Erica Sadun

ADDISON-WESLEY

iOS 5

Podręcznik programisty

Najlepsze przepisy dla każdego dewelopera!

Helion

Tytuł oryginału: The iOS 5 Developer's Cookbook:
Core Concepts and Essential Recipes for iOS Programmers (3rd Edition)

Tłumaczenie: Robert Górczyński (wstęp, rozdz. 2 – 15), Łukasz Suma (rozdz. 1)

ISBN: 978-83-246-5121-4

Authorized translation from the English edition, entitled: THE IOS 5 DEVELOPER'S COOKBOOK: CORE CONCEPTS AND ESSENTIAL RECIPES FOR IOS PROGRAMMERS, Third Edition; ISBN 0321832078; by Erica Sadun; published by Pearson Education, Inc, publishing as Addison Wesley.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem: ftp://ftp.helion.pl/przyklady/ios5pp.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/ios5pp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

14

Możliwości urządzenia

Każde urządzenie iPhone to mieszanka unikatowych, współdzielonych, tymczasowych i trwałych cech. Wspomniane cechy to między innymi aktualna fizyczna orientacja urządzenia, nazwa modelu, stan baterii oraz dostęp do komponentów sprzętowych. W tym rozdziale skoncentrujemy się na urządzeniu, począwszy od konfiguracji kompilacji, aż po aktywne czujniki sprzętowe. Przedstawię przykłady dostarczające różnych informacji na temat urządzenia, w którym została uruchomiona aplikacja. Dowiesz się, jak w trakcie działania aplikacji przetestować sprzęt pod kątem wymaganych cech, a następnie opisać te cechy w pliku *Info.plist*. Poznasz sposoby pobierania informacji z czujników i nasłuchiwania powiadomień, dzięki którym można wywoływać metody po wykryciu zmian w stanie czujników. W rozdziale koncentruję się na sprzęcie, systemie plików i czujnikach dostępnych w iPhone'ie. Ponadto pomogę Ci w programowym wykorzystaniu możliwości oferowanych przez iPhone'a.

Uzyskanie dostępu do informacji podstawowych

Klasa `UIDevice` udostępnia klucze właściwości charakterystycznych dla urządzeń (np. dla iPhone'a i iPoda touch) pozwalające na sprawdzenie między innymi nazwy urządzenia, nazwy i wersji używanego systemu operacyjnego itd. Wymieniona klasa to proste rozwiązanie, które pozwala na szybkie pobranie pewnych informacji dotyczących systemu. Każda metoda jest metodą egzemplarza wywoływaną jako Singleton, np. `[UIDevice currentDevice]`.

Poniżej wymieniono kilka właściwości klasy `UIDevice` pozwalających na pobranie pewnych informacji o urządzeniu.

- **`systemName`** — przechowuje nazwę aktualnie używanego systemu operacyjnego. W przypadku bieżącej generacji urządzeń iOS istnieje tylko jedna wersja systemu działającego na platformie: iPhone OS. Firma Apple nie uaktualniła jeszcze nazwy systemu, aby ogólnie odnosiła się do urządzeń iOS.
- **`systemVersion`** — podaje wersję oprogramowania firmware aktualnie zainstalowanego w urządzeniu, np. 4.2, 4.3, 5.0 itd.
- **`model`** — zawiera ciąg tekstowy opisujący platformę, np. iPhone, iPad lub iPod touch. Jeżeli rodzina urządzeń iOS zostanie powiększona, pojawią się dodatkowe ciągi tekstowe opisujące nowe urządzenia. Właściwość `localizedModel` to zlokalizowana wersja wymienionej właściwości.

- **`userInterfaceIdiom`** — określa styl interfejsu używanego w bieżącym urządzeniu, czyli iPhone (dla urządzeń iPhone i iPod touch) lub iPad. Mogą pojawić się jeszcze inne style, jeżeli firma Apple wprowadzi kolejne urządzenia iOS.
- **`name`** — zawiera ciąg tekstowy opisujący nazwę przypisaną urządzeniu w aplikacji iTunes przez użytkownika, np. *iPhone Janka* lub *Mój telefon*. Ta nazwa jest wykorzystywana także do utworzenia nazwy lokalnej dla urządzenia.

Poniżej przedstawiono kilka przykładów użycia wymienionych właściwości:

```
UIDevice *device = [UIDevice currentDevice];
NSLog(@"Nazwa systemu: %@", device.systemName);
NSLog(@"Model: %@", device.model);
NSLog(@"Nazwa urządzenia: %@", device.name);
```

W przypadku bieżących wydań systemu iOS masz możliwość sprawdzenia typu urządzenia dzięki prostemu testowi zwracającego wartość boolowską:

```
#define IS_IPAD (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
```

Można przyjąć założenie, że powyższy test zwróci wartość `NO`, gdy aktualnym urządzeniem nie będzie iPad (czyli w przypadku urządzeń iPhone i iPod touch). Jeśli firma Apple wypuści na rynek nową rodzinę urządzeń, wtedy będziesz musiał odpowiednio uaktualnić kod. Oto przykład rozwiązania:

```
if (IS_IPAD)
{
   // Kod przeznaczony dla iPada.
}
else if (IS_IPHONE)
{
   // Kod przeznaczony dla urządzeń iPhone i iPod touch.
}
else if (IS_OTHER_DEVICE)
{
   // Kod przeznaczony dla innych urządzeń iOS.
}
```

Dodanie ograniczeń związanych z możliwościami urządzeń

Plik *Info.plist* pozwala na określenie wymagań aplikacji podczas jej umieszczania w iTunes. Zdefiniowane ograniczenia informują iTunes, jakie funkcje są wymagane przez daną aplikację.

Każde urządzenie iOS jest wyposażone w pewien zestaw funkcji. Niektóre posiadają wbudowane aparaty i GPS, natomiast inne są ich pozbawione. Jeszcze inne mają żyroskop, autofocus itd. Masz więc możliwość zdefiniowania funkcji wymaganych do działania Twojej aplikacji.

Umieszczając klucz `UIRequiredDeviceCapabilities` w pliku *Info.plist*, powodujesz, że iTunes ogranicza możliwość instalacji Twojej aplikacji jedynie do urządzeń wyposażonych w zdefiniowane funkcje. Listę funkcji podajesz jako tablicę ciągów tekstowych lub w postaci słownika.

Tablica zawiera każdą wymaganą funkcję, a wszystkie elementy tablicy (funkcje) muszą być dostępne w urządzeniu. Z kolei słownik pozwala na wyraźne zdefiniowanie funkcji jako wymaganej lub niedozwolonej w danym urządzeniu. Klucze słownika oznaczają możliwości, natomiast wartości określają, czy funkcja musi być obecna (boolowska wartość `YES`), czy nie (`NO`).

Aktualnie dostępne klucze słownika zostały wymienione w tabeli 14.1. Używaj tylko tych funkcji, które są absolutnie wymagane przez aplikację lub nie mogą być w żaden sposób obsłużone. Jeżeli w aplikacji możesz umieścić pewne rozwiązanie pozwalające na obejście braku pewnej funkcji w urządzeniu, to nie definiuj tej funkcji za pomocą klucza słownika. Wszystkie funkcje wymienione w tabeli 14.1 zostały przedstawione w pozytywnym kontekście. Kiedy stosujesz wykluczenia zamiast wymogu dostępności danej funkcji, wtedy zastosuj jej negatywne znaczenie. Przykładowo zdefiniuj, że urządzenie nie może posiadać autofocusa lub żyroskopu bądź że dostęp do serwisu Game Center jest nieobsługiwany.

Tabela 14.1. *Możliwości oferowane przez urządzenie*

Klucz	Sposób użycia
`telephony`	Aplikacja wymaga programu Telefon lub używa schematu URL `tel://`.
`wifi`	Aplikacja wymaga dostępu do sieci lokalnej 802.11.
`sms`	Aplikacja wymaga programu Wiadomości lub używa schematu URL `sms://`.
`still-camera`	Aplikacja wymaga obecności aparatu w urządzeniu i może używać interfejsu kontrolki `UIImagePicker` do wykonywania zdjęć tym aparatem.
`auto-focus-camera`	Aplikacja wymaga dodatkowego osprzętu (autofocus) w celu wykonywania zdjęć makro lub wyjątkowo ostrych zdjęć na potrzeby wykrywania danych na zdjęciach.
`opengles-1`	Aplikacja wymaga obsługi OpenGL ES 1.1.
`opengles-2`	Aplikacja wymaga obsługi OpenGL ES 2.0.
`camera-flash`	Aplikacja wymaga dostępności flesza w aparacie.
`video-camera`	Aplikacja wymaga aparatu, który ma możliwość nagrywania filmów.
`accelerometer`	Aplikacja wymaga obecności przyśpieszeniomierza, który pozwala na znacznie więcej niż prostą zmianę orientacji `UIViewController`.
`gyroscope`	Aplikacja wymaga obecności żyroskopu w urządzeniu.
`location-services`	Aplikacja wymaga użycia technologii Core Location.
`gps`	Aplikacja używa Core Location i wymaga dodatkowej dokładności podczas określania położenia urządzenia za pomocą GPS.
`magnetometer`	Aplikacja używa Core Location i wymaga zdarzeń związanych z poruszaniem się — to znaczy kierunku podróży. (Ten czujnik jest wbudowany w kompasie).
`gamekit`	Aplikacja wymaga dostępu do serwisu Game Center (system iOS 4.1 i nowsze).
`microphone`	Aplikacja używa mikrofonu wbudowanego lub (obsługiwanego) osprzętu oferującego mikrofon.
`armv6`	Aplikacja jest skompilowana *jedynie* dla zbioru instrukcji armv6 (iOS 3.1 i nowsze).
`armv7`	Aplikacja jest skompilowana *jedynie* dla zbioru instrukcji armv7 (iOS 3.1 i nowsze).
`peer-peer`	Aplikacja używa połączeń Bluetooth typu jeden do jednego w serwisie Game Center (iOS 3.1 i nowsze).

Przeanalizujmy przykład aplikacji oferującej możliwość wykonywania zdjęć, jeśli urządzenie jest wyposażone w aparat. Jeżeli taka aplikacja działa w urządzeniach niewyposażonych w aparat, np. wcześniejszych wersjach iPoda touch, nie umieszczaj w słowniku wymagania dostępności aparatu. Zamiast tego dostępność aparatu sprawdź z poziomu kodu w aplikacji i wyświetlaj interfejs aparatu, gdy taki znajduje się w urządzeniu. Dodanie klucza w słowniku spowoduje wyeliminowanie potencjalnych klientów, którzy posiadają urządzenia bez aparatu — wczesne wersje urządzenia iPod touch (od pierwszej do trzeciej generacji) oraz iPada pierwszej generacji.

