Rob Miles

Microsoft® XNA® Game Studio 4.0

Projektuj i buduj gry dla konsoli Xbox 360®, urządzeń z systemem Windows® Phone 7 i własnego PC

Helion

Tytuł oryginału: Microsoft® XNA® Game Studio 4.0: Learn Programming Now!: How to program for Windows Phone 7, Xbox 360, Zune devices, and more

Tłumaczenie: Mikołaj Szczepaniak (wstęp, rozdz. 1 – 3, 10 – 16);
Jacek Kowolik (rozdz. 4 – 9, 17 – 19)

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-246-3410-1

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/games4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.
Printed in Poland.

Spis treści

Część III Pisanie poprawnych gier

Rozdział 3.

Pobieranie danych wejściowych od gracza

W tym rozdziale:

- Dowiesz się, jak we frameworku Microsoft XNA są reprezentowane pady i klawiatury.
- Odkryjesz struktury języka programowania C#, które umożliwiają uzyskiwanie danych wejściowych od gracza.
- Napiszesz kilka naprawdę niezbyt mądrych gier, które skutecznie przestraszą graczy.

Wprowadzenie

Poznałeś już podstawy programowania gier komputerowych. Wiesz, że program jest w istocie sekwencją wyrażeń, z których każde wykonuje pojedynczą czynność. Przekonałeś się, że wyrażenia umieszcza się w metodach odpowiedzialnych za określone zadanie oraz że metody (wraz z potrzebnymi danymi) należą do klas. Sam program operuje na wartościach danych, które są przechowywane w zmiennych określonych typów. Program podejmuje też decyzje na podstawie wartości zawartych w tych zmiennych. (Jeśli uważasz, że żadne z powyższych zdań nie ma sensu, czytaj tak długo rozdział 2., zatytułowany „Programy, dane i ładne kolory", aż te stwierdzenia będą dla Ciebie oczywiste).

W tym rozdziale będziesz miał okazję poszerzyć swoją wiedzę o techniki uzyskiwania danych wejściowych ze świata zewnętrznego, dzięki którym gry mogą reagować na zachowania gracza. Szybko odkryjesz, że po uzyskaniu tej wiedzy spektrum Twoich możliwości w roli twórcy gier komputerowych będzie nieporównanie szersze. Będziesz potrafił napisać kilka niezbyt jeszcze przemyślanych gier, w tym *Color Nerve*, *Mind Reader*, *The Thing That Goes Bump in the Night* oraz *Gamepad Racer*.

Zanim jednak przystąpimy do omawiania padów, musimy zdecydować, jak powinien działać Twój przyszły program. Przeanalizuj następujące wyrażenie języka C# zaczerpnięte z metody `Update` w ramach poprzedniej wersji programu nastrojowej lampy:

```
if (redCountingUp) redIntensity++;
```

To jeden z testów umożliwiających kontrolę intensywności czerwonego składnika wyświetlanego koloru. Wyrażenie w tej formie można opisać słowami: „Jeśli zmienna logiczna `redCountingUp` ma wartość `true`, zwiększ wartość zmiennej `redIntensity` o 1".

Projekt programu: kontroler wielokolorowej lampy

W rozdziale 2. stworzyłeś prostą grę — swoiste światło, które zmieniało kolor w czasie. Wspomniałem też, że podobne rozwiązania najpewniej będą w przyszłości stosowane na statkach kosmicznych. Lampa zmieniająca kolor być może nie nadaje się do czytania książek, ale skutecznie poprawia nastrój. Przyjmijmy teraz, że kapitan naszego statku kosmicznego potrzebuje lampy, której kolor sam będzie mógł ustawić. Oznacza to, że tym razem przygotujesz lampę sterowaną za pośrednictwem pada konsoli Xbox. Użytkownik będzie naciskał czerwony, niebieski, zielony i żółty przycisk na padzie, aby zwiększać intensywność poszczególnych barw. Aby stworzyć opisane rozwiązanie, musisz dowiedzieć się, jak odczytywać stan pada.

Przytoczone wyrażenie jest przetwarzane dla każdego wywołania metody `Update` (obecnie ta częstotliwość wynosi 60 razy na sekundę), zatem jeśli zmienna `redCountingUp` ma wartość `true`, intensywność czerwonej barwy będzie stopniowo zwiększana.

Chcesz teraz napisać kod, który będzie wyrażał zdanie: „Jeśli na pierwszym padzie jest naciśnięty czerwony przycisk, zwiększ wartość zmiennej `redIntensity` o 1”. Jeśli gracz będzie przytrzymywał czerwony przycisk, ekran będzie stawał się coraz bardziej czerwony. Oznacza to, że jedyna niezbędna zmiana będzie polegała na takiej modyfikacji przytoczonego testu, aby odczytywał on stan przycisku na padzie. Stworzenie lampy sterowanej przez użytkownika jest więc zadziwiająco proste.

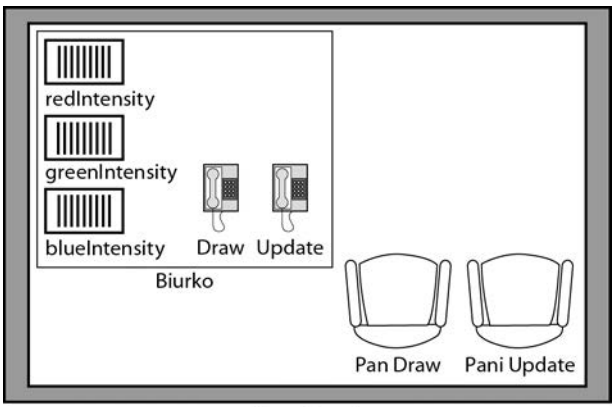
Odczytywanie stanu pada

Pady w rzeczywistości są bardzo skomplikowanymi urządzeniami. Są połączone z głównym urządzeniem albo za pośrednictwem magistrali USB (od ang. *universal serial bus*), albo przez połączenie bezprzewodowe. Sposób współpracy programów z padami nie zależy od sposobu ich połączenia z konsolą czy komputerem. Połączenie z padem można wykorzystać zarówno do odczytywania stanu przycisków i gałek analogowych, jak i do wysyłania poleceń do pada (na przykład w celu włączenia lub wyłączenia efektu wibracji). Konsola Xbox i framework XNA oferują obsługę maksymalnie czterech podłączonych jednocześnie padów.

Pady i klasy

Informacje o stanie pada są reprezentowane we frameworku XNA przez składowe klasy nazwanej `GamePadState`. Zadaniem tej klasy jest zapewnianie połączenia pomiędzy programem a fizycznym padem w dłoniach gracza. Aby zrozumieć, jak należy używać tej klasy, musisz pogłębić swoją wiedzę na temat działania wszystkich klas.

