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

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.

Spis treści

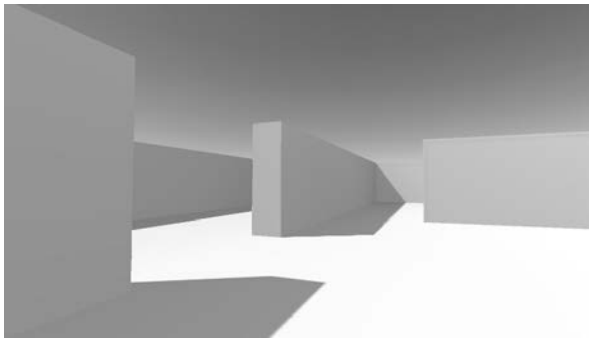
2 Budowa demo pokazującego przestrzeń 3D

W tym rozdziale:

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

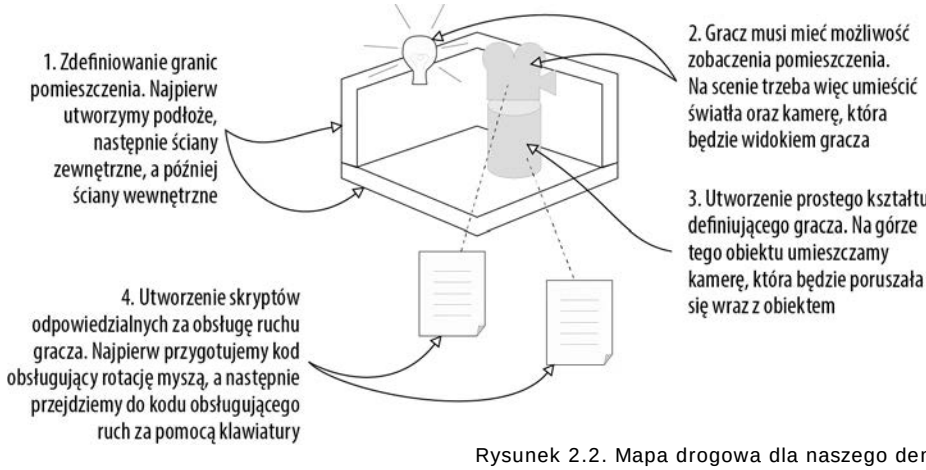
Unity znacznie ułatwia rozpoczęcie pracy początkującym programistom, choć konieczne jest zwrócenie uwagi na kilka kwestii przed przystąpieniem do budowy kompletnej sceny. Nawet podczas pracy z tak elastycznym narzędziem jak Unity niezbędne jest odkrycie celu, który będziesz próbował osiągnąć. Trzeba 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 drogowa dla naszego demo 3D

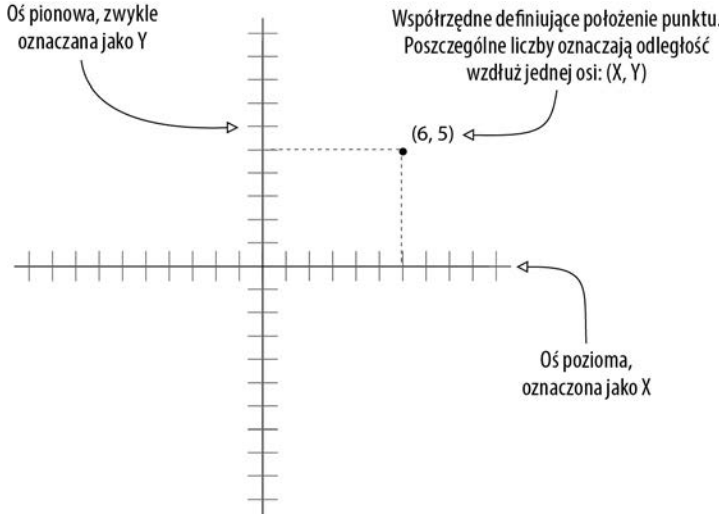
Nie przerażaj się zawartością przedstawionej powyżej mapy drogowej! Wprawdzie wygląda na to, że czeka nas dużo pracy, ale na szczęście Unity bardzo nam wszystko ułatwi. Kolejne punkty rozdziału dotyczące skryptów odpowiedzialnych za obsługę ruchu są dość obszerne, ale tylko dlatego, że przeanalizujemy 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