Sposób: pobieranie informacji dodatkowych o urządzeniu

Funkcje `sysctl()` i `sysctlbyname()` pozwalają na pobieranie informacji dotyczących systemu. To są standardowe funkcje platformy UNIX pozwalające na wykonywanie zapytań do systemu operacyjnego w celu pobrania informacji szczegółowych dotyczących sprzętu i systemu. Jeśli zajrzysz do pliku */usr/include/sys/sysctl.h* w komputerze Mac, to poznasz ogólny zasięg działania wymienionych funkcji. We wspomnianym pliku znajdziesz obszerną listę stałych, które można wykorzystać w charakterze parametrów wymienionych wcześniej funkcji.

Wspomniane stałe pozwalają na pobranie pewnych informacji, np. częstotliwości taktowania procesora, ilości dostępnej pamięci itd. W sposobie 14.1 zademonstrowałam pobranie tego typu informacji. Kod zawiera kategorię `UIDevice` odpowiedzialną za zbieranie informacji o urządzeniu, a następnie ich przekazywanie poprzez serię wywołań metod.

Sposób 14.1. *Rozszerzone zbieranie informacji o urządzeniu*

```
@implementation UIDevice (Hardware)
+ (NSString *) getSysInfoByName:(char *)typeSpecifier
{
   // Pobranie informacji dostarczanych przez funkcję sysctl().
   size_t size;
   sysctlbyname(typeSpecifier, NULL, &size, NULL, 0);

   char *answer = malloc(size);
   sysctlbyname(typeSpecifier, answer, &size, NULL, 0);

   NSString *results = [NSString stringWithCString:answer
      encoding: NSUTF8StringEncoding];
   free(answer);

   return results;
}

- (NSString *) platform
{
   return [UIDevice getSysInfoByName:"hw.machine"];
}

- (NSUInteger) getSysInfo: (uint) typeSpecifier
{
   size_t size = sizeof(int);
   int results;
   int mib[2] = {CTL_HW, typeSpecifier};
   sysctl(mib, 2, &results, &size, NULL, 0);
   return (NSUInteger) results;
}

- (NSUInteger) cpuFrequency
{
   return [UIDevice getSysInfo:HW_CPU_FREQ];
}

- (NSUInteger) busFrequency
{
   return [UIDevice getSysInfo:HW_BUS_FREQ];
}
```

```
- (NSUInteger) totalMemory
{
   return [UIDevice getSysInfo:HW_PHYSMEM];
}

- (NSUInteger) userMemory
{
   return [UIDevice getSysInfo:HW_USERMEM];
}

- (NSUInteger) maxSocketBufferSize
{
   return [UIDevice getSysInfo:KIPC_MAXSOCKBUF];
}
@end
```

Pobierz kod przedstawionego sposobu

Aby pobrać kod przedstawionego tutaj sposobu, przejdź na stronę *ftp://ftp.helion.pl/przyklady/ios5pp.zip*, lub jeśli pobrałeś obraz dysku zawierający wszystkie przykłady przedstawione w książce, to przejdź do katalogu przeznaczonego dla rozdziału 14. i otwórz projekt dla tego sposobu.

Być może zastanawiasz się, dlaczego przedstawiona kategoria zawiera metodę `platform`, skoro standardowa klasa `UIDevice` na żądanie zwraca nazwę modelu urządzenia. Odpowiedź kryje się w możliwości rozróżniania różnych typów urządzeń.

Standardowo iPhone 3GS jest określany po prostu jako `iPhone`, podobnie jak iPhone 4. Z kolei przedstawiona kategoria podaje `iPhone2,1` dla modelu 3GS i `iPhone3,1` dla modelu iPhone 4. W ten sposób masz możliwość programowego odróżnienia pierwszej generacji iPhone'a (`iPhone1,1`) od iPhone'a 3G (`iPhone1,2`).

Każdy z wymienionych modeli oferuje różne wbudowane funkcje. Poznanie numeru modelu urządzenia, w którym została uruchomiona aplikacja, pozwala na określenie, czy dane urządzenie będzie potrafiło obsłużyć konkretną funkcję, np. GPS lub kompas.

Monitorowanie poziomu naładowania baterii w iPhone'ie

Masz możliwość programowego śledzenia poziomu naładowania baterii w iPhone'ie i jej aktualnego stanu. Poziom naładowania baterii jest przedstawiany za pomocą wartości zmiennoprzecinkowej z zakresu od `0.0` (w pełni rozładowana) do `1.0` (w pełni naładowana). Wartość ta przedstawia przybliżony poziom naładowania baterii, możesz ją wykorzystać przed rozpoczęciem operacji, której przeprowadzenie wymaga wyjątkowo dużej ilości energii.

Na przykład przed przeprowadzeniem serii długotrwałych obliczeń matematycznych możesz o tym ostrzec użytkownika i zaproponować podłączenie urządzenia do gniazdka sieciowego. Poziom naładowania baterii sprawdzasz poprzez wywołanie klasy `UIDevice`. Wartość zwrotna jest podawana w odstępach co 5%.

```
NSLog(@"Poziom naładowania baterii: %0.2f%",
   [[UIDevice currentDevice] batteryLevel] * 100);
```

Stan baterii ma cztery możliwe wartości. Urządzenie może być ładowane (to znaczy podłączone do źródła prądu), w pełni naładowane, odłączone od źródła prądu lub w stanie nieznanym. Stan baterii można odczytać dzięki właściwości `batteryState` klasy `UIDevice`:

```
NSArray *stateArray = [NSArray arrayWithObjects:
   @"Stan baterii jest nieznany",
   @"Urządzenie nie jest podłączone do źródła zasilania",
   @"Bateria jest ładowana",
   @"Bateria jest w pełni naładowana", nil];

UIDevice *device = [UIDevice currentDevice];
NSLog(@"tan baterii: %@", [stateArray objectAtIndex:device.batteryState]);
```

Wymienione stany nie mogą być traktowane jako trwałe, a jedynie jako chwilowe, opisujące to, co aktualnie dzieje się z urządzeniem. To nie są opcje i nie są ze sobą łączone w celu przygotowania ogólnych informacji o baterii. Wymienione wartości odzwierciedlają ostatnią zmianę, która dotyczy stanu baterii.

Bardzo łatwo możesz monitorować zmiany stanu baterii poprzez udzielanie odpowiedzi na powiadomienia informujące o zmianie stanu baterii. W ten sposób wychwytujesz chwilowe zdarzenia, takie jak naładowanie baterii, podłączenie urządzenia do źródła zasilania w celu naładowania baterii oraz odłączenie urządzenia od źródła zasilania.

Aby rozpocząć monitorowanie właściwości `batteryMonitoringEnabled`, przypisz wartość `YES`. Podczas monitorowania klasa `UIDevice` generuje powiadomienia po każdym wystąpieniu zmiany w stanie baterii lub poziomie jej naładowania. Przedstawiona poniżej metoda nasłuchuje obu powiadomień. Zwróć uwagę, że wartości możesz sprawdzać także bezpośrednio bez konieczności oczekiwania na powiadomienie. Firma Apple nie daje żadnej gwarancji dotyczącej częstotliwości generowania uaktualnień informujących o zmianie poziomu naładowania. Jednak na podstawie przedstawionego fragmentu kodu przekonasz się, że tego rodzaju powiadomienia są generowane całkiem regularnie.

```
- (void) viewDidLoad
{
   // Włączenie monitorowania baterii.
   [[UIDevice currentDevice] setBatteryMonitoringEnabled:YES];

   // Dodanie obserwatorów stanu baterii i poziomu jej naładowania.
   [[NSNotificationCenter defaultCenter] addObserver:self
      selector:@selector(checkBattery)
      name:UIDeviceBatteryStateDidChangeNotification
      object:nil];
   [[NSNotificationCenter defaultCenter] addObserver:self
      selector:@selector(checkBattery)
      name:UIDeviceBatteryLevelDidChangeNotification
      object:nil];
}
```

Włączanie i wyłączanie czujnika zbliżeniowego

Obecnie czujnik zbliżeniowy jest komponentem dostępnym jedynie w iPhone'ie, urządzenia iPod touch i iPad nie oferują tego czujnika. O ile nie masz ważnego powodu trzymania smartfona iPhone blisko ciała (lub na odwrót), włączenie czujnika zbliżeniowego niewiele daje.

Po włączeniu czujnika zbliżeniowego ma on jedno podstawowe zadanie — wykrycie dużego obiektu znajdującego się przed urządzeniem. Po wykryciu obiektu następuje wyłączenie ekranu i wygenerowanie powiadomienia. Odsunięcie obiektu od urządzenia powoduje włączenie ekranu z powrotem. W ten sposób unikasz przypadkowego naciśnięcia przycisków lub wybrania numeru uchem, kiedy prowadzisz rozmowę telefoniczną. Niektóre kiepsko zaprojektowane ochrony na urządzenie uniemożliwiają poprawne działanie czujnika zbliżeniowego.

Aplikacja Google Mobile dostępna w sklepie iTunes App Store wykorzystuje czujnik zbliżeniowy do rozpoczęcia sesji nagrywania. Kiedy umieścisz telefon blisko głowy, aplikacja rozpocznie nagrywanie Twojego polecenia, a następnie zinterpretuje je po odsunięciu urządzenia od głowy. Programiści aplikacji nie przejęli się wyłączeniem ekranu, ponieważ do poprawnego nagrywania informacji nie jest wymagany graficzny interfejs użytkownika.

Przedstawiona poniżej metoda demonstruje sposób pracy z czujnikiem zbliżeniowym dostępnym w iPhone'ie. Kod wykorzystuje klasę `UIDevice` w celu włączenia czujnika zbliżeniowego i nasłuchuje powiadomień `UIDeviceProximityStateDidChangeNotification` w celu przechwycenia informacji o zmianie stanu czujnika. Dwa dostępne stany to czujnik włączony lub wyłączony. Kiedy właściwość `proximityState` klasy `UIDevice` zwraca wartość `YES`, oznacza to aktywację czujnika zbliżeniowego.

```
- (void) toggle: (id) sender
{
   UIDevice *device = [UIDevice currentDevice];

   // Określenie, czy czujnik zbliżeniowy jest aktualnie monitorowany, i jego ewentualne włączenie.
   BOOL isEnabled = device.proximityMonitoringEnabled;
   device.proximityMonitoringEnabled = ! isEnabled;
}

- (void) stateChange: (NSNotificationCenter *) notification
{
   // Zapis informacji powiadomienia.
   NSLog(@"Czujnik zbliżeniowy %@",
   [UIDevice currentDevice].proximityState ?
      @"spowoduje wyłączenie ekranu" :
      @"spowoduje z powrotem włączenie ekranu");
}

- (void) viewDidLoad
{
   // Dodanie obserwatora czujnika zbliżeniowego.
   [[NSNotificationCenter defaultCenter]
      addObserver:self
      selector:@selector(stateChange)
      name:@"UIDeviceProximityStateDidChangeNotification"
      object:nil];
}
```

Sposób: użycie przyśpieszeniomierza w celu ustalenia „góry” urządzenia

iPhone zawiera trzy czujniki pozwalające na pomiar przyśpieszenia urządzenia względem trzech osi: lewo/prawo (`x`), góra/dół (`y`) i do przodu/do tyłu (`z`). Wymienione wartości wskazują na siły działające na iPhone, zarówno grawitację, jak i ruch urządzeniem przez użytkownika. Możesz otrzymać ciekawe wartości opisujące wspomniane siły podczas obracania iPhone'em wokół głowy (siła dośrodkowa) lub rzucania iPhone'a z wysokiego budynku (spadanie swobodne). Niestety, nie będziesz w stanie odczytać tych wartości, gdy z iPhone'a pozostanie jedynie sterta złomu.

