

Le Programmeur

Développez des applications originales pour

iPhone et iPod Touch

**Revendez
vos créations
sur iTunes
App Store**



Jean-Marc Delprato

PEARSON

LE PROGRAMMEUR

Développez des applications originales pour iPhone et iPod Touch

Revendez vos créations sur iTunes App Store

Jean-Marc Delprato

PEARSON


Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Avec la contribution de Fabien Schwob

Collaboration éditoriale
et mise en pages : Digit Books (Dominique Buraud)

ISBN : 978-2-7440-4127-3

Copyright © 2010 Pearson Education France

Tous droits réservés

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

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

Sommaire

Préface	1	5. Les vues de votre application	107
1. Le SDK de l'iPhone et de l'iPod Touch	7	6. Les contrôles spécifiques	153
2. Vue d'ensemble du développement pour iPhone	29	7. Lire et écrire des données	207
3. À la découverte du SDK de l'iPhone	45	8. Le multimédia	245
4. Interactions avec vos applications	75	9. Réagir aux gestes de l'utilisateur	287
		10. Publication et marketing	317
		Index	343

Table des matières

Préface	1	La filiation entre l'iPhone et Mac OS X	32
Organisation de cet ouvrage	2	Une première approche de Cocoa	34
Prérequis	4	Le langage de programmation	
Conventions typographiques	5	Objective-C	35
Remerciements	5	Les couches technologiques de l'iPhone OS	
À propos de l'auteur	6	et l'intégration de Cocoa Touch	37
		Les spécificités et les limites de l'iPhone	39
Chapitre 1. Le SDK de l'iPhone et de l'iPod Touch	7	En route vers Xcode	43
Les éléments à réunir	8	Chapitre 3. À la découverte du SDK de l'iPhone	45
Installer le SDK	13	Créer votre premier projet pour iPhone	46
Avant d'aller plus loin : préparer votre projet	17	Maîtriser Xcode	46
Établir un cahier des charges sommaire	18	Structure générale de tous les projets pour iPhone	49
Les types d'applications de l'iPhone : faites jaillir toutes sortes d'idées !	19	Maîtriser Interface Builder	52
Les bonnes idées d'applications et les projets à éviter	23	Ajouter des objets avec Interface Builder	54
Concevoir et designer son application sur papier	24	Les objets de la bibliothèque d'Interface Builder	58
Chapitre 2. Vue d'ensemble du développement pour iPhone	29	Tester et déboguer votre premier projet	62
Les outils à votre disposition	30	Compiler votre projet	63
À la découverte de Cocoa Touch et de l'Objective-C	32	Ajouter l'icône de votre application	66
		Pour aller plus loin : importer des projets officiels d'Apple	68
		En route vers le dialogue avec l'utilisateur	72

Chapitre 4. Interactions avec vos applications	75	Interagir avec des réglettes, des alertes, des	
Le schéma Modèle-Vue-Contrôleur	78	contrôles segmentés et des sous-vues	173
Développer la vue et associer des actions aux		Ajouter une réglette	173
contrôles	80	Afficher une alerte à l'utilisateur	177
Le concept d'outlets et d'actions	81	Ajouter un contrôle segmenté et une sous-	
Associer les outlets et les actions au projet	84	vue	179
Dessiner l'interface et associer des		Intégrer une roulette à l'application	184
méthodes aux objets	91	Les roulettes de temps	185
Les événements associés à l'iPhone	96	Intégrer une roulette personnalisée	190
Les accesseurs et les mutateurs	98	Définir des éléments dépendants dans une liste	
La logique interne de l'Objective-C	98	de propriétés	199
Les propriétés en Objective-C	98	Pour aller plus loin	206
Le délégué de l'application	99	Chapitre 7. Lire et écrire des données	207
Finaliser notre projet de quiz	101	Lire des données	209
Pour aller plus loin	106	Créer une liste simple à partir de contenu	
Chapitre 5. Les vues de votre application	107	externe	209
Hiérarchiser les vues de votre application	109	Mettre en forme les cellules	215
Choisir un système de navigation	111	Réagir au contact de l'utilisateur	221
Organiser une application multivues	116	Créer un lecteur de flux RSS	223
La navigation par onglets	116	Enregistrer des données	231
La navigation par boutons	121	Sauvegarder les Préférences Système	232
Modifier l'animation de transition	127	Écrire un fichier	237
MultiConvertisseur, une application multivues		Pour aller plus loin	244
composée d'actions et d'outlets	131	Chapitre 8. Le multimédia	245
Le design des vues de votre application	143	Les images et les photographies	248
Les tendances du design d'applications	144	Afficher des images	248
Du papier à Photoshop : la refonte de		Adapter les dimensions de la vue	
MultiConvertisseur	147	d'image	253
Pour aller plus loin	150	Charger une image depuis une source	
Chapitre 6. Les contrôles spécifiques	153	externe	258
Interagir avec des contrôles spécifiques : la		Charger des images et des PDF avec la	
création d'un utilitaire	154	classe UIWebView	261
Le modèle Utility Application	154	Charger la photothèque de l'utilisateur	264
Le cahier des charges et l'ébauche		L'audio et la vidéo	275
préliminaire	158	Déclencher un son	275
La logique interne des contrôleurs	160	Lire une vidéo	280
L'habillage de l'interface et la liaison des		Le dessin et les animations	282
outlets et actions	169	Pour aller plus loin	286

Chapitre 9. Réagir aux gestes de l'utilisateur	287
Gérer la rotation de l'affichage	289
Contrôler la manipulation de l'écran	297
Déplacer un élément du bout des doigts	300
Gérer la navigation à partir de gestes	304
Interagir avec l'accéléromètre de l'iPhone	311
Pour aller plus loin	316
Chapitre 10. Publication et marketing	317
Adhérer au programme de développeur iPhone	318
Déployer une application iPhone	322
Créer un certificat de développeur	322
Associer des iPhones enregistrés	324
Créer un identifiant d'application	325
Créer un profil de provisionnement	326
Tester une application sur votre iPhone	327
Soumettre votre application à Apple	328
Populariser votre application	335
Pour aller plus loin	341
Index	343



Préface

"Design is not just what it looks like and feels like. Design is how it works."

Steve Jobs, 30 novembre 2003, *New York Times*

San Francisco, janvier 2007. Devant un parterre de journalistes largement acquis à sa cause, Steve Jobs a la voix empreinte d'un ton plus solennel qu'à l'accoutumée. En coulisses, l'équipe technique d'Apple pressent que la grand-messe de cette année marquera à coup sûr un virage spectaculaire dans l'industrie de l'informatique et des loisirs numériques, bien au-delà du simple sort réservé à la marque à la pomme. Une pointe d'excitation gagne les premiers rangs de la salle du MacWorld, alors que Steve Jobs entame son discours inaugural. "De temps à autre, un produit révolutionnaire apparaît sur le marché et change toute la donne. On peut s'estimer vraiment chanceux de travailler sur un seul produit de cette trempe au cours de sa carrière. Aujourd'hui, Apple s'apprête à lever le voile sur trois produits à l'importance aussi capitale. Trois produits en un : l'iPhone".

Vous connaissez la suite : la fièvre iPhone a gagné l'ensemble de la planète, en dépassant allègrement les prévisions les plus folles et en faisant exploser tous les compteurs. Par son ergonomie si intuitive, sa facilité d'accès, ses fonctionnalités riches et l'étendue des services qu'il offre, le mobile d'Apple a redéfini l'usage du téléphone. C'est un appareil de communication au sens large, qui vous permet d'interroger les services les plus modernes tout en profitant des fonctions téléphoniques de base. Mais c'est aussi un formidable terrain de jeu pour tout type de loisirs numériques, du visionnage de séquences vidéos à la lecture de votre audiOTHèque personnelle. Et un outil de productivité sans commune mesure avec les téléphones antérieurs. Inutile de poursuivre l'inventaire des services qu'il rend : l'iPhone

révolutionne l'usage du mobile et s'affirme comme la plateforme par excellence de l'utilisateur nomade.

Figure de proue de ces nouveaux usages aux innombrables aspects, l'App Store d'iTunes introduit un mode de distribution original. Sacralisant l'application comme la brique essentielle de tout usage mobile, la boutique en ligne diffuse directement les créations de ses propres utilisateurs. Modeste utilitaire rendant de fiers services au quotidien, jeu vidéo spectaculaire ou lecteur multimédia spécialisé dans un domaine : vous adaptez librement votre téléphone à vos usages, en lui greffant toutes les applications dont vous avez besoin. Une fonction manque à l'appel ? Développez-la vous-même et venez enrichir le catalogue commun ! Au-delà du plaisir de s'impliquer dans cet effort global, vous découvrez un nouveau moyen de rétribuer votre passion et vous pouvez ainsi prétendre à des revenus réguliers.

L'iPhone est bien cette hydre aux cent mille têtes. Cent mille applications qui exploitent chacune de ses possibilités dans leurs derniers retranchements. Cent mille idées originales, qui se complètent mutuellement et offrent un panorama d'une richesse incroyable. Et cent mille développeurs, à la passion et l'inventivité contagieuses, qui regorgent de créativité.

Cent mille développeurs... dont vous ! Grâce à cet ouvrage, vous pénétrez dans les coulisses de l'iPhone et vous découvrez toute la richesse de ses rouages. Décryptez son système, profitez de ses contrôles originaux et modernes et réinventez à votre tour l'usage mobile. De l'écran multi-touch à l'accéléromètre en passant par l'accès aux services web ou l'utilisation du capteur photo/vidéo, vous disposez d'un arsenal très riche de fonctions qui laissent libre cours à tous vos projets. Apple vous offre un kit de développement complet qui, une fois maîtrisé, donne naissance à d'innombrables applications aux multiples talents.

Si le développement web a profondément marqué le monde de l'informatique depuis les années 90, la programmation pour plateformes mobiles en prend naturellement la succession et complète son approche. Prenez part à votre tour à cette révolution et figurez parmi ces pionniers qui redéfinissent nos usages quotidiens. Ludique et intuitif, le développement pour iPhone est accessible à tous et vous mènera vers des terres encore inexplorées.

Et si vous commenciez dès aujourd'hui ?

Organisation de cet ouvrage

Cet ouvrage se compose de 10 chapitres, qui vous offrent un panorama complet du développement pour iPhone et iPod Touch. De la création de vos premiers projets à leur publication sur iTunes, en passant par l'accès à des médias externes ou l'intégration de vidéos et de fichiers audios à vos applications, vous avez toutes les clés en main pour devenir un développeur professionnel et briller sur l'App Store.

- Le Chapitre 1, *Le SDK de l'iPhone et de l'iPod Touch*, présente l'étendue de vos possibilités. Vous y apprendrez à installer tous les éléments nécessaires pour amorcer la programmation. Vous travaillerez sur papier en amont, afin de poser toutes vos idées initiales et d'entamer le cycle de développement dans les meilleures conditions, en réfléchissant au cadre et à l'intérêt de vos applications.
- Le Chapitre 2, *Vue d'ensemble du développement pour iPhone*, pose les bases de votre environnement de programmation. Vous découvrirez par le détail l'ensemble des outils qui constituent l'arsenal du développeur, en particulier Xcode et Interface Builder. Les principales règles et syntaxes de l'Objective-C n'auront plus de secrets pour vous !
- Le Chapitre 3, *À la découverte du SDK de l'iPhone*, vous accompagne dans vos premiers pas sous Xcode et Interface Builder. Vous y apprendrez à créer votre premier projet, à utiliser le Simulateur d'iPhone et à profiter des modèles prédéfinis pour offrir un cadre idéal à l'ensemble de vos idées d'applications.
- Le Chapitre 4, *Interactions avec vos applications*, expose l'ensemble des solutions à votre disposition pour interagir avec l'utilisateur de votre application. Création de boutons déclenchant des fonctions, modifications de l'interface de votre application, gestion des événements... Cet examen en profondeur du dialogue qui s'établit entre votre application et ses utilisateurs s'achève par un exemple concret : un quiz auquel on répond par de simples pressions du bout des doigts.
- Le Chapitre 5, *Les vues de votre application*, vous présente tous les outils permettant de définir les interfaces de vos applications. Vous y apprendrez à créer des applications riches s'articulant autour de systèmes de navigation complets qui donnent accès à de multiples écrans. Onglets, barres d'outils et séries de boutons : hiérarchisez le contenu de votre application et soignez les transitions entre chaque écran à l'aide d'effets d'animation. Vous y apprendrez également tout l'art du design d'interfaces pour iPhone, en travaillant minutieusement les écrans de votre application du papier à Photoshop.
- Le Chapitre 6, *Les contrôles spécifiques*, s'attarde sur tous les éléments d'interface plus complexes, comme les réglettes, les interrupteurs ou les roulettes à manipuler du bout des doigts. Vous y apprendrez à bâtir des projets plus complexes, du cahier des charges à l'implémentation de tous les contrôleurs et des éléments d'interface, en tirant partie de l'ensemble des composants exclusifs du mobile d'Apple. Vos applications gagneront ainsi en ergonomie et en réactivité.
- Le Chapitre 7, *Lire et écrire des données*, passe en revue l'ensemble des solutions permettant d'interagir avec du contenu externe. Vous préparerez ainsi le contenu de votre application dans un fichier annexe, facile à mettre à jour, ou vous le récupérerez directement à travers la connexion WiFi ou 3G de l'iPhone. Création d'un lecteur de flux RSS, sauvegarde des paramètres spécifiques à une application dans les réglages de l'iPhone,

enregistrement de champs saisis par l'utilisateur... Enrichissez votre application et laissez l'utilisateur l'adapter à ses besoins ! Les tableaux et listes d'éléments, à manipuler au doigt, constituent un socle idéal pour héberger les informations liées à votre application.

