowałeś kod wprawiający obiekt w ruch! Ten nowy kod składa się w większości z kodu szablonu domyślnego dla nowego skryptu oraz dwóch nowych wierszy. Przeanalizujemy więc działanie tych dwóch nowych wierszy.

Pierwszy dodany wiersz zawiera deklarację zmiennej określającej szybkość obracania się obiektu. Tę zmienną umieściliśmy na początku definicji klasy. Mamy dwa powody zdefiniowania szybkości obracania się obiektu jako zmiennej. Pierwszy to standardowa reguła programowania: „żadnych magicznych liczb”. Drugi wiąże się ze specyfiką, z jaką Unity wyświetla zmienne publiczne. Unity wykonuje pewne użyteczne zadania związane ze zmiennymi publicznymi umieszczanymi w komponentach skryptów, jak to wyjaśniłem w poniższej wskazówce.

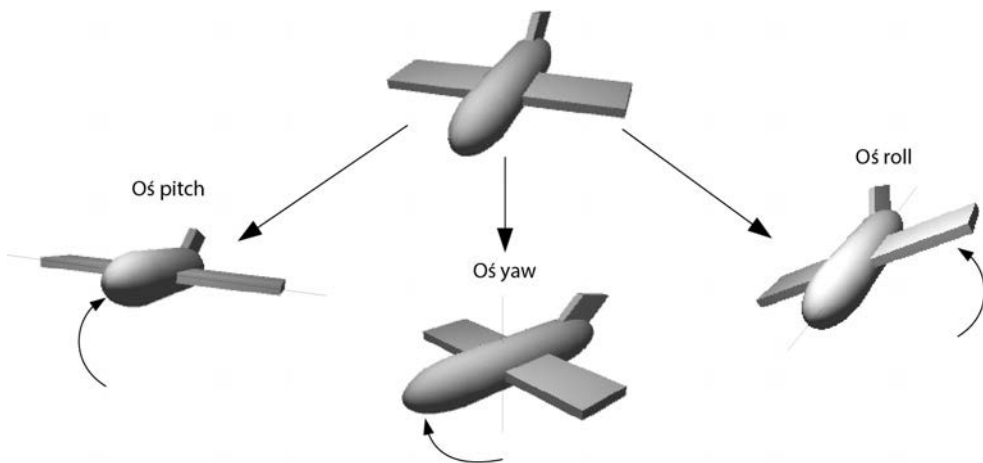
WSKAZÓWKA Zmienne publiczne są udostępniane w panelu *Inspector*, co pozwala na zmianę wartości komponentu po jego dodaniu do obiektu gry. Ten mechanizm nazywany jest serializacją wartości, ponieważ Unity zachowuje zmodyfikowany stan zmiennej.

Na rysunku 2.12 pokazałem wygląd omawianego komponentu skryptu w panelu *Inspector*. Możesz wpisać nową wartość, która zostanie użyta zamiast domyślnej wartości zdefiniowanej w kodzie. Otrzymujemy więc wygodny sposób na dostosowanie ustawień

komponentów w różnych obiektach poprzez możliwość pracy za pomocą edytora graficznego zamiast używania wartości definiowanych na stałe w kodzie.

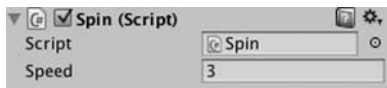
Drugi wiersz kodu dodany w skrypcie przedstawionym na listingu 2.1 to wywołanie metody `Rotate()`. Ponieważ znajduje się ono w metodzie `Update()`, metoda `Rotate()` będzie wykonywana w trakcie każdej klatki. Metoda `Rotate()` pochodzi z klasy `Transform` i jest wywoływana przez użycie notacji z kropką w komponencie skryptu dla danego obiektu. (W większości języków programowania zorientowanego obiektowo przyjmowane jest założenie o wykorzystaniu `this.transform` po napotkaniu wywołania `transform`). Zastosowana tutaj transformacja powoduje obrót o podaną liczbę stopni (`speed`) w trakcie każdej klatki, a skutkiem jest płynny obrót obiektu. Mógłbyś w tym miejscu zapytać, dlaczego w nawiasie wywołania `Rotate()` mamy `(0, speed, 0)` zamiast, powiedzmy, `(speed, 0, 0)`.

Przypomnij sobie o istnieniu trzech osi w przestrzeni 3D, oznaczonych jako X, Y i Z. Dość łatwo można zrozumieć, jak te osie przekładają się na położenie i ruch. Jednak te osie mogą być wykorzystane także do opisanego rotacji. W aeronautyce rotacja jest opisywana w podobny sposób, a programiści zajmujący się grafiką 3D bardzo często posługują się terminami zapożyczonymi właśnie z aeronautyki: *pitch*, *yaw* i *roll*. Na rysunku 2.13 możesz zobaczyć, co oznaczają te pojęcia — *pitch* to rotacja względem osi X, *yaw* to rotacja względem osi Y, a *roll* to rotacja względem osi Z.



Rysunek 2.13. Osie pitch, yaw i roll w aeronautyce

Biorąc pod uwagę możliwość opisanego rotacji względem osi X, Y i Z, można powiedzieć, że trzema parametrami metody `Rotate()` są wartości dla wymienionych osi. Ponieważ chcemy jedynie obracać graczem na boki, a nie w górę i w dół, zmienia się wartość dla osi Y, a dla osi X i Z wartością pozostaje 0. Chyba domyślasz się już, co się stanie po zmianie parametrów na `(speed, 0, 0)` i uruchomieniu gry. Zrób to i przekonaj się sam!



Rysunek 2.12. Panel Inspector wyświetla wartość zmiennej publicznej zadeklarowanej w kodzie skryptu

Do omówienia mamy jeszcze jedną subtelną kwestię dotyczącą rotacji i osi współrzędnych w przestrzeni 3D. Jest to opcjonalny, czwarty parametr metody `Update()`.

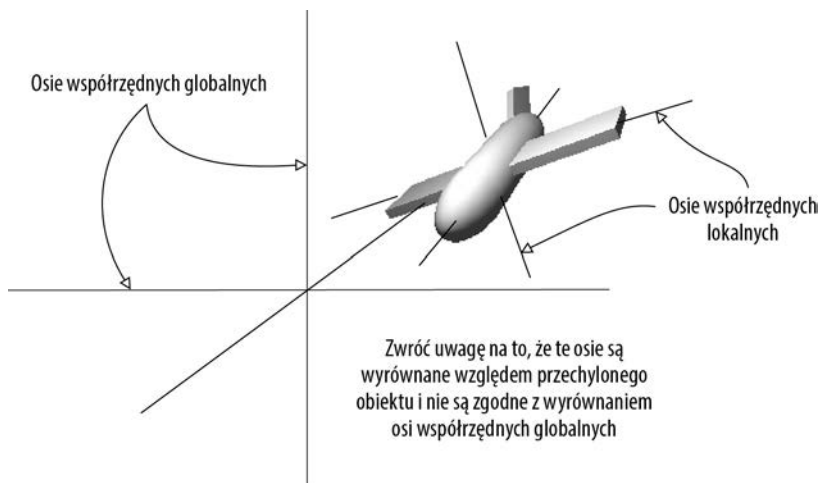
2.3.3. Współrzędne lokalne kontra globalne

Domyślnie metoda `Rotate()` operuje na tak zwanych współrzędnych lokalnych. Oprócz nich istnieją jeszcze współrzędne globalne. Konieczność użycia współrzędnych lokalnych lub globalnych można wskazać metodzie za pomocą opcjonalnego, czwartego parametru. W omawianym przykładzie będzie to `Space.Self` lub `Space.World`, jak pokażę poniżej.

```
Rotate(0, speed, 0, Space.World)
```

Powróćmy jeszcze na chwilę do wcześniejszego wyjaśnienia dotyczącego współrzędnych przestrzeni 3D i zastanówmy się nad tym, gdzie znajduje się punkt określony jako $(0, 0, 0)$. W którym kierunku jest zwrócona oś X? Czy układ współrzędnych sam w sobie może się poruszać?