Z lektury punktu „Gry i klasy" zamieszczonego w rozdziale 2. wiesz już, czym jest klasa. Zawiera ona dane (zmienne, które mogą reprezentować rozmaite informacje) oraz metody (kod wykonujący operacje). Klasę możesz traktować jako odpowiednik biura z centralnie ustawionym biurkiem, na którym leżą zmienne, oraz pracownikami pełniącymi podobne funkcje jak metody. Na rysunku 3.1 pokazano plan biura dla klasy `Game1`, która — jak już wiesz — jest podstawowym elementem gry opartej na frameworku XNA.



RYSUNEK 3.1. Klasa Game1 jako plan biura

Klasa zawiera pewne zmienne trzymane na biurku (w tym przypadku zmienne reprezentują intensywność poszczególnych barw) oraz dwie metody, które nazwaliśmy panem `Draw` i panią `Update`. Każda metoda dysponuje własnym telefonem. Programy mogą dzwonić na te telefony, aby żądać od poszczególnych metod wykonywania swoich zadań.

> **Wielka programistka mówi: w rzeczywistości klasy to nie biura** Nasza wielka programistka przeczytała ten tekst i stwierdziła, że jest śmieszny. Mówi, że klasy nie przypominają biur, ale uznała, że skoro to porównanie ma ułatwić zrozumienie sposobu konstruowania programów, w takim postrzeganiu klas nie ma niczego złego.

Podczas uruchamiania gry system XNA tworzy **egzemplarz** (ang. *instance*) klasy `Game1`, który będzie bezpośrednim adresatem żądań wykonania metod `Draw` i `Update`. Podczas tworzenia egzemplarza klasy instrukcje zawarte w metodach tej klasy są ładowane do pamięci, a na potrzeby zmiennych należących do tego egzemplarza jest przydzielana przestrzeń w pamięci.

Pliki klas stanowią źródło swoistych planów klas, dzięki czemu podczas działania programu istnieje możliwość tworzenia egzemplarzy poszczególnych klas. W otaczającym nas świecie stworzenie biura gry wymaga budowy pomieszczenia, umieszczenia w nim biurka i kilku telefonów oraz zatrudnienia pana `Draw` i pani `Update`. Proces tworzenia egzemplarza klasy

jest dość podobny. Okazuje się jednak, że dla oszczędności pamięci działający program posługuje się tylko jedną kopią kodu metody — ta jedna kopia jest współdzielona przez wszystkie egzemplarze tej samej klasy.

Uwaga Pamiętaj, że opisane operacje są realizowane w czasie wykonywania programu. Proces tworzenia egzemplarzy klas nie jest wykonywany przez kompilator. Zadaniem kompilatora jest konwersja kodu źródłowego języka C# na instrukcje, które będzie można wykonać na docelowym urządzeniu. W momencie, w którym Twój program przejmuje kontrolę nad tym urządzeniem, rola kompilatora się kończy, a komputer wykonuje wygenerowane przez ten kompilator rozkazy języka maszynowego.

Znajdowanie pada

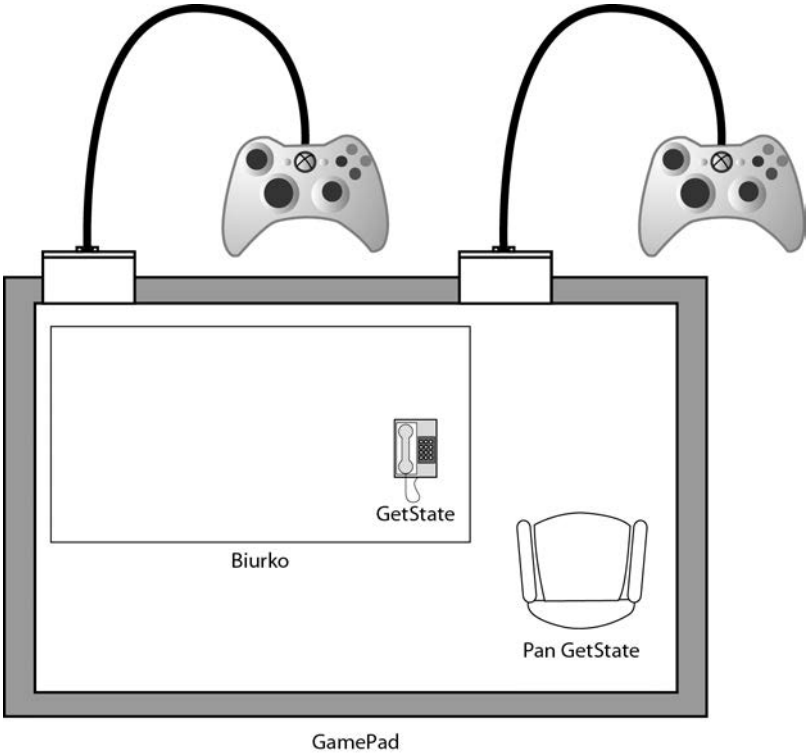
Framework XNA przejmuje odpowiedzialność także za wiele innych aspektów działania gry. Jednym z dostępnych rozwiązań jest klasa `GamePad` połączona ze wszystkimi padami. Nie musisz wiedzieć, jak pad jest fizycznie połączony z urządzeniem. Możesz postrzegać ten proces jako mikroskopijne chochliki wędrujące w przewodzie i przenoszące na jeszcze mniejszych karteczkach komunikaty, np: „Pierwszy gracz nacisnął czerwony przycisk". Na rysunku 3.2 pokazano, jak wyglądałaby klasa `GamePad`, gdyby była biurem.

Klasa `GamePad` zawiera metodę, nazwaną `GetState`, która zwraca bieżący stan jednego z padów. Po wywołaniu metody `GetState` jej kod lokalizuje jeden z padów, odczytuje jego ustawienia, po czym wysyła uzyskane informacje z powrotem do wyrażenia, które wywołało tę metodę (i które może teraz użyć tych danych).

Metoda `GetState` otrzymuje na wejściu parametr identyfikujący pad, którego stan ma być odczytany. **Parametr** ten umożliwia przekazywanie informacji do metody z poziomu jej wywołania. Z parametrami mieliśmy do czynienia już wcześniej — w Twoim pierwszym programie, gdzie przekazywałeś parametry typu `Color` do metody `Clear`, aby wskazać kolor, który miał być użyty do wypełnienia ekranu.

W przypadku metody `GetState` parametr identyfikuje pad, którego stan chcesz odczytać. Jeśli porównujesz ten mechanizm z funkcjonowaniem biur, możesz traktować parametry jako fragmenty instrukcji przekazywanych współpracownikom przez telefon. Kiedy pan `GetState` odbiera telefon, słyszy w słuchawce: „Podaj mi stan pierwszego pada". Informacje na temat stanu tego pada są następnie odsyłane do struktury `GamePadState` (patrz rysunek 3.3).

Jeśli chcesz, możesz traktować parametry jako zbiór elementów wypełnionego formularza, jednak w rzeczywistości jest to struktura języka C# zawierająca zarówno składowe widoczne na rysunku 3.3, jak i pewne dane dodatkowe.