Aby obiekt otrzymywał informacje z przyśpieszeniomierza, należy zdefiniować go jako delegata. Obiekt ustawiony jako delegat musi implementować protokół `UIAccelerometerDelegate`:

```
[[UIAccelerometer sharedAccelerometer] setDelegate:self];
```

Po przypisaniu w obiekcie delegata wywoływana jest metoda `accelerometer:didAccelerate:`, w której możesz udzielić odpowiedzi na dane otrzymane z przyśpieszeniomierza. Struktura `UIAcceleration` przekazuje metodzie delegata trzy wartości zmiennoprzecinkowe dla osi x, y i z. Każda wartość mieści się w zakresie od `-1.0` do `1.0`.

```
float x = acceleration.x;
float y = acceleration.y;
float z = acceleration.z;
```

W sposobie 14.2 wspomniane wartości pomagają w określeniu „góry" urządzenia. Kod oblicza arcus tangens pomiędzy wektorami przyśpieszenia X i Y i zwraca kąt. Po otrzymaniu nowych danych z przyśpieszeniomierza kod obraca egzemplarz `UIImageView` zawierający obraz strzałki (zobacz rysunek 14.1) w taki sposób, aby wskazać odpowiedni kierunek. Udzielana użytkownikowi odpowiedź w czasie rzeczywistym gwarantuje, że strzałka zawsze będzie wskazywała górę, niezależnie od sposobu umieszczenia urządzenia przez użytkownika.

Rysunek 14.1. *Odrobina matematyki pozwala na wskazanie kierunku „góra" dzięki użyciu funkcji arcus tangens względem sił działających na osie x i y. W omawianym przykładzie strzałka zawsze wskazuje górę niezależnie od sposobu, w jaki użytkownik umieści urządzenie*

Sposób 14.2. *Przechwytywanie zdarzeń przyśpieszeniomierza*

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
   didAccelerate:(UIAcceleration *)acceleration
{
   // Określenie kierunku góra na podstawie wartości przyśpieszeniomierza dla osi x i y.
   float xx = -acceleration.x;
   float yy = acceleration.y;
   float angle = atan2(yy, xx);
   [arrow setTransform:CGAffineTransformMakeRotation(angle)];
}
```

```
- (void) viewDidLoad
{
   // Inicjalizacja delegata w celu rozpoczęcia przechwytywania zdarzeń przyśpieszeniomierza.
   [UIAccelerometer sharedAccelerometer].delegate = self;
}
```

Pobierz kod przedstawionego sposobu

Aby pobrać kod przedstawionego tutaj sposobu, przejdź na stronę *ftp://ftp.helion.pl/przyklady/ios5pp.zip*, lub jeśli pobrałeś obraz dysku zawierający wszystkie przykłady przedstawione w książce, to przejdź do katalogu przeznaczonego dla rozdziału 14. i otwórz projekt dla tego sposobu.

Synchroniczne pobieranie bieżącego kąta przyśpieszeniomierza

Czasami możesz chcieć pobrać dane z przyśpieszeniomierza bez konieczności definiowania delegata. Przedstawione poniżej metody przeznaczone do używania w ramach kategorii `UIDevice` pozwalają na synchroniczne pobieranie bieżącego kąta urządzenia względem płaszczyzn x i y — płaszczyzny przodu urządzenia iOS. Rozwiązanie polega na przygotowaniu nowej pętli działania, zaczekaniu na zdarzenie przyśpieszeniomierza, pobraniu kąta z wywołania zwrotnego, a następnie opuszczeniu pętli działania i zwróceniu pobranego kąta.

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
   didAccelerate:(UIAcceleration *)acceleration
{
   float xx = acceleration.x;
   float yy = -acceleration.y;
   device_angle = M_PI / 2.0f - atan2(yy, xx);

   if (device_angle > M_PI)
   {
      device_angle -= 2 * M_PI;
   }

   CFRunLoopStop(CFRunLoopGetCurrent());
}

- (float) orientationAngle
{
   // Zdefiniowanie bieżącego delegata.
   id priorDelegate = [UIAccelerometer sharedAccelerometer].delegate;
   [UIAccelerometer sharedAccelerometer].delegate = self;

   // Oczekiwanie na odczyt wartości.
   CFRunLoopRun();

   // Przywrócenie delegata.
   [UIAccelerometer sharedAccelerometer].delegate = priorDelegate;

   return device_angle;
}
```

To podejście nie jest odpowiednie, jeśli chodzi o ciągłe otrzymywanie wartości przyśpieszeniomierza — do tego należy bezpośrednio wykorzystać wywołania zwrotne. Jednak doskonale nadaje się do okazjonalnego sprawdzania wartości kąta. Przedstawione powyżej metody zapewniają prosty i bezpośredni dostęp do bieżącej wartości kąta, pod jakim znajduje się urządzenie.

Obliczanie kąta względnego

Obsługa zmiany orientacji ekranu oznacza, że związki w interfejsie dla danego kąta, pod jakim znajduje się urządzenie, muszą być obsługiwane we wszystkich ćwiartkach, po jednej dla każdej dostępnej orientacji ekranu. Podobnie jak wtedy, gdy `UIViewController` automatycznie obraca widok na ekranie, tak samo tutaj konieczne są pewne obliczenia matematyczne w celu uwzględnienia zmiany orientacji.

Przedstawiona poniżej metoda przeznaczona dla kategorii `UIDevice` oblicza kąt, pod jakim znajduje się urządzenie, dzięki czemu ten kąt pozostaje w zgodzie z orientacją urządzenia. W ten sposób otrzymujesz wartość przesunięcia od pionu, która odpowiada aktualnemu sposobowi wyświetlania graficznego interfejsu użytkownika.

```
- (float) orientationAngleRelativeToOrientation:(UIDeviceOrientation) someOrientation
{
   float dOrientation = 0.0f;
   switch (someOrientation)
   {
      case UIDeviceOrientationPortraitUpsideDown:
         {dOrientation = M_PI; break;}
      case UIDeviceOrientationLandscapeLeft:
         {dOrientation = -(M_PI/2.0f); break;}
      case UIDeviceOrientationLandscapeRight:
         {dOrientation = (M_PI/2.0f); break;}
      default:
         break;
   }

   float adjustedAngle = fmod(self.orientationAngle - dOrientation, 2.0f * M_PI);
   if (adjustedAngle > (M_PI + 0.01f))
   {
      adjustedAngle = (adjustedAngle - 2.0f * M_PI);
   }
   return adjustedAngle;
}
```

Powyższa metoda używa reszty z dzielenia w postaci liczby zmiennoprzecinkowej, aby obliczyć różnicę pomiędzy aktualnym kątem urządzenia i przesunięciem kątowym interfejsu. W ten sposób zostanie obliczona bardzo ważna wartość przesunięcia kątowego względem pionu.

Praca z podstawową orientacją

Klasa `UIDevice` używa wbudowanej właściwości `orientation` w celu pobrania fizycznej orientacji urządzenia. Urządzenie iOS obsługuje siedem możliwych wartości dla wymienionej właściwości:

- **`UIDeviceOrientationUnknown`** — orientacja jest aktualnie nieznana;
- **`UIDeviceOrientationPortrait`** — przycisk *Początek* znajduje się na dole;
- **`UIDeviceOrientationPortraitUpsideDown`** — przycisk *Początek* znajduje się na górze;
- **`UIDeviceOrientationLandscapeLeft`** — przycisk *Początek* znajduje się po prawej stronie;
- **`UIDeviceOrientationLandscapeRight`** — przycisk *Początek* znajduje się po lewej stronie;
- **`UIDeviceOrientationFaceUp`** — ekran znajduje się na górze;
- **`UIDeviceOrientationFaceDown`** — ekran znajduje się na dole.

Podczas typowej sesji aplikacji urządzenie może przejść przez dowolne lub wszystkie wymienione orientacje. Choć orientacja jest ustalana w połączeniu z przyśpieszeniomierzem, to w żaden sposób nie jest powiązana z wbudowaną wartością kątową.

System iOS oferuje dwa wbudowane makra pomagające w określeniu, czy orientacja urządzenia jest pionowa, czy pozioma: `UIDeviceOrientationIsPortrait()` i `UIDeviceOrientationIsLandscape()`. Uznałam za wygodne rozbudowę klasy `UIDevice` o możliwość sprawdzania orientacji na podstawie właściwości wbudowanych w urządzenie.

```
@property (nonatomic, readonly) BOOL isLandscape;
@property (nonatomic, readonly) BOOL isPortrait;

- (BOOL) isLandscape
{

 return UIDeviceOrientationIsLandscape(self.orientation);
}
- (BOOL) isPortrait
{
   return UIDeviceOrientationIsPortrait(self.orientation);
}
```

Twój kod może bezpośrednio nasłuchiwać powiadomień o zmianie orientacji urządzenia. W tym celu do wzorca Singleton, jakim jest `currentDevice`, należy wysłać wiadomość `beginGeneratingDeviceOrientation` ↪`Notifications`. Następnie trzeba dodać obserwatora w celu przechwytywania powiadomień `UIDevice` ↪`OrientationDidChangeNotification`. Jak możesz oczekiwać, zakończenie nasłuchiwania następuje po wywołaniu `endGeneratingDeviceOrientationNotification`.

> **Wskazówka**
>
> W trakcie pisania tej książki system iOS niepoprawnie zgłaszał orientację po pierwszym uruchomieniu aplikacji. Uaktualnienie orientacji następuje jedynie po zmianie położenia urządzenia lub wywołaniu metody `UIViewController`. Aplikacja uruchomiona w orientacji pionowej może nie być tak traktowana aż do chwili, gdy użytkownik obróci urządzeniem, a następnie ponownie obróci je do właściwej orientacji. Ten błąd występuje zarówno w symulatorze, jak i w rzeczywistym urządzeniu, a ponadto łatwo go odtworzyć. Rozwiązaniem może być użycie orientacji kątowej przedstawionej w jednym z przykładów zaprezentowanych w rozdziale.

Sposób: użycie przyśpieszeniomierza do poruszania obiektami po ekranie

Po zastosowaniu odrobiny kodu przyśpieszeniomierz iPhone'a można wykorzystać do poruszania obiektami na ekranie, odpowiadając w czasie rzeczywistym na poruszanie urządzeniem przez użytkownika. W sposobie 14.3 przedstawiłam animowanego motyla, którym użytkownik może poruszać po ekranie.