Okazuje się, że każdy obiekt ma swój punkt początkowy, jak również kierunek dla trzech wymienionych osi, a układ współrzędnych porusza się wraz z obiektem. W takim przypadku mamy do czynienia z tak zwanymi współrzędnymi lokalnymi. Cała scena 3D także ma własny punkt początkowy i kierunek dla trzech osi, ale jej układ współrzędnych nigdy się nie porusza. W takim przypadku mówimy o współrzędnych globalnych. Dlatego też po podaniu współrzędnych lokalnych lub globalnych metodzie `Rotate()` po prostu wskazujesz jej, według których osi X, Y i Z ma być przeprowadzona rotacja (patrz rysunek 2.14).



Rysunek 2.14. Osie współrzędnych lokalnych kontra osie współrzędnych globalnych

Jeżeli dopiero zaczynasz pracę z grafiką 3D, przedstawiona koncepcja może na początku wydawać się nieco dziwna. Poszczególne osie zostały pokazane na rysunku 2.14 (zauważ, że „lewa” strona względem samolotu ma inny kierunek niż „lewa” strona sceny). Jednak

najłatwiejszym sposobem na zrozumienie zagadnienia współrzędnych lokalnych i globalnych jest praca z konkretnym przykładem.

Najpierw zaznacz obiekt przedstawiający gracza, a następnie nieco go przechył (na przykład o mniej więcej 30 stopni względem osi X). To spowoduje zmianę współrzędnych lokalnych i dlatego rotacje lokalna i globalna będą teraz wyglądały inaczej. Spróbuj uruchomić skrypt *Spin* wraz z czwartym parametrem o wartości `Space.World` oraz bez niego. Jeżeli masz trudności z wizualizacją tego, co się dzieje w grze, usuń komponent skryptu z obiektu gracza i zamiast tego spróbuj obrócić przechylony sześciąt umieszczony przed graczem. Powinieneś zobaczyć, że obiekt obraca się wokół różnych osi, w zależności od tego, które ze współrzędnych (lokalne czy globalne) są używane.

2.4. Skrypt `MouseLook` pozwalający na rozglądanie się

Teraz zajmiemy się zdefiniowaniem rotacji w odpowiedzi na ruch wykonywany myszą (chodzi o rotację obiektu, do którego został dołączony skrypt; w omawianym przykładzie będzie to obiekt gracza). W tym celu wykonamy wiele czynności i progresywnie będziemy dodawać postaci w grze kolejne możliwości w zakresie poruszania się. Najpierw gracz będzie mógł obracać się jedynie na boki, a dopiero później dodamy możliwość rotacji w górę i w dół. Ostatecznie gracz zyska możliwość rozglądania się we wszystkich kierunkach (jednoczesna rotacja pozioma i pionowa) — tego rodzaju zachowanie jest określane mianem *mouse-look*.

Biorąc pod uwagę to, że mamy do czynienia z trzema różnymi rodzajami zachowania rotacji (poziomą, pionową oraz w obu kierunkach jednocześnie), pracę rozpoczynamy od utworzenia frameworka przeznaczonego do obsługi wszystkich trzech rodzajów. Utwórz nowy skrypt C#, nadaj mu nazwę *MouseLook*, a następnie umieść w nim kod przedstawiony na listingu 2.2.

Listing 2.2. Framework `MouseLook` wraz z typem wycieniowym dla ustawienia `Rotation`

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
    public enum RotationAxes {
        MouseXAndY = 0,
        MouseX = 1,
        MouseY = 2
    }

    public RotationAxes axes = RotationAxes.MouseXAndY;

    void Update() {
        if (axes == RotationAxes.MouseX) {
            // Miejsce na kod obsługujący rotację poziomą.
        }
        else if (axes == RotationAxes.MouseY) {
            // Miejsce na kod obsługujący rotację pionową.
        }
    }
}
```

← Zdefiniowanie struktury danych w postaci typu wycieniowego w celu powiązania nazwy z ustawieniami.

← Zadeklarowanie zmiennej publicznej przeznaczonej do ustawienia w edytorze Unity.

← Miejsce na umieszczenie kodu obsługującego tylko rotację poziomą.

← Miejsce na umieszczenie kodu obsługującego tylko rotację pionową.

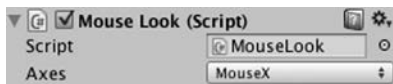
```

    }
    else {
        // Miejsce na kod obsługujący rotację zarówno poziomą, jak i pionową.
    }
}
}

```

←
Miejsce na umieszczenie kodu obsługującego rotację zarówno poziomą, jak i pionową.

Zwróć uwagę na to, że typ wyliczeniowy jest używany do wyboru rodzaju rotacji (poziomej lub pionowej) dla skryptu *MouseLook*. Zdefiniowanie struktury w postaci typu wyliczeniowego pozwala na ustawienie wartości za pomocą nazw, a nie liczb. W tym drugim przypadku trzeba jeszcze pamiętać znaczenie poszczególnych liczb — czy 0 oznacza rotację poziomą? A może pionową? Po zadeklarowaniu zmiennej publicznej jako typu wyliczeniowego w panelu *Inspector* będzie ona wyświetlona w postaci rozwijanego menu (patrz rysunek 2.15), co jest użyteczne podczas wyboru ustawień.



Rysunek 2.15. Zmienna publiczna typu wyliczeniowego jest w panelu *Inspector* wyświetlana jako rozwijane menu

Usuń komponent *Spin* (tak samo jak wcześniej usunąłeś komponent *Capsule Collider*) i zamiast niego do obiektu przedstawiającego gracza dołącz ten nowy skrypt. Za pomocą rozwijanego menu *Axes* w panelu *Inspector* możesz zmieniać kierunek rotacji. Mając przygotowane miejsce na obsługę rotacji poziomej i pionowej, możemy przystąpić do umieszczenia kodu w obu gałęziach konstrukcji warunkowej.

2.4.1. Rotacja pozioma na podstawie ruchu myszy

Pierwsza i najprostsza gałąź kodu źródłowego jest odpowiedzialna za obsługę rotacji poziomej. Pracę rozpocznij od wpisania tych samych poleceń rotacji, które na listingu 2.1 były użyte do wprowadzenia obiektu w ruch. Nie zapomnij o zadeklarowaniu zmiennej publicznej określającej szybkość rotacji. Zadeklaruj więc nową zmienną po *axes*, ale jeszcze przed metodą *Update()*. Tej nowej zmiennej nadaj nazwę *sensitivityHor*, ponieważ *speed* to zbyt ogólna nazwa, gdy w grę wchodzi wiele rotacji. Zmiennej *sensitivityHor* przypisz wartość 9; w tym miejscu potrzebna nam będzie większa wartość, kiedy tylko kod rozpocznie jej skalowanie (tym zajmiemy się za chwilę). Kod źródłowy skryptu na tym etapie powinien wyglądać jak ten pokazany na listingu 2.3.

Listing 2.3. Początkowy kod rotacji poziomej, która jeszcze nie reaguje na ruch myszą

```

...
public RotationAxes axes = RotationAxes.MouseXAndY;
public float sensitivityHor = 9.0f;
void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, sensitivityHor, 0);
    }
}
...

```

← Kod zapisany pochyloną czcionką znajdował się już w skrypcie. Pozostawiłem go tutaj w celu pokazania kontekstu.

← Deklaracja zmiennej publicznej określającej szybkość obracania się obiektu.

← Wywołanie *Rotate()* umieszczamy tutaj, aby było wykonywane w trakcie każdej klatki.

W rozwijanym menu *Axes* komponentu *MouseLook* wybierz opcję oznaczającą rotację poziomą (*MouseX*) i uruchom skrypt. Zobaczysz, że widok będzie się obracał jak wcześniej, tylko znacznie szybciej, ponieważ szybkość rotacji względem osi Y wynosi teraz 9 zamiast 3. Kolejnym krokiem jest zmodyfikowanie rotacji w taki sposób, aby reagowała na ruch myszą. Wprowadzamy więc nową metodę, o nazwie `Input.GetAxis()`. Klasa `Input` zawiera wiele metod przeznaczonych do obsługi urządzeń przekazujących dane wejściowe. Przykładem tego rodzaju urządzenia jest mysz. Wartości zwrótnie metody `GetAxis()` są skorelowane z ruchem myszy i są one ujemne lub dodatnie, w zależności od kierunku ruchu. Ta metoda pobiera parametr w postaci nazwy osi, na przykład oś pozioma jest określana jako *Mouse X*.

Jeżeli wartość dla danej osi pomnożymy przez liczbę określającą szybkość rotacji, wtedy rotacja będzie przeprowadzana jako wynik ruchu myszą. Szybkość będzie skalowana zgodnie z ruchem myszy, do zera lub nawet w przeciwnym kierunku. Wywołanie `Rotate()` powinno teraz wyglądać tak, jak pokazałem na listingu 2.4.

Listing 2.4. Wywołanie `Rotate()` dostosowane tak, aby reagowało na ruch myszą