- Le Chapitre 8, *Le multimédia*, évoque toutes les possibilités les plus avancées de l'iPhone et de l'iPod Touch. Affichage et retouche de photos personnelles, utilisation du capteur photo/vidéo de l'iPhone, lecture de fichiers audio ou de séquences vidéo : enrichissez vos applications d'un contenu multimédia très divers ! En marge de ces solutions, vous apprendrez également à dessiner sur l'écran de l'iPhone à l'aide de fonctions avancées du kit de développement.
- Le Chapitre 9, *Réagir aux gestes de l'utilisateur*, vous livre tous les secrets des capteurs intégrés à l'iPhone et à l'iPod Touch. Détection du mouvement, gestion de l'écran multi-touch, réaction aux gestes de l'utilisateur, prise en compte de l'inclinaison et de l'accéléromètre du mobile... Composez des applications au comportement riche et intuitif, afin d'accompagner l'utilisateur dans tout type de situations.
- Le Chapitre 10, *Publication et marketing*, vous assiste dans l'ultime étape du développement pour iPhone : la commercialisation de tous vos projets sur l'App Store d'iTunes. Vous y apprendrez à investir dans un compte de développeur professionnel grâce auquel vous soumettrez vos applications finalisées à Apple. Soignez la demande de validation et mettez tous les atouts de votre côté afin de rencontrer l'adhésion du public.

Prérequis

Nous avons conçu cet ouvrage afin de le rendre accessible au plus grand nombre. Vous ne devez pas nécessairement disposer d'une expérience antérieure en programmation, ni connaître les rouages du développement orienté objet ou de l'instanciation de classes. Toutefois, le kit de développement de l'iPhone et le langage Objective-C s'inscrivent dans une longue tradition d'outils et puisent leurs racines dans un socle commun à de nombreux langages. Les programmeurs Java, par exemple, reconnaîtront rapidement les principaux traits de leur langage de prédilection et pourront envisager des projets plus complexes. Les développeurs web profiteront de leur expérience et de leur maîtrise des outils graphiques pour entamer le design d'interfaces ambitieuses et positionner au mieux les objets avec lesquels les utilisateurs interagissent. Il n'est pas non plus inutile de disposer d'une solide expérience en PHP si vous envisagez de lier votre application à un service web ou à l'une des nombreuses API disponibles sur la Toile (YouTube, Flickr, Google Maps, etc.).

À l'inverse, vos connaissances en Objective-C vous seront d'une grande aide pour envisager de vous recycler vers d'autres langages ou plateformes. Le développement pour iPhone et

Mac OS X s'articule autour d'une même base et de la même panoplie d'outils. Vous pourrez ainsi naturellement poursuivre l'expérience vers la création de vos propres applications bureautiques.

Conventions typographiques

Cet ouvrage utilise des polices de caractères différentes pour distinguer certains éléments :

- **Police fixe.** Cette police met en valeur les éléments du langage ainsi que les noms de fichiers ou de répertoires.
- **Police fixe grasse.** Dans les extraits de code, cette police permet de mettre en exergue les modifications apportées aux programmes.
- Les noms d'applications ou les termes mis en exergue sont en *italique*.

Les encadrés

Ils précisent des éléments destinés à attirer votre attention sur un cas particulier.

Remerciements

Ce livre n'aurait pas été possible sans le soutien indéfectible, le savoir-faire et les multiples talents de Dominique Buraud. Son regard professionnel, son attention constante et sa passion, de la première ébauche de cet ouvrage à sa réalisation, auront éclairé l'ensemble de ce projet. Nombreux sont les auteurs français à lui devoir une fière chandelle pour avoir concrétisé leurs souhaits, en œuvrant pour la défense d'une littérature technique de grande qualité dans notre langue. L'auteur est fier de pouvoir aujourd'hui rejoindre leurs rangs.

Ce livre n'aurait pas non plus été possible sans l'aide précieuse et fidèle de Fabien Schwob, qui a consciencieusement relu chacune des pages et les a éclairées de son savoir-faire technique. Si le développement Cocoa vous intéresse, n'hésitez pas à suivre son site web passionnant dédié à cet environnement, à l'adresse <http://www.cocoa.fr/>.

Enfin, les remerciements de l'auteur vont naturellement à toute l'équipe de Pearson France, en particulier Patricia Moncorgé. Leur passion et leur talent ont rendu cet ouvrage possible !

À propos de l'auteur

Jean-Marc Delprato est journaliste technique spécialisé dans la presse informatique depuis 1991. Après un DEA de Lettres Modernes à la Sorbonne et des études d'ingénieur informatique, il collabore à de nombreux magazines (*Player One*, *Code(r)*, *PC Team*, *Windows News*, *Computer Arts*, *Joystick...*) et a traduit ou rédigé une vingtaine d'ouvrages techniques. Passionné de webdesign et d'infographie, il s'intéresse notamment à PHP et à l'architecture de l'information. Avec l'apparition de nouvelles plateformes de développement, il a conçu des applications pour de multiples services web et API renommées. Vous trouverez notamment ses travaux à travers ses ouvrages *Intégrer Google Maps à une application Ajax*, *Créer des scripts pour Second Life* et *Créer des applications pour Facebook*. L'iPhone lui est apparu comme une extension naturelle de chacune de ces plateformes et comme un terrain de jeu idéal pour déployer des interfaces riches et interactives autour d'un contenu original.



Le SDK de l'iPhone et de l'iPod Touch

Au sommaire de ce chapitre

- Les éléments à réunir
- Installer le SDK
- Avant d'aller plus loin : préparer votre projet

Ce n'est pas un hasard si le catalogue de l'iPhone comprend aujourd'hui plus de 100 000 applications différentes ; à l'image de son terminal mobile si intuitif et ergonomique, Apple a mis en place un kit de développement (SDK, *Software Development Kit*) puissant et efficace qui offre un formidable cadre aux programmeurs du monde entier. Quel que soit votre

niveau d'expérience, vous disposez d'un environnement riche et accessible, à travers lequel vous mettrez en forme vos projets les plus ambitieux. Votre périple à travers le développement d'applications pour iPhone commence donc irrémédiablement par son téléchargement et sa prise en main.

Tout au long de cet ouvrage, nous utiliserons les trois outils principaux qui composent un tel kit de développement : *Xcode*, l'interface où vous programmerez vos applications, *Interface Builder*, l'outil de design et de conception et le *Simulateur d'iPhone* pour tester vos programmes. Retrouvez-vous les manches : votre carrière de développeur pour iPhone et iPod Touch commence dès aujourd'hui !

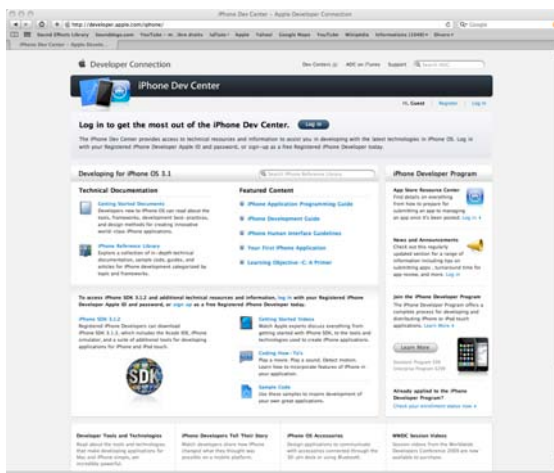
Les éléments à réunir

Pour profiter du SDK distribué gratuitement par Apple, vous devez disposer d'un Macintosh doté d'un processeur Intel et de Mac OS X 10.5.3 ou ultérieur. Les récents iMac 27 pouces ou les MacBook Pro exécutant Snow Leopard s'acquittent parfaitement de cette tâche et offrent un espace de travail très confortable pour tirer parti de tels outils. Si vous n'êtes pas encore équipé et si votre budget est plus limité, vous pouvez également envisager l'achat d'un Mac mini – vous offrirez alors une seconde jeunesse à un vieux moniteur.

Ouvrez votre navigateur web et rendez-vous à l'adresse <http://developer.apple.com/iphone>. Vous découvrirez le Centre de développement pour iPhone d'Apple. Cet espace essentiel décrit les principales techniques de développement et illustre même certaines procédures à travers une série de vidéos. N'hésitez pas à y revenir régulièrement pour découvrir les dernières nouveautés et pour vous entretenir avec d'autres développeurs aguerris. Sachez toutefois que l'ensemble des documents est en anglais.

Figure 1.1

La page d'accueil du Centre de développement pour iPhone.



Cliquez sur le bouton Log in, au centre de l'écran, puis saisissez votre identifiant Apple ID. Vous l'utilisez probablement déjà pour vous connecter à l'App Store d'iTunes afin d'acheter de nouvelles applications. Si vous ne disposez pas encore de ce précieux sésame, cliquez sur le lien Join now et remplissez le formulaire afin d'ouvrir un compte. Ce n'est d'ailleurs pas une mauvaise idée de créer un second identifiant Apple ID, réservé au SDK et à votre compte de développeur.

Figure 1.2

Authentifiez-vous à l'aide de votre identifiant Apple ID ou créez-en un nouveau.



Figure 1.3

Vous vous apprêtez à télécharger le kit de développement, cliquez sur le bouton Continue.

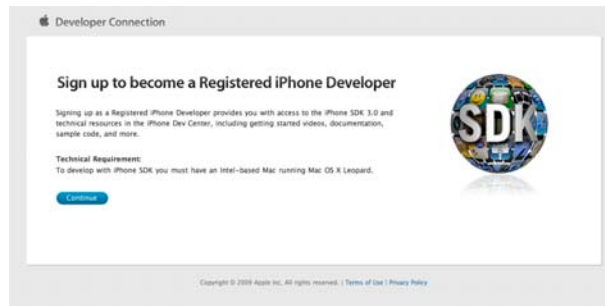
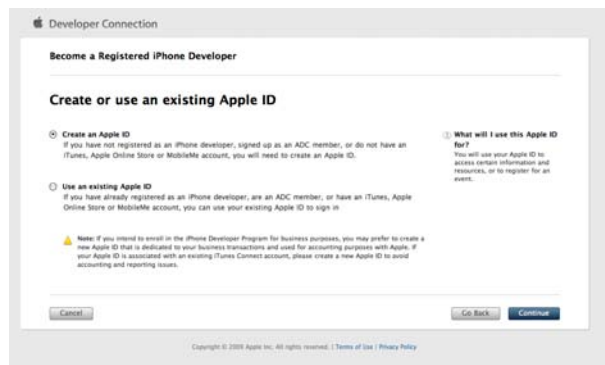


Figure 1.4

Vous ne disposez pas encore de compte ? Créez-en un gratuitement en quelques secondes.



Si vous envisagez de souscrire à une licence payante afin de commercialiser vos propres applications, vous avez tout intérêt à prendre le temps de soigner les détails qui vous concernent. Remplissez le formulaire complet, en indiquant votre nom, votre adresse e-mail et votre pays d'origine.

Figure 1.5

Remplissez le formulaire d'inscription en détaillant vos informations personnelles.

The screenshot shows the 'Developer Connection' page for becoming a registered iPhone developer. The progress bar indicates four steps: Sign in, Account info (current), Agree to license, and Email verification. The main heading is 'Create an Apple ID and complete your personal profile' with a note that all form fields are required. The form is divided into three sections: 'Create Apple ID' with fields for Desired Apple ID, Password (30 characters), and Re-enter Password (30 characters); 'Security Information' with fields for Birthdate (month and day), Security Question, and Answer; and 'Personal Information' with fields for First Name, Last Name, Email address, Confirm Email, Company / Organization, and Country (a dropdown menu). There are also informational links for Apple ID, Security Information, and Your contact info.

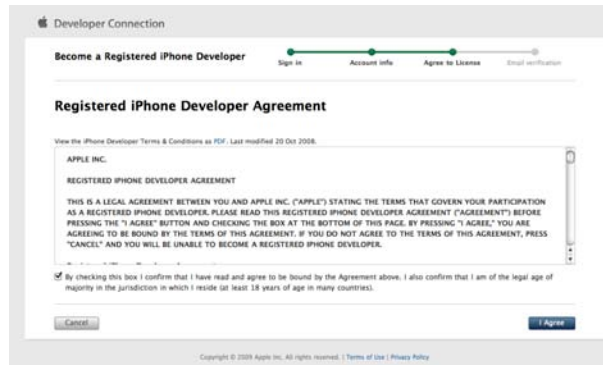
Figure 1.6

Apple vous demande d'emblée le type d'applications que vous êtes susceptible de programmer.