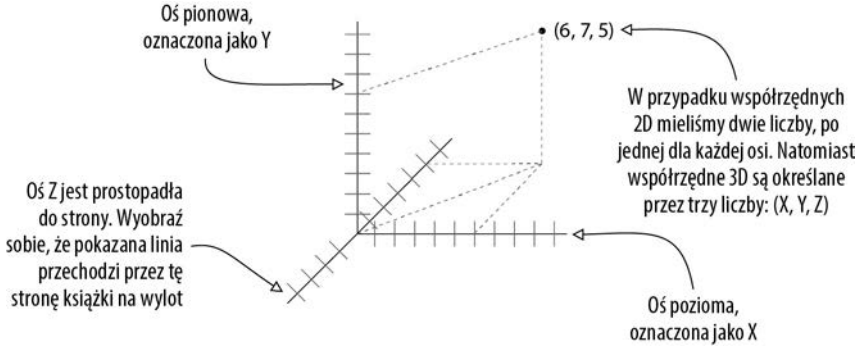
Przypomnij sobie prosty plan, od którego ułożenia rozpoczęliśmy pracę. Trzeba zwrócić uwagę na trzy elementy: pomieszczenie, widok i kontrolki. W przypadku tych wszystkich elementów trzeba wiedzieć, jak ich położenie i ruch będą przedstawiane w trójwymiarowych symulacjach komputerowych. Jeżeli dopiero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

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.

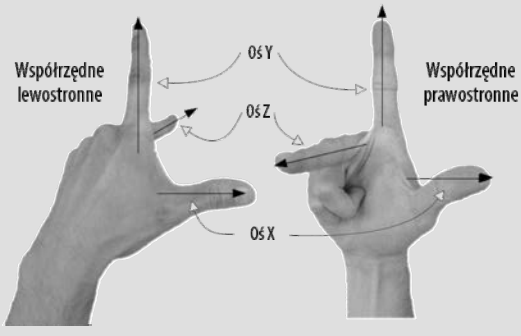
W widoku *Scene* w Unity możesz zobaczyć wyświetlone trzy osie definiujące układ współrzędnych 3D. Z kolei w panelu *Inspector* masz możliwość wpisania trzech liczb definiujących położenie danego obiektu w przestrzeni 3D. Nie tylko będziesz tworzyć kod źródłowy umieszczający obiekty za pomocą trzech 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.



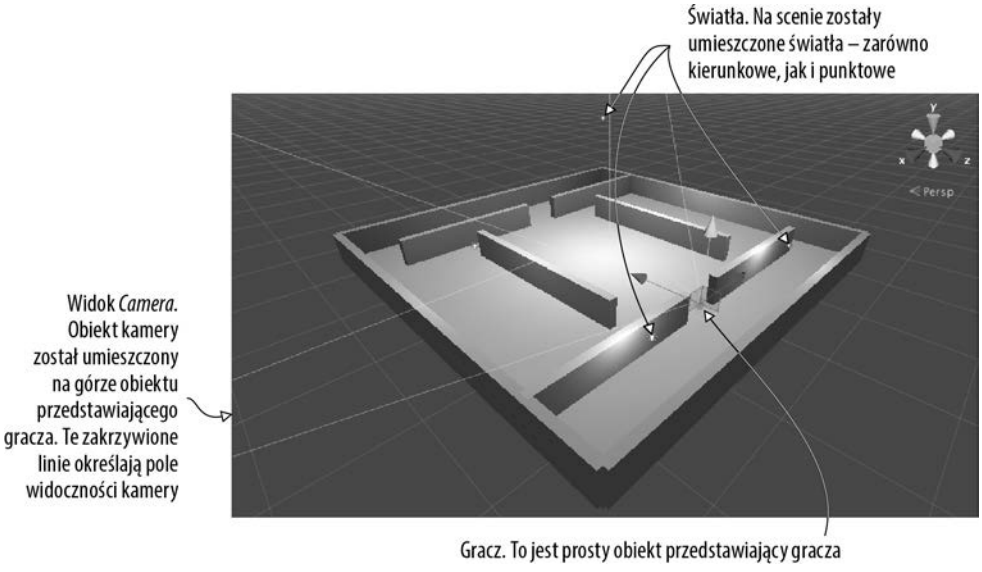
Kierunek wskazywany przez oś Z zależy od używanej dłoni