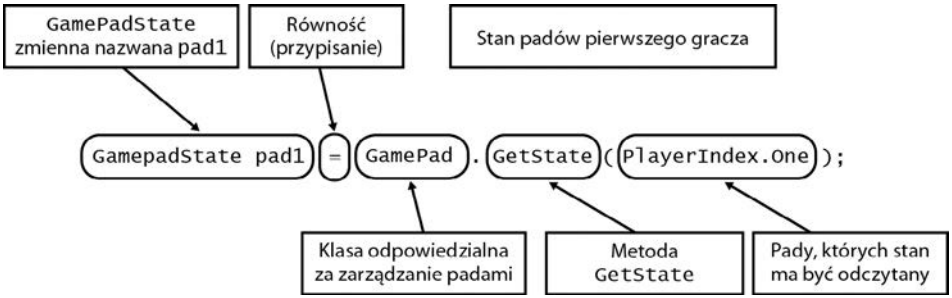


RYSUNEK 3.2. Klasa GamePad jako biuro

GamePadState	
Przyciski	
Zielony (A)	ButtonState.Pressed
Czerwony (B)	ButtonState.Released
Niebieski (X)	ButtonState.Released
Żółty (Y)	ButtonState.Released
Start	ButtonState.Released
Back	ButtonState.Released

RYSUNEK 3.3. Struktura GamePadState reprezentująca stan pada z naciśniętym zielonym przyciskiem (A)

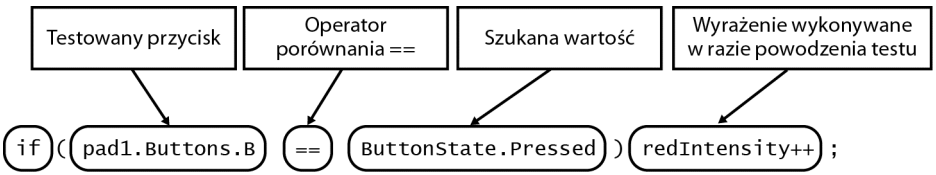
Jeśli więc pani `Update` chce uzyskać informacje na temat stanu jednego z padów konsoli Xbox, wywołuje metodę `GetState` klasy `GamePad` i prosi: „Czy możesz mi przekazać stan pada pierwszego gracza"? Pan GetState zrywa się z fotela, wypełnia formularz `GamePadState` i odsyła go do pani `Update`. Na rysunku 3.4 pokazano wyrażenie języka C# uzyskujące stan pada i przypisujące ten stan do zmiennej typu `GamePadState`.



RYSUNEK 3.4. Uzyskiwanie statusu pada

Testowanie statusu pada

Skoro dysponujesz już statusem pada, możesz wykorzystać ten status w swoim programie do sprawdzenia, czy gracz wcisnął jakiś przycisk. Na rysunku 3.5 pokazano kompletne wyrażenie języka C#, które wykonuje odpowiedni test.



RYSUNEK 3.5. Testowanie przycisku na padzie

Wyrażenie w tej formie porównuje stan czerwonego przycisku (*B*) z wartością `ButtonState.Pressed`. Jeśli obie wartości są równe, przycisk jest wciśnięty, zatem metoda `Update` powinna zwiększyć intensywność barwy czerwonej. Tych samych zasad możesz używać do zarządzania wartościami reprezentującymi intensywność barw niebieskiej i zielonej, co oznacza, że w swojej grze możesz posługiwać się metodą `Update` zawierającą następujący kod:

```
protected override void Update(GameTime gameTime)
{
  // Umożliwia wyjście z gry.
  if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    this.Exit();

  GamePadState pad1 = GamePad.GetState(PlayerIndex.One);

  if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
  if (pad1.Buttons.X == ButtonState.Pressed) blueIntensity++;
  if (pad1.Buttons.A == ButtonState.Pressed) greenIntensity++;

  base.Update(gameTime);
}
```

W opisanej metodzie `Update` brakuje już tylko obsługi żółtego przycisku. Gdy gracz wciska żółty przycisk, program powinien zwiększyć intensywność barw zielonej i żółtej, zatem w przypadku spełnienia odpowiedniego warunku musi wykonać dwa wyrażenia. Okazuje się, że realizacja tego zadania jest bardzo łatwa — wystarczy umieścić te dwa wyrażenia w bloku kontrolowanym przez odpowiedni warunek, tak jak w poniższym kodzie:

```
if (pad1.Buttons.Y == ButtonState.Pressed)
{
  redIntensity++;
  greenIntensity++;
}
```

Z blokami miałeś do czynienia już wcześniej. Blokiem jest na przykład ciało metody (sekwencja wyrażeń składających się na tę metodę). W języku programowania C# **blok** ma postać pewnej liczby wyrażeń otoczonych nawiasami klamrowymi. Jeśli warunek w wyrażeniu `if` jest spełniony (ma wartość `if`), powyższy kod wykonuje oba wyrażenia, ponieważ znajdują się one w bloku kontrolowanym przez ten warunek.

> **Wielka programistka mówi: bloki to podstawa** Nasza wielka programistka stara się stosować bloki za wyrażeniami `if`, nawet jeśli formalnie nie jest to konieczne. Mówi, że takie rozwiązanie poprawia czytelność tekstu programu i że znacznie ułatwia dodawanie ewentualnych wyrażeń w przyszłości (jeśli zajdzie taka potrzeba).

Jeśli umieścisz powyższe wyrażenia w metodzie `Update` w ramach jednej z wcześniejszych wersji programu *Mood Light*, otrzymasz komunikaty ostrzeżeń kompilatora, ponieważ nowa wersja metody `Update` nie będzie używała wszystkich zmiennych utworzonych w starszych wersjach programu. Aby wyeliminować te ostrzeżenia, musisz usunąć wyrażenia tworzące nieużywane zmienne. Wielka programistka nie lubi, gdy programy zawierają zmienne, które nigdzie nie są używane. Mówi, że to wygląda nieprofesjonalnie, z czym trudno się nie zgodzić.

> **Przykładowy kod: ręczna, wielokolorowa lampa** Wszystkie przykładowe projekty są dostępne pod adresem *ftp://ftp.helion.pl/przyklady/games4.zip*. Przykładowy projekt z omówioną w tym punkcie metodą `Update` znajduje się w katalogu *01 Manual MoodLight* w zasobach kodu źródłowego dla tego rozdziału. Możesz zwiększać jasność kolorów na ekranie, naciskając przyciski na swoim padzie.

Stosowanie klawiatury

Framework XNA obsługuje nie tylko pady, ale też klawiatury. Wielu użytkowników ze zdziwieniem odkrywa możliwość podłączenia klawiatury USB do konsoli Xbox 360 i używania jej tak jak w przypadku komputerów PC. Jeśli chcesz napisać program

Pomysł na grę: Color Nerve

Od czasu do czasu będziemy podejmowali próby tworzenia nowych koncepcji gier. Początkowo nasze gry będą bardzo proste, ale z czasem tworzone programy staną się coraz bardziej skomplikowane i interesujące. Do utworzenia swojej pierwszej gry możesz użyć kodu zawartego w projekcie *Manual MoodLight*. Gra wykorzystuje też pewien aspekt opisany w rozdziale 2. Jak zapewne pamiętasz, jeśli stale zwiększasz jakąś wartość, prędzej czy później osiąga ona punkt, w którym nie mieści się w przydzielonym dla niej obszarze pamięci, powodując tzw. przepełnienie. Właśnie dlatego ekran wypełniony jasnym, białym kolorem nagle stawał się czarny. Okazuje się, że ten sam efekt możesz wykorzystać do utworzenia swojej pierwszej „bardzo niemądrej gry".