```
...
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
```

← Zwróć uwagę na użycie metody `GetAxis()`
w celu pobrania danych wejściowych przekazanych przez mysz.

Kliknij przycisk *Start*, a następnie poruszaj myszą. Gdy będziesz poruszać myszą na boki, widok w grze również będzie poddawany rotacji w kierunku wybranym myszą. To jest całkiem dobre rozwiązanie! Kolejnym krokiem jest zapewnienie możliwości rotacji pionowej zamiast poziomej.

2.4.2. Rotacja pionowa z ograniczeniami

Do obsługi rotacji poziomej wykorzystaliśmy metodę `Rotate()`, a w przypadku rotacji pionowej zastosujemy zupełnie inne rozwiązanie. Wprowadzicie wymieniona metoda jest wygodnym sposobem na używanie transformacji, ale jednocześnie mało elastycznym. Najlepiej sprawdza się podczas przyrostowej rotacji bez żadnych ograniczeń — z taką sytuacją mamy do czynienia w rotacji poziomej. Z kolei rotacja pionowa wymaga pewnych ograniczeń odnośnie do tego, na ile widok może być przechylony w górę bądź w dół. Na listingu 2.5 przedstawiłem kod źródłowy odpowiedzialny za rotację pionową, który należy umieścić w skrypcie *MouseLook*. Dokładne omówienie sposobu działania tego kodu znajdziesz pod listingiem.

Listing 2.5. Kod w skrypcie *MouseLook* odpowiedzialny za rotację pionową

```
...
public float sensitivityHor = 9.0f;
public float sensitivityVert = 9.0f;
```

← Deklaracje zmiennych używanych w rotacji pionowej.

```
public float minimumVert = -45.0f;
public float maximumVert = 45.0f;
```

```
private float _rotationX = 0; ← Deklaracja zmiennej prywatnej określającej kąt pionowy.
```

```

void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
        _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
        _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);
    }

    float rotationY = transform.localEulerAngles.y;

    transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
}
...

```

Inkrementacja kąta pionowego na podstawie ruchu myszą.

Ograniczenie kąta pionowego, aby mieścił się między wartościami minimalną i maksymalną.

Zachowanie tego samego kąta Y (brak rotacji poziomej).

Utworzenie nowego wektora na podstawie przechowywanych wartości rotacji.

W panelu *Inspector* przejdź do komponentu *MouseLook*, a następnie z rozwijanego menu *Axes* wybierz opcję wskazującą rotację pionową (*MouseY*) i uruchom skrypt. Zauważ, że widok nie porusza się na boki. Natomiast ruch myszą w górę lub w dół powoduje rotację widoku w wybranym kierunku. Rotacja pionowa zatrzymuje się na ustalonych wartościach granicznych (minimalnej i maksymalnej).

W tym kodzie wprowadziłem kilka nowych koncepcji wymagających wyjaśnienia. Przede wszystkim tym razem nie używamy metody *Rotate()*, więc potrzebujemy zmiennej (tutaj o nazwie *_rotationX*, ponieważ rotacja pionowa odbywa się względem osi *X*) przeznaczonej do przechowywania kąta rotacji. Metoda *Rotate()* inkrementuje bieżącą rotację, a omawiany tutaj kod bezpośrednio ustawia kod rotacji. Innymi słowy, mamy różnicę między poleceniami „dodaj 5 do kąta” i „ustaw kąt na 30”. Nadal konieczne jest przeprowadzenie inkrementacji kąta rotacji, ale do tego celu w kodzie wykorzystujemy operator *-=*: odjęcie wartości od kąta rotacji zamiast ustawienia kąta dla tej wartości. Rezygnując ze stosowania metody *Rotate()*, zyskujemy możliwość zmiany kąta rotacji na wiele różnych sposobów, a nie tylko jego inkrementacji. Wartość rotacji jest mnożona przez *Input.GetAxis()*, podobnie jak w kodzie odpowiedzialnym za rotację poziomą, z wyjątkiem tego, że teraz korzystamy z wartości *Mouse Y*, ponieważ interesuje nas pionowa oś ruchu myszy.

Operacje na kącie rotacji są przeprowadzane w jeszcze kolejnym wierszu. Wykorzystujemy metodę *Mathf.Clamp()* do utrzymania kąta rotacji między wartościami minimalną i maksymalną. Te ograniczenia zostały nałożone za pomocą zdefiniowanych wcześniej w kodzie zmiennych publicznych i gwarantują, że widok będzie obracany jedynie o maksymalnie 45 stopni w górę lub w dół. Metoda *Clamp()* nie jest typowa wyłącznie dla rotacji, ale jest ogólnie użyteczna do utrzymywania pewnych zmiennych liczbowych w wyznaczonych granicach. Umieść znak komentarza przed wywołaniem *Clamp()* i zobacz, co się stanie. Teraz rotacja nie zostanie zatrzymana po 45 stopniach i gracz będzie mógł się obrócić do góry nogami! Oglądanie świata z takiej pozycji bez wątplenia nie jest pożądane, stąd wspomniane wcześniej ograniczenia.

Ponieważ właściwość *kąta* w *transform* jest typu *Vector3*, konieczne jest utworzenie nowego egzemplarza wymienionego typu wraz z kątem rotacji przekazanym konstruktorowi. Metoda *Rotate()* automatyzuje ten proces za nas, inkrementując kąt rotacji, a następnie tworząc nowy wektor.

DEFINICJA Wektor to wiele liczb przechowywanych razem w postaci pojedynczej jednostki. Na przykład `Vector3` to trzy liczby (oznaczone jako x , y i z).

OSTRZEŻENIE Powodem utworzenia nowego obiektu typu `Vector3` zamiast zmiany wartości w istniejącym wektorze w transformacji jest to, że wspomniane wartości są w przypadku transformacji tylko do odczytu. Próba zmiany wartości wektora jest najczęstszym błędem popełnianym przez początkujących.

Kąty Eulera kontra kwaterniony

Prawdopodobnie zastanawiasz się, dlaczego nazwą właściwości jest `localEulerAngles` zamiast na przykład `localRotation`. Przede wszystkim musisz zacząć od poznania koncepcji o nazwie **kwaterniony**.

Kwaterniony to jeszcze inna konstrukcja matematyczna przeznaczona do reprezentowania rotacji. Różni się od kątów Eulera, czyli używanego przez nas dotąd podejścia opartego na osiach X , Y i Z . Czy pamiętasz wcześniejsze informacje o osiach określanych mianem *pitch*, *yaw* i *roll*? To jest właśnie metoda przedstawienia rotacji za pomocą kątów Eulera. Natomiast kwaterniony są... inne. Trudno jest tak dokładnie wyjaśnić, czym one są, ponieważ to skomplikowany aspekt wyższej matematyki, wymagający poruszenia tematu czterech wymiarów. Jeżeli chciałbyś poznać szczegółowo to zagadnienie, zajrzyj na stronę <http://www.flipcode.com/documents/matrfaq.html#Q47>.

Znacznie łatwiej jest odpowiedzieć na pytanie, dlaczego kwaterniony są używane do przedstawienia rotacji. Otóż interpolacja między wartościami rotacji (to znaczy przejście przez wiele wartości pośrednich w celu stopniowej zmiany z jednej wartości na inną) odbywa się znacznie łagodniej i naturalniej, gdy w jej trakcie są wykorzystywane kwaterniony.

Powracając do początkowego pytania, możemy powiedzieć, że nazwa `localRotation` przedstawia kwaternion, a nie kąty Eulera. Unity również oferuje właściwość kątów Eulera w celu łatwiejszego przeprowadzania operacji z wykorzystaniem rotacji. Właściwość kątów Eulera jest automatycznie konwertowana na postać kwaternionów oraz z postaci kwaternionów. Unity w tle wykonuje trudne operacje matematyczne i dlatego programista nie musi się tym przejmować.

Mamy jeszcze jeden wariant rotacji wymagający utworzenia kodu w skrypcie *MouseLook*: rotację przeprowadzaną jednocześnie w poziomie i w pionie.

2.4.3. Jednoczesna rotacja w poziomie i w pionie

Ostatni fragment kodu również nie używa metody `Rotate()`, dokładnie z tego samego powodu co poprzedni: rotacja w pionie jest ograniczona dwiema wartościami, minimalną i maksymalną, po inkrementacji. To oznacza, że rotacja pozioma musi być bezpośrednio obliczona teraz. Przypomnij sobie, jak metoda `Rotate()` automatyzuje proces inkrementacji kąta rotacji (patrz listing 2.6).

Listing 2.6. Rotacja pozioma i pionowa w skrypcie MouseLook