Sposób 14.3. *Poruszanie obiektami na ekranie na podstawie informacji z przyśpieszeniomierza*

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
   didAccelerate:(UIAcceleration *)acceleration
{
   // Wyodrębnienie komponentów przyśpieszeniomierza.
   float xx = -acceleration.x;
```

```
    float yy = acceleration.y;

    // Przechowywanie ostatniej wartości kąta.
    mostRecentAngle = atan2(yy, xx);

    // Czy nastąpiła zmiana kierunku?
    float accelDirX = SIGN(xvelocity) * -1.0f;
    float newDirX = SIGN(xx);
    float accelDirY = SIGN(yvelocity) * -1.0f;
    float newDirY = SIGN(yy);

    // Przyśpieszenie. Aby zwiększyć „lepkość", należy zmniejszyć dodawaną wartość.
    if (accelDirX == newDirX)
    {
       xaccel = (abs(xaccel) + 0.85f) * SIGN(xaccel);
    }
    if (accelDirY == newDirY)
    {
       yaccel = (abs(yaccel) + 0.85f) * SIGN(yaccel);
    }

    //  Wprowadzenie zmiany przyśpieszenia względem bieżącej prędkości.
    xvelocity = -xaccel * xx;
    yvelocity = -yaccel * yy;
}

- (void) tick
{
    // Wyzerowanie transformacji przed zmianą położenia.
    butterfly.transform = CGAffineTransformIdentity;

    // Przesunięcie motyla zgodnie z bieżącym wektorem prędkości.
    CGRect rect = CGRectOffset(butterfly.frame, xvelocity, 0.0f);
    if (CGRectContainsRect(self.view.bounds, rect))
    {
       butterfly.frame = rect;
    }

    rect = CGRectOffset(butterfly.frame, 0.0f, yvelocity);
    if (CGRectContainsRect(self.view.bounds, rect))
    {
       butterfly.frame = rect;
    }

    // Obrót motyla niezależnie od położenia.
    butterfly.transform = CGAffineTransformMakeRotation(mostRecentAngle + M_PI_2);
}

- (void) initButterfly
{
    CGSize size;

    // Wczytanie klatek animacji.
    NSMutableArray *butterflies = [NSMutableArray array];
    for (int i = 1; i <= 17; i++)
    {
       NSString *fileName = [NSString stringWithFormat:@"bf_%d.png", i];
       UIImage *image = [UIImage imageNamed:fileName];
       size = image.size;
       [butterflies addObject:image];
    }
```

```
    // Rozpoczęcie animacji.
    butterfly = [[UIImageView alloc] initWithFrame:(CGRect){.size=size}];
    [butterfly setAnimationImages:butterflies];
    butterfly.animationDuration = 0.75f;
    [butterfly startAnimating];

    // Określenie początkowej prędkości i przyśpieszenia motyla.
    xaccel = 2.0f;
    yaccel = 2.0f;
    xvelocity = 0.0f;
    yvelocity = 0.0f;

    // Dodanie motyla.
    butterfly.center = RECTCENTER(self.view.bounds);
    [self.view addSubview:butterfly];

    // Aktywacja przyśpieszeniomierza.
    [[UIAccelerometer sharedAccelerometer] setDelegate:self];

    // Uruchomienie licznika czasu.
    [NSTimer scheduledTimerWithTimeInterval: 0.03f
       target: self selector: @selector(tick)
       userInfo: nil repeats: YES];
}
```

Pobierz kod przedstawionego sposobu

Aby pobrać kod przedstawionego tutaj sposobu, przejdź na stronę *ftp://ftp.helion.pl/przyklady/ios5pp.zip*, lub jeśli pobrałeś obraz dysku zawierający wszystkie przykłady przedstawione w książce, to przejdź do katalogu przeznaczonego dla rozdziału 14. i otwórz projekt dla tego sposobu.

Sekret rozwiązania kryje się w zastosowaniu licznika czasu. Zamiast bezpośrednio odpowiadać na zmiany w przyśpieszeniomierzu jak w przypadku kodu przedstawionego w sposobie 14.2, w omawianym przykładzie wywołania zwrotne przyśpieszeniomierza dokonują pomiaru aktualnej siły. Następnie procedura stosuje te siły względem motyla, zmieniając jego ramkę. Poniżej wymieniłam kilka punktów, na które warto zwrócić uwagę.

- Kiedy kierunek siły pozostaje taki sam, motyl przyśpiesza. Wzrost prędkości jest skalowany względem stopnia siły przyśpieszenia w kierunku X lub Y.
- Metoda `tick` wywoływana przez licznik czasu powoduje przesunięcie motyla poprzez dodanie wektora prędkości do punktu początkowego motyla.
- Zakres ruchu motyla jest ograniczony i po dotarciu do krawędzi ekranu motyl przestaje się poruszać w danym kierunku. W ten sposób motyl cały czas pozostaje widoczny na ekranie. Metoda `tick` sprawdza, czy motyl dotarł do krawędzi ekranu. Na przykład jeśli motyl dotarł do krawędzi pionowej, to nadal może poruszać się poziomo.
- Motyl zmienia orientację i zawsze jest skierowany w dół. Odbywa się to poprzez zastosowanie polecenia obrotu w metodzie `tick`. Zachowaj ostrożność podczas używania transformacji względem ramki lub przesunięcia punktu środkowego. Przed zastosowaniem przesunięcia zawsze zeruj wartości, a następnie ponownie stosuj wszelkie zmiany kąta. W przeciwnym razie mogą wystąpić niepożądane zjawiska, np. powiększenie, zmniejszenie lub zdeformowanie ramki.

Dodanie nieco blasku

Wprawdzie zastosowana w kodzie sposobu 14.3 animacja motyla przyciąga oko, ale efekt można jeszcze poprawić poprzez zastosowanie w warstwie motyla emitera Core Animation. Emiter generuje cząsteczki w czasie rzeczywistym, używając do tego zestawu własnych właściwości. Przedstawiony poniżej fragment kodu powoduje utworzenie cząsteczek przypominających pewnego rodzaju chmurę kurzu poruszającą się wraz z motylem, powstałą jakby na skutek ruchu skrzydeł motyla.

```
// Dodanie emitera cząsteczek.
float multiplier = 0.25f;

CAEmitterLayer *emitter = [CAEmitterLayer layer];
emitter.emitterPosition = RECTCENTER(butterfly.bounds);
emitter.emitterMode = kCAEmitterLayerOutline;
emitter.emitterShape = kCAEmitterLayerCircle;
emitter.renderMode = kCAEmitterLayerAdditive;
emitter.emitterSize = CGSizeMake(100 * multiplier, 0);

// Utworzenie cząsteczki emitera.
CAEmitterCell* particle = [CAEmitterCell emitterCell];
particle.emissionLongitude = M_PI;
particle.birthRate = multiplier * 100.0;
particle.lifetime = multiplier;
particle.lifetimeRange = multiplier * 0.35;
particle.velocity = 180;
particle.velocityRange = 130;
particle.emissionRange = 1.1;
particle.scaleSpeed = 1.0;
particle.color = [[UIColor orangeColor] colorWithAlphaComponent:0.1f].CGColor;
particle.contents = (__bridge id)[UIImage imageNamed:@"spark.png"].CGImage;
particle.name = @"particle";

emitter.emitterCells = [NSArray arrayWithObject:particle];
[butterfly.layer addSublayer:emitter];
```

Sposób: podstawy Core Motion

Framework Core Motion centralizuje przetwarzanie danych ruchu. Wprowadzona w iOS 4 SDK technologia Core Motion zastępuje bezpośredni dostęp do przyśpieszeniomierza, który wcześniej przedstawiłam. Technologia oferuje punkt centralny monitorowania trzech kluczowych czujników ruchu znajdujących się w urządzeniu. Wspomniane czujniki to żyroskop informujący o obrocie urządzenia, kompas wskazujący kierunek oraz przyśpieszeniomierz wykrywający zmiany sił wzdłuż trzech osi. Czwarty punkt nazywany „ruchem urządzenia" łączy wszystkie trzy wymienione czujniki w pojedynczy system monitorowania.

Technologia Core Motion używa niezmodyfikowanych wartości wymienionych czujników w celu utworzenia możliwych do odczytu wartości, najczęściej w postaci wektorów siły. Wspomniane elementy składają się z wymienionych poniżej właściwości.

- **Położenie urządzenia** (**`attitude`**) — orientacja urządzenia względem pewnego punktu odniesienia. Położenie jest przedstawione za pomocą trzech kątów o wartościach wyrażanych w radianach.
- **Obrót** (**`rotationRate`**) — wartość, o jaką urządzenie jest obrócone względem każdej z trzech osi. Obrót obejmuje wyrażone w radianach wartości kąta dla osi x, y i z na sekundę.

- **Grawitacja** (**`gravity`**) — bieżący wektor przyśpieszenia urządzenia będący skutkiem normalnej grawitacji mierzonej w wartościach G względem trzech osi x, y i z. Każda jednostka przedstawia standardową siłę grawitacji powodowaną przez Ziemię (9,8 metra na sekundę).
- **Przyśpieszenie spowodowane przez użytkownika** (**`userAcceleration`**) — wektor przyśpieszenia powodowanego przez użytkownika. Podobnie jak w przypadku grawitacji przyśpieszenie jest mierzone w wartościach G względem trzech osi x, y i z. Po dodaniu wektorów grawitacji i przyśpieszenia powodowanego przez użytkownika otrzymujemy całkowite przyśpieszenie, jakie powoduje urządzenie.
- **Pole magnetyczne** (**`magneticField`**) — wektor przedstawiający ogólne wartości pola magnetycznego w sąsiedztwie urządzenia. Pole magnetyczne jest mierzone w mikroteslach wzdłuż osi x, y i z. Zapewniana jest także kalibracja dokładności w celu poinformowania aplikacji o jakości pomiaru.

Sprawdzanie dostępności czujników

Jak przeczytałeś we wcześniejszej części rozdziału, plik *Info.plist* możesz wykorzystać do wskazania wymaganych lub niedozwolonych do działania aplikacji czujników urządzenia. Masz więc możliwość sprawdzenia dostępności wszystkich czujników obsługiwanych przez technologię Core Motion.