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 podłoże 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

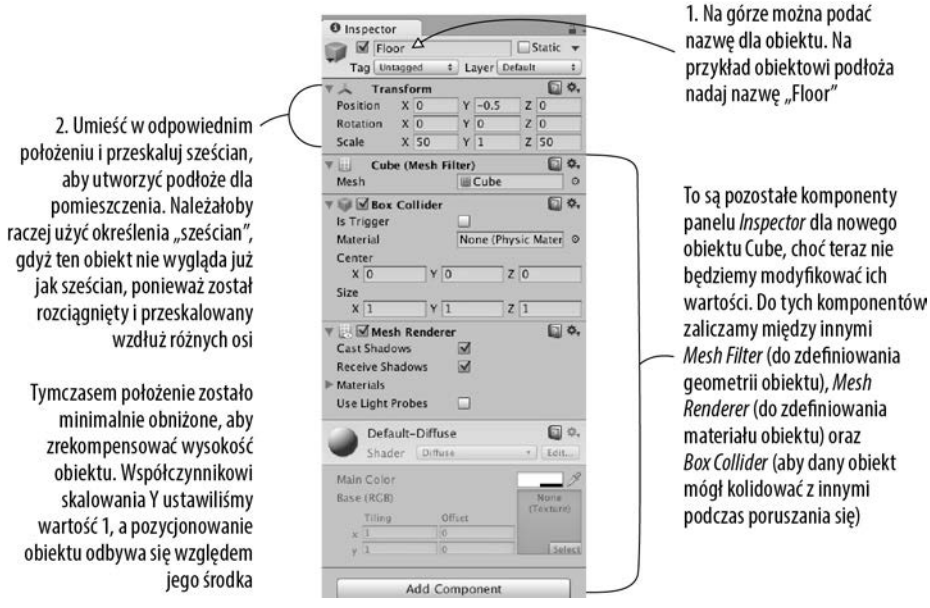
2.2.1. Sceneria — podłoże, ściany zewnętrzne i wewnętrzne

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”).

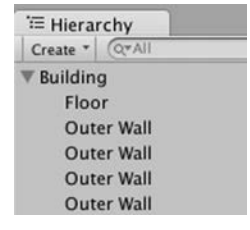
> **WSKAZÓWKA** Przypomnij sobie również o kontrolkach nawigacyjnych, za pomocą których można oglądać scenę pod różnymi kątami oraz na odmiennych



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 Przeciąganie obiektów na siebie w panelu *Hierarchy* powoduje zdefiniowanie połączeń między nimi. Obiekt, który ma dołączone do siebie inne obiekty, jest określany mianem **obiektu 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 świateł.

2.2.2. Światła i kamera

Scena 3D jest oświetlana najczęściej za pomocą światła kierunkowego oraz serii świateł 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*.

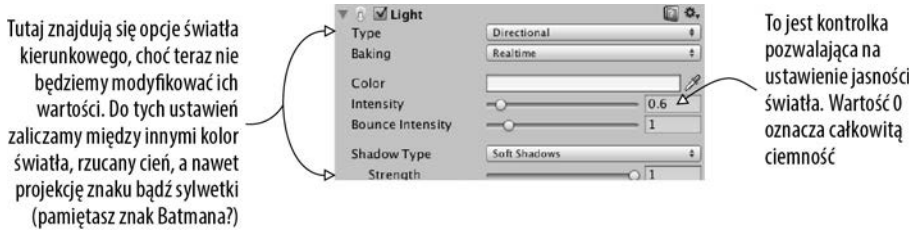
Rodzaje światła
Istnieje możliwość utworzenia wielu różnych typów światła zdefiniowanych na podstawie tego, jak i gdzie emitują one promienie światła. Do dyspozycji mamy trzy rodzaje światła: punktowe, reflektor i kierunkowe.