The screenshot shows the 'Complete your professional profile' section of the registration form. It includes a note that all form fields are required. The first section, 'What is your primary market?', features a grid of radio button options: Business, Education, Entertainment, Finance, Games, Health & Fitness, Lifestyle, Medical, Music, Navigation, News, Photography, Productivity, Reference, Social Networking, Sports, Travel, Utilities, and Weather. The second section, 'What type of application(s) do you plan on developing for iPhone and iPod touch?', has an identical grid of radio button options. The third section, 'Please select the primary category for your application(s)', includes radio buttons for Free Applications, Commercial Applications, and Enterprise (In-House) Applications. The final section, 'How many years have you been developing on Apple platforms?', has radio buttons for New to Apple platforms and < 1 year.

Figure 1.7

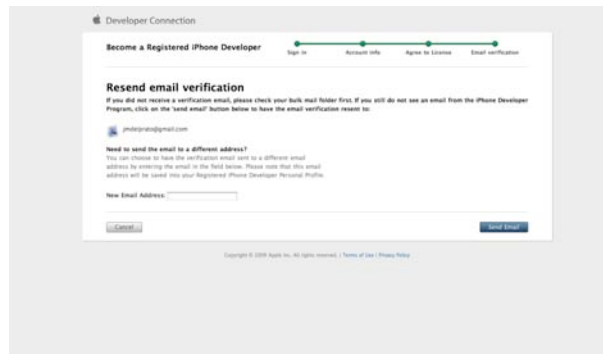
Validez les conditions générales d'utilisation en cliquant sur le bouton I Agree.



Quelques secondes plus tard, vous recevez un e-mail : suivez le lien qu'il contient puis saisissez le code à cinq chiffres dans le champ prévu à cet effet. Votre compte est enfin prêt et vous accédez à nouveau au Centre de développement d'Apple.

Figure 1.8

Surveillez vos e-mails : vous devez encore valider votre compte.

**Figure 1.9**

Saisissez le code de vérification pour finaliser votre compte.

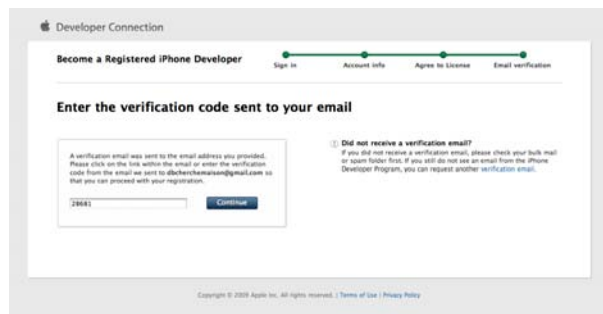
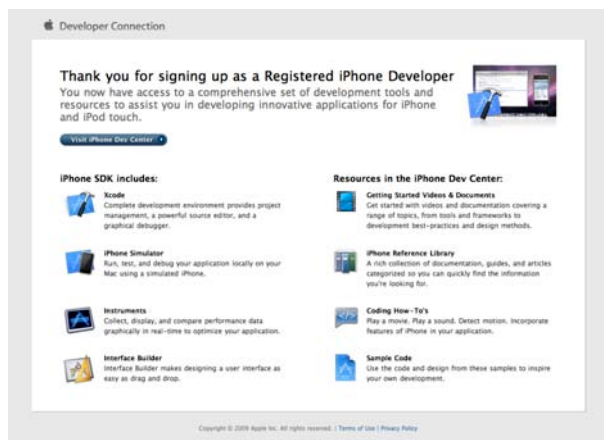


Figure 1.10

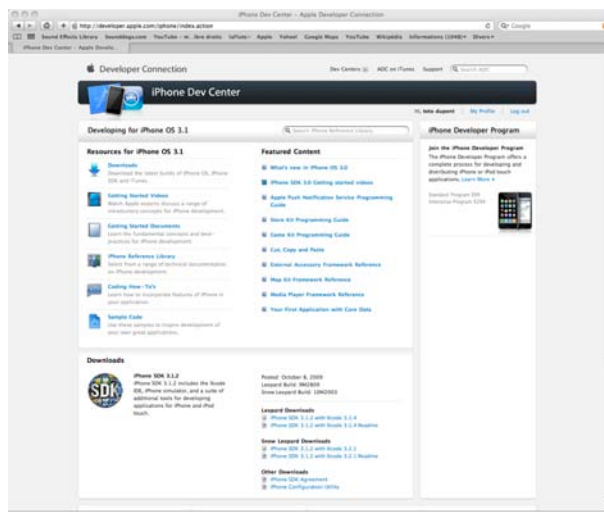
De retour au Centre de développement d'Apple, vous découvrez les différents éléments qui composent le SDK.



Sur la page qui apparaît à l'écran, choisissez la version du SDK qui correspond à votre système d'exploitation. Vous constatez en effet qu'il existe des déclinaisons spécifiques à Leopard ou Snow Leopard. Armez-vous de patience : à l'heure où nous mettons ce livre sous presse, la dernière version en date avoisine les 2,73 Go.

Figure 1.11

Sélectionnez la version du SDK qui correspond à votre système d'exploitation, Leopard ou Snow Leopard.



Signer un contrat de développeur auprès d'Apple

Si le téléchargement du kit de développement est purement gratuit, vous découvrez deux formules d'abonnement sur la colonne de droite du Centre d'Apple. Pour 99 dollars, vous pouvez en effet opter pour une véritable licence de développeur. Celle-ci vous permettra de déposer vos projets sur votre propre iPhone et de les commercialiser à travers l'App

Store d'iTunes. La licence d'entreprise, proposée à 299 dollars, concerne essentiellement les sociétés disposant de plus de 500 employés qui envisagent de développer des applications en interne, pour leurs propres besoins. Si l'achat d'une licence n'est absolument pas impératif pour suivre tous les exemples de cet ouvrage, vous devriez l'envisager malgré tout ; c'est l'unique moyen de rentabiliser votre apprentissage et de commercialiser vos propres applications. Nous reviendrons largement sur cette souscription et sur la vente d'applications à travers le Chapitre 10.

Installer le SDK

Vous disposez désormais d'un fichier "iphone_sdk_3.x.x_with_xcode_3.1.x.dmg" sur votre disque dur. Effectuez un double-clic sur son icône afin de monter cette image. Dans la fenêtre qui apparaît à l'écran, cliquez sur l'icône "iPhone SDK".

Figure 1.12

Le téléchargement est terminé, procédez à l'installation du SDK.

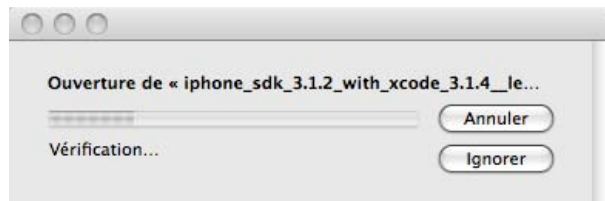


Figure 1.13

L'installation du SDK greffe tous les outils nécessaires à Mac OS X.



Validez l'installation du kit de développement, puis cliquez sur le bouton Continuer à l'ouverture du programme (voir Figures 1.14 et 1.15).

Acceptez ensuite les multiples conditions d'utilisation, puis sélectionnez les versions du SDK que vous souhaitez installer. Pour suivre les exemples de cet ouvrage, vous pouvez vous contenter des options par défaut. Toutefois, si vous envisagez de développer pour une version plus spécifique de l'iPhone OS, vérifiez que vous l'avez bien sélectionnée dans la

liste figurant à l'écran. Le Simulateur d'iPhone disposera ainsi de l'image appropriée du système (voir Figures 1.16 et 1.17).

Figure 1.14

Cliquez une première fois sur le bouton Continuer pour lancer l'installation du SDK.

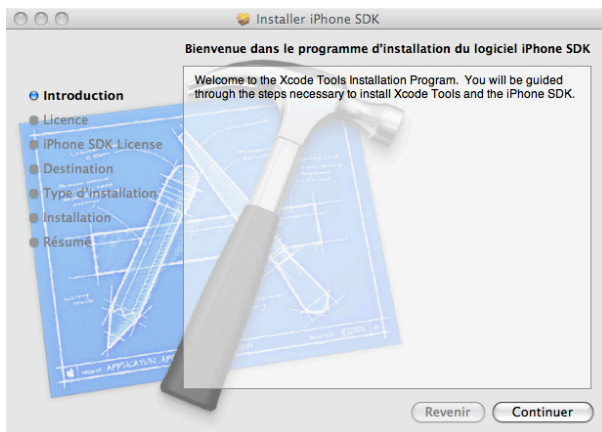


Figure 1.15

Tout est prêt, procédez à l'installation de Xcode et de ses différents outils.

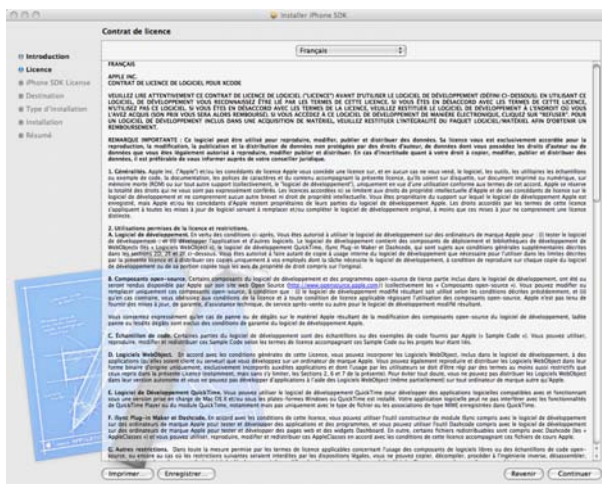


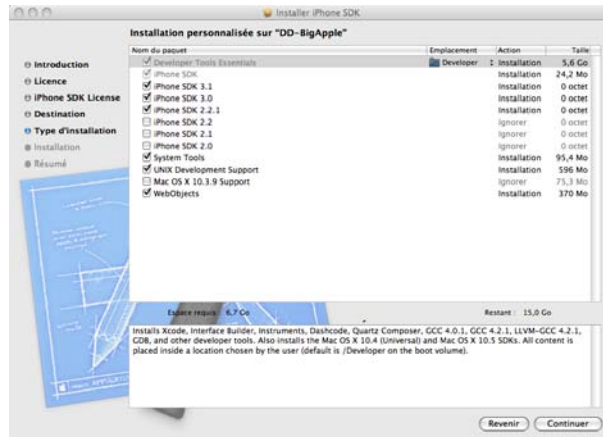
Figure 1.16

Acceptez les conditions d'utilisation.



Figure 1.17

Sélectionnez la version du SDK appropriée puis cliquez sur Continuer.



Par défaut, les outils du SDK s'installent dans le dossier /Developer de votre disque dur. Vous devez disposer de 7,3 Go d'espace libre pour profiter de l'ensemble des outils. Modifiez éventuellement le répertoire d'installation, puis cliquez sur Choisir.

Figure 1.18

Choisissez le répertoire d'installation et validez.

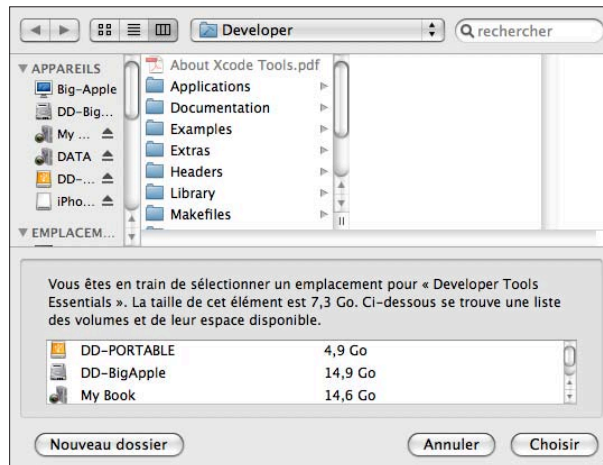


Figure 1.19

Cliquez enfin sur le bouton Installer pour lancer la procédure.

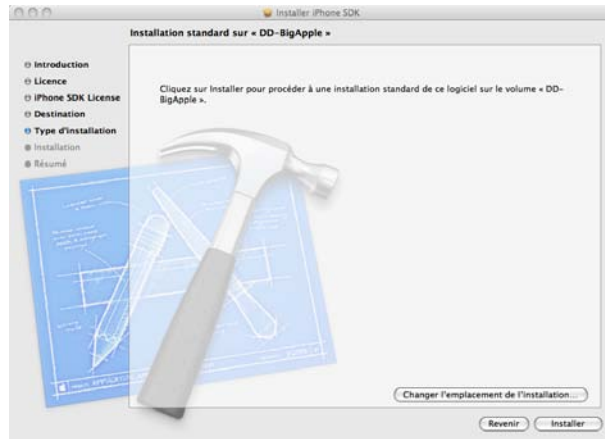


Figure 1.20

Patiencez : dans quelques secondes, tout sera prêt !