Color Nerve to gra dla co najmniej dwóch graczy. Każdy gracz w swojej kolejce naciska na padzie jeden lub wiele przycisków. (Pozostali gracze muszą uważnie obserwować ekran, aby mieć pewność, że gracz, do którego należy ruch, rzeczywiście nacisnął przynajmniej jeden przycisk). Każdy gracz może nacisnąć w swojej kolejce dowolną liczbę przycisków. Jeśli jednak ekran nagle zmieni kolor (z powodu zmiany intensywności przynajmniej jednej z barw z 255 na 0), gracz przegrywa, a dalszą rozgrywkę prowadzą pozostali gracze. Ostatni gracz, który pozostaje w grze, jest zwycięzcą.

Gra może mieć przebieg taktyczny. Przyciski można naciskać bardzo krótko, ale niektórzy gracze na początku gry mogą wykazać się mocnymi nerwami i przytrzymywać przyciski przez dłuższy czas, wpędzając w tarapaty pozostałych graczy. Mogą oni też podejmować próby odgadywania, która barwa „przekręciła licznik", tak aby nieco dłużej przytrzymać odpowiedni przycisk w swoim ruchu. Gra sprawdza się na imprezach, ponieważ może w niej uczestniczyć wiele osób, a jej zasady są wyjątkowo proste. W rozdziale 4., zatytułowanym „Wyświetlanie obrazów", udoskonalisz tę grę, dodając obrazy w miejsce pustego ekranu.

współpracujący z klawiaturą, wystarczy, że dodasz do swojego programu odpowiedni kod, na przykład taki jak poniżej:

```
KeyboardState keys = Keyboard.GetState();

if (keys.IsKeyDown(Keys.R)) redIntensity++;
if (keys.IsKeyDown(Keys.B)) blueIntensity++;
if (keys.IsKeyDown(Keys.G)) greenIntensity++;
if (keys.IsKeyDown(Keys.Y))
{
    redIntensity++;
    greenIntensity++;
}
```

Łatwo zauważyć, że prezentowany proces bardzo przypomina działanie padów, ale istnieją też nieznaczne różnice. Na potrzeby metody `GetState` nie musisz określać klasy `Keyboard`, dla której klawiatury chcemy uzyskać stan, ponieważ framework XNA obsługuje tylko jedną klawiaturę. Egzemplarz klasy `KeyboardState` zwracany przez to wywołanie

w rzeczywistości nie jest kartką papieru — jest obiektem udostępniającym metody, które możesz w swoim programie wykorzystać do określenia, czy poszczególne klawisze są wciśnięte. Zamiast sprawdzać, czy stan przycisku ma przypisaną wartość `ButtonState.Pressed`, program może wykonać metodę `IsKeyDown`. Na wejściu metody `IsKeyDown` przekazujesz parametr identyfikujący klawisz, którego stan chcesz sprawdzić, jak w poniższym przykładzie:

```
if (keys.IsKeyDown(Keys.R)) redIntensity++;
```

Kod jest wyrażeniem warunkowym zwiększającym wartość zmiennej `redIntensity`, w przypadku gdy klawisz *R* jest wciśnięty. Jeśli wskazany klawisz jest wciśnięty, metoda `IsKeyDown` zwraca wartość `true` — w przeciwnym razie zwraca wartość `false`. Oznacza to, że możemy wykorzystać to wyrażenie do sterowania aktualizowaniem wartości zmiennej `redIntensity`.

Zatrzymywanie gry za pomocą klawisza Escape

Metoda `Update` generowana w czasie tworzenia nowej gry na bazie frameworku XNA zawiera test, który sprawdza, czy na pierwszym padzie wciśnięto przycisk *Back* (wstecz), oraz wywołuje metodę `Exit` (zatrzymującą grę), jeśli ten warunek jest spełniony. Jeśli używasz klawiatury zamiast pada, nie będziesz mógł nacisnąć tego przycisku i — tym samym — zatrzymać gry. Możesz jednak dodać test sprawdzający, czy na klawiaturze nie wciśnięto klawisza *Escape*. Klawisz *Escape* należy do tzw. klawiszy sterujących (ang. *control key*). Nie reprezentują one znaków drukowalnych, a jedynie mają sygnalizować działania, których oczekujemy od programu. Innymi przykładami klawiszy sterujących są *Enter* i *Backspace*. Do sprawdzania, czy naciśnięto klawisz *Escape*, służy ta sama metoda `IsKeyDown`.

```
if (keys.IsKeyDown(Keys.Escape)) Exit();
```

Powyższy kod zatrzymuje grę w momencie naciśnięcia klawisza *Escape*.

Jednoczesne używanie pada i klawiatury

Jeśli chcesz jednocześnie używać pada i klawiatury, musisz testować stan obu tych urządzeń. Oznacza to, że metoda `Update` powinna mieć teraz następującą postać:

```
protected override void Update(GameTime gameTime)
{
  GamePadState pad1 = GamePad.GetState(PlayerIndex.One);

  if (pad1.Buttons.Back == ButtonState.Pressed) Exit();
  if (pad1.Buttons.B == ButtonState.Pressed) redIntensity++;
  if (pad1.Buttons.X == ButtonState.Pressed) blueIntensity++;
  if (pad1.Buttons.A == ButtonState.Pressed) greenIntensity++;
```

```
  if (pad1.Buttons.Y == ButtonState.Pressed)
  {
    redIntensity++;
    greenIntensity++;
  }

  KeyboardState keys = Keyboard.GetState();

  if (keys.IsKeyDown(Keys.Escape)) Exit();

  if (keys.IsKeyDown(Keys.R)) redIntensity++;
  if (keys.IsKeyDown(Keys.B)) blueIntensity++;
  if (keys.IsKeyDown(Keys.G)) greenIntensity++;
  if (keys.IsKeyDown(Keys.Y))
  {
    redIntensity++;
    greenIntensity++;
  }
  base.Update(gameTime);
}
```

Kod w tej formie jest niedoskonały, ponieważ wykonujemy tę samą czynność dwukrotnie, tyle że w wyniku dwóch różnych testów. Gdyby wielka programistka miała kiedykolwiek okazję zapoznać się z tym kodem, z pewnością nie byłaby zachwycona. Na szczęście język C# oferuje możliwość łączenia w programie dwóch warunków i wykonywania kodu, jeśli przynajmniej jeden z nich jest spełniony (ma wartość `true`). Warunki można łączyć za pomocą logicznego operatora alternatywy (*lub*), którego wynik jest prawdziwy, jeśli przynajmniej jeden z jego operandów ma wartość `true`. W kodzie programu operator zapisywany jest w formie dwóch pionowych kresek (`||`):

```
GamePadState pad1 = GamePad.GetState(PlayerIndex.One);
KeyboardState keys = Keyboard.GetState();

if (pad1.Buttons.B == ButtonState.Pressed ||
    keys.IsKeyDown(Keys.R)) redIntensity++;
```

Operator logiczny *lub* (ang. *or*) umieszcza się pomiędzy dwoma wyrażeniami logicznymi, które mogą mieć albo wartość `true`, albo wartość `false`. Jeśli choć jedno z tych wyrażeń ma wartość `true`, cały warunek logiczny także ma wartość `true`.