```
...
else {
    _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
    _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);
    float delta = Input.GetAxis("Mouse X") * sensitivityHor; ←
```

Zmienna *delta* określa wielkość, o jaką zmieni się rotacja

```
float rotationY = transform.localEulerAngles.y + delta;
transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
}
...
```

Inkrementacja kąta rotacji o podaną wartość delta.

Kilka pierwszy wierszy odpowiedzialnych za pracę z `_rotationX` działa dokładnie w taki sam sposób, jaki przedstawiłem w poprzednim punkcie rozdziału. Musisz jedynie pamiętać, że rotacja według osi X obiektu jest rotacją pionową. Skoro rotacja pozioma nie jest dłużej obsługiwana przez metodę `Rotate()`, to zamiast niej wykorzystujemy wiersze definiujące `delta` i `rotationY`. **Delta** to powszechnie stosowane w matematyce określenie „wielkości zmiany”. Dlatego też przeprowadzane tutaj obliczenie `delta` ma ustalić, o ile powinna zmienić się rotacja. Ta wielkość zmiany jest następnie dodawana do bieżącego kąta rotacji, co prowadzi do obliczenia nowego żądanego kąta rotacji.

Na koniec oba kąty, poziomy i pionowy, są używane do utworzenia nowego wektora, który następnie będzie przypisany właściwości kąta komponentu transformacji.

Wyłączenie fizyki rotacji gracza

Wprawdzie nie ma to żadnego znaczenia w tworzonym tutaj projekcie, ale w większości nowoczesnych gier typu FPS stosowana jest skomplikowana symulacja fizyki, która ma wpływ na wszystko to, co znajduje się na scenie. Może spowodować na przykład, że obiekty będą się odbijać lub wywracać. Tego rodzaju zachowanie wygląda i sprawdza się doskonale w przypadku większości obiektów. Jednak rotacja graczem powinna być kontrolowana wyłącznie za pomocą myszy i symulacja fizyki nie powinna mieć na nią wpływu.

Dlatego też skrypty wykorzystujące dane wejściowe myszy zwykle ustawiają właściwość `freezeRotation` w komponencie *Rigidbody* dołączonym do obiektu przedstawiającego gracza. W skrypcie *MouseLook* umieść przedstawioną poniżej metodę `Start()`.

```
...
void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true;
}
...
```

Sprawdzenie, czy istnieje komponent o podanej nazwie.

(*Rigidbody* to kolejny komponent, który może mieć obiekt. Symulacja fizyki działa na komponente *Rigidbody* i przeprowadza operacje na obiektach, do których został on dołączony).

Jeżeli zgubiłeś się podczas wprowadzania różnych zmian w trakcie pracy nad skrypcem *MouseLook*, na listingu 2.7 przedstawiłem jego pełny kod źródłowy. Gotowy skrypt znajdziesz również w materiałach dodatkowych przygotowanych dla tej książki.

Listing 2.7. Gotowy skrypt MouseLook

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
    public enum RotationAxes {
        MouseXAndY = 0,

```

```

    MouseX = 1,
    MouseY = 2
}

public RotationAxes axes = RotationAxes.MouseXAndY;

public float sensitivityHor = 9.0f;
public float sensitivityVert = 9.0f;

public float minimumVert = -45.0f;
public float maximumVert = 45.0f;

private float _rotationX = 0;

void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true;
}

void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
        _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
        _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

        float rotationY = transform.localEulerAngles.y;

        transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
    else {
        _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
        _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

        float delta = Input.GetAxis("Mouse X") * sensitivityHor;
        float rotationY = transform.localEulerAngles.y + delta;

        transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
}
}
}

```

Po wybraniu odpowiedniej opcji w rozwijanym menu *Axes* komponentu *MouseLook* i uruchomieniu nowego skryptu będziesz mógł się rozglądać we wszystkich kierunkach za pomocą myszy. Doskonale! Jednak nadal stoisz w miejscu i możesz rozglądać się wokół, jakbyś był zamontowany na wieżyczce. Kolejnym krokiem jest więc umożliwienie graczowi poruszania się po scenie.

2.5. Komponent danych wejściowych klawiatury – kontrolki pierwszej osoby

Rozglądanie się na podstawie danych wejściowych przekazywanych przez mysz jest bardzo ważnym aspektem gry typu FPS, ale nie jedynym. Gracz musi mieć również możliwość poruszania się w odpowiedzi na naciskane klawisze klawiatury. Przystępujemy teraz do utworzenia komponentu uzupełniającego funkcjonalność komponentu działającego na podstawie danych wejściowych myszy. Utwórz nowy skrypt C#, nadaj mu nazwę *FPSInput* i dołącz go do obiektu przedstawiającego gracza (ma on już dodany skrypt *MouseLook*). W komponencie *MouseLook* chwilowo wybierz jedynie rotację poziomą.

WSKAZÓWKA Kierowanie postacią gracza za pomocą klawiatury i myszy zostało tutaj osiągnięte z użyciem oddzielnych skryptów. Nie musisz stosować tego rodzaju struktury kodu, tylko możesz umieścić wszystko w pojedynczym skrypcie zapewniającym sterowanie postacią gracza. Jednak system oparty na komponentach (czyli podobnie jak w samym Unity) okazuje się znacznie elastyczniejszy i dlatego też jest bardziej użyteczny, gdy funkcjonalność zostaje umieszczona w kilku mniejszych komponentach.

Kod źródłowy opracowany w poprzednim punkcie rozdziału ma wpływ jedynie na rotację, a teraz chcemy zmieniać również położenie obiektu. Jak pokazałem na listingu 2.8, odwołujemy się do wcześniejszego kodu rotacji, ale przed dodaniem obsługi danych wejściowych myszy. Wprowadź ten kod w skrypcie *FPSInput*, przy czym wywołanie *Rotate()* zastąp metodą *Translate()*. Po kliknięciu przycisku *Start* widok będzie się poruszał, zamiast obracać się wokół własnej osi. Spróbuj zmienić wartości parametrów i sprawdź, jaki ma to wpływ na ruch obiektu. (W szczególności zamień miejscami pierwszą i drugą wartość). Po zakończeniu eksperymentów możemy powrócić do implementacji obsługi danych wejściowych pochodzących z klawiatury.

Listingi 2.8. Kod pochodzący z pierwszego listingu, ale zawierający kilka drobnych zmian

```
using UnityEngine;
using System.Collections;

public class FPSInput : MonoBehaviour {
    public float speed = 6.0f;
    void Update() {
        transform.Translate(0, speed, 0);
    }
}
```

Wprowadź to nie jest wymagane, ale prawdopodobnie zechcesz zwiększyć szybkość.

Zmiana wywołania *Rotate()* na *Translate()*.

2.5.1. Reakcja na naciśnięcie klawisza

Kod odpowiedzialny za obsługę ruchu na podstawie danych wejściowych pochodzących z klawiatury (przedstawiony na listingu 2.9) jest podobny do kodu obsługującego rotację myszą. Tutaj również wykorzystujemy metodę *GetAxis()*, i to bardzo podobnie jak

wcześniej. Spójrz na kod z listingu 2.9, na którym pokazałem sposób użycia metody `GetAxis()`.

Listing 2.9. Obsługa ruchu na podstawie klawiatury

```
...
void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    transform.Translate(deltaX, 0, deltaZ);
}
...
```

„Horizontal” i „Vertical”
to bezpośrednie nazwy stosowane
podczas mapowania klawiatury.