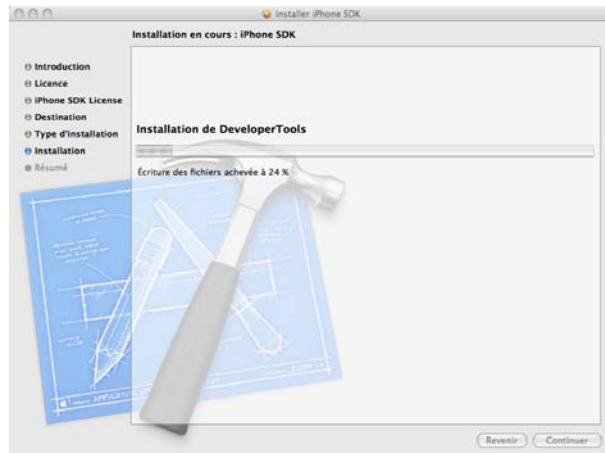
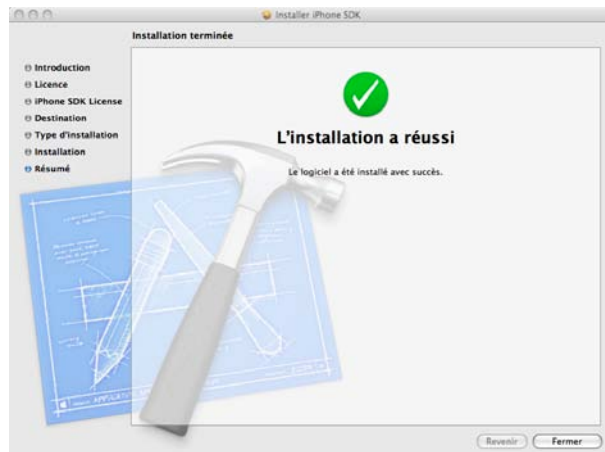


Figure 1.21

Voilà, vous disposez de l'ensemble des éléments du SDK.



Quelques minutes plus tard, le SDK est installé sur votre Mac. Déposez éventuellement Xcode dans votre Dock ; vous en ferez une utilisation constante à travers tous les chapitres de cet ouvrage.

Les autres solutions pour développer

Malgré le cadre ludique et convivial offert par le SDK, de nombreux développeurs ont cherché à s'en affranchir et à explorer d'autres pistes pour programmer des applications pour iPhone. Parmi de telles solutions alternatives, les frameworks *QuickConnect* et *Phone-Gap* présentent un grand intérêt. Ils s'adressent en priorité aux développeurs web et reposent essentiellement sur du code JavaScript, HTML et CSS pour designer les interfaces de vos applications et interagir avec l'utilisateur. Vous pouvez ainsi exploiter la quasi-totalité des contrôleurs spécifiques de l'iPhone sans apprendre l'Objective-C pour autant. Nous n'aborderons pas ces techniques de programmation alternatives au cours de cet ouvrage et nous vous invitons à vous reporter au livre *Développez des applications pour iPhone avec HTML, CSS et JavaScript* de Lee S. Barney, paru aux éditions Pearson. Sachez toutefois que de telles pratiques commencent à se répandre et qu'elles constituent une solution de choix pour tous les développeurs sous Windows.

Avant d'aller plus loin : préparer votre projet

Vous avez désormais réuni tous les éléments nécessaires au développement de votre première application pour l'iPhone. Vous avez également pris le temps de personnaliser votre environnement de développement afin d'acquérir rapidement de nouveaux réflexes. Comme nous le verrons par la suite, vous basculerez sans cesse entre les différents outils du SDK : vous devez ainsi vous familiariser avec leur interface pour en exploiter toute la puissance. Même si Apple a conduit un réel effort pour proposer des outils riches et clairs, vous devez impérativement les apprivoiser et ne pas vous laisser déborder par ces nouvelles possibilités !

Pour l'heure, nous vous recommandons ainsi de ne pas foncer tête baissée dans cet environnement nouveau ; à l'instar de n'importe quel projet de développement, la programmation d'une application pour iPhone requiert une solide phase de préparation. N'oubliez jamais que, quelle que soit son échelle, votre application entretient des rapports privilégiés avec ses utilisateurs – non seulement tous ceux qui la jugeront utile l'auront sans cesse à portée de main ou dans le creux de leur poche, mais ils interagiront aussi directement au doigt sur les contrôles que vous avez prévus. Peu de logiciels bénéficient d'une telle immersion dans l'univers immédiat de leurs adeptes ! Sans pour autant jeter sur le papier des extraits de codes sources ou de scripts, vous devez impérativement passer par une phase initiale de design. La conception de l'interface et des interactions de votre application joue un grand

rôle sur le développement à proprement parler : sans le niveler par le bas et l'enclaver de contraintes, cette première approche va au contraire révéler l'étendue de vos possibilités et préparer la phase de développement. L'iPhone requiert une approche très naturelle et instinctive : il est difficile de concevoir des interactions riches et ergonomiques en se lançant directement dans la programmation.

Vous êtes, par exemple, motivé à l'idée de programmer une application de recettes de cuisine ? Songez que les utilisateurs n'auront peut-être pas toujours les deux mains libres pour manipuler votre logiciel. Vous devez ainsi prévoir une solution pour naviguer rapidement et intuitivement à travers vos menus et vos recettes, sans multiplier les gestes pour autant. Mettez-vous dans la peau de vos utilisateurs et revenez, l'espace d'un instant, vers deux outils qui valent tout à fait le SDK d'Apple : le papier et le crayon !

Établir un cahier des charges sommaire

Du point de vue du développeur, le mobile tactile d'Apple offre un formidable terrain de jeu au potentiel quasi infini. Pour vous en convaincre, prêtez-vous à une petite expérience. Réunissez plusieurs amis qui utilisent un téléphone portable traditionnel. Présentez-leur une première série d'applications que vous avez glanées sur l'App Store, dont une poignée d'utilitaires et de jeux. Faites-leur ensuite part de votre projet de développer à votre tour des applications. Les idées les plus originales vont automatiquement fuser dans l'assemblée, des interfaces les plus loufoques et novatrices jusqu'aux ambitieux projets censés rendre de colossaux services au quotidien. C'est une source inestimable d'idées.

L'iPhone et l'iPod Touch sont deux terminaux très intuitifs, qui deviennent rapidement des extensions naturelles du bon vieux carnet et de l'agenda. Grâce à leur interface tactile, à leurs possibilités de communication (WiFi et 3G) et à l'accéléromètre qu'ils embarquent, on envisage immédiatement des déclinaisons infinies de tous les logiciels auxquels nous sommes habitués. Là encore, faites preuve de patience ! En revenant au papier et au crayon et en préparant votre projet en amont, vous ne devez pas partir dans tous les sens pour autant. Ne bridez pas votre imagination, mais concentrez-vous sur une ou deux idées principales pour bien débuter. Organisez une séance de brainstorming au cours de laquelle vous relèverez une série de pistes : imaginez avant tout les applications qui manquent encore à l'appel, ou celles que vous rêvez de perfectionner. Lorsque la silhouette générale de votre projet vous apparaît, passez à la phase de design. Esquissez les différents écrans que parcourront les utilisateurs et imaginez un système de navigation. Clarifiez le contenu que vous souhaitez leur présenter, puis dressez la liste complète de tous les éléments qui constitueront votre application. Vous bâtissez ainsi rapidement un cahier des charges, dont vous pourrez éventuellement vous écarter par la suite si une meilleure idée ou un système de navigation plus ergonomique vous venaient durant la programmation à proprement parler !

Quel que soit votre niveau en programmation, cette phase initiale est incontournable. Au fur et à mesure de vos progrès, l'expérience vous apprendra à viser plus juste et à ne retenir que les éléments réellement impératifs pour mener à bien le développement. Mais gardez cet adage à l'esprit : si votre application tient la route sur le papier et se voit soutenue par de solides fondations, vous avez déjà accompli la moitié du chemin.

Les types d'applications de l'iPhone : faites jaillir toutes sortes d'idées !

Vous rêvez de rencontrer le succès sur l'App Store d'iTunes et de voir votre application intronisée sur sa page d'accueil ? Vous n'avez pas encore d'idée bien précise et vous souhaitez vous inspirer de projets auréolés de prestige ? Commencez votre phase de conception initiale par une petite visite sur le centre de téléchargement d'Apple ! Vous y puiserez systématiquement de nouvelles idées et vous saurez rapidement où se situe votre projet vis-à-vis de la "concurrence". La communauté des utilisateurs a-t-elle réellement besoin d'une millième version de la calculatrice, par exemple ? Ou bien votre projet est-il suffisamment innovant et ergonomique pour mériter de figurer parmi toutes ces déclinaisons d'ores et déjà disponibles ? Vous serez rapidement en mesure d'estimer le succès potentiel de votre propre projet !

Voici les différentes catégories d'applications pour iPhone et iPod Touch, triées par popularité. En les envisageant sous l'aspect de leur *programmation* et non de leur *utilisation*, leur parcours fera probablement jaillir d'innombrables idées !

Divertissement / Jeux. Indéniablement la catégorie la plus populaire de l'App Store, les jeux figurent aussi parmi les applications les plus onéreuses. Elles présentent donc un fort rendement potentiel ! On distingue essentiellement trois types de jeux : les adaptations de grands classiques, menées par de prestigieux studios (*Resident Evil*, *Crash Bandicoot*, *Need for Speed*, *Myst*...), les jeux de physique qui tirent partie de l'accéléromètre de l'iPhone (*Topple*, *Labyrinth*...) et les petits jeux indépendants, conçus autour d'un principe nouveau et accrocheur (casse-têtes, énigmes, puzzles...). À moins de tomber dans la dernière catégorie, votre projet ne devra souffrir d'aucune faiblesse sur le plan visuel et sonore. Certains studios indépendants se sont même constitués en vue de n'offrir que des jeux pour l'iPhone : c'est notamment le cas des français d'Agharta Studio (*1112 Episode 01*), qui reprennent les principes fondateurs du jeu vidéo avec ses corps de métiers soigneusement cloisonnés. S'il restera toujours de la place pour des projets de moindre ampleur menés par des programmeurs indépendants, le développement d'un jeu pour l'iPhone est une activité ambitieuse : comptez plusieurs semaines ou mois de programmation. C'est toutefois l'un des secteurs les plus lucratifs et pour lequel le bouche-à-oreille fonctionne le mieux, comme en témoigne l'aventure d'Ethan Nicholas le développeur du jeu *iShoot*. Vendu 2,99 dollars

et décliné dans une version gratuite, son titre s'est rapidement arraché sur l'App Store au point de faire de son auteur l'un des premiers millionnaires d'iTunes ! Sans prétendre à une telle audience, votre application sera forcément sous les feux des projecteurs si vous l'introduisez dans cette catégorie, consultée quotidiennement par des centaines de milliers d'utilisateurs.

Utilitaires/Productivité. Deuxième catégorie la plus populaire, les utilitaires constituent l'approche la plus originale de l'iPhone et la plus saluée par Apple. La plupart de ces applications tirent parti des outils exclusifs au terminal d'Apple, en particulier son interface tactile et son accéléromètre. C'est aussi une catégorie qui tolère les applications minimalistes et moins visuelles, si elles remplissent leur rôle et offrent de véritables services au quotidien ! C'est d'ailleurs le seul critère à l'origine de leur succès : si vous développez un utilitaire pour iPhone, vous ne devez jamais formuler des promesses sans les tenir. Combien d'applications se targuent de remplacer votre dictaphone ou votre boîte à outils avant de se révéler de bien piètres compagnons ? N'oubliez pas que si vous fixez un prix de départ à votre projet, les utilisateurs ne seront pas en mesure de l'essayer sans saisir leur numéro de carte bancaire : les attentes vis-à-vis de vos promesses n'en sont que décuplées. Nous y reviendrons, mais il est très intéressant de décliner votre projet en une version gratuite et une version payante si vous envisagez de développer un utilitaire. Parmi les utilitaires les plus populaires, on distingue essentiellement les outils de type "couteau-suisse" (niveau à bulle, altimètre, télécommande universelle, lampe de poche...), les calculatrices ou utilitaires spécialisés (conversion, finances, accordeur de guitare, etc.) et les adaptations d'outils de bureautique traditionnels (transformation de l'appareil photo de l'iPhone en webcam, dictaphone numérique, radio-réveil...).