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.

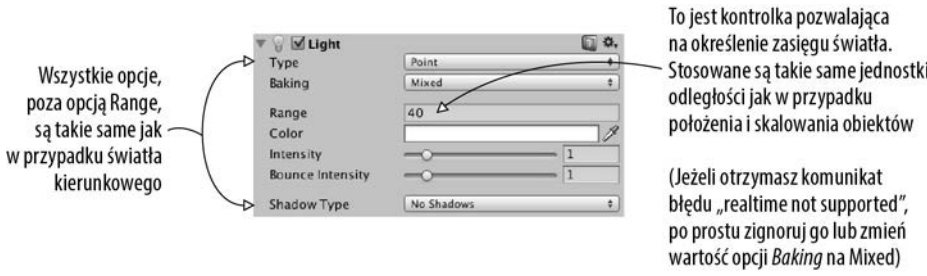
Ponieważ inaczej niż w przypadku rotacji położenie światła kierunkowego nie wpływa na kierowane przez niego promienie światła, z technicznego punktu widzenia można je umieścić w dowolnym miejscu na scenie. Zalecam umieszczenie tego światła ponad pomieszczeniem, aby intuicyjnie było odbierane jako słońce, 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 świateł, 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 świateł, aby mieć pewność, że wszystkie ściany zostały oświetlone. Nie przesadzaj z liczbą tych świateł, ponieważ jeżeli w grze umieścić zbyt wiele świateł, 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.

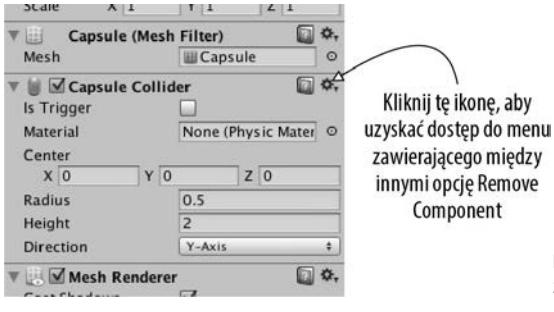


Rysunek 2.9. Ustawienia światła punktowego wyświetlane w panelu Inspector

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 pomieszczenia 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)`.

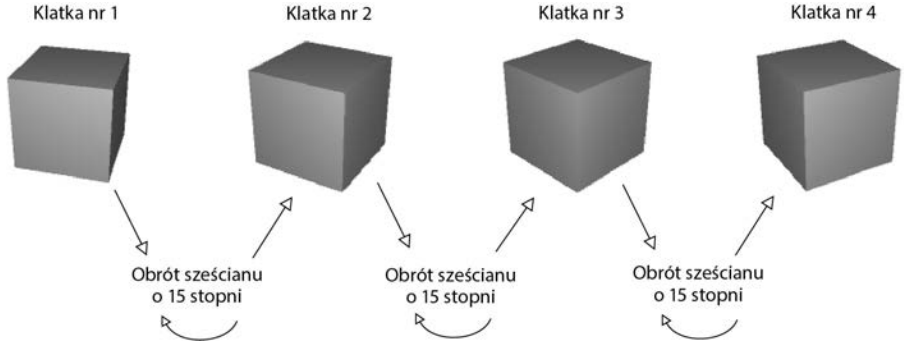
W ten sposób utworzyliśmy wszystkie obiekty niezbędne w budowanej scenie. Pozostało już tylko napisanie kodu źródłowego odpowiedzialnego za poruszanie obiektem gracza.

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 liczby zmiennoprzecinkowe jako typu double, czyli podwójnej precyzji.