```
if (motionManager.gyroAvailable)
{
   [motionManager startGyroUpdates];
}

if (motionManager.magnetometerAvailable)
{
   [motionManager startMagnetometerUpdates];
}

if (motionManager.accelerometerAvailable)
{
   [motionManager startAccelerometerUpdates];
}

if (motionManager.deviceMotionAvailable)
{
   [motionManager startDeviceMotionUpdates];
}
```

Rozpoczęcie generowania uaktualnień nie powoduje wywoływania metod delegata, jak miało to miejsce w przypadku klasy `UIAccelerometer`. Zamiast tego to programista jest odpowiedzialny za przechwycenie wartości. Ewentualnie możesz użyć mechanizmu opartego na bloku, którego kod będzie wykonywany podczas każdego uaktualnienia wartości (np. `startAccelerometerUpdatesToQueue:withHandler:`).

Obsługa bloku

Sposób 14.4 to praktycznie kod sposobu 14.3, ale zmodyfikowany do użycia technologii Core Motion. Polecenia z wywołań delegata zostały przeniesione do bloku, a wartości `x` i `y` są odczytywane z właściwości przyśpieszeniomierza. Poza tym kod pozostał bez zmian. W tym przykładzie możesz poznać podstawy Core Motion: utworzenie nowego menedżera ruchu. Pierwszym jego zadaniem jest sprawdzenie dostępności przyśpieszeniomierza. Następnie rozpoczyna się przekazywanie uaktualnionych wartości poprzez użycie nowej kolejki operacyjnej istniejącej przez cały czas działania aplikacji.

Sposób 14.4. *Podstawowy sposób użycia Core Motion*

```
@implementation TestBedViewController
- (void) tick
{
   butterfly.transform = CGAffineTransformIdentity;

   // Przesunięcie motyla zgodnie z bieżącym wektorem prędkości.
   CGRect rect = CGRectOffset(butterfly.frame, xvelocity, 0.0f);
   if (CGRectContainsRect(self.view.bounds, rect))
   {
      butterfly.frame = rect;
   }

   rect = CGRectOffset(butterfly.frame, 0.0f, yvelocity);
   if (CGRectContainsRect(self.view.bounds, rect))
   {
      butterfly.frame = rect;
   }

   butterfly.transform = CGAffineTransformMakeRotation(mostRecentAngle + M_PI_2);
}

- (void) shutDownMotionManager
{
   NSLog(@"Zamknięcie menedżera ruchu");
   [motionManager stopAccelerometerUpdates];
   motionManager = nil;

   [timer invalidate];
   timer = nil;
}

- (void) establishMotionManager
{
   if (motionManager)
   {
      [self shutDownMotionManager];
   }

   NSLog(@"Utworzenie menedżera ruchu.");

   // Utworzenie menedżera ruchu.
   motionManager = [[CMMotionManager alloc] init];
   if (motionManager.accelerometerAvailable)
   {
      [motionManager startAccelerometerUpdatesToQueue:[[NSOperationQueue alloc] init]
         withHandler:^(CMAccelerometerData *data, NSError *error)
      {
         // Wyodrębnienie komponentów przyśpieszeniomierza.
         float xx = -data.acceleration.x;
         float yy = data.acceleration.y;
         mostRecentAngle = atan2(yy, xx);

         // Czy nastąpiła zmiana kierunku?
         float accelDirX = SIGN(xvelocity) * -1.0f;
         float newDirX = SIGN(xx);
         float accelDirY = SIGN(yvelocity) * -1.0f;
         float newDirY = SIGN(yy);
```

```
            // Przyśpieszenie. Aby zwiększyć „lepkość", należy zmniejszyć dodawaną wartość.
            if (accelDirX == newDirX)
            {
               xaccel = (abs(xaccel) + 0.85f) * SIGN(xaccel);
            }
            if (accelDirY == newDirY)
            {
               yaccel = (abs(yaccel) + 0.85f) * SIGN(yaccel);
            }

            // Wprowadzenie zmiany przyśpieszenia względem bieżącej prędkości.
            xvelocity = -xaccel * xx;
            yvelocity = -yaccel * yy;
         }];
      }

      // Uruchomienie licznika czasu.
      timer = [NSTimer scheduledTimerWithTimeInterval: 0.03f
         target: self selector: @selector(tick)
         userInfo: nil repeats: YES];
}

- (void) initButterfly
{
   CGSize size;

   // Uruchomienie licznika czasu.
   NSMutableArray *butterflies = [NSMutableArray array];
   for (int i = 1; i <= 17; i++)
   {
      NSString *fileName = [NSString stringWithFormat:@"bf_%d.png", i];
      UIImage *image = [UIImage imageNamed:fileName];
      size = image.size;
      [butterflies addObject:image];
   }

   // Rozpoczęcie animacji.
   butterfly = [[UIImageView alloc] initWithFrame:(CGRect){.size=size}];
   [butterfly setAnimationImages:butterflies];
   butterfly.animationDuration = 0.75f;
   [butterfly startAnimating];

   // Określenie początkowej prędkości i przyśpieszenia motyla.
   xaccel = 2.0f;
   yaccel = 2.0f;
   xvelocity = 0.0f;
   yvelocity = 0.0f;

   // Dodanie motyla.
   butterfly.center = RECTCENTER(self.view.bounds);
   [self.view addSubview:butterfly];
}

- (void) loadView
{
   [super loadView];
   self.view.backgroundColor = [UIColor whiteColor];
   [self initButterfly];
}
@end
```

Pobierz kod przedstawionego sposobu

Aby pobrać kod przedstawionego tutaj sposobu, przejdź na stronę *ftp://ftp.helion.pl/przyklady/ios5pp.zip*, lub jeśli pobrałeś obraz dysku zawierający wszystkie przykłady przedstawione w książce, to przejdź do katalogu przeznaczonego dla rozdziału 14. i otwórz projekt dla tego sposobu.

Metody `establishMotionManager` i `shutDownMotionManager` pozwalają aplikacji na uruchamianie i zamykanie menedżera ruchu na żądanie. Wymienione metody są wywoływane z poziomu delegata aplikacji, gdy aplikacja staje się aktywna oraz kiedy jej działanie jest wstrzymywane.

```
- (void) applicationWillResignActive:(UIApplication *)application
{
   [tbvc shutDownMotionManager];
}

- (void) applicationDidBecomeActive:(UIApplication *)application
{
   [tbvc establishMotionManager];
}
```

Wymienione metody zapewniają elegancki sposób zamykania i wznawiania usług ruchu w odpowiedzi na aktualny stan aplikacji.

Sposób: pobieranie i używanie informacji o położeniu urządzenia

Wyobraź sobie iPada postawionego na biurku. Możesz przechylić iPada i spojrzeć na obraz wyświetlany na jego ekranie. A teraz wyobraź sobie obracanie tym iPadem leżącym płasko na biurku, ale podczas obracania urządzeniem obraz pozostaje nieruchomy i zachowuje doskonałą pozycję względem otaczającego go świata. Niezależnie od tego, jak będziesz obracał iPadem, obraz pozostanie nieruchomy, ponieważ widok będzie samodzielnie się uaktualniał w celu zrównoważenia fizycznego ruchu urządzeniem. Na takiej zasadzie działa kod przedstawiony w sposobie 14.5.

Sposób 14.5. *Użycie uaktualnień o położeniu urządzenia w celu zmiany położenia obrazu*

```
- (void) shutDownMotionManager
{
   NSLog(@"Wyłączenie menedżera ruchu");
   [motionManager stopDeviceMotionUpdates];
   motionManager = nil;
}

- (void) establishMotionManager
{
   if (motionManager)
   {
      [self shutDownMotionManager];
   }

   NSLog(@"Włączenie menedżera ruchu.");

   // Włączenie menedżera ruchu.
```

```
    motionManager = [[CMMotionManager alloc] init];
    if (motionManager.deviceMotionAvailable)
    {
        [motionManager startDeviceMotionUpdatesToQueue:[NSOperationQueue currentQueue]
        withHandler: ^(CMDeviceMotion *motion, NSError *error) {
            CATransform3D transform;
            transform = CATransform3DMakeRotation(motion.attitude.pitch, 1, 0, 0);
            transform = CATransform3DRotate(transform, motion.attitude.roll, 0, 1, 0);
            transform = CATransform3DRotate(transform, motion.attitude.yaw, 0, 0, 1);
            imageView.layer.transform = transform;
        }];
    }
}
```

Pobierz kod przedstawionego sposobu

Aby pobrać kod przedstawionego tutaj sposobu, przejdź na stronę *ftp://ftp.helion.pl/przyklady/ios5pp.zip*, lub jeśli pobrałeś obraz dysku zawierający wszystkie przykłady przedstawione w książce, to przejdź do katalogu przeznaczonego dla rozdziału 14. i otwórz projekt dla tego sposobu.

Obraz dostosowuje się niezależnie od sposobu, w jaki będziesz trzymał urządzenie. Oprócz tego możesz podnieść urządzenie i zmieniać jego orientację w przestrzeni. Jeżeli odwrócisz urządzenie i spojrzysz na nie, wtedy zobaczysz odwrócony „dół" obrazu. Urządzeniem możesz obracać w obu osiach: jedna przebiega pionowo od przycisku *Początek* do aparatu, natomiast druga poziomo od lewej do prawej strony iPada. Kolejna oś — pierwsza, którą poznasz — przechodzi jakby „na wylot" przez środek urządzenia, od punktu znajdującego się pod iPadem aż do punktu nad urządzeniem. Kiedy obracasz urządzeniem, reakcją obrazu jest utworzenie wirtualnego świata w tym urządzeniu.

W kodzie sposobu 14.5 pokazano, jak uzyskać powyższy efekt za pomocą kilku prostych przekształceń geometrycznych. Kod tworzy menedżera ruchu, pobiera uaktualnienia dotyczące ruchu urządzenia, a następnie na podstawie otrzymanych danych przeprowadza transformację obrazu.

Użycie zdarzeń ruchu do wykrycia wstrząśnięcia urządzeniem

Kiedy iPhone wykrywa zdarzenie ruchu, przekazuje je do obiektu aktualnie posiadającego przypisany status *First Responder*, czyli do podstawowego obiektu w łańcuchu odpowiedzi. Obiekty, które mogą mieć przypisany wymieniony status, potrafią przetwarzać zdarzenia. Zaliczamy do nich wszystkie widoki i okna, a także obiekt aplikacji.

Łańcuch odpowiedzi definiuje hierarchię obiektów, które mogą odpowiadać na zdarzenia. Kiedy obiekt znajdujący się na początku łańcucha odpowiedzi otrzymuje zdarzenie, to zdarzenie nie jest przekazywane dalej, ale obsłużone przez dany obiekt. Jeśli obiekt nie potrafi obsłużyć zdarzenia, wówczas przekazuje je do kolejnego obiektu w łańcuchu.

Obiekt może odpowiadać na zdarzenia, gdy ma przypisany status *First Responder* za pomocą metody o nazwie `becomeFirstResponder`. W przedstawionym poniżej fragmencie kodu obiekt `UIViewController` będzie odpowiadał na zdarzenia, gdy jego widok pojawi się na ekranie. Po usunięciu widoku z ekranu obiekt przestanie odpowiadać na zdarzenia.