Actualités. Généralement constellée d'applications gratuites, cette catégorie constitue avant tout une "vitrine" pour de nombreux sites d'informations qui prétendent ainsi à une plus large audience en s'invitant sur le téléphone de leurs utilisateurs réguliers ! Prenez l'application du *Monde*, par exemple. Elle n'offre pas un contenu fondamentalement différent du portail de ce quotidien, mais elle présente une ergonomie savamment étudiée, qui en fait un meilleur outil de consultation que le navigateur intégré à l'iPhone. Avec son "zapping" et ses images qui apparaissent en plein écran, on accède bien plus rapidement à l'information qu'à travers le site web officiel du journal. De très nombreux sites web ou médias traditionnels déclinent actuellement leur contenu à l'iPhone. Si vous possédez un blog WordPress, vous avez probablement déjà remarqué le plug-in permettant de transformer l'affichage de votre site afin qu'il épouse les dimensions de l'iPhone. Il s'agit ici d'offrir une expérience plus riche encore à vos lecteurs : votre source est clairement identifiée à travers sa propre icône et bénéficie d'une consultation autonome, au même titre que le parcours des derniers e-mails ou SMS. C'est un moyen idéal d'accompagner votre propre blog et de renforcer sa visibilité et sa popularité ! Comme vous pouvez l'imaginer, ce type d'applications tire

essentiellement parti des fonctions de communication de l'iPhone : vous profitez de votre flux RSS pour rapatrier automatiquement vos derniers billets, que vous mettez en forme à l'aide du SDK. Par ailleurs, c'est également l'une des catégories que vous devez le mieux maîtriser si vous souhaitez devenir un développeur iPhone professionnel : de nombreuses sociétés sont actuellement à la recherche de programmeurs capables d'adapter leur contenu à ce terminal et vous pouvez envisager de répondre à ces missions.

Photographie/Musique. Même si l'appareil photo intégré à l'iPhone ne figure pas parmi les plus performants du marché, de très nombreuses applications permettent d'en améliorer le rendu et de retoucher rapidement ses photos. Succès garanti ! C'est notamment le cas de *PhotoEffect*, une application que l'on doit au développeur japonais Kyosuke Takahashi. À travers une série de plug-ins et de filtres inspirés de Photoshop, les utilisateurs modifient les clichés qu'ils ont pris en flagrant déclin dans leur photothèque. L'application *Photo Collage* emprunte le même chemin : elle réunit vos plus belles photos autour de compositions originales, calculées automatiquement. Attention toutefois : le développement de filtres visant à modifier des images n'est pas aisé et ne devrait pas constituer votre première approche du SDK de l'iPhone. Retenez toutefois qu'il est possible d'interagir à travers une application avec des fichiers stockés sur la mémoire Flash de votre mobile et de charger/enregistrer des données. Dans un premier temps, vous pouvez vous contenter de proposer une série de cadres et de bordures que les utilisateurs apposeront sur leurs photos préférées. Si le principe vous paraît minimaliste, sachez qu'il existe un réel public pour ce type d'applications simples dont le seul but est d'étendre les possibilités offertes par l'iPhone et de mettre en scène la vie de tous les jours.

L'iPhone et l'iPod Touch ne sont pas en reste du côté de la musique, baladeur numérique oblige ! Là encore, il s'agit d'applications très ambitieuses qui se déclinent en des synthétiseurs ou des séquenceurs. Plus précisément, on distingue deux types d'applications musicales : les "initiations" à un instrument, que l'on manipule aux doigts (*MiniPiano*, *PocketGuitar...*), et les applications de création qui enregistrent de véritables fichiers. La première sous-catégorie partage de nombreux titres avec les applications de jeux et de divertissement : les applications sont ludiques et ne prétendent pas véritablement à la rigueur d'un conservatoire de musique ! L'autre sous-catégorie ne contient que des applications très ambitieuses, qui tirent avantage de toutes les fonctions les plus avancées du SDK : chargement de données, enregistrement de fichiers, interface ergonomique... N'ayez crainte : vous serez en mesure de développer de tels utilitaires à l'issue de la lecture de cet ouvrage.

Navigation. Dans sa dernière version en date, l'iPhone intègre un GPS performant. De nombreuses applications s'appuient sur cet outil et sur une variété de services en ligne afin de cartographier vos déplacements ou de préparer vos trajets. C'est l'un des domaines où le développement web et la programmation pour iPhone épousent la même direction : vous tirez généralement partie d'API très bien référencées sur Internet (Google Maps, analyse

du trafic en temps réel, flux RSS contenant la liste de stations Vélib par exemple...) et vous déclinez ainsi des services en ligne au terminal d'Apple. Prenez garde cependant : ce type d'applications ne tolère pas les inexactitudes et vous êtes souvent tributaire de données dont la publication vous échappe – vous devrez probablement surveiller la validité des liens et des informations à intervalles réguliers.

Réseaux sociaux. Facebook, Twitter et MySpace ont le vent en poupe et s'imposent comme les fers de lance du Web 2.0 : tous ces services se déclinent donc naturellement à l'iPhone, qui se révèle un compagnon idéal pour en profiter pendant tous vos déplacements ! Bien entendu, les maisons mères de chacun de ces services ont d'ores et déjà décliné plusieurs outils pour se connecter et accéder à leurs réseaux sociaux. Mais à l'image des utilitaires de navigation, il vous reste une confortable marge de manœuvre pour étendre leurs capacités et profiter d'API disponibles sur le Web. Vous souhaitez afficher rapidement le statut de tous vos amis sur votre mobile, tel qu'il apparaît sur Facebook ? Pas de problème ! Des fonctions spécialement dédiées à ce type d'activités existent. Votre véritable travail consiste ainsi à imaginer une interface attractive, qui améliore considérablement l'ergonomie des outils déjà existants. Inspirez-vous des applications disponibles sur l'App Store afin de découvrir les options qui manquent encore à l'appel. Il s'agit toutefois d'une catégorie très largement balisée sur le circuit de distribution d'Apple : il est difficile de proposer des applications payantes dans cette catégorie, à moins de révolutionner fondamentalement les possibilités d'interactions.

Enseignement. Vous vous sentez l'âme d'un pédagogue et vous pouvez vous prévaloir d'être le spécialiste d'un domaine ? Envisagez de bâtir une application *encyclopédique*, qui se manipule à la manière d'un dictionnaire. Comme nous le verrons par la suite, ce type d'applications est soumis à de nombreuses contraintes, en particulier au niveau de la réactivité et du stockage des informations. Si vous répondez précisément à une demande, vous pouvez toutefois prétendre à une forte adhésion de la part des utilisateurs auprès de qui vous deviendrez une référence incontournable. C'est aussi une catégorie d'applications qui exige des mises à jour moins régulières : sitôt votre fonds encyclopédique organisé, son actualisation ne vous accapara pas durant de longs mois. Mais en vous spécialisant autour d'un domaine très précis, vous réduisez d'autant plus votre audience potentielle. Toutefois, le prix que vous fixez est généralement supérieur aux simples utilitaires. Il vous revient de préparer soigneusement le terrain en compulsant d'emblée une solide base de données, afin de rentabiliser au mieux votre investissement en temps !

Forme et santé/Médecine/Sports. On ne s'improvise pas médecin ou entraîneur professionnel, mais la demande concernant les applications de "surveillance" ou "d'initiation sportive" est très forte. Si vous jouez la carte de la transparence, en précisant que vous n'êtes pas un spécialiste, mais plutôt un passionné, vous éveillerez la curiosité de tous les utilisateurs partageant vos goûts. Calculatrices de poids, conseils diététiques, outils visant

à optimiser une pratique sportive : imaginez les utilitaires qui vous aident à vivre au mieux votre passion ou qui améliorent votre propre hygiène de vie ! Là encore, le bouche-à-oreille fonctionne particulièrement bien avec ce type d'applications, que l'on chérit au même titre qu'un compagnon.

Les bonnes idées d'applications et les projets à éviter

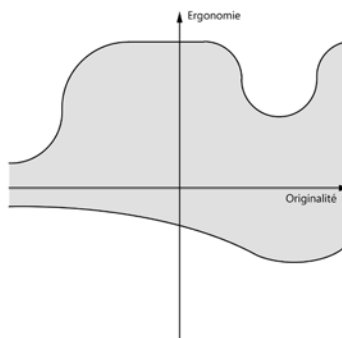
Ce rapide tour d'horizon des applications disponibles sur l'App Store d'iTunes devrait vous donner de premières pistes pour élaborer votre propre projet. Sachez distinguer les applications vouées à l'échec des projets les plus prometteurs : c'est justement en travaillant sur papier que vous identifierez rapidement les écueils et les pertes de temps inutiles. Analysez également votre niveau de compétences actuel : certes, le développement d'une application ambitieuse pourrait largement améliorer votre maîtrise du SDK et de la programmation pour iPhone, mais l'ampleur de la tâche pourrait aussi vous rebuter au point d'abandonner votre projet.

Dans un premier temps, vous devez absolument bannir toute estimation de la rentabilité de votre application. Vous mettez bien en place un projet afin de vous adresser à une large audience, mais celle-ci ne doit pas peser sur vos choix conceptuels. Par exemple, vous savez que les cadres et les bordures à ajouter automatiquement à des photos sont très populaires – mais si vous n'avez pas de compétences particulières pour les dessiner et proposer des modèles originaux et si vous tentez malgré tout d'imposer vos créations, vous risquez bien de décevoir vos utilisateurs. Sur le papier, n'envisagez que les applications qui vous tiennent réellement à cœur et n'essayez pas purement de copier des projets déjà auréolés de succès.

En revanche, ne négligez pas l'estimation du temps nécessaire au développement de votre application. Si vous êtes encore débutant et que vous ne maîtrisez pas les rouages du SDK de l'iPhone sur le bout des doigts, vous avez tout intérêt à entamer votre première approche par un projet simple qui ne vous tiendra pas en haleine de longs mois. Par ailleurs, gardez à l'esprit que deux valeurs principales font et défont le succès d'une application : son ergonomie et son originalité. Reportez-vous à la Figure 1.22 pour découvrir le parfait mélange de ces deux valeurs. En règle générale, on retiendra qu'une application très originale tolère certains défauts d'ergonomie puisqu'elle présente une forte valeur ajoutée. À l'inverse, une application peu originale (comme une énième calculatrice) ne pourra souffrir d'aucun souci ergonomique – c'est précisément votre choix de système de navigation et d'arborescence qui en fera tout le succès !

Figure 1.22

Le rapport entre l'ergonomie et l'originalité dans le succès d'une application.



Si votre projet appartient à la catégorie des jeux vidéo, veillez à clarifier très précisément son cadre et son interface, en reprenant à votre sauce les grandes recettes de cette industrie. Scénario, ergonomie, interactions, intérêt : ne négligez aucun aspect. Dans la mesure du possible, constituez-vous en équipe et déclinez différentes responsabilités aux intervenants. Pour garantir un véritable plaisir de jeu, vous avez tout intérêt à demander l'assistance d'un graphiste ou d'un illustrateur sonore... à moins que vous ne cumuliez déjà tous les talents ou que votre projet n'appelle pas d'effets visuels spécifiques ! Réfléchissez à deux fois avant d'entamer le développement d'une telle application : on découvre généralement de nombreuses contraintes en cours de route, comme le chargement d'images ou la nécessité d'évoluer avec le matériel de l'iPhone ou de l'iPod Touch.

Mises à part ces quelques recommandations, il n'y a pas réellement de projets à éviter. Tout du moins, toute idée mérite d'être étudiée sur le papier. C'est précisément cet exercice qui va révéler tous les défauts de conception et les faiblesses de votre projet : vous identifierez rapidement les écueils et vous dresserez un solide plan de travail afin de vous consacrer pleinement au développement de votre application.

Concevoir et designer son application sur papier

Vous avez réuni une série d'idées et la silhouette générale de votre application s'esquisse déjà dans votre esprit. Sans plus attendre, jetez sur papier toutes vos pistes ! N'hésitez pas à emprunter certains outils aux méthodes conceptuelles traditionnelles, comme Merise par exemple¹. Dans un premier temps, relevez tous les objectifs de votre application. Vous définissez ainsi rapidement un premier cahier des charges, qui vous servira de main-courante tout au long du développement. Triez tous ces buts par ordre d'importance, afin d'identifier

1. Pour plus de détails concernant les méthodes conceptuelles, reportez-vous à l'ouvrage *Processus d'ingénierie du logiciel* de Claude Pinet, paru aux Éditions Pearson.

des fonctions qui pourraient éventuellement être corrigées ou modifiées par la suite. L'enjeu consiste à établir rapidement un plan de développement, avec des étapes clairement identifiées.

Comme vous le verrez par la suite, notre premier exemple est un simple quiz autour du cinéma. Une rapide analyse sur papier révèle les objectifs et aspects suivants :

- Enchaîner une série de questions préparées à l'avance.
- Offrir aux utilisateurs la possibilité de répondre en pressant l'une des propositions au doigt (type QCM).
- Vérifier les réponses soumises par les utilisateurs.
- Calculer un score total en fin de questionnaire.
- Sauvegarder ce score pour le comparer à des essais ultérieurs ou aux résultats des amis.
- Envisager des mises à jour des questions.