Podobnie jak wcześniej, wartości zwracane przez metodę `GetAxis()` są mnożone przez wartość określającą szybkość (`speed`), co ma na celu ustalenie wielkości ruchu. Wcześniej interesował nas ruch myszą w dowolnym kierunku, a teraz przekazujemy `Horizontal` lub `Vertical`. Te nazwy można uznać za abstrakcje dla ustawień danych wejściowych w Unity. Jeżeli wybierzesz opcję menu *Edit/Project Settings/Input*, w oknie Unity wyświetlone zostaną lista abstrakcji nazw danych wejściowych oraz dokładne kontrolki mapowane na te nazwy. Klawisze kursora w lewo oraz w prawo, a także litery *A* i *D* są mapowane na *Horizontal*, podczas gdy klawisze kursora w górę oraz w dół, a także litery *W* i *S* są mapowane na *Vertical*.

Zwróć uwagę na to, że wartości ruchu są stosowane względem współrzędnych *X* i *Z*. Jak prawdopodobnie zauważyłeś podczas eksperymentowania z metodą `Translate()`, współrzędna *X* powoduje przesunięcie na boki, a współrzędna *Z* powoduje przesunięcie do przodu i do tyłu.

Po dodaniu do skryptu tego nowego kodu będziesz w stanie poruszać się po scenie za pomocą klawiszy kursora lub *WASD*, co jest standardem w większości gier typu FPS. Skrypt obsługujący ruch jest niemal ukończony, pozostało nam już do wykonania tylko kilka drobiazgów.

2.5.2. Ustawienie współczynnika ruchu niezależnie od szybkości komputera

W tym momencie to nie będzie jeszcze oczywiste, ponieważ kod testujesz tylko na jednym komputerze (własny), ale po uruchomieniu go na innym komputerze zauważysz odmienną szybkość poruszania się gracza. Wyjaśnienie jest proste: pewne komputery potrafią przetwarzać kod źródłowy i grafikę szybciej niż inne. Dlatego też w obecnej wersji projektu szybkość poruszania się gracza będzie zależała od szybkości komputera. Taki mechanizm nazywamy **zależnością od liczby klatek generowanych na sekundę**, ponieważ działanie kodu obsługującego ruch zależy od liczby klatek generowanych na sekundę w trakcie gry.

Na przykład wyobraź sobie uruchomienie tego demo na dwóch różnych komputerach. Pierwszy osiąga 30 klatek na sekundę, a drugi 60 klatek na sekundę. To oznacza, że na drugim komputerze metoda `Update()` będzie wywoływana dwukrotnie częściej niż na pierwszym, a ta sama wartość określająca szybkość (tutaj: 6) będzie stosowana w trakcie każdego wywołania. W przypadku 30 klatek na sekundę współczynnik ruchu

wynosi 180 jednostek na sekundę, a przy 60 klatkach na sekundę ten współczynnik wynosi już 360 jednostek na sekundę. W większości gier tak duża różnica w szybkości poruszania się gracza jest niepożądana.

Rozwiązaniem jest takie zmodyfikowanie kodu odpowiedzialnego za ruch gracza, aby działał **niezależnie od liczby klatek generowanych na sekundę**. W takim przypadku szybkość ruchu nie będzie zależała od liczby generowanych klatek na sekundę. Sposób uzyskania takiego efektu polega na niestosowaniu w każdej klatce tej samej wartości określającej szybkość. Zamiast tego wspomniana wartość powinna być zmieniana w zależności od wydajności komputera, na którym została uruchomiona gra. Należy więc pomnożyć tę wartość przez inną wartość, nazywaną `deltaTime`, jak pokażalem na listingu 2.10.

Listing 2.10. Niezależna od liczby klatek obsługa ruchu postacią gracza oparta na wartości `deltaTime`

```
...
void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    transform.Translate(deltaX * Time.deltaTime, 0, deltaZ * Time.deltaTime);
}
...
```

To była prosta zmiana. Klasa `Time` ma wiele właściwości i metod przeznaczonych do pracy z wartościami czasu, a jedną z tych właściwości jest `deltaTime`. Skoro `delta` oznacza wielkość zmiany, to `deltaTime` przedstawia wielkość zmiany w czasie, a dokładnie ilość czasu, jaka upłynęła między wygenerowaniem dwóch klatek. Czas między wygenerowaniem klatek zależy od liczby klatek na sekundę, na przykład przy 30 klatkach na sekundę wartość `deltaTime` wynosi 1/30 sekundy. Dlatego też pomnożenie wartości szybkości przez `deltaTime` pozwala na odpowiednie dostosowanie szybkości w różnych komputerach.

Po tej zmianie szybkość ruchu będzie taka sama we wszystkich komputerach. Jednak nasz skrypt nie jest jeszcze w pełni ukończony. Kiedy będziesz się poruszać po pomieszczeniu, zauważysz, że możesz przechodzić przez ściany. Musimy więc wprowadzić odpowiednie zmiany w kodzie, aby to uniemożliwić.

2.5.3. Komponent *Character Controller* i wykrywanie kolizji

Bezpośrednia zmiana transformacji obiektu nie stosuje mechanizmu wykrywania kolizji, więc postać może przechodzić przez ściany. Aby użyć mechanizmu wykrywania kolizji, musimy wykorzystać komponent *Character Controller*. Dzięki niemu obiekt porusza się jak postać w grze, co dotyczy również kolizji ze ścianami. Przypomnij sobie, że wcześniej — podczas definiowania gracza — już dołączyliśmy ten komponent. Teraz użyjemy go wraz z utworzonym w skrypcie *FPSInput* kodem odpowiedzialnym za obsługę ruchu (spójrz na listing 2.11).

Listing 2.11. Użycie komponentu *Character Controller* do obsługi ruchu postaci gracza

```

...
private CharacterController _charController;
void Start() {
    _charController = GetComponent<CharacterController>();
}
void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    Vector3 movement = new Vector3(deltaX, 0, deltaZ);
    movement = Vector3.ClampMagnitude(movement, speed);

    movement *= Time.deltaTime;
    movement = transform.TransformDirection(movement);
    _charController.Move(movement);
}
...

```

Zmienna pozwalająca na odwołanie się do komponentu *Character Controller*.

Dostęp do innych komponentów dołączonych do tego samego obiektu.

Ograniczenie szybkości ruchu ukośnego do takiej samej szybkości jak wzdłuż osi.

Przekształcenie wektora ruchu ze współrzędnych lokalnych na globalne.

Nakazanie komponentowi *Character Controller* przesunięcia o podany wektor.

W powyższym fragmencie kodu pojawiło się kilka nowych koncepcji. Pierwsza z nich dotyczy wprowadzenia zmiennej pozwalającej na odwołanie się do komponentu *Character Controller*. Ta zmienna po prostu tworzy lokalne odwołanie do obiektu (obiekty kodu — nie pomył go z obiektem na scenie). Wiele skryptów może zawierać odwołania do tego jednego egzemplarza *Character Controller*.

Na początku ta zmienna jest pusta, więc zanim będziemy mogli użyć tego odwołania, musimy przypisać zmiennej obiekt, do którego chcemy się później odwoływać. W tym miejscu do gry wchodzi metoda `GetComponent()`; jej wartością zwrrotną są inne komponenty dołączone do tego samego obiektu typu `GameObject`. Zamiast przekazywać parametr w nawiasie okrągłym, wykorzystujemy składnię C# zdefiniowania typu w nawiasie ostrym.

Gdy mamy odwołanie do komponentu *Character Controller*, możemy wywołać w nim metodę `Move()`. Tej metodzie przekazujemy wektor, podobnie jak kod rotacji na podstawie myszy używał wektora jako wartości dla rotacji. Także tutaj wartości mają pewne ograniczenia. Wykorzystujemy `Vector3.ClampMagnitude()` do ograniczenia rzędu wielkości dla szybkości ruchu. W tym miejscu zostało użyte wywołanie `ClampMagnitude()`, ponieważ w przeciwnym razie ruch odbywający się po przekątnej byłby szybszy niż ruch bezpośrednio wzdłuż osi (przeciwprostokątnej i przyprostokątnej w trójkącie prostym).