```
- (BOOL)canBecomeFirstResponder
{
   return YES;
}

// Przypisanie statusu First Responder po wyświetleniu widoku na ekranie.
- (void)viewDidAppear:(BOOL)animated
{
   [super viewDidAppear:animated];
   [self becomeFirstResponder];
}

// Rezygnacja ze statusu First Responder po usunięciu widoku z ekranu.
- (void)viewWillDisappear:(BOOL)animated
{
   [super viewWillDisappear:animated];
   [self resignFirstResponder];
}
```

Obiekty z przypisanym statusem *First Responder* otrzymują wszystkie zdarzenia dotyku i ruchu. Wywołania zwrotne ruchu odpowiadają wywołaniom zwrotnym dotknięć, omówionym w rozdziale 8. zatytułowanym „Gesty i dotknięcia".

- **`motionBegan:withEvent:`** — ta metoda wskazuje na początek zdarzenia ruchu. W trakcie pisania niniejszej książki istniał tylko jeden rodzaj rozpoznawanego zdarzenia: wstrząśnięcie. To się może zmienić w przyszłości, więc w kodzie warto umieścić procedurę sprawdzania typu ruchu.
- **`motionEnded:withEvent:`** — ta metoda jest wywoływana na końcu zdarzenia ruchu.
- **`motionCancelled:withEvent:`** — podobnie jak w przypadku dotknięć, także ruch może zostać przerwany przez połączenie przychodzące lub inne zdarzenie systemowe. Firma Apple zaleca implementację w kodzie produkcyjnym wszystkich trzech wymienionych tutaj metod (i podobnie wszystkich czterech metod dotyczących obsługi dotknięć).

Przedstawiony poniżej fragment kodu pokazuje przykład użycia metod obsługi zdarzeń ruchu. Po przetestowaniu tej aplikacji w rzeczywistym urządzeniu możesz mieć kilka spostrzeżeń. Przede wszystkim z punktu widzenia użytkownika zdarzenia rozpoczynające i kończące ruch występują niemal jednocześnie. Odtwarzanie dźwięku w obu wymienionych rodzajach zdarzeń jest zabójcze. Ponadto istnieje tendencja w kierunku wykrywania gestu wstrząśnięcia urządzeniem na boki. iPhone lepiej sprawdza się podczas wykrywania wstrząśnięcia urządzeniem na boki niż do przodu i do tyłu. Wreszcie zastosowana przez firmę Apple implementacja używa lekkiego podejścia blokowania. Nie ma możliwości wygenerowania nowego zdarzenia ruchu przed upływem sekundy od chwili przetworzenia poprzedniego. Takie samo blokowanie jest używane w gestach „wstrząśnij, aby pomieszać" i „wstrząśnij, aby cofnąć".

```
- (void)motionBegan:(UIEventSubtype)motion
   withEvent:(UIEvent *)event
{
   // Odtwórz dźwięk po rozpoczęciu gestu wstrząśnięcia.
   if (motion != UIEventSubtypeMotionShake)
   {
      return;
   }
   [self playSound:startSound];
}
```

```
- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
   // Odtwórz dźwięk po zakończeniu gestu wstrząśnięcia.
   if (motion != UIEventSubtypeMotionShake)
   {
      return;
   }
   [self playSound:endSound];
}
```

Sposób: użycie przyśpieszeniomierza do wykrycia gestu wstrząśnięcia

W sposobie 14.6 przedstawiłam rozwiązanie powielające przygotowany przez firmę Apple system wykrywania ruchu, ale bez konieczności przekazywania zdarzeń obiektowi z przypisanym statusem *First Responder*. Projekt jest zbudowany w oparciu o dwa kluczowe parametry: poziom czułości określający granicę, która musi być przekroczona w celu rozpoznania wstrząśnięcia, oraz czas blokady określający częstotliwość, z jaką mogą być generowane zdarzenia wstrząśnięcia.

Sposób 14.6. *Wykrywanie gestu wstrząśnięcia za pomocą przyśpieszeniomierza*

```
@implementation AccelerometerHelper
- (id) init
{
   if (!(self = [super init]))
   {
      return self;
   }

   self.triggerTime = [NSDate date];

   // Bieżący wektor siły.
   cx = UNDEFINED_VALUE;
   cy = UNDEFINED_VALUE;
   cz = UNDEFINED_VALUE;

   // Ostatni wektor siły.
   lx = UNDEFINED_VALUE;
   ly = UNDEFINED_VALUE;
   lz = UNDEFINED_VALUE;

   // Poprzedni wektor siły.
   px = UNDEFINED_VALUE;
   py = UNDEFINED_VALUE;
   pz = UNDEFINED_VALUE;

   self.sensitivity = 0.5f;
   self.lockout = 0.5f;

   // Przypisanie delegata dla przyśpieszeniomierza.
   [[UIAccelerometer sharedAccelerometer] setDelegate:self];

   return self;
}
```

```
- (void) setX: (float) aValue
{
   px = lx;
   lx = cx;
   cx = aValue;
}

- (void) setY: (float) aValue
{
   py = ly;
   ly = cy;
   cy = aValue;
}

- (void) setZ: (float) aValue
{
   pz = lz;
   lz = cz;
   cz = aValue;
}

- (float) dAngle
{
   if (cx == UNDEFINED_VALUE)
   {
      return UNDEFINED_VALUE;
   }
   if (lx == UNDEFINED_VALUE)
   {
      return UNDEFINED_VALUE;
   }
   if (px == UNDEFINED_VALUE)
   {
      return UNDEFINED_VALUE;
   }

   // Obliczenia dla pierwszej pary.
   float dot1 = cx * lx + cy * ly + cz * lz;
   float a = ABS(sqrt(cx * cx + cy * cy + cz * cz));
   float b = ABS(sqrt(lx * lx + ly * ly + lz * lz));
   dot1 /= (a * b);

   // Obliczenia dla drugiej pary.
   float dot2 = lx * px + ly * py + lz * pz;
   a = ABS(sqrt(px * px + py * py + pz * pz));
   dot2 /= a * b;

   // Zwrot różnicy pomiędzy kątami wektorów.
   return acos(dot2) - acos(dot1);
}

- (BOOL) checkTrigger
{
   if (lx == UNDEFINED_VALUE)
   {
      return NO;
   }

   // Sprawdzenie, czy można przetworzyć nowe dane
   if ([[NSDate date] timeIntervalSinceDate:self.triggerTime] < self.lockout)
```

```
    {
        return NO;
    }

    // Pobranie aktualnej zmiany kąta.
    float change = [self dAngle];

    // Jeżeli dwie próbki nie zostały jeszcze zebrane, wartością zwrotną jest NO.
    if (change == UNDEFINED_VALUE)
    {
        return NO;
    }

    // Czy obliczona wartość jest większa od granicznej?
    if (change > self.sensitivity)
    {
        self.triggerTime = [NSDate date];
        return YES;
    }
    else
    {
        return NO;
    }
}

- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // Przystosowanie wartości do standardowego systemu współrzędnych.
    [self setX:-acceleration x];
    [self setY:acceleration.y];
    [self setZ:acceleration.z];

    // Wszystkie zdarzenia przyśpieszeniomierza.
    if (self.delegate && [self.delegate respondsToSelector:@selector(ping)])
    {
        [self.delegate performSelector:@selector(ping)];
    }

    // Wszystkie zdarzenia wstrząśnięcia.
    if ([self checkTrigger] && self.delegate &&
        [self.delegate respondsToSelector:@selector(shake)])
    {
        [self.delegate performSelector:@selector(shake)];
    }
}
@end
```

Pobierz kod przedstawionego sposobu

Aby pobrać kod przedstawionego tutaj sposobu, przejdź na stronę *ftp://ftp.helion.pl/przyklady/ios5pp.zip*, lub jeśli pobrałeś obraz dysku zawierający wszystkie przykłady przedstawione w książce, to przejdź do katalogu przeznaczonego dla rozdziału 14. i otwórz projekt dla tego sposobu.

Klasa `AccelerometerHelper` przechowuje trzy wartości przyśpieszeniomierza. Każda z wartości przedstawia wektor siły w przestrzeni 3D. Każda kolejna para tych trzech wartości może być analizowana w celu określenia kąta pomiędzy dwoma wektorami. W omawianym przykładzie kąty pomiędzy pierwszymi dwoma elementami

i dwoma kolejnymi pomagają w ustaleniu wystąpienia gestu wstrząśnięcia. Kąt wyszukuje parę, w której drugi kąt jest większy od pierwszego. Jeżeli wartość kątowa wzrosła wystarczająco dużo, wówczas wykryty zostanie gest wstrząśnięcia.

Klasa pomocnicza nie wywołuje metod delegata aż do przekazania drugiego zdarzenia. Blokada uniemożliwia przeprowadzanie kolejnych wywołań aż do chwili upłynięcia zdefiniowanej ilości czasu. Takie rozwiązanie jest zaimplementowane poprzez przechowywanie godziny ostatniego zdarzenia wstrząśnięcia. Wszystkie wstrząśnięcia, które nastąpią przed upływem zdefiniowanej ilości czasu, zostaną zignorowane. Następnie mogą być generowane nowe gesty wstrząśnięcia.

Dostarczany przez firmę Apple system wykrywania wstrząśnięcia wykorzystuje znacznie bardziej skomplikowaną analizę danych przyśpieszeniomierza. Zgodnie z informacjami technicznymi na ten temat system wyszukuje oscylacji w około ośmiu do dziesięciu kolejnych punktach danych. W kodzie sposobu 14.6 zastosowano mniej skomplikowane podejście i zademonstrowano, jak pracować z nieprzetworzonymi danymi przyśpieszeniomierza w celu dostarczenia wyników przygotowanych na podstawie wspomnianych danych.

Sposób: używanie ekranów zewnętrznych

Istnieje wiele sposobów używania ekranów zewnętrznych z iOS 5. Weźmy np. iPada 2, który oferuje wbudowaną funkcję reprodukcji wideo na zewnętrznym ekranie. Dzięki użyciu przewodu VGA lub HDMI treść możesz wyświetlać zarówno w iPadzie, jak i na zewnętrznym ekranie. Począwszy od systemu iOS 5, wybrane urządzenia pozwalają na bezprzewodową reprodukcję wideo do Apple TV za pomocą technologii AirPlay. Wspomniana funkcja reprodukcji jest wyjątkowo użyteczna, ale w iOS nie jesteś ograniczony jedynie do kopiowania treści z jednego ekranu na drugi.

Klasa `UIScreen` pomaga w wykryciu i niezależnym użyciu zewnętrznego ekranu, który można potraktować jako nowe okno i dla którego można przygotować treść oddzielną od wyświetlanej na ekranie urządzenia iOS. Zewnętrzny ekran można podłączyć zarówno przewodem, jak i bezprzewodowo do Apple TV za pomocą AirPlay (począwszy od iOS 5 i iPada 2).

Geometria ma duże znaczenie i zaraz dowiesz się dlaczego. Obecnie urządzenia iOS korzystają z ekranów następujących urządzeń: stary iPhone o rozdzielczości 320×480 pikseli, Retina w iPhone'ie o rozdzielczości 640×960 pikseli, stary iPad o rozdzielczości 1024×768 pikseli i Retina w iPadzie o rozdzielczości 2048×1536 pikseli. Typowy obraz ma rozdzielczość 720×480 pikseli (480i i 480p), VGA ma rozdzielczości 1024×768 i 1280×720 (720p), a ponadto jest jeszcze dostępna rozdzielczość HD 1920×1080 pikseli (poprzez HDMI).

Dodaj do tego problemy z tak zwanym overscanem oraz inne ograniczenia wyświetlacza docelowego, a wyjście wideo szybko stanie się wyzwaniem pod względem geometrii. Na szczęście firma Apple oferuje odpowiednie rozwiązanie w systemie iOS 5. Zamiast próbować tworzyć związek typu jeden do jednego pomiędzy ekranem wbudowanym w urządzenie i zewnętrznym, po prostu utwórz treść w oparciu o dostępne właściwości ekranu. Możesz więc utworzyć okno, wypełnić je treścią, a następnie wyświetlić.

Ogólnie rzecz biorąc, jeśli zamierzasz tworzyć aplikacje korzystające z zewnętrznego ekranu, upewnij się o posiadaniu co najmniej jednego przewodu każdego z wymienionych typów (kompozytowe, komponentowe, VGA i HDMI), jak również obsługującego technologię AirPlay iPada, co pozwoli Ci na przetestowanie każdej możliwej konfiguracji. Przewody produkowane przez firmy trzecie mogą nie działać, więc upewnij się, że kupujesz przewody produkcji Apple.

Wykrywanie ekranu

Klasa `UIScreen` informuje o liczbie dostępnych ekranów. Jeżeli ich liczba jest większa niż jeden, wtedy wiadomo, że podłączony został ekran zewnętrzny. Pierwszym elementem tablicy `screens` zawsze jest ekran wbudowany w urządzenie iOS.