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

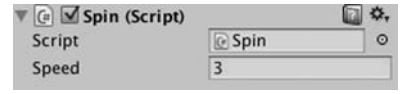
W celu dodania komponentu skryptu do obiektu gracza należy go przeciągnąć z panelu *Project* i upuścić na obiekcie `Player` w panelu *Hierarchy*. Teraz po naciśnięciu przycisku *Start* zobaczysz, że widok się obraca. W ten sposób przygotowałeś kod wprawiający obiekt w ruch! Ten nowy kod składa się w 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

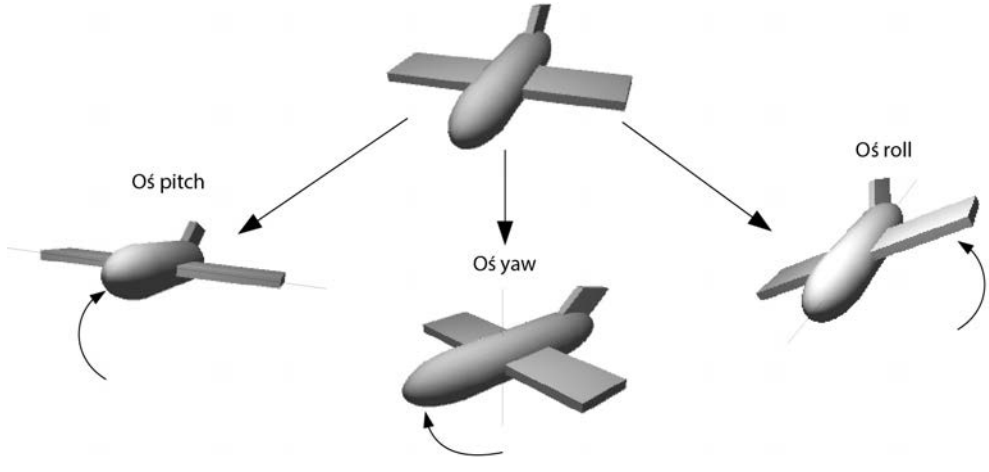
komponentów w różnych obiektach poprzez możliwość pracy za pomocą edytora graficznego zamiast używania wartości definiowanych na stałe w kodzie.



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

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 opisania 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 opisania 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!

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

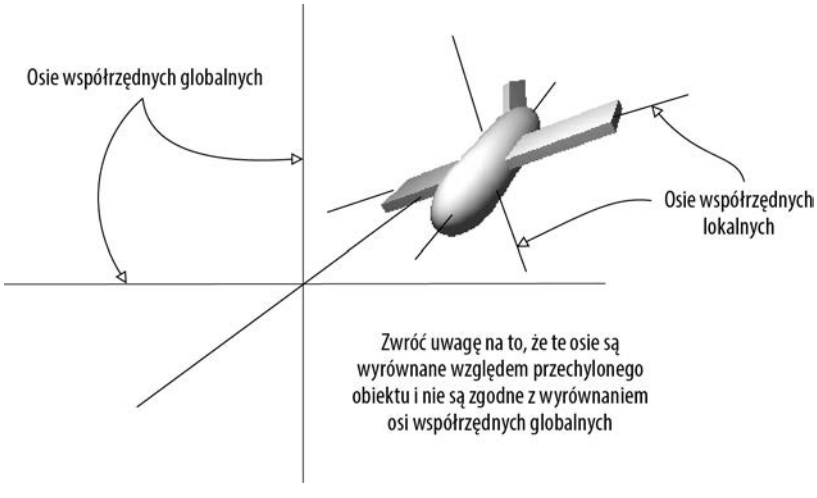
2.3.3. Współrzędne lokalne kontra globalne

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 pokazałem 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 przechyl (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 sześcian 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 wyliczeniowym 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 wyliczeniowego 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 samym 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 zwrotne 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. Wprawdzie 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;

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

private float _rotationX = 0;
```

Deklaracje zmiennych używanych w rotacji pionowej.

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 wątpienia 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 *Rigibody* 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.

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

Jeżeli zgubiłeś się podczas wprowadzania różnych zmian w trakcie pracy nad skryptem *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);
  }
}
```

Wprawdzie 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 pokazałem 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 (obiektu kodu — nie pomyl 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 zwrotną 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 pomyl 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

G

H

I

J

K

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

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/nowosci

ISBN 978-83-283-3519-6

Informatyka w najlepszym wydaniu

cena: 69,00 zł