Mamy jeszcze jeden, trudniejszy aspekt przedstawionego tutaj ruchu wektora, związany z przestrzenią lokalną i globalną, o których wspomniałem już wcześniej, podczas omawiania rotacji. Przyjmujemy założenie o utworzeniu wektora wraz z wartością pozwalającą na ruch w lewą stronę. To będzie lewa strona gracza, która może być w zupełnie innym kierunku niż lewa strona świata. Mówimy więc o lewej stronie w przestrzeni lokalnej, a nie globalnej. Metodzie `Move()` trzeba przekazać wektor ruchu zdefiniowany w przestrzeni globalnej i dlatego trzeba będzie przeprowadzić konwersję wektora przestrzeni lokalnej na wektor przestrzeni globalnej. Tego rodzaju konwersja wymaga wyjątkowo skomplikowanych obliczeń matematycznych. Na szczęście

Unity zajmuje się tymi obliczeniami i nasze zadanie sprowadza się po prostu do wywołania metody `TransformDirection()` w celu transformacji kierunku.

DEFINICJA Słowo **transformacja** użyte w charakterze czasownika oznacza konwersję z jednego układu współrzędnych na inny. (Jeżeli nie pamiętasz, czym jest układ współrzędnych, zajrzyj do punktu 2.3.3). Nie pomył tego znaczenia z innymi znaczeniami transformacji, obejmującymi między innymi komponent *Transform* i czynność przesunięcia obiektu na scenie. Mamy więc do czynienia z dość przeciążonym pojęciem, ponieważ jego wszystkie znaczenia odwołują się do tej samej koncepcji bazowej.

Przetestuj teraz kod odpowiedzialny za obsługę ruchu. Jeżeli jeszcze tego nie zrobiłeś, ustaw w komponencie *MouseLook* rotację — zarówno poziomą, jak i pionową. Masz możliwość pełnego rozglądania się po scenie oraz latania po niej za pomocą klawiatury. To całkiem dobre rozwiązanie, jeśli chcesz umożliwić graczowi latanie po scenie. Co zrobić w sytuacji, gdy gracz ma mieć możliwość poruszania się jedynie po ziemi?

2.5.4. Dostosowanie komponentów do ruchu po ziemi, a nie do lotu

Skoro mechanizm wykrywania kolizji działa, skrypt może obsługiwać grawitację, co spowoduje, że gracz pozostanie na ziemi. Zdefiniuj zmienną grawitacji, a następnie wykorzystaj wartość grawitacji dla osi Y, jak pokazałem na listingu 2.12.

Listing 2.12. Dodanie grawitacji do kodu obsługującego ruch

```
...
public float gravity = -9.8f;
...
void Update() {
    ...
    movement = Vector3.ClampMagnitude(movement, speed);
    movement.y = gravity;           ← Użyj wartości grawitacji zamiast po prostu zera.
    ...
}
```

W ten sposób dodaliśmy siłę przytrzymującą gracza na ziemi. Jednak nie zawsze będzie ona skierowana prosto do dołu, ponieważ za pomocą myszy gracz może spoglądać w górę oraz w dół. Na szczęście mamy wszystko, co jest potrzebne do wprowadzenia odpowiednich poprawek. Wystarczy dokonać kilku mniejszych zmian dostosowujących konfigurację komponentów w obiekcie gracza. Przede wszystkim w komponencie *MouseLook* w obiekcie gracza ustaw rotację tylko poziomą. Następnie komponent *MouseLook* dodaj do obiektu kamery i wybierz rotację tylko pionową. Tak, to prawda, mamy dwa różne obiekty reagujące na ruchy wykonywane myszą!

Skoro obiekt gracza może teraz wykonywać rotację jedynie poziomą, nie istnieje już żaden problem związany z działaniem siły grawitacji, gdy gracz spogląda w górę lub w dół. Obiekt kamery jest dołączony do obiektu gracza (pamiętasz, jak to zrobiliśmy w panelu *Hierarchy*?), więc pomimo zdefiniowanej w kamerze rotacji pionowej niezależnej od gracza podlega ona również rotacji poziomej wraz z obiektem gracza.

Dopracowanie skryptu

Metoda `RequireComponent()` ma zagwarantować dołączenie także innych komponentów wymaganych przez ten skrypt. Czasami inne komponenty są uznawane za opcjonalne (to znaczy kod można odczytać w następujący sposób: „Jeśli ten komponent również został dołączony, to...”), a czasami chcemy, aby były obowiązkowe. Na początku skryptu dodaj metodę wymuszającą spełnienie zależności i przekazującą żądany komponent jako parametr.

Podobnie po umieszczeniu metody `AddComponentMenu()` na początku skryptu zostanie on dodany do menu komponentów w edytorze graficznym Unity. W wywołaniu tej metody należy podać nazwę wskazującą nazwę dla dodawanego elementu menu. Następnie ten skrypt będzie wybrany po kliknięciu widocznego na dole panelu *Inspector* przycisku *Add Component*. To bardzo użyteczne rozwiązanie.

Po dodaniu na początku skryptu obu wymienionych metod jego początek wygląda tak, jak pokazałem w poniższym fragmencie kodu.

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
    ...
}
```

Na listingu 2.13 zaprezentowałem w pełni ukończony skrypt. Po wprowadzeniu drobnych usprawnień dotyczących konfiguracji komponentu w obiekcie przedstawiającym gracza otrzymałeś możliwość poruszania się po przygotowanej scenie. Nawet pomimo użycia zmiennej grawitacji nadal można wykorzystać ten skrypt do obsługi latania — wystarczy zmiennej `gravity` przypisać wartość 0.

Listing 2.13. Ukończony skrypt FPSInput

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
    public float speed = 6.0f;
    public float gravity = -9.8f;

    private CharacterController _charController;

    void Start() {
        _charController = GetComponent<CharacterController>();
    }

    void Update() {
        float deltaX = Input.GetAxis("Horizontal") * speed;
        float deltaZ = Input.GetAxis("Vertical") * speed;
        Vector3 movement = new Vector3(deltaX, 0, deltaZ);
        movement = Vector3.ClampMagnitude(movement, speed);

        movement.y = gravity;
    }
}
```

```
movement *= Time.deltaTime;
movement = transform.TransformDirection(movement);
_charController.Move(movement);
}
}
```

Gratuluję ukończenia budowy tego projektu 3D! W tym rozdziale podałem naprawdę dużo informacji i teraz powinieneś już dość dobrze wiedzieć, jak w Unity tworzyć kod odpowiedzialny za wykonywanie ruchu. Wprawdzie to pierwsze demo może być ekscytujące, ale musimy wykonać jeszcze sporo pracy, zanim grę będzie można uznać za ukończoną. W naszym planie opisaliśmy ten projekt jako prostą scenę gry FPS. A co to za gra FPS, w której nie może strzelać? Zrób sobie zasłużoną przerwę od projektu przedstawionego w tym rozdziale, a później przygotuj się na kolejny krok.

2.6. Podsumowanie

W tym rozdziale dowiedziałeś się, że:

- układ współrzędnych w przestrzeni 3D jest zdefiniowany za pomocą osi X, Y i Z;
- obiekty i światło definiują scenę;
- gracz na scenie typu FPS to w zasadzie kamera;
- kod odpowiedzialny za obsługę ruchu stosuje niewielkie transformacje przeprowadzane w każdej klatce;
- kontrolki w grze typu FPS to rotacja myszą oraz poruszanie się za pomocą klawiatury.

Skorowidz

3ds Max, 361

A

akcje w grze, 293
Android, 335
Android SDK, 360
animacja, 99, 175
 postaci gracza, 195
 szkieletowa, 195
animowane sprite'y, 130
APK, Android Application Package, 338
aplikacje do grafiki 3D, 360
atlas, 131
Audacity, 362
automat skończony, 85

B

biblioteki kodu, 372
bilbord, 257
Blender, 361, 363
 modelowanie ławki, 363
bufor, 258
buforowanie pobranego obrazu, 258

C

celownik, 76, 78
cienie, 179
culling mask, 181
cyfrowe audio, 267

D

delta, 62
deserializacja, 252
deweloper, 24
dodawanie
 cieni, 179
 elementów do magazynu, 224
 funkcjonalności, 230

graficznych wskaźników celownika, 76
grawitacji, 68
menedżera DataManager, 320
nieprzyjaciół, 71
pocisków, 71
poziomów gry, 322
przycisku, 232
dopasowanie, 141
dostęp do danych menedżera, 227
dostosowanie widoku kamery, 177
dostrzeganie przeszkód, 83
drzwi, 207
dziedziczenie, 26
dźwięk 2D, 270

E

edytory grafiki 2D, 361
efekty
 dźwiękowe, 265
 specjalne, 116
 systemu cząstek, 120
egzemplarz, 87
eksport
 modelu, 114
 sceny, 104
elementy słownika, 225
emisja zdarzenia, 172
ETC, Ericsson Texture Compression, 340
etykiety tekstowe, 157