W powyższym kodzie wartość zmiennej `redIntensity` jest zwiększana, jeśli wciśnięty jest czerwony przycisk na padzie lub przycisk *R* na klawiaturze (lub oba jednocześnie). Dokładnie o takie działanie właśnie nam chodziło. Co więcej, w grę *Color Nerve* można teraz grać zarówno przy użyciu pada, jak i przy użyciu klawiatury (lub korzystając z obu tych urządzeń jednocześnie). W tym i innych przypadkach mówi się o operatorach logicznych, ponieważ każdy z nich zwraca wartość logiczną, a nie wynik liczbowy. Istnieją jeszcze inne operatory logiczne, których będziesz używał podczas tworzenia bardziej skomplikowanych programów.

Uwaga Jeśli zrozumienie działania tego operatora logicznego sprawia Ci kłopot, wróć do istoty problemu, który próbujesz rozwiązać. Chcesz na przykład, aby Twój program wykonał pewne wyrażenie (w tym przypadku `redIntensity++`), wtedy gdy wciśnięto czerwony przycisk na padzie lub gdy wciśnięto klawisz *R* na klawiaturze. Jeśli przynajmniej jeden z tych warunków jest spełniony, wówczas posługujesz się operatorem *lub* (`||`) do połączenia obu testów i wykonania odpowiedniego wyrażenia.

Przykładowy kod: Color Nerve Przykładowy projekt implementujący tę grę można znaleźć w katalogu *02 Color Nerve* w zasobach dla tego rozdziału. Możesz zmieniać kolory na ekranie, naciskając przyciski na padzie lub klawisze na klawiaturze.

Dodawanie wibracji

Komunikacja pomiędzy padem a grą odbywa się w obu kierunkach. Twoje możliwości nie ograniczają się tylko do odczytywania stanu przycisków pada — możesz dodatkowo wysyłać do pada polecenia włączające lub wyłączające mechanizm wibracji. Także w tym przypadku nie musisz dysponować szczegółową wiedzą o sposobie dostarczania odpowiednich komunikatów. Wystarczy znajomość funkcji frameworku XNA używanych do sterowania efektami wibracji.

Oznacza to, że Twoja gra *Color Nerve* może być jeszcze bardziej wciągająca i atrakcyjna, jeśli pad będzie wibrował w czasie zbliżania się wartości intensywności barw do ich górnych granic. To ciekawe, jak podobne rozwiązania mogą poprawić oceny nawet najprostszych gier. W kolejnych kilku grach będziesz bardzo często korzystał z efektu wibracji.

Sterowanie wibracjami pada

Klasa `GamePad` udostępnia metodę, nazwaną `SetVibration`, która umożliwia programowi sterowanie silnikami odpowiedzialnymi za wibracje:

```
GamePad.SetVibration(PlayerIndex.One, 0, 1);
```

Metoda `SetVibration` używa trzech parametrów. Pierwszy parametr identyfikuje pad, który ma wibrować. Drugi parametr, który musi mieć wartość z przedziału od `0` do `1`, steruje natężeniem wibracji generowanych przez lewy silnik. Im większa jest ta liczba, tym bardziej intensywne są wibracje pada. Trzeci parametr steruje prawym silnikiem dokładnie tak samo, jak drugi parametr steruje lewym silnikiem. Powyższe wyrażenie powoduje, że prawy silnik pierwszego pada wibruje z pełną prędkością., natomiast lewy silnik jest odpowiedzialny za wibracje z niską częstotliwością.

Jeśli sobie wyobrazisz, że biuro `GamePad` zatrudnia pana `SetVibration`, będziesz mógł przyjąć, że ten pracownik otrzymuje polecenia ustawiania wibracji konkretnego pada

wraz z parametrami obrotów lewego i prawego silnika. Po wywołaniu tej metody pad zaczyna wibrować i wibruje tak długo, aż kolejne wywołanie tej metody nie zmieni ustawień jego silników. Innymi słowy, metodę `SetVibration` można traktować jako przełącznik, który można ustawiać w wielu różnych pozycjach. Początkowo oba silniki pada mają ustawioną szybkość równą `0`, co oznacza brak wibracji.

Testowanie wartości intensywności

Gra musi oczywiście zdecydować, w którym momencie należy włączyć wibracje. W tym celu musi sprawdzać wartości reprezentujące intensywność barw i włączać silnik wibracji, jeśli któraś z tych wartości będzie zbyt duża. Program może na przykład włączać silniki, jeśli intensywność którejkolwiek z barw — czerwonej, zielonej lub niebieskiej — będzie większa niż `220`. Oznacza to, że program musi sprawdzać wartości intensywności w następujący sposób:

```
if (redIntensity > 220)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
```

Powyższy kod zawiera nową formę warunku. We wcześniejszych przykładach posługiwaliśmy się warunkami sprawdzającymi, czy dwie wartości były sobie równe. Tym razem kod sprawdza, czy jedna wartość jest większa od drugiej. Znak większości (`>`) jest kolejnym przykładem operatora logicznego. Operator umieszczony pomiędzy dwiema wartościami zwraca wartość `true`, jeśli wartość na lewo od tego operatora jest większa od wartości na prawo od operatora (w przeciwnym razie operator zwraca wartość `false`). Dokładnie takiego działania oczekiwałeś.