Cette première liste d'objectifs est encore vague et ne fait l'impasse sur aucun aspect du quiz. Procédez de même en identifiant clairement les actions de votre application. Dans le cas de développements plus complexes, vous pouvez même faire appel à d'autres outils des méthodes de conceptualisation, comme le *Modèle Conceptuel de Traitement* par exemple. Ce dernier vise à définir les données qui régissent votre application, à établir les relations qui s'exercent entre les différents blocs de contenu et à modéliser les interactions des utilisateurs. Vous déclinerez ensuite ce large schéma en autant de *scripts* et de *fonctions* au sein de votre application.

Le travail sur papier (voir Figure 1.23) vous encourage également à imaginer l'interface de votre application. Là encore, n'oubliez pas que l'iPhone introduit de nouveaux concepts que vous devez pleinement exploiter.

Votre projet ne doit pas donner l'impression d'être une modeste adaptation d'un outil de Windows ou de Mac OS X : envisagez toutes les possibilités que vous offrent l'accéléromètre et l'interface tactile. N'oubliez pas non plus que l'iPhone et l'iPod Touch ont la possibilité de basculer leur affichage en mode Paysage ou Portrait. Comme nous le verrons par la suite, il est possible de programmer deux interfaces distinctes, pour chacun de ces cas de figure. Dessinez également sur papier toutes les interactions avec l'utilisateur. À quels moments est-il invité à saisir une requête ? Sous quelle forme ? À l'aide de quels contrôles ? Comment va-t-il manipuler l'interface ? Avez-vous prévu des boutons ou des onglets simples facilitant la navigation ? Multipliez les esquisses de votre application sans toutefois imaginer des effets visuels trop ambitieux. Pour l'heure, réfléchissez en termes de "blocs" et de "contrôleurs", en dessinant des zones rectangulaires que vous remplacerez par le contenu et les boutons de votre application. N'oubliez jamais qu'un utilisateur de

téléphone portable dispose d'un temps plus court pour manipuler une application que face à un ordinateur, la souris en main : évitez de multiplier les niveaux d'arborescence et faites en sorte qu'il accède rapidement au contenu recherché !

Figure 1.23

L'ébauche préliminaire d'une application sur papier.

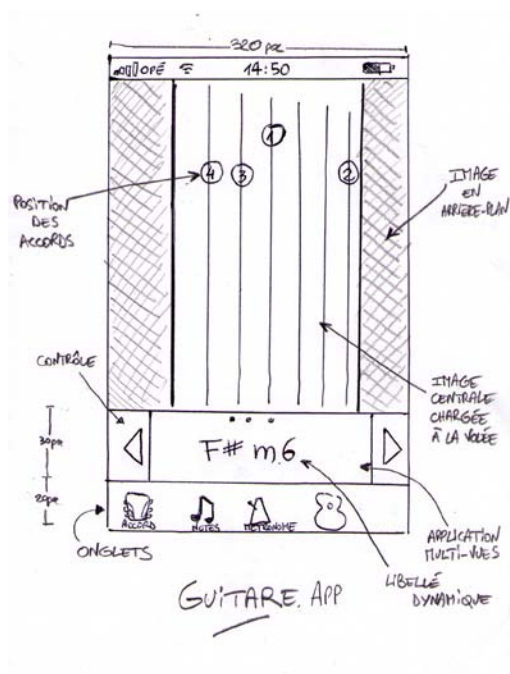


Figure 1.24

Travaillez en amont le dialogue homme-machine.

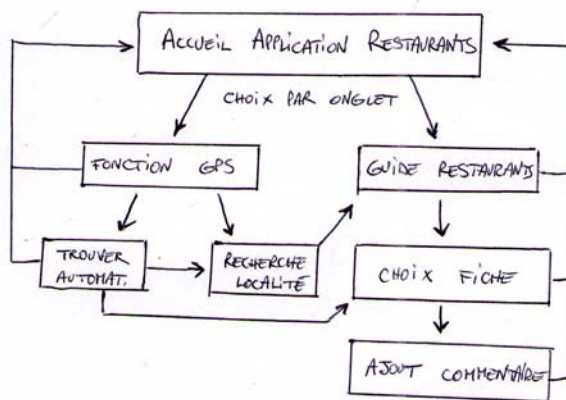


Figure 1.25

Certaines applications de l'iPhone misent sur une interface réellement originale pour offrir un usage très intuitif !



Vous êtes fin prêt et vous estimez que votre application tient la route sur le papier ? Il est grand temps de découvrir le SDK de l'iPhone dans le détail et de le confronter à votre ébauche !



2

Vue d'ensemble du développement pour iPhone

Au sommaire de ce chapitre

- Les outils à votre disposition
- À la découverte de Cocoa Touch et de l'Objective-C
- En route vers Xcode

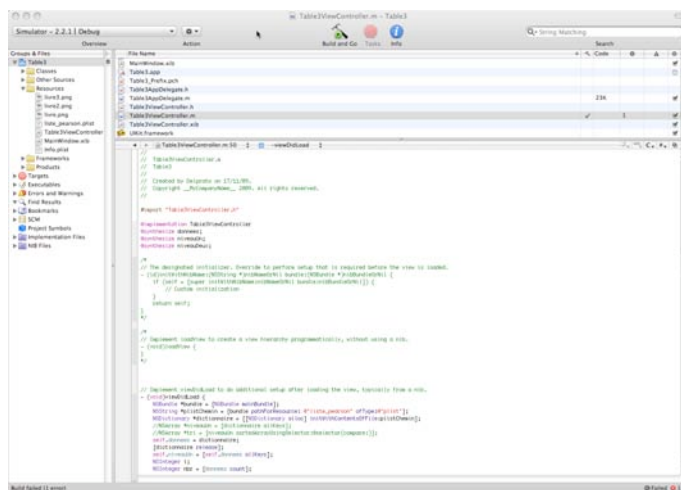
Vous avez installé le kit de développement pour iPhone d'Apple et vous avez jeté sur papier les bases de votre première application. Vous voilà bien parti pour entamer la programmation de votre projet ! Mais avant d'entrer dans le vif du sujet, attardons-nous sur tous les aspects généraux du développement pour iPhone. Si l'ergonomie et le design si particuliers du mobile d'Apple sont de précieux atouts qui vont contribuer au succès de votre application, ils vous obligent à conserver en mémoire un certain nombre de limites et de contraintes. N'y voyez pas un carcan qui bride votre imagination : adoptez plutôt ces règles afin d'optimiser au maximum votre application !

Les outils à votre disposition

En téléchargeant, puis en installant le SDK d'Apple, vous greffez une série d'outils nouveaux à Mac OS X. Parmi ceux-ci, deux pièces maîtresses vont rapidement vous devenir indispensables : l'interface de développement Xcode et l'émulateur d'iPhone. Le premier est un riche environnement de développement (IDE, *Integrated Development Environment*), capable de supporter des scripts dans de nombreux langages. L'application réunit un grand nombre d'outils facilitant la création, le débogage et la compilation de votre code source, mais visant également à optimiser votre application afin d'améliorer ses performances et de respecter au mieux les contraintes matérielles du mobile d'Apple.

Figure 2.1

Xcode devient votre environnement de développement privilégié.



Sachez par ailleurs que Xcode figure aujourd'hui parmi les rares applications optimisées pour les processeurs multicœurs. Certes, nous ne pousserons pas cet outil dans ses retranchements en développant des projets pour iPhone au point de mettre à mal la plus grosse configuration de Mac Pro, mais sachez-le : si vous disposez d'une machine puissante, Xcode saura l'exploiter au mieux. Il soutient même les cœurs "virtualisés" dont héritent les

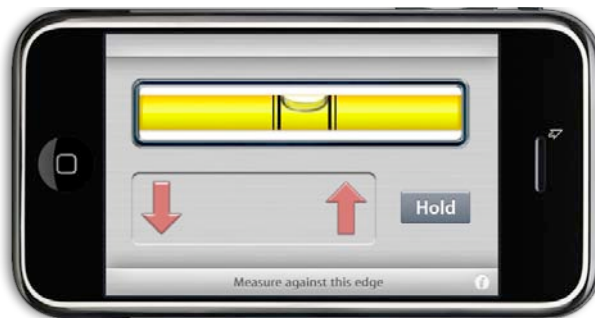
processeurs de nouvelle génération, comme les Xeon de série 5500 ou plus généralement l'architecture Nehalem d'Intel, qui se décline actuellement à tous les modèles de Mac Pro, MacBook et iMac.

La plongée dans le développement pour l'iPhone suscitera peut-être des vocations pour découvrir plus largement l'API *Cocoa* et le langage de programmation *Objective-C* : si vous aspirez par la suite à des projets de plus grande envergure, vous disposez d'ores et déjà de l'outil idéal afin de poursuivre votre carrière de développeur !

En installant le SDK de l'iPhone, vous héritez également d'un émulateur du mobile d'Apple (voir Figure 2.2). Très pratique pour tester rapidement vos applications, il constitue d'ailleurs le seul moyen d'exécuter vos projets si vous ne disposez que d'un compte gratuit de développeur. Là encore, il vous faudra investir dans une véritable licence afin de transférer vos programmes vers un véritable iPhone ou iPod Touch et d'entamer la vente de vos applications à travers l'iTunes App Store. Émulation logicielle oblige, cet outil ne gère donc pas l'accéléromètre de l'iPhone et ne permet pas non plus de capturer des photos. Rassurez-vous : il vous sera tout à fait possible d'intégrer ces deux éléments à vos projets, comme nous le verrons par la suite. Cet émulateur nous sera essentiel lors de la compilation de nos projets et des phases de test ; il s'intègre pleinement au cycle de développement de vos applications et constitue un élément central pour valider vos codes sources.

Figure 2.2

Grâce à l'émulateur intégré au SDK, Mac OS X embarque un iPhone.

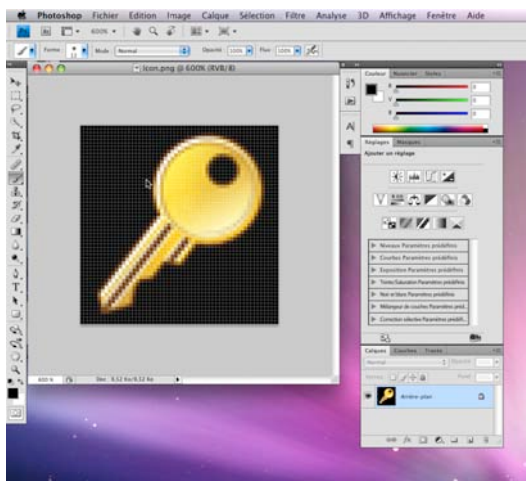


Au-delà de cette première panoplie indispensable, vous devez naturellement compléter ces outils par quelques applications dont vous ferez un usage ponctuel. Là encore, reportez-vous au plan que vous avez esquissé sur papier. Votre projet implique-t-il de nombreux effets visuels et sonores ? Vous aurez besoin d'un éditeur graphique (comme Photoshop CS4 ou The Gimp) et d'un synthétiseur/séquenceur afin de produire les images, pictogrammes et enregistrements audios liés à votre projet. Heureusement, la réputation de Mac OS X dans ce domaine n'est plus à faire et le système d'exploitation d'Apple est livré avec une batterie de logiciels qui œuvrent dans ce sens, la suite *iLife* en tête.

Complétez-les par vos propres références : vous voilà fin prêt pour entamer le développement de votre première application. Mis à part la création de l'icône de votre application et les éléments graphiques liés à nos exemples, nous n'aurons hélas pas le loisir de nous attarder sur la prise en main de tous ces outils. S'ils ne vous sont pas particulièrement familiers, n'hésitez pas à vous reporter sur les innombrables sites d'experts, toujours de bon conseil pour se former en autodidacte ! Nous vous accompagnerons toutefois dans les principales démarches liées à l'iPhone, comme la création d'une icône associée à votre application par exemple (voir Figure 2.3).

Figure 2.3

Chaque application pour l'iPhone s'identifie par sa propre icône : préparez-la à l'aide d'un logiciel de retouche d'image.



À la découverte de Cocoa Touch et de l'Objective-C

La filiation entre l'iPhone et Mac OS X

Lors de sa conférence inaugurale à la convention MacWorld 2007, où l'iPhone a été présenté pour la première fois, Steve Jobs s'est fendu d'une réplique qui a fait frémir d'enthousiasme l'assemblée : "Les logiciels pour téléphones portables sont généralement de simples ersatz à l'ergonomie limitée. Avec l'iPhone, nous proposons des logiciels qui ont d'ores et déjà cinq années d'avance sur n'importe quel autre terminal mobile. Comment avons-nous atteint un tel résultat ? Tout simplement en articulant l'iPhone autour d'une base solide : notre mobile exécute Mac OS X".