F

fizyka, 212
 rotacji, 62
FMOD Studio, 373
format JSON, 251
formaty plików, 104, 113, 266
FPS, first-person shooter, 44
FPS CONTROL, 374
framework MouseLook, 57
funkcja lambda, 259
funkcje anonimowe, 259

G

generowanie, 78
 nieba, 109, 237
 GIMP, 362
 Git, 360
 gra
 2D, 125
 3D, 175
 RPG, 293
 typu FPS, 176
 gracz, 67
 graficzny
 interfejs użytkownika, 152, 159
 wskaźnik, 76
 grafika, 97
 2D, 126
 3D, 360
 grawitacja, 68

H

HTML5, 332
 HUD, heads-up display, 152

I

IDE, integrated development environment, 25
 IDE
 MonoDevelop, 38
 iloczyn skalarny, 194
 implementacja
 akcji skoku, 188
 funkcjonalnych przedmiotów, 205
 import
 efektów dźwiękowych, 266
 klipów muzycznych, 282, 268
 modelu, 114
 obrazów interfejsu użytkownika, 155
 plików graficznych, 106
 postaci, 178
 integracja
 systemu zdarzeń, 170
 z obiektami, 211
 interaktywne urządzenia, 205
 interaktywność interfejsu użytkownika, 161
 interfejs
 audio, 274
 IGameManager, 242, 294
 użytkownika, 31, 151, 229
 internet, 235
 interpolacja liniowa, 186

iOS, 335
 IPA, iOS App Package, 336

J

język
 C#, 36
 JavaScript, 36
 PHP, 263
 JSON, 251

K

kafelkowanie, 106
 kamera, 50
 rotacja postaci, 185
 tryb 2D, 130
 kanał alfa, 105
 kąty Eulera, 61
 klasa
 GameObject, 50
 Managers, 243
 NetworkService, 245
 WeatherManager, 243
 klatka, 38
 klawiatura, 33
 klip, 197
 klucz do drzwi, 229
 kod
 definiujący przycisk, 147
 do obsługi ruchu, 297
 kamery, 181
 kontrolujący muzykę, 285
 menedżera AudioManager, 274
 menedżera InventoryManager, 220
 menedżera PlayerManager, 221
 odkrywający karty, 142
 odpowiedzialny za ustawienia wartości, 202
 PHP, 263
 rotacji poziomej, 58
 skryptu ObjectiveTrigger, 315
 sztucznej inteligencji, 86
 wyświetlający celownik, 78
 wyzwalacza, 214
 kolizje, 94
 gracza, 52
 z przeszkodą, 212
 kompilacja, 330
 aplikacji, 328
 aplikacji internetowej, 331
 na platformy mobilne, 335

- komponent, 26
 - AudioManager, 283
 - AudioSource, 283
 - Character Controller, 66
 - nasłuchujący zdarzeń, 173
 - skryptu, 37
 - UIButton, 146
 - UIController, 162
- kompresja tekstur, 339
- konfiguracja
 - animacji, 195
 - atmosfery, 238
 - graficznego interfejsu użytkownika, 156
 - narzędzi kompilacji, 336
- konsole do gier, 29
- kontrola
 - dźwięku, 271
 - poziomu głośności, 285, 287
- kontroler animacji, 199
- kontrolka skoku, 299
- kontrolki dotyczące muzyki, 284
- kontrolowanie przepływu misji, 309
- kotwica, 160
- kwaterniony, 61

Ł

- łączenie zasobów i kodu, 293

M

- magazyn, 217
 - InventoryManager, 230
 - przedmiotów, 226
- mapowanie tekstury, 367
- marketing gry, 353
- materiał, 92, 99
 - skyboxu, 110
- Maya, 361
- Maya LT, 372
- mechanika gry, 352
- menedżer
 - AudioManager, 274
 - DataManager, 320
 - gracza, 218
 - ImagesManager, 255
 - InventoryManager, 220, 225, 231
 - magazynu, 218
 - menedżerów, 222
 - MissionManager, 316
 - PlayerManager, 221, 315

- Mercurial, 360
- metoda
 - Active(), 215
 - Deactivate(), 215
 - RequireComponent(), 69
 - Rotate(), 60
 - ScreenPointToRay(), 73
 - SendMessage(), 146
- metody publiczne, 137
- MMO, massively multiplayer online, 235
- model, 98
 - 3D, 112, 176
- modelowanie ławki, 363
- modyfikacja wykrywania ziemi, 190
- monitor zmieniający kolor, 210, 211
- monitorowanie
 - aktualnej pogody, 261
 - stanu postaci, 85
- MonoDevelop, 38
- muzyka, 265
 - odtwarzana w tle, 281
- MVC, model-view-controller, 206
- myszka, 33, 133

N

- naciśnięcie klawisza, 64
- nadrzędna struktura gry, 309
- narzędzia
 - programistyczne, 359
 - zewnętrzne, 359
- narzędzie
 - do obsługi dźwięku, 290
 - FMOD, 290
- nasłuchiwanie zdarzeń, 171, 172
- nawigacja po scenie, 355
- niebo, 109, 237

O

- obiekt
 - GameObject, 50
 - karty, 133
 - List, 223
 - Mesh, 32, 98
 - sprite'ów, 133
 - WeatherManager, 247
- obiekty
 - główne, 198
 - nadrzędnego, 49
 - potomne, 49

obliczanie położenia kamery, 183
 obraz, 157
 nieba, 110
 tekstury, 107
 typu lightmap, 180
 obrót obiektu, 54
 obsługa
 drzwi, 213
 efektów dźwiękowych, 273, 280
 graficznych wskaźników, 76
 interfejsu użytkownika, 304
 kontrolki, 168
 krawędzi, 190
 postępów gracza, 317
 reaktywnych nieprzyjaciół, 79
 ruchu, 65, 297
 urządzenia, 210
 zmiany położenia gracza, 187
 obsługiwane formaty plików, 266
 ocena dopasowania, 141
 oddawanie strzałów, 90
 odkrycie karty, 134
 odpowiedzi na zdarzenia, 170
 odtwarzanie
 dźwięku, 265, 270, 280
 muzyki, 283
 muzyki w pętli, 282
 okno
 Build Settings, 327
 HUD, 172
 magazynu, 305
 opóźnione wczytywanie, 281
 oprogramowanie audio, 362
 orbitowanie kamery, 181

P

panel
 Animator, 200
 Console, 35
 Hierarchy, 34
 Inspector, 34, 49, 118
 Project, 35
 pasek narzędzi, 31
 perspektywa trzeciej osoby, 177
 Photoshop, 361
 plan podłoga, 102
 planowanie
 projektu, 44
 układu, 154
 platforma IOS, 336
 platformy mobilne, 29

plik
 describe.meta, 27
 manifest, 348
 TestPlugin.h, 344
 TestPlugin.java, 348
 TestPlugin.m, 344
 pliki APK, 338
 płomień, 118
 płótno, 156
 płynna rotacja, 186
 pocisk, 76
 podprocedura, 76
 podręcznik Unity, 371
 pola tekstowe, 167
 porównywanie odkrytych kart, 142
 poruszanie obiektami, 53
 prefabrykat nieprzyjaciela, 87
 Pro Tools, 362
 prognoza pogody, 241
 programista, 24
 programowanie
 gier, 372
 interaktywności interfejsu użytkownika, 161
 komponentu UIButton, 146
 komponentu UIViewController, 162
 kontrolki ruchu, 184
 menedżera gry, 219
 ruchu, 53
 programowe wczytywanie obrazów, 135
 projekt gry, 352
 projektowanie poziomu, 101
 promień światła, 73
 przeciąganie
 obiektów, 89
 obrazów, 107
 przekazanie danych do serwera, 260
 przeszkody, 83
 przetwarzanie danych XML, 248, 249
 przycisk zerujący grę, 146
 przyciski, 157
 przyśpieszenie, 189
 punkt widzenia, 52

R

raycasting, 72, 73, 192
 rigging, 195
 rodzaje światła, 50
 rotacja
 pionowa, 59, 61
 postaci gracza, 184
 pozioma, 58, 61

rozglądanie się, 57
rozglaszanie zdarzeń, 171, 322
RPC, 254
RPG, role-playing game, 292
ruch, 54
 postaci gracza, 175