Powyższy kod powoduje, że w momencie przekroczenia wartości `220` przez zmienną reprezentującą intensywność barwy czerwonej pad zaczyna wibrować z powodu działania prawego silnika wibrującego. Jeśli dodasz ten kod do metody `Update` gry *Color Nerve*, przekonasz się, że pad zaczyna wibrować po zwiększeniu intensywności koloru czerwonego. Program zawiera jednak pewien błąd. Gdy intensywność barwy czerwonej spada do poziomu `0`, pad nie przestaje wibrować. Musisz jeszcze dodać kod, który wyłączy silnik w momencie, w którym wartość intensywności koloru spadnie poniżej poziomu `220`. Okazuje się, że jest to bardzo proste — wystarczy dodać część `else` do istniejącego warunku:

```
if (redIntensity > 220)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
else
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
}
```

Wyrażenie następujące po słowie `else` jest wykonywane wówczas, jeśli warunek nie jest spełniony, czyli jeśli ma wartość `false`. (Część `else` możesz dodać do dowolnego utworzonego przez siebie warunku `if`). Oznacza to, że gdy intensywność barwy czerwonej wraca do poziomu `0`, pad przestaje wibrować. Możesz dodatkowo rozszerzyć te testy za pomocą operatora **lub** (`||`), tak aby program testował wartości reprezentujące intensywność wszystkich barw:

```
if ( redIntensity > 220 ||
     greenIntensity > 220 ||
     blueIntensity > 220 )
{
    GamePad.SetVibration(PlayerIndex.One, 0, 1);
}
else
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
}
```

Wibracje pada są teraz zależne od wszystkich wartości reprezentujących intensywność barw. Grę możesz dodatkowo udoskonalić, eksperymentując z rozmaitymi rodzajami wibracji dla różnych kolorów, a nawet dodatkowo korzystając z silnika niskiej częstotliwości. Za sterowanie tym silnikiem odpowiada drugi parametr metody `SetVibration`:

```
GamePad.SetVibration(PlayerIndex.One, 1, 0);
```

Powyższy wiersz kodu włącza wibracje o niskiej częstotliwości. Możesz też eksperymentować z odmiennymi progami uruchamiania wibracji.

Program wciąż zawiera jednak pewne niedociągnięcie. Jeśli po jego uruchomieniu zostaną włączone wibracje pada, nie wyłączą się one same w momencie zakończenia programu. Musisz więc dodać kod wyłączający wibracje z chwilą opuszczania gry. Gra kończy się wtedy, gdy gracz naciska przycisk *Back* na swoim padzie. Odpowiedni test tego przycisku znajduje się w metodzie `Update`. Jeśli przycisk *Back* jest wciśnięty, następuje wywołanie metody `Exit`, która zatrzymuje i wyłącza grę:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    this.Exit();
```

Metoda `Exit` służy do eleganckiego usunięcia z ekranu elementów wizualnych gry i wyłączenia samego programu. Do Twojego programu należy odpowiedzialność za wyłączenie silników pada przed wywołaniem tej metody, dlatego w przypadku wciśnięcia przycisku *Back* Twój program musi wykonać dodatkowe wyrażenie. Potrzebujemy zatem kolejnego bloku:

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
{
    GamePad.SetVibration(PlayerIndex.One, 0, 0);
    this.Exit();
}
```

Teraz, gdy gracz naciśnie przycisk *Back*, aby zakończyć program, silniki wibrujące pada zostaną wyłączone.

Wielka programistka mówi: w razie wątpliwości sprawdź to sam Wielka programistka uważa, że jeśli znajdujesz się w sytuacji, w której nie jesteś pewien, czy jakiś warunek jest spełniony we wszystkich możliwych przypadkach, powinieneś dodać kod eliminujący potencjalne wątpliwości. Testując zachowanie opisanego w tym podrozdziale mechanizmu wibracji, odkryłem, że po opuszczeniu gry pad nadal wibruje tylko w starszych wersjach frameworku XNA, natomiast w nowszych wersjach wibracje są automatycznie wyłączane. Aby mieć absolutną pewność, że wibracje zostaną wyłączone niezależnie od wersji frameworku XNA, w której działa Twoja gra, powinieneś użyć powyższego kodu do samodzielnego wyłączenia wibracji.

Przykładowy kod: Color Nerve z wibracjami Katalog z kodem źródłowym dla tego projektu zawiera przykładowy projekt *03 Color Nerve with Vibes* z wersją gry *Color Nerve* korzystającą z efektu wibrowania padów.

Pomysł na grę: Tajne komunikaty w formie wibracji

Skoro już wiesz, że odczytywanie stanu przycisków pada i sterowanie jego silnikami wibrującymi jest dość proste, możesz przystąpić do tworzenia ciekawszych rozwiązań przy użyciu frameworku XNA, szczególnie jeśli dysponujesz padami bezprzewodowymi. Możesz tworzyć gry polegające na czytaniu w cudzych myślach, tak aby Twój asystent sprawiał wrażenie dysponowania umiejętnością precyzyjnego określania, o czym aktualnie myślisz. Osoby postronne nie będą wiedziały, że obaj ukryliście w kieszeniach pady konsoli Xbox i że używacie ich do przesyłania sobie sygnałów w formie wibracji. Kod realizujący to działanie jest zadziwiająco prosty — powinieneś bez trudu poradzić sobie z jego zrozumieniem:

```
protected override void Update(GameTime gameTime)
{
    // Umożliwia wyjście z gry.
    if(GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
        this.Exit();
    }

    GamePadState pad1 = GamePad.GetState(PlayerIndex.One);
    GamePadState pad2 = GamePad.GetState(PlayerIndex.Two);

    if (pad1.Buttons.A == ButtonState.Pressed)
    {
```

```
GamePad.SetVibration(PlayerIndex.Two, 0, 1);
    }
    else
    {
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
    }

    if (pad2.Buttons.A == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 1);
    }
    else
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
    }

    base.Update(gameTime);
}
```

Metoda `Update` odczytuje stan przycisku *A* na padzie pierwszego gracza. Jeśli przycisk jest wciśnięty, włącza szybki silnik wibracji w padzie drugiego gracza. Program powtarza ten sam proces w przeciwnym kierunku, przesyłając sygnał z drugiego pada do pierwszego pada. W ten sposób możesz łatwo przesyłać bezprzewodowe sygnały pomiędzy dwoma padami. Zwróć uwagę na części `else` w obu wyrażeniach warunkowych — jeśli przycisk nie jest wciśnięty, wibracje są wyłączane.

Ten sam mechanizm możesz wykorzystać także do praktycznych psikusów — możesz na przykład umieścić pad pod łóżkiem swojej ofiary, poczekać, aż zgasi światło i położy się spać. Spróbuj następnie włączyć maksymalne wibracje w ukrytym padzie. Tylko nie miej do mnie pretensji, jeśli już nigdy nie odzyskasz swojego pada!

Przykładowy kod: Komunikaty w formie wibracji Przykładowy projekt w katalogu *04 Mind Reader* w ramach zasobów z kodem źródłowym dla tego rozdziału zawiera wersję programu do przekazywania komunikatów w formie wibracji. Pamiętaj tylko, żeby mądrze korzystać z tego programu. Program dodatkowo powoduje wyświetlenie czarnego ekranu, zatem jego prawidłowe działanie może nie być oczywiste.

Pomysł na grę: Gamepad Racer

Ostatni pomysł gry proponowany w tym rozdziale jest co prawda wyjątkowo niemądry, ale może dostarczać mnóstwo zabawy. Pierwszą rzeczą, którą musisz zrobić, jest znalezienie wielkiego, możliwie gładkiego stołu. Umieść kilka książek pod nogami na jednym końcu stołu, tak aby jego powierzchnia nie była pozioma. Jeśli umieścisz

pad konsoli Xbox na wyższym końcu tego stołu i włączysz jego wibracje, pad zsunie się po powierzchni stołu aż do drugiego, niższego końca. Możesz teraz poeksperymentować z kątem nachylenia stołu. Z moich testów wynika, że już po kilku próbach można tak dobrać ten kąt, aby podróż pada przez cały stół przy pełnej mocy silników wibrujących zajmowała około 30 sekund. Jeśli ułożysz cztery pady w linii blisko najwyższej krawędzi stołu, gracze będą mogli wybrać swoich „zawodników", po czym przystąpić do wyścigu w dół stołu.