Cette architecture n'est pas le fruit du hasard : après des années de recherche et de développement, Mac OS X regorge de fonctions et de services dont la plupart correspondent très précisément aux besoins des utilisateurs d'iPhone. Connexions réseau, synchronisation, options d'économie d'énergie, sécurité renforcée, support de la vidéo, bibliothèques

graphiques et sonores... Le champ d'action de l'iPhone s'inscrit parfaitement dans tous ces domaines d'application, ce qui évite aux ingénieurs d'Apple de "réinventer la roue". Sans céder au prosélytisme pour autant ni se faire les dithyrambes d'Apple, on conviendra que ce lien de parenté entre l'iPhone et Mac OS X se traduit pour l'utilisateur final par une ergonomie singulière et surtout une grande intégration entre le mobile et les systèmes avec lesquels il se synchronise (voir Figure 2.4).

Figure 2.4

Les paramètres système de l'iPhone décrivent le fonctionnement du système d'exploitation embarqué et s'apparentent à ceux de Mac OS X – vous définissez notamment les fonctionnalités Bluetooth et WiFi.



En tant que développeur et designer, vous pouvez surfer sur cette tendance dès l'ébauche sur papier de votre application. Certes, comme nous le verrons par la suite, l'iPhone est doté d'une série de contrôleurs qui évoquent déjà largement les outils intégrés à Mac OS X (champs, boutons, fenêtres de réglages, clavier virtuel...). Mais de manière fine et discrète, vous avez tout intérêt à concevoir votre application en songeant à l'ergonomie des logiciels les plus populaires pour les machines d'Apple, afin de renforcer encore ce lien de parenté. Mac OS X bénéficie d'une interface très intuitive et nous pousse à contracter des réflexes qui échappent encore au monde Windows : en déclinant ces concepts et ces systèmes de navigation à vos propres applications, vous avez toutes les chances de captiver un plus large public – votre application sera louée pour son ergonomie et sa prise en main intuitive. Sans compter qu'Apple voit d'un bon œil les développeurs qui s'inspirent de tous ces jalons posés au fil des années ! À ce titre, nous vous invitons à consulter le guide officiel des interfaces homme-machine de l'iPhone, à l'adresse <http://tinyurl.com/iphoneIHM>. Particulièrement clair et complet, ce guide décrit en détails toutes les bonnes pratiques aboutissant à une application intuitive et ergonomique. Vous découvrirez ainsi à quel point l'emplacement de vos boutons et contrôleurs ne doit jamais être le fruit du hasard.

Plus intéressant encore vis-à-vis de la tâche qui nous occupe aujourd'hui : la parenté entre l'iPhone et Mac OS X se prolonge jusqu'au noyau du système et à son environne-

ment de développement. Si vous avez déjà développé des applications pour le système d'Apple, vous êtes familier de Cocoa et de son langage de programmation de prédilection, Objective-C. Sachez que tous ces éléments se déclinent à l'iPhone et que vous serez rapidement en mesure d'adapter vos compétences au terminal d'Apple. Si vous n'avez encore jamais programmé d'applications pour Mac OS X, pas de panique : ce n'est absolument pas un passage obligé pour développer vos projets pour l'iPhone ou l'iPod Touch et vous pourrez facilement partir d'un bon pied en déclinant les grands concepts que vous maîtrisez déjà à travers d'autres langages de programmation. Si l'iPhone est la toute première plate-forme pour laquelle vous allez développer, n'ayez crainte. À l'image de son interface ergonomique et intuitive, les outils de développement demeurent clairs et efficaces. Reportez-vous éventuellement à la fin de ce chapitre pour déterminer votre profil de développeur et estimer votre niveau de compétence.

Une première approche de Cocoa

Si le parallèle entre Mac OS X et l'iPhone est évident, il faut tout de même garder à l'esprit les spécificités du terminal d'Apple. Il s'agit avant tout d'un appareil nomade, que l'on exploite lors de ses déplacements : les utilisateurs ne le manipulent pas à la souris et l'iPhone introduit plusieurs fonctionnalités inédites, comme le Multi-Touch et l'accéléromètre. Pour toutes ces raisons, Apple ne qualifie pas explicitement le système de l'iPhone de "Mac OS X" mais plutôt "d'iPhone OS". Mais le fort rapprochement entre les deux systèmes persiste malgré tout : pour vous en convaincre, vous pouvez parcourir les longues conditions d'utilisation (dans Réglages > Général > Informations > Mention légale sur votre iPhone) où figure le nom de toutes les bibliothèques et sources utilisées.

Les applications pour Mac OS X et pour iPhone partagent ainsi les mêmes racines, autour de l'environnement de développement *Cocoa*. Cette pure plate-forme de développement pour Mac s'inscrit dans l'héritage du système NeXTSTEP, créé en 1989 par la société NeXT de Steve Jobs et racheté par Apple en décembre 1996. Avant d'aller plus loin, retenez que cet environnement est très largement orienté objet et qu'il constitue l'un des socles de Mac OS X, en tirant parti de la puissance des outils de développement d'Apple comme Xcode. Sous Mac OS X, Cocoa est partie prenante du système et constitue à la fois un environnement d'exécution d'applications (comme Safari par exemple) et de développement, autour d'un ensemble de frameworks orientés objets. La plupart des applications s'exécutant sous Mac OS X sont ainsi des applications Cocoa ! Du côté de l'iPhone, il s'agit même du seul environnement disponible : des applications intégrées au système, comme Mail ou Safari par exemple, sont des applications Cocoa.

En manipulant Xcode pour développer vos applications pour iPhone, vous vous inscrivez donc pleinement dans cet environnement. Cocoa est un colossal ensemble de composants

orientés objet (des *classes*), que vous pouvez directement exploiter ou adapter à vos besoins afin de construire toutes les "briques" de votre application. La quasi-totalité des comportements du système, des interactions avec l'utilisateur, de l'interface graphique et de la lecture/écriture de données sont couverts par cette large bibliothèque de classes. Un environnement si riche est une vraie aubaine pour les programmeurs : la politique de développement d'Apple est clairement normée et Cocoa automatise de nombreux aspects du développement d'une application. Nous le verrons très rapidement dans la suite de cet ouvrage : si un domaine précis échappe à sa juridiction, vous pouvez très facilement créer une sous-classe afin d'élargir son champ d'action. Dans la mesure où l'iPhone introduit justement de nombreux comportements inédits, en particulier l'interface tactile mais aussi le GPS et l'accéléromètre, Cocoa a été revu et corrigé afin de disposer d'emblée des classes adéquates. Dans le cadre du développement pour iPhone et iPod Touch autour du SDK délivré par Apple, on parle ainsi de l'API "Cocoa Touch" afin de marquer cette différence.

Ouvrages à consulter

Le développement d'applications Cocoa vous intéresse et vous souhaitez élargir votre apprentissage de l'Objective-C à l'ensemble de plate-forme Mac OS X ? N'hésitez pas à vous reporter à l'ouvrage *Programmation Cocoa sous Mac OS X* d'Aaron Hillegass, paru aux éditions Pearson (<http://www.pearson.fr/livre/?GCOI=27440100726260>). Pour une exploration en détails du langage Objective-C, nous ne saurons que vous recommander la lecture de *Objective-C 2.0, le langage de programmation iPhone et Cocoa sur Mac OS X*, de Pejvan Beigui, également paru aux éditions Pearson (<http://www.pearson.fr/livre/?GCOI=27440100024070>). Ces deux ouvrages complètent le présent manuel et vous permettent de prolonger votre apprentissage au-delà du seul cadre de l'iPhone.

Le langage de programmation Objective-C

Dans la mesure où les environnements de développement Cocoa et Cocoa Touch sont intégralement écrits dans le langage Objective-C, celui-ci devient le langage de référence pour le développement d'applications Cocoa. Vous l'utiliserez constamment lorsque vous mettrez au point votre application pour iPhone ! Un bref rappel historique s'impose avant d'aller plus loin. À la fin des années 1970, le langage C connaît une immense popularité : Brian Kernighan et Denis Ritchie, du Laboratoire AT&T Bell, l'ont mis en place quelques années plus tôt et il est même devenu la pierre angulaire des systèmes d'exploitation UNIX, totalement écrits dans ce langage. Au début des années 80, les pratiques de génie logiciel évoluent et donnent naissance à la programmation structurée : les larges programmes sont divisés en de plus petits sous-ensembles, plus faciles à déboguer et à exécuter de manière autonome. Mais en conservant l'aspect traditionnel du C, une telle pratique n'optimise pas l'écriture du code pour autant ; les programmes gagnant en complexité, il est de plus en plus

difficile de réutiliser du code et il faut prévoir de larges structures de contrôle afin de lier les différentes "parties" d'un programme.

C'est à cette époque que les premières extensions du C vers l'orienté objet apparaissent : Bjarne Stroustrup développe le C++ en 1979. La notion de classes et d'objets réutilisables résout les problèmes évoqués précédemment et les développeurs peuvent envisager des programmes effectuant des traitements très complexes sans écrire des millions de lignes de code pour autant. L'Objective-C est une approche parallèle, initiée par Brad Cox à la même période. Là encore, il s'agit d'une couche supplémentaire au C visant à l'autoriser à créer et manipuler des objets. Si vous n'avez encore jamais eu la moindre expérience avec un langage orienté objet, sachez qu'Objective-C constitue une formidable approche pour s'initier aux concepts de classes, d'objets et d'héritage. Grâce à un nombre de types réduit et à sa grande intégration avec Cocoa, Objective-C s'avère relativement intuitif à l'usage tout en produisant des programmes rigoureux, bâtis autour de classes et sous-classes automatiquement hiérarchisées. Comme nous le verrons par la suite, il s'approche davantage de Java que de C++. En tant que surcouche du C, il respecte de manière très stricte sa syntaxe. Toutefois, toutes les opérations liées à l'orienté objet épousent une forme originale et intuitive : en Objective-C, on "n'appelle" pas une méthode comme en C++, on "envoie un message" à un objet. Là encore, nous aurons tout le loisir de détailler cette spécificité autour d'exemples concrets, mais retenir ce concept général : le traitement de ces "messages" est très largement automatisé et vous autorise une grande souplesse dans la programmation de votre code. L'objet destinataire a ainsi la possibilité de décliner le message ou de le traiter et il n'est donc pas nécessaire d'implémenter toutes les méthodes définies dans le code source. Une autre particularité d'Objective-C que vous devez retenir tient au système de "catégories de classes". Il est très facile de réunir et de trier les classes et sous-classes sous une même bannière : à travers l'interface de développement Xcode, vous "visualisez" très précisément vos morceaux de code et vous facilitez leur débogage ou leur mise à jour. Idéal pour s'inscrire parfaitement dans le système des "mises à jour" proposé par l'iTunes App Store, qui permet aux utilisateurs d'une application de profiter rapidement des derniers changements imaginés par ses développeurs !

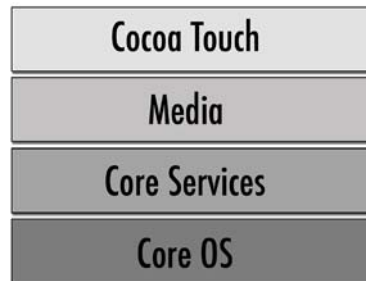
En 2007, Objective-C a connu une sérieuse évolution qui apporte son lot de nouveautés, en particulier l'introduction de propriétés. Apple l'évoque sous le nom "d'Objective-C 2.0" et c'est devenu le langage de programmation par excellence des applications Cocoa depuis Mac OS X 10.5. Lorsque vous développez des programmes pour l'iPhone et l'iPod Touch, vous utilisez ainsi le langage de programmation Objective-C 2.0 sous l'environnement de développement Cocoa Touch.

Les couches technologiques de l'iPhone OS et l'intégration de Cocoa Touch

Vous connaissez désormais l'aspect théorique des deux pièces centrales du développement pour iPhone. Voyons à présent de manière schématique comment ces deux composants s'imbriquent dans iPhone OS. Reportez-vous à la Figure 2.5 qui présente les différentes couches technologiques du système de l'iPhone.

Figure 2.5

Les couches technologiques de l'iPhone OS.



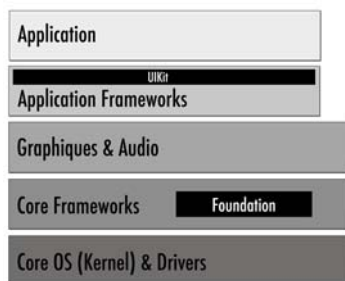
Quatre couches successives se superposent afin de réunir toutes les fonctions de l'iPhone. Les couches inférieures, Core OS et Core Services, correspondent aux interfaces et éléments fondamentaux, notamment le noyau du système, la gestion des fichiers, les accès réseau et les options d'alimentation. Les couches supérieures sont largement écrites en Objective-C et réunissent toutes les technologies propres à l'iPhone. La couche Media comprend ainsi la gestion du dessin (OpenGL ES, Quartz, Core Animation...), mais aussi le support de la vidéo et de l'audio (Core Audio). La dernière couche contient l'environnement de Cocoa Touch. Elle se divise en deux frameworks complémentaires, qui constituent le socle de toutes les applications pour iPhone : *UIKit* et *Foundation*. Comme son nom l'indique, *UIKit* contient tous les objets apparaissant dans l'interface utilisateur d'une application et délivre aux programmeurs de nombreux outils pour représenter des éléments à l'écran. De nombreuses classes prédéfinies vous permettent de gérer les fenêtres, les vues et les contrôleurs de votre application. Le framework *Foundation* gère notamment le comportement des objets, l'écriture et la lecture de fichiers et les opérations réseau. C'est aussi à ce niveau que se trouvent toutes les classes permettant d'accéder aux contacts et aux informations de l'utilisateur, ainsi qu'aux technologies exclusives de l'iPhone, comme l'accéléromètre ou le GPS par exemple.