S

samouczki, 371
scena
 dodanie cieni, 179
 nawigacja, 355
 teksturowanie, 103
 umieszczenie obiektów, 47
 widok z góry, 297
 zewnątrzna, 237
sceneria 3D, 100
separacja zadań, 163
serializacja, 318
shader, 111
 Additive, 119
siatka, 365
 kart, 138
skok, 188
skróty klawiszowe, 356
skrypt, 37
 BasicUI, 228, 230
 build.xml, 347
 CheckpointTrigger, 262
 DataManager, 318
 DeviceOperator, 208, 210
 Fireball, 94
 FPSInput, 69, 173
 GameEvent, 171, 249, 313
 InventoryPopup, 306
 Managers, 275
 MemoryCard, 137
 MobileTestObject, 343
 MouseLook, 57, 62
 NetworkService, 261
 ObjectiveTrigger, 315
 otwierający drzwi, 207
 PlatformTest, 331
 PlayerCharacter, 296
 PlayerManager, 303
 PointClickMovement, 301, 305
 RayShooter, 74, 76, 162, 273
 ReactiveTarget, 80
 RelativeMovement, 189, 212
 SceneController, 88, 136, 149

SettingsPopup, 166, 174, 280, 287
StartupController, 312
TestPlugin, 345
UIController, 163, 279
ukrywający tył karty, 134
usuwany obiekt, 216
WanderingAI, 84, 85
WeatherController, 239, 253
WebLoadingBillboard, 257
WebTestObject, 333
skybox, 109, 111
słownik, 225
słowo kluczowe this, 82
sprite'y, 129
SSAO, 25
strzały, 72, 90
strzelanie pociskiem, 92
suwak, 167
SVN, 360
system
 animacji Mecanim, 196
 cząstek, 99, 116, 120
 Mecanim, 116
 zdarzeń, 170
sztuczna inteligencja, 82
 nieprzyjaciela, 295

Ś

ściany
 wewnętrzne, 48
 zewewnętrzne, 48
światła domyślne, 370
światło, 50
 kierunkowe, 50
 punktowe, 50
 reflektora, 50

T

tablica sprite'ów, 138
tasowanie kart, 140
technika
 raycastingu, 72
 SSAO, 25
tekst 3D, 144
tekstura, 104
teksturowanie sceny, 103
testowa strona internetowa, 332
TexturePacker, 362
trafienie, 79, 80, 92

transformacja, 53, 68
 translacja, 34
 tryb

- bezpośredni, 153
- edytora 2D, 128
- zachowany, 153

 tweens, 81
 tworzenie

- atlasu, 131
- dopasowania, 141
- geometrii siatki, 364
- gry 2D, 125
- gry 3D, 175
- kontrolera animacji, 199
- obiektu, 88, 90
- obiektu karty, 133
- obiektu sprite'ów, 133
- plótna, 156
- poziomu, 102
- prefabrykatu nieprzyjaciela, 87
- prefabrykatu pocisku, 90
- sceny zewnętrznej, 237
- siatki kart, 138
- urządzeń, 206
- wyskakującego okna, 164

 typ wyliczeniowy, 220
 typy

- plików, 105, 113
- zasobów graficznych, 98

U

uaktualnianie gry, 170
 układ współrzędnych 3D, 43, 45
 ukrycie niedopasowanych kart, 143
 Unity3D, 372
 Unity3D Student, 371
 uruchamianie kodu, 37
 usługa Play Games, 373
 usługi sieciowe, 241
 ustawienia

- dla systemu cząstek, 118
- dotyczące jakości, 330
- importu, 178
- kamery, 131
- kotwicy, 160
- Player Settings, 328, 329
- profilu provisioning, 338
- światła kierunkowego, 51
- światła punktowego, 52
- współczynnika ruchu, 65

usunięcie

- komponentu, 52
- obiektu GameObject, 91

 uszkodzenie gracza, 95
 używanie

- komponentu Character Controller, 67
- MonoDevelop, 38
- obrazu tekstury, 107
- PlayerPrefs, 169
- raycastingu, 192
- siły, 212
- urządzeń, 299

V

Visual Studio, 359

W

wczytywanie obrazów, 135, 254
 wdrożenie gry, 325
 Web API, 241
 WebGL, 332
 wektor, 61
 whiteboxing, 100
 widok

- 2D, 128
- 3D, 43
- Game, 31
- Scene, 31

 współczynnik ruchu, 65
 współrzędne

- globalne, 56
- lewostronne, 47
- lokalne, 56
- prawostronne, 47
- tekstury, 115

 wtyczki, 341

- Android, 345
- iOS, 342

 wybór

- formatu pliku, 104
- materiału, 108

 wyciszenie między ścieżkami muzycznymi, 288
 wykonywanie żądań HTTP, 245
 wykrywanie

- kolizji, 66
- ziemi, 190

 wyłączenie fizyki rotacji, 62
 wypalanie cieni, 180
 wyskakujące okna, 164

wysyłanie danych, 261
wyświetlanie
 informacji, 40
 obrazów 2D, 129
 obrazów kart, 135
 obrazu, 257
 okna ustawień dźwięku, 279
 punktacji, 144
 wyniku, 145
wytłoczenie, 366
wywołania zwrotne, 246
wywołanie
 LoadLevel, 149
 Rotate(), 59
wywoływanie efektów dźwiękowych, 272
wyzwalacz, 214
wzorce projektowe, 219

X

Xcode, 338, 360
XML, 249

Z

zapełnienie, 197
zarządzanie
 danymi magazynu, 217
 pamięcią, 91
 stanem gry, 217
zasoby graficzne, 97
zastąpienie interfejsu użytkownika, 302
zbieranie przedmiotów, 216
zdarzenia myszy, 164
zdarzenie health, 303
zdefiniowanie zapełnionego dźwięku, 271
zmiana
 obrazu sprite'a, 136
 poziomu głośności, 276
 sceny, 252

Ż

żądania HTTP, 236, 254
żądanie
 danych WWW, 244
 typu POST, 261

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄZKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Warunkiem stworzenia dobrej gry jest staranne opracowanie jej mechaniki, czyli poszczególnych akcji i systemu tych akcji. Gry, które odniosły największy sukces, charakteryzują się innowacyjną i interesującą mechaniką. Niemniej nawet jeśli już opracowałeś projekt gry, łącznie z jej mechaniką, grafiką i dźwiękiem, potrzebujesz bardzo dobrego narzędzia, aby stworzyć grę, która osiągnie sukces i zyska popularność. Takim narzędziem jest Unity — zintegrowane środowisko do tworzenia trójwymiarowych i dwuwymiarowych gier komputerowych oraz innych materiałów interaktywnych.



Niniejsza książka jest przeznaczona dla osób, które osiągnęły biegłość w programowaniu i teraz chcą pisać gry za pomocą Unity. Wyczerpująco przedstawiono kolejne kroki podejmowane podczas pisania gry w Unity. Nauka została oparta na przykładach: w książce znalazło się kilka projektów różnych gier. Opisano również metody wdrażania gier na różnych platformach, między innymi internetowej i mobilnej — Unity jest środowiskiem w pełni niezależnym od platformy sprzętowej. Nie zabrakło też informacji o narzędziach przydatnych do projektowania grafiki 3D oraz o innych zasobach, dzięki którym praca programisty jest efektywna i bardzo satysfakcjonująca!

W tej książce omówiono między innymi:

- programowanie poruszania się postaci po świecie 3D, raycasting i sztuczną inteligencję
- zasoby takie jak modele i tekstury: tworzenie i import
- GUI oraz implementację interaktywnych urządzeń i elementów w grze
- obsługę krótszych i dłuższych ścieżek dźwiękowych
- komunikację z internetem oraz obsługę różnych platform sprzętowych

Joseph Hocking jest inżynierem oprogramowania. Specjalizuje się w tworzeniu interaktywnych aplikacji. Pracuje w firmie Synapse Games, gdzie zajmuje się programowaniem gier internetowych i mobilnych, takich jak ostatnio wydana *Tyrant Unleashed*. Jest również wykładowcą w Columbia College Chicago. Mieszka w Chicago.

**Unity — i oto tworzysz
zadziwiający świat
dla swoich graczy!**

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowości>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-283-3519-6



9 788328 335196

Informatyka w najlepszym wydaniu

cena: 69,00 zł