Kod tej gry jest wyjątkowo prosty; działanie metody `Update` sprowadza się do włączenia wszystkich silników wibrujących w padach:

```
protected override void Update(GameTime gameTime)
{
    // Umożliwia wyjście z gry.
    if(GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    {
        GamePad.SetVibration(PlayerIndex.One, 0, 0);
        GamePad.SetVibration(PlayerIndex.Two, 0, 0);
        GamePad.SetVibration(PlayerIndex.Three, 0, 0);
        GamePad.SetVibration(PlayerIndex.Four, 0, 0);
        this.Exit();
    }

    GamePad.SetVibration(PlayerIndex.One, 1, 1);
    GamePad.SetVibration(PlayerIndex.Two, 1, 1);
    GamePad.SetVibration(PlayerIndex.Three, 1, 1);
    GamePad.SetVibration(PlayerIndex.Four, 1, 1);

    base.Update(gameTime);
}
```

Jedyną komplikacją jest konieczność wyłączenia wszystkich silników wibrujących w momencie zakończenia gry. Umieść teraz wszystkie pady na szczycie „toru wyścigowego" i uruchom ten program. Aby zatrzymać grę, naciśnij przycisk *Back* na pierwszym padzie.

Przykładowy kod: Gamepad Racer Przykładowy projekt w katalogu *05 GamepadRacer* w ramach zasobów z kodem źródłowym dla tego rozdziału zawiera wersję programu wprawiającego w ruch ścigające się pady.

Uwaga Uważnie zmieniając wartości decydujące o szybkości działania silników wibrujących, możesz „sabotować" wybrane pady, tak aby zawsze wygrywali ci sami zawodnicy. Pamiętaj jednak, że nie pochwalam podobnych zachowań.

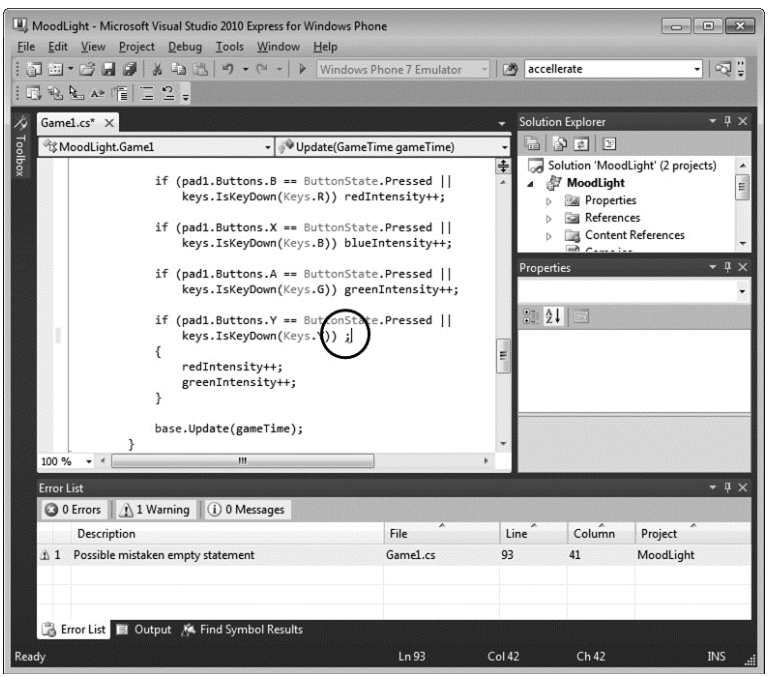
Błędy w programie

Twój młodszy brat wciąż próbuje nauczyć się programować, ale nadal napotyka problemy. Twierdzi, że ta książka zawiera błędy, ponieważ proponowane tutaj programy po zapisaniu na komputerze nie działają prawidłowo. Próbuje zmusić do działania grę *Color Nerve*, ale za każdym razem, gdy uruchamia ten program, intensywność żółtej barwy jest zwiększana, i to niezależnie od tego, czy naciska na padzie odpowiedni przycisk. Analizujesz jego program i znajdujesz następujący fragment kodu w metodzie `Update`:

```
if (pad1.Buttons.Y == ButtonState.Pressed ||
    keys.IsKeyDown(Keys.Y)) ;
{
    redIntensity++;
    greenIntensity++;
}
```

To jedyna część programu, w której jest zwiększana intensywność żółtej barwy, zatem wydaje się, że użyty warunek jest ignorowany.

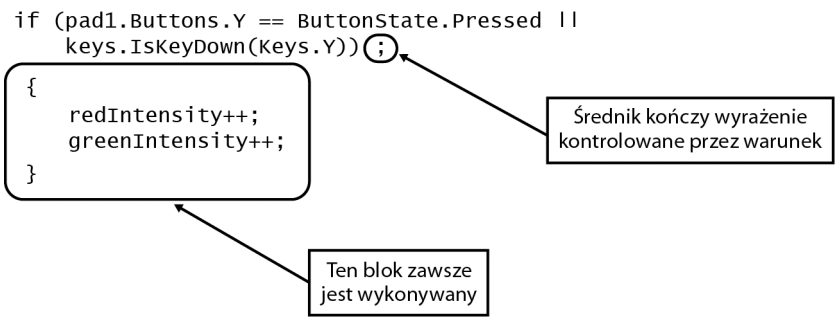
Kod sprawia wrażenie całkowicie prawidłowego, jest też poprawnie kompilowany i wykonywany, a mimo to intensywność żółtej barwy zdaje się nieustannie rosnąć. Warto w tym momencie przyjrzeć się komunikatom wyświetlanym w środowisku Microsoft Visual Studio i sprawdzić, czy kompilator nie próbuje Ci czegoś zasygnalizować w związku z tym kodem. Na rysunku 3.6 pokazano kod Twojego brata po kompilacji.



RYSUNEK 3.6. Komunikat ostrzeżenia w środowisku Visual Studio

Twoją uwagę powinien zwrócić lewy dolny narożnik ekranu z komunikatem *Possible mistaken empty statement* (możliwy błąd pustego wyrażenia). Jeśli dwukrotnie klikniesz ten komunikat, kursor zostanie przeniesiony w miejsce bezpośrednio za wyrażeniem `if` (na rysunku 3.6 wyróżniłem to miejsce okręgiem).