```
#define SCREEN_CONNECTED ([UIScreen screens].count > 1)
```

Każdy ekran podaje swoje wymiary (fizyczne, wyrażone w punktach) oraz skalę ekranu (punkty względem pikseli). Dwa standardowe powiadomienia pozwalają na obserwację podłączania i odłączania ekranów od urządzenia.

```
// Rejestracja otrzymywania powiadomień o włączeniu i wyłączeniu ekranu.
[[NSNotificationCenter defaultCenter]
   addObserver:self selector:@selector(screenDidConnect:)
   name:UIScreenDidConnectNotification object:nil];
[[NSNotificationCenter defaultCenter]
   addObserver:self selector:@selector(screenDidDisconnect:)
   name:UIScreenDidDisconnectNotification object:nil];
```

Połączenie oznacza *dowolnego* rodzaju połączenie, zarówno przewodowe, jak i bezprzewodowe w technologii AirPlay. Po otrzymaniu tego rodzaju uaktualnienia upewnij się o policzeniu ekranów i dostosowaniu interfejsu użytkownika do właściwości nowych ekranów.

Do Ciebie należy zadanie konfiguracji okien po podłączeniu i odłączeniu ekranu od urządzenia iOS. Każdy ekran powinien posiadać własne okno przeznaczone do zarządzania treścią wyświetlaną na tym ekranie. Po wykryciu odłączenia ekranu nie zatrzymuj okna w pamięci, usuń je i powtórnie je utwórz po wykryciu ponownego podłączenia ekranu.

Pobieranie informacji o rozdzielczości ekranu

Każdy ekran posiada właściwość `availableModes`, to tablica obiektów rozdzielczości uporządkowanych w kolejności od najmniejszej do największej. Z kolei każdy tryb posiada właściwość `size` wskazującą rozdzielczość ekranu wyrażoną w pikselach. Wiele ekranów obsługuje różne rozdzielczości. Na przykład wyświetlacz VGA może obsługiwać kilka różnych rozdzielczości, których liczba zależy od producenta ekranu. Zawsze dostępna jest przynajmniej jedna rozdzielczość, ale jeśli ekran udostępnia ich więcej, to użytkownikowi również powinieneś pozwolić na ich wybór. W tym celu odpowiednio przygotuj kod aplikacji.

Konfiguracja wyjścia wideo

Po pobraniu obiektu zewnętrznego ekranu z tablicy `[UIScreen screens]` sprawdź dostępne tryby i wybierz jeden z nich, który będzie następnie używany. Ogólnie rzecz biorąc, zawsze powinieneś wybierać ostatni element tablicy, udostępniając tym samym użytkownikowi największą możliwą rozdzielczość, ewentualnie możesz wybrać pierwszy — oznaczający najniższą rozdzielczość.

W celu rozpoczęcia przesyłania strumienia wideo musisz utworzyć egzemplarz `UIWindow` i dostosować jego wielkość do używanej rozdzielczości ekranu. W oknie trzeba umieścić nowy widok pozwalający na wyświetlenie treści. Kolejny krok to przypisanie okna zewnętrznemu ekranowi i uczynienie go podstawowym i widocznym. W ten sposób okno zostanie wyświetlone i przygotowane do użycia. Kiedy zakończysz pracę z oknem na ekranie zewnętrznym, ekran w urządzeniu ponownie ustaw jako podstawowy. Nie pomijaj tego kroku! Dla użytkownika nie ma nic bardziej irytującego niż przekonanie się, że drogie urządzenie iOS nie reaguje na dotknięcia ekranu.

```
self.outwindow = [[UIWindow alloc] initWithFrame:theFrame];
outwindow.screen = secondaryScreen;
[outwindow makeKeyAndVisible];
[delegate.view.window makeKeyAndVisible];
```

Dodanie obiektu DisplayLink

Obiekt `DisplayLink` to rodzaj licznika czasu, który jest zsynchronizowany z częstotliwością odświeżania ekranu. Częstotliwość odświeżania ramki możesz zmienić poprzez modyfikację właściwości `frameInterval`. Wartość domyślna tej właściwości wynosi 1. Ustawienie wyższej wartości spowoduje zmniejszenie częstotliwości odświeżania. Tak więc wartość 2 oznacza zmniejszenie o połowę częstotliwości odświeżania. Obiekt `DisplayLink` utwórz po podłączeniu ekranu do urządzenia. Klasa `UIScreen` implementuje metodę, która zwraca obiekt `DisplayLink` dla ekranu. Dla wymienionego obiektu możesz zdefiniować cel oraz wskazać wywoływany selektor.

Obiekt `DisplayLink` regularnie wywołuje zdefiniowaną metodę, pozwalając Ci w ten sposób na uaktualnienie treści wyświetlanej na ekranie. Zmniejszenie częstotliwości powoduje mniejsze obciążenie procesora, ale kosztem utraty niektórych ramek. Trzeba koniecznie pamiętać o tym koszcie, zwłaszcza w przypadku bezpośredniej manipulacji interfejsami wymagającymi wiele pracy ze strony procesora w celu udzielenia odpowiedzi na działania użytkownika.

W dalszej części rozdziału znajdziesz kod sposobu 14.7, w którym użyto najczęściej spotykanych trybów dla pętli działania, zapewniając tym samym najmniejsze opóźnienie. Po zakończeniu pracy z obiektem `DisplayLink` wywołaj metodę `invalidate`, która powoduje usunięcie obiektu z pętli działania.

Kompensacja overscanningu

Poprzez przypisanie wartości właściwości `overscanCompensation` klasa `UIScreen` pozwala na kompensację w przypadku utraty pikseli na krawędziach ekranu. Ta technika została przedstawiona w dokumentacji Apple; w zasadzie odpowiada sytuacji, gdy chcesz obciąć treść lub dopełnić ją czarną przestrzenią.

VIDEOkit

W sposobie 14.7 przedstawiłam VIDEOkit, czyli klienta ekranu zewnętrznego. Zawiera on wszystkie funkcje niezbędne do konfiguracji i użycia ekranu zewnętrznego podłączonego zarówno przewodowo, jak i bezprzewodowo. Monitorowanie ekranu włączasz poprzez wywołanie `startupWithDelegate:`. Przekaż podstawowy kontroler widoku, którego zadaniem będzie utworzenie treści na zewnętrznym ekranie.

Sposób 14.7. *VIDEOkit*

```
@interface VIDEOkit : NSObject
{
   UIImageView *baseView;
}
@property (nonatomic, weak)   UIViewController *delegate;
@property (nonatomic, strong) UIWindow *outwindow;
@property (nonatomic, strong) CADisplayLink *displayLink;
+ (void) startupWithDelegate: (id) aDelegate;
@end

@implementation VIDEOkit
@synthesize delegate;
```

```
@synthesize outwindow, displayLink;

static VIDEOkit *sharedInstance = nil;

- (void) setupExternalScreen
{
   // Sprawdzenie w poszukiwaniu brakujących ekranów.
   if (!SCREEN_CONNECTED)
   {
      return;
   }

   // Konfiguracja ekranu zewnętrznego.
   UIScreen *secondaryScreen = [[UIScreen screens] objectAtIndex:1];
   UIScreenMode *screenMode = [[secondaryScreen availableModes] lastObject];
   CGRect rect = CGRectMake(0.0f, 0.0f, screenMode.size.width, screenMode.size.height);

   // Utworzenie nowego okna.
   self.outwindow = [[UIWindow alloc] initWithFrame:CGRectZero];
   outwindow.screen = secondaryScreen;
   outwindow.screen.currentMode = screenMode; // Podziękowania dla Scotta Lawrence'a.
   [outwindow makeKeyAndVisible];
   outwindow.frame = rect;

   // Dodanie do okna podstawowego widoku.
   baseView = [[UIImageView alloc] initWithFrame:rect];
   baseView.backgroundColor = [UIColor clearColor];
   [outwindow addSubview:baseView];

   // Ustawienie jako okno główne.
   [delegate.view.window makeKeyAndVisible];
}

- (void) updateScreen
{
   // Przerwanie, jeśli ekran został odłączony.
   if (!SCREEN_CONNECTED && outwindow)
   {
      self.outwindow = nil;
   }

   // (Ponowna) inicjalizacja, jeśli nie ma okna na ekranie zewnętrznym.
   if (SCREEN_CONNECTED && !outwindow)
   {
      [self setupExternalScreen];
   }

   // Przerwanie w przypadku napotkania dziwnego błędu.
   if (!self.outwindow)
   {
      return;
   }

   // Przeprowadzenie uaktualnienia.
   SAFE_PERFORM_WITH_ARG(delegate, @selector(updateExternalView:), baseView);
}

- (void) screenDidConnect: (NSNotification *) notification
{
   NSLog(@"Ekran został podłączony");
```

```
    UIScreen *screen = [[UIScreen screens] lastObject];
    if (displayLink)
    {
        [displayLink removeFromRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
        [displayLink invalidate];
        self.displayLink = nil;
    }

    // Sprawdzenie istnienia obiektu DisplayLink.
    if (!displayLink)
    {
        self.displayLink = [screen displayLinkWithTarget:self
            selector:@selector(updateScreen)];
        [displayLink addToRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
    }
}

- (void) screenDidDisconnect: (NSNotification *) notification
{
    NSLog(@"Ekran został odłączony.");
    if (displayLink)
    {
        [displayLink removeFromRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
        [displayLink invalidate];
        self.displayLink = nil;
    }
}