Kompilator języka C# próbuje przekazać nam jakąś informację na temat tego wyrażenia. Jeśli wrócimy do oryginalnego listingu, odkryjemy, że Twój brat dopisał średnik bezpośrednio za warunkiem. Problem w tym, że średnik kończy wyrażenie kontrolowane przez ten warunek. Oznacza to, że jeśli zostanie wciśnięty czerwony przycisk na padzie, klawisz *R* na klawiaturze lub krzyżak, program nie wykona żadnej operacji (wykona wyrażenie puste), po czym wykona kolejne wyrażenia niezależnie od stanu pada czy klawiatury, doprowadzając do efektu, na który skarży się Twój młodszy brat. Działanie tego programu opisano na rysunku 3.7.



RYSUNEK 3.7. Skutek dopisania średnika

Usunięcie tego średnika spowoduje, że kompilator nie będzie już wyświetlał swojego ostrzeżenia, a program zacznie działać prawidłowo. Twój brat zaczyna powoli zmieniać swoją opinię na Twój temat; zaproponował nawet wyniesienie śmieci, mimo że dzisiaj przypada Twoja kolejka.

Wielka programistka mówi: pomaganie innym ludziom jest dobrym rozwiązaniem Wielka programistka obserwowała Twoje postępowanie z rosnącą aprobatą. Powiedziała, że zawsze warto podejmować próby pomagania innym programistom, którzy napotykają na rozmaite problemy podczas pisania swoich programów. Zdarza się, że samo wyjaśnienie postronnemu obserwatorowi przez programistę negatywnych skutków ukrytego błędu w jakimś fragmencie kodu ułatwia temu pierwszemu szybką identyfikację źródła usterki. Oznacza to, że możesz dość łatwo, już przez samą obecność, zyskać sławę nieustraszonego pogromcy błędów. Co więcej, obserwowanie błędów popełnianych przez innych może być dla Ciebie cennym źródłem wskazówek, na co zwracać uwagę, gdy w Twoich programach coś nie działa tak, jak należy. A… i jeszcze może ominąć Cię kolejka wynoszenia śmieci.

Podsumowanie

W tym rozdziale nauczyłeś się całkiem sporo i wreszcie udało Ci się stworzyć gry, przy których gracze mogą miło spędzać czas. Dowiedziałeś się, jak framework XNA umożliwia programom interakcję z fizycznymi urządzeniami (za pośrednictwem metod wywoływanych dla klas), i opanowałeś sztukę podejmowania decyzji w programie zależnie od informacji uzyskiwanych z tych urządzeń. Swoją wiedzę wykorzystałeś do stworzenia kilku prostych (i niezbyt mądrych) gier

Przegląd rozdziału w pytaniach

Żaden rozdział nie byłby kompletny bez przeglądu. Oto przegląd dla tego rozdziału. Zapewne pamiętasz już zasady — zdecyduj tylko, czy poniższe zdania są prawdziwe, czy fałszywe. W dodatku A na końcu tej książki znajdziesz odpowiedzi, które pozwolą Ci rozstrzygnąć, czy po lekturze tego rozdziału jesteś zwycięzcą, czy przegranym.

1. Jeśli klasa jest biurem, metoda jest biurkiem w tym biurze.
2. Kompilator tworzy wszystkie egzemplarze klas w programie.
3. Wyrażenie `if` musi zawierać część `else`.
4. Parametr służy do przekazywania informacji do klasy.
5. Część `else` wyrażenia `if` zawsze jest wykonywana.
6. W programie na bazie frameworku XNA informacje o stanie pada są reprezentowane przez wartość typu `byte`.
7. Metoda `GamePad.GetState` może być używana do sprawdzania, czy naciśnięto jakiś przycisk na padzie (to trudne pytanie; przed udzieleniem odpowiedzi możesz zajrzeć do rozdziału).
8. Blok składa się z pewnej liczby wyrażeń języka C# otoczonych nawiasami klamrowymi.
9. W języku C# warunek `(true || false)` oznacza „prawda lub fałsz” i zawsze ma wartość `true`.
10. W języku C# warunek `(redIntensity > 220)` ma wartość `true`, jeśli wartość zmiennej `greenIntensity` jest większa niż `220`.
11. Wibracje pada zawsze są automatycznie wyłączane w momencie zatrzymania wykonywania gry przez system XNA.

Część II

Obrazy, dźwięk i tekst

W tej części:

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

Każdy pasjonat gier komputerowych zadaje sobie czasem pytania typu:

„Niesamowite, jak oni to zrobili?" albo „Jak można uzyskać taki efekt?". A co się dzieje, jeśli to pytanie przychodzi do głowy programiście? Jego następną myślą zwykle jest: „Muszę spróbować sam to zrobić!". Na co czekasz? Dziś już nic nie stoi na przeszkodzie! Odkąd Microsoft udostępnił framework XNA, każdy może wymyślić i stworzyć swoją własną profesjonalną grę komputerową, uruchomić ją na prawdziwej konsoli, a nawet wprowadzić na rynek dzięki usłudze Xbox Live® lub Windows® Phone Marketplace.

Jeśli zawsze marzyłeś o profesjonalnym programowaniu gier, ale nie do końca wiedziałeś, jak się za to zabrać, oto książka, która wszystko zmieni! Ten podręcznik będzie dla Ciebie doskonałym przewodnikiem, prowadzącym od opanowania podstaw frameworku XNA, przez zasady programowania w użytecznym języku C#, aż do tworzenia własnych gier. Przejdziesz niezwykłą drogę: na jej początku czeka Cię instalacja i przygotowanie środowiska programistycznego, potem projektowanie realistycznej grafiki gry i jej ciekawe udźwiękowienie, a uwieńczeniem Twoich działań będzie stworzenie profesjonalnych projektów dla konsoli Xbox 360®, telefonów z systemem Windows® Phone 7 czy komputerów PC. Co ważne, nie musisz być zaawansowanym i doświadczonym programistą. Wszystko, czego będziesz potrzebował do realizacji tej misji, znajdziesz w tej książce!

Nauczysz się m.in.:

- pisać kod tworzący zachowania w grze i sterujący nimi
- projektować wygląd gry — grafikę, czcionki, kolory, oświetlenie i efekty trójwymiarowe
- budować obiekty gry, tzw. sprajty
- nagrywać i odtwarzać dźwięki
- przetwarzać dane wejściowe uzyskiwane z padów i klawiatur
- korzystać z ekranu dotykowego i akcelerometru w systemie Windows® Phone
- dodawać stopery, systemy punktacji, elementy zagrażające graczowi i postęp w rozgrywce
- tworzyć gry wieloosobowe i sieciowe

Opanuj sztukę programowania gier i zmieniaj własne pomysły w ekscytującą rzeczywistość!

Rob Miles uczy programowania od ponad dwudziestu pięciu lat. Jest ekspertem w dziedzinie programowania w języku Visual C#, przy użyciu frameworku XNA oraz Microsoft MVP for Windows Phone Development. Oprócz pisania własnych gier i aplikacji oraz pracy dydaktycznej na uniwersytecie Rob brał udział w wielu różnych komercyjnych projektach informatycznych.

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

Informatyka w najlepszym wydaniu

ISBN: 978-83-246-3410-1
9 788324 634101
Cena: 89,00 zł