- (id) init
{
    if (!(self = [super init]))
    {
        return self;
    }

    // Obsługa utworzenia okna na ekranie zewnętrznym.
    if (SCREEN_CONNECTED)
    {
        [self screenDidConnect:nil];
    }

    // Rejestracja powiadomień informujących o podłączeniu lub odłączeniu ekranu.
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(screenDidConnect:)
        name:UIScreenDidConnectNotification object:nil];
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(screenDidDisconnect:)
        name:UIScreenDidDisconnectNotification object:nil];

    return self;
}

- (void) dealloc
{
    [self screenDidDisconnect:nil];
    self.outwindow = nil;
}
```

```
+ (VIDEOkit *) sharedInstance
{
   if (!sharedInstance)
   {
      sharedInstance = [[self alloc] init];
   }
   return sharedInstance;
}

+ (void) startupWithDelegate: (id) delegate
{
   [[self sharedInstance] setDelegate:delegate];
}
@end
```

Pobierz kod przedstawionego sposobu

Aby pobrać kod przedstawionego tutaj sposobu, przejdź na stronę *ftp://ftp.helion.pl/przyklady/ios5pp.zip*, lub jeśli pobrałeś obraz dysku zawierający wszystkie przykłady przedstawione w książce, to przejdź do katalogu przeznaczonego dla rozdziału 14. i otwórz projekt dla tego sposobu.

Wewnętrzna metoda `init` rozpoczyna nasłuchiwanie zdarzeń informujących o podłączeniu lub odłączeniu ekranu, a następnie tworzy lub usuwa okno. Metoda delegata (`updateExternalView:`) jest wywoływana przez obiekt `DisplayLink`. Parametrem jest widok znajdujący się w zewnętrznym oknie; we wspomnianym widoku delegat może wyświetlić przygotowaną treść.

W omawianym przykładzie kontroler widoku przechowuje wartość koloru lokalnego, a następnie używa jej na ekranie zewnętrznym.

```
- (void) updateExternalView: (UIImageView *) aView
{
   aView.backgroundColor = color;
}

- (void) action: (id) sender
{
   color = [UIColor randomColor];
}
```

Każde naciśnięcie przycisku powoduje wygenerowanie przez kontroler widoku nowego koloru. Kiedy VIDEOkit nakazuje kontrolerowi uaktualnienie zewnętrznego widoku, wspomniany kolor jest używany jako kolor tła. Możesz się przekonać, że tło na ekranie zewnętrznym jest natychmiast uaktualniane nowo wygenerowanym kolorem.

I jeszcze jedno: sprawdzenie dostępnej ilości wolnego miejsca na dysku

Klasa `NSFileManager` pozwala na sprawdzenie wolnej ilości miejsca pozostałej w urządzeniu iOS, jak również całkowitej ilości miejsca na dysku dostarczanej przez urządzenie iOS. W listingu 14.1 zademonstrowałam, jak sprawdzić obie wartości i wyświetlić je w postaci łatwego w odczycie ciągu tekstowego. Zwrócone wartości podają ilość miejsca wyrażoną w bajtach.

Listing 14.1. *Pobranie informacji o wielkości systemu plików i pozostałej ilości wolnego miejsca*

```
- (NSString *) commaFormattedStringWithLongLong: (long long) num
{
   // Wygenerowanie odpowiednio sformatowanego ciągu tekstowego.
   // Alternatywnym rozwiązaniem jest użycie egzemplarza NSNumberFormatter.
   if (num < 1000)
   {
      return [NSString stringWithFormat:@"%d", num];
   }
   return [[self commasForNumber:num/1000]
      stringByAppendingFormat:@",%03d", (num % 1000)];
}

- (void) action: (UIBarButtonItem *) bbi
{
   NSFileManager *fm = [NSFileManager defaultManager];
   NSDictionary *fattributes = [fm fileSystemAttributesAtPath:NSHomeDirectory()];
   NSLog(@"Całkowita ilość miejsca na dysku: %@",
      [self commaFormattedStringWithLongLong:[[fattributes
      objectForKey:NSFileSystemSize] longLongValue]]);
   NSLog(@"Wolna ilość miejsca na dysku: %@",
      [self commasForNumber:[[fattributes
      objectForKey:NSFileSystemFreeSize] longLongValue]]);
}
```

Podsumowanie

W tym rozdziale przedstawiłam podstawowe sposoby interakcji z urządzeniem iPhone. Dowiedziałeś się, jak pobrać informacje dotyczące urządzenia, sprawdzić poziom naładowania baterii oraz nasłuchiwać zdarzeń czujnika zbliżeniowego. Poznałeś też sposoby rozróżniania urządzeń iPod touch, iPhone i iPad oraz określania modelu urządzenia, w którym została uruchomiona aplikacja. Przedstawiłam przyśpieszeniomierz i jego użycie w kilku przykładach, począwszy od prostego „wyszukania" aż po bardziej skomplikowany algorytm wykrywania gestu wstrząśnięcia. Poznałeś także technologię Core Motion i dowiedziałeś się, jak tworzyć bloki pozwalające na udzielanie odpowiedzi w czasie rzeczywistym na zdarzenia urządzenia. Wreszcie omówiłam temat dodawania obsługi ekranów zewnętrznych w aplikacjach. Zanim przejdziesz dalej, poświęć chwilę i przemyśl kilka kwestii dotyczących sposobów przedstawionych w rozdziale.

- Przyśpieszeniomierz w iPhone'ie stanowi uzupełnienie oferowanego przez urządzenie interfejsu dotykowego. Dane przyśpieszeniomierza możesz wykorzystać w celu rozbudowy interakcji z użytkownikiem poza „dotknij tutaj" i umożliwić aplikacji reakcję na obrót urządzeniem.
- Niskiego poziomu wywołania mogą być przyjazne dla aplikacji zgłaszanej do sklepu iTunes App Store. Nie zależą od API firmy Apple, które może być zmienione w aktualnym wydaniu oprogramowania typu firmware. Wywołania systemu UNIX mogą być kłopotliwe, ale wiele z nich jest w pełni obsługiwanych przez urządzenia z rodziny iOS.
- Nie zapominaj o ograniczeniach urządzeń. Przed przeprowadzeniem operacji wymagającej intensywnego przetwarzania pliku sprawdź ilość wolnej pamięci w urządzeniu. Podobnie przed operacją wymagającą użycia pełnej mocy procesora sprawdź, czy urządzenie jest podłączone do źródła zasilania.

- Wykorzystuj technologię Core Motion. Dostarczane przez nią dane w czasie rzeczywistym są podstawą dla wielu interesujących aplikacji na platformie iOS.
- Skoro technologia AirPlay pozwala na pozbycie się przewodów łączących urządzenie iOS z zewnętrznym ekranem, masz możliwość użycia wyjścia wideo w celu utworzenia jeszcze bardziej ekscytujących projektów, niż wcześniej sądziłeś. Technologia AirPlay i zewnętrzne ekrany mogą pomóc w przekształceniu urządzenia iOS na zdalnego pilota do gier i narzędzi wyświetlających treść na dużym ekranie i kontrolujących tę treść za pomocą mniejszego urządzenia iOS.
- Podczas zgłaszania aplikacji do sklepu iTunes App Store używaj pliku *Info.plist* w celu określenia funkcji urządzenia wymaganych dla danej aplikacji. iTunes używa wspomnianej listy, ustalając, czy dana aplikacja może zostać pobrana w konkretnym urządzeniu i czy będzie w nim prawidłowo działała.

Skorowidz

E

F

G

H

I

J

K

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

Ź

ZAOSZCZĘDŹ CZAS
DZIĘKI POMOCNYM PRZYKŁADOM!

iOS 5. Podręcznik programisty

Oto książka poświęcona programowaniu na platformie iOS na którą czekałeś!

iOS to jeden z najpopularniejszych systemów operacyjnych wykorzystywanych w urządzeniach mobilnych. Znajdziesz go we wszystkich sprzętach Apple – iPhone'ach, iPadach i iPodach. Różnorodność urządzeń gwarantuje Ci dużą liczbę użytkowników, ale z drugiej strony dostarcza wielu nowych wyzwań. Zastanawiasz się, jak poradzić sobie z codziennymi problemami pojawiającymi się w trakcie tworzenia aplikacji dla iOS?

W tej książce Erica Sadun – autorka bestsellerów i guru programowania na platformie iOS – umieściła wszystkie informacje potrzebne do tego, abyś jak najszybciej zaczął tworzyć znakomite aplikacje dla urządzeń mobilnych na iOS 5. Znajdziesz tu gotowy do natychmiastowego użycia i łatwy w rozbudowie kod źródłowy, co pozwoli Ci uniknąć konieczności szukania własnych rozwiązań metodą prób i błędów. Przykłady przygotowane i przetestowane przez autorkę dotyczą praktycznie każdej dziedziny programowania na platformie iOS, od tworzenia interfejsu użytkownika, poprzez kontrolery widoku, gesty i dotknięcia, aż do obsługi sieci oraz kwestii bezpieczeństwa.

- Opanowanie iOS 5 SDK, podstaw języka programowania Objective-C oraz cyklu życiowego tworzenia oprogramowania na platformie iOS
- Projektowanie i dostosowanie do własnych potrzeb interfejsu użytkownika przy użyciu modułu Interface Builder i języka Objective-C
- Organizowanie aplikacji za pomocą kontrolerów widoku, widoków i animacji, w tym najnowszego kontrolera widoku strony oraz własnych kontenerów
- Maksymalne wykorzystanie dotknięć i gestów oraz tworzenie własnych procedur rozpoznawania gestów
- Praca z technologiami Core Image i Core Text
- Implementacja w pełni wyposażonych widoków tabel, włącznie z edycją komórek, zmianą ich kolejności oraz stosowaniem własnych komórek
- Tworzenie magazynów zarządzanych danych, dodawanie, usuwanie, pobieranie i wyświetlanie danych
- Wyświetlanie komunikatów dla użytkownika w postaci okien dialogowych, pasków postępu, powiadomień lokalnych i typu push, okien typu Popover oraz polecenia ping
- Żądanie i obsługa informacji pochodzących od użytkowników
- Nawiązywanie połączenia z sieciami i usługami, obsługa uwierzytelniania oraz zarządzanie pobieraniem danych
- Instalacja aplikacji w urządzeniach iOS, przekazywanie aplikacji testerom wersji beta oraz do sklepu iTunes App Store

Erica Sadun jest autorką bestsellerów oraz współautorką licznych książek o programowaniu. Opracowała dziesiątki aplikacji na platformę iOS, oferuje usługi konsultingowe w zakresie szybkiego tworzenia prototypów aplikacji. Jej artykuły są publikowane w wielu serwisach, między innymi Ars Technica, O'Reilly i LifeHacker. Obecnie regularnie publikuje w serwisie TUAW. Uzyskała tytuł doktora nauk informatycznych na Georgia Institute of Technology.

helion.pl
księgarnia internetowa

Nr katalogowy: 13130

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

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

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-5121-4

Cena 119.00 zł

Informatyka w najlepszym wydaniu