La couche Cocoa Touch et les deux frameworks *UIKit* et *Foundation* constituent ainsi le socle du développement pour iPhone. En règle générale, vous trouverez systématiquement une méthode ou une fonction dans l'un de ces deux frameworks capables de prendre en charge la plupart des opérations du terminal d'Apple ; vous avez d'ailleurs tout intérêt à les exploiter directement afin d'éviter de faire appel aux couches inférieures d'iPhone OS.

Toutefois, nous aurons ponctuellement besoin d'utiliser un framework situé à un niveau sous-jacent afin d'exercer un plus grand contrôle sur le comportement d'une application. Le framework UIKit vous permet d'afficher simplement du texte à l'écran : comme vous le verrez à travers certains exemples de cet ouvrage, il est parfois préférable de le suppléer par un framework de la couche Media afin de mieux personnaliser l'intégration de tels objets. Reportez-vous à la Figure 2.6 pour visualiser de manière plus détaillée l'imbrication des frameworks dans la couche Cocoa Touch.

Figure 2.6

Les frameworks de la couche Cocoa Touch.



Cette vue d'ensemble vous permet de découvrir tous les rouages internes de la couche Cocoa Touch. Dès le chapitre suivant, nous plongerons dans les bases d'Objective-C 2.0 à travers l'utilisation de Xcode et nous mettrons ce canevas général en application !

Tirer partie de ses connaissances d'autres langages

Dans la suite de cet ouvrage, nous partons du principe que vous n'avez pas encore d'expérience particulière dans le développement d'applications pour iPhone et iPod Touch. Il n'est pas non plus nécessaire de se familiariser avec Cocoa et le développement sous Mac OS X pour partir d'un bon pied : les outils du SDK sont clairs et intuitifs, de plus, l'iPhone présente un certain nombre de spécificités par rapport aux applications Mac OS X.

Toutefois, comme nous l'avons vu, le développement d'applications pour iPhone s'articule autour d'un langage orienté objet. Vous devez disposer de connaissances générales autour du concept d'objets et de classes pour mieux appréhender toutes les subtilités d'Objective-C 2.0 et de Cocoa Touch. À ce titre, sachez que des expériences comparables dans d'autres langages de programmation peuvent tout à fait se transposer et vous feront gagner un temps précieux. Les férus de Java et de C++ sont choyés et devraient rapidement trouver leurs marques, au-delà d'une période d'initiation à la nomenclature et à la syntaxe particulières d'Objective-C.

La programmation n'a jamais couvert un champ aussi large qu'aujourd'hui. Il existe des dizaines de langages, adaptés à des domaines très spécifiques comme le développement web par exemple. Tous les langages de programmation structurée trouvent un écho à travers Objective-C. Ainsi, si vous avez déjà développé en ActionScript 3.0 ou en PHP 5, vous êtes familier avec le concept des classes et de l'héritage. Le développement pour iPhone

est relativement spécifique, dans la mesure où le programmeur endosse bien souvent le rôle de designer d'interface et qu'il doit travailler avec un cadre limité. Là encore, une expérience en développement web (où il faut bien souvent prendre en compte les feuilles de style et d'autres éléments de nature ergonomique, comme le placement de boutons ou d'images !) est un sérieux atout : c'est en multipliant les casquettes que vous parviendrez rapidement aux résultats les plus aboutis, sans lésiner sur le moindre aspect comportemental ou esthétique de votre application. Comme nous l'avons vu au chapitre précédent, il existe également deux frameworks puisant leurs racines dans JavaScript : QuickConnect et PhoneGap. Ils permettent de développer des applications ambitieuses sans nécessairement passer par du code en Objective-C. Dans la mesure où ces frameworks n'exploitent pas le SDK de l'iPhone, nous ne les passons pas en revue au cours de cet ouvrage et nous vous invitons à vous reporter au livre *Développez des applications pour iPhone avec HTML, CSS et JavaScript* de Lee S. Barney, paru aux éditions Pearson.

Les spécificités et les limites de l'iPhone

Nous avons déjà évoqué au chapitre précédent le caractère singulier de l'iPhone et les nombreuses implications de cette ergonomie sur le design de votre application. Maintenant que vous disposez d'une connaissance plus approfondie de l'architecture d'iPhone OS, vous devez retenir que votre application accapare totalement le système lors de son exécution. Téléchargée sur le terminal d'un utilisateur, elle est représentée par son icône individuelle, aux côtés des applications livrées en standard avec l'iPhone ; si l'utilisateur clique sur cette icône, votre application s'exécute et occupe totalement l'écran. Lorsque l'utilisateur presse le bouton principal, votre application se termine et le système s'affiche à nouveau. Mis à part quelques démons et services de bas niveau (comme la possibilité de basculer automatiquement vers un appel téléphonique, lorsque vous exécutez une application), votre application dispose d'un contrôle total des ressources logicielles et matérielles du système, y compris les périphériques intégrés comme l'appareil photo ou l'accéléromètre.

N'oubliez jamais que vous programmez une application pour un périphérique mobile. Aussi moderne soit-il, il s'inscrit dans des limites matérielles très strictes : même si vous disposez d'un environnement d'exécution tout entier réservé à votre application, vous devez vous borner à respecter un certain cadre. Parcourons tous ces aspects ensemble.

Une taille d'affichage limitée. C'est indéniablement l'un des atouts de l'iPhone et de l'iPod Touch : ils sont affublés d'un superbe écran, clair et lumineux. Au moment de leur sortie respective, cet écran dépassait largement la qualité et la résolution de la plupart des terminaux mobiles. Mais à l'heure où la vidéo en haute définition connaît un fort engouement, vous devez toutefois inscrire votre application dans un cadre plus limité : la résolution de l'écran de l'iPhone est de 480 × 320 pixels. Plus précisément, votre application dispose d'un cadre

de 460 × 320 pixels : 20 pixels de hauteur sont réservés au menu supérieur qui affiche le nom de l'opérateur, le niveau de batterie actuel et la portée du signal WiFi.

Autant dire que vous devrez rivaliser d'ingéniosité et profiter au mieux du framework UIKit pour mettre en place un système de navigation ergonomique, qui s'articule autour d'onglets et de fenêtres successives ! L'iPhone dispose toutefois d'un mode "d'auto-rotation" qui vous permet de profiter d'un espace de 480 × 300 pixels en mode Paysage (là encore, les informations sur l'opérateur sont préservés dans une barre de 20 pixels de hauteur). Nous y reviendrons.

Figure 2.7

L'application du quotidien Le Monde dispose d'un système de navigation original, qui exploite au mieux l'espace disponible sur l'écran de l'iPhone.



Une seule fenêtre à la fois. Contrairement aux applications bureautiques, les applications pour iPhone ne disposent que d'une seule "fenêtre" pour s'exécuter. Là encore, cette limite pèse avant tout sur vos choix en matière de design et d'interface utilisateur.

Une seule application à la fois. Comme nous l'avons vu, votre application dispose d'un contrôle quasi total des ressources de l'iPhone au moment de son exécution. Si l'utilisateur presse le bouton principal, celle-ci s'arrête : vous ne pouvez donc pas envisager de prolonger ses traitements en "tâche de fond", pendant que l'utilisateur manipule d'autres applications ou services. Il s'agit toutefois d'une limite que les ingénieurs d'Apple cherchent à remettre en cause et qui pourrait être évincée lors d'une prochaine mise à jour du système, avec des terminaux disposant d'une plus grande quantité de mémoire vive par exemple (vous avez dit *iPhone 4* ?).

Une réactivité limitée. Dans le même ordre d'esprit, sachez qu'Apple a voulu préserver coûte que coûte la réactivité et la souplesse de l'iPhone : dans la mesure où il s'agit avant tout d'un téléphone portable, les utilisateurs doivent être en mesure de basculer rapidement d'une activité à l'autre. Lorsqu'ils pressent l'icône de votre application, celle-ci doit se char-

ger immédiatement. À l'inverse, lorsqu'on appuie sur le bouton principal, on revient instantanément à l'écran d'accueil du système. Si votre application est conçue pour enregistrer et charger des paramètres personnalisés (profil de l'utilisateur, données, etc.), sachez qu'elle ne dispose que de cinq secondes au maximum pour s'acquitter de ces tâches. Nous y reviendrons, mais vous devrez vous efforcer d'optimiser au mieux les opérations de sauvegarde afin de vous assurer qu'aucune donnée n'est perdue lorsque l'utilisateur quitte promptement votre application. La Figure 2.8 illustre ce type d'applications.

Figure 2.8

*La sauvegarde automatique
du circuit parcouru
avec Crash Bandicoot
Nitro Kart 3D.*



Des ressources système limitées. L'un des intérêts de la programmation pour un terminal mobile comme l'iPhone est de connaître à l'avance le matériel exploité par les utilisateurs : vous savez à quoi vous attendre et vous n'êtes pas en droit d'exiger une "configuration minimale"... puisque vous la connaissez déjà ! La principale limite réside dans la quantité de mémoire vive disponible pour votre application. Toutes les déclinaisons actuelles de l'iPhone embarquent 128 Mo de RAM : une quantité tout à fait raisonnable pour envisager des applications riches sur le plan visuel, comme le démontrent les projets d'éditeurs traditionnels du jeu vidéo (*Need For Speed, Episode 1112...*). Même si les applications s'exécutent de manière individuelle, sachez que le système accapare environ la moitié de ces ressources pour fonctionner. Vous disposez ainsi d'une marge confortable de 64 Mo de mémoire vive pour exécuter votre application. Par ailleurs, l'iPhone n'est pas en mesure d'écrire un fichier d'échange (*swap*) à l'heure actuelle. Vous ne disposez ainsi que de la mémoire physique du terminal et vous ne pouvez donc pas exploiter les 8, 16 ou 32 Go d'espace de stockage pour gonfler artificiellement cette valeur. Il s'agit toutefois d'une valeur qui risque d'être remise en cause lors de la sortie de la prochaine version de l'iPhone. Heureusement, Cocoa Touch et Objective-C 2.0 sont dotés de nombreuses classes et de plusieurs mécanismes visant à limiter les fuites de mémoire et à gérer au mieux les ressources du système.

Figure 2.9

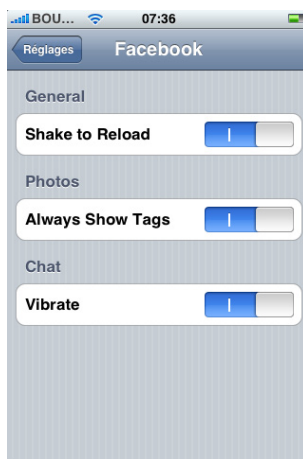
Malgré ses ressources plutôt limitées, l'iPhone est en mesure d'afficher de vibrantes animations en 3D !



Un accès limité aux données. C'est l'un des chevaux de bataille d'Apple avec l'iPhone et l'iTunes App Store : la marque à la pomme veille scrupuleusement à préserver les données personnelles de ses utilisateurs. Votre application ne peut lire et écrire des données que dans une portion de l'espace de stockage qui lui est spécifiquement réservée (qu'on qualifie généralement de *sandbox*, ou bac à sable). Vous ne pouvez donc pas envisager de rédiger des "virus" qui viendraient effacer le carnet d'adresses des utilisateurs !

Figure 2.10

L'enregistrement des paramètres de l'application Facebook dans sa zone dédiée.



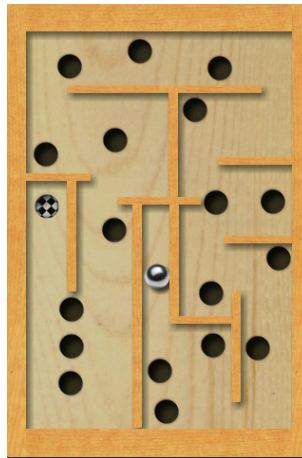
Fort heureusement, toutes ces limites ont été soigneusement définies et anticipées par Apple et le SDK de l'iPhone. Sous Xcode et à travers l'émulateur, vous serez ainsi en mesure de tester précisément le comportement de votre application et vous ne pourrez pas mettre en vente un projet qui viendrait faire planter l'iPhone. Par ailleurs, de nombreuses méthodes et procédures prédéfinies vous permettent de gérer automatiquement les ressources du système, en particulier la mémoire vive disponible, sans multiplier les lignes de code. Vous pouvez ainsi vous focaliser sur le comportement de votre application, sans vous arracher les cheveux sur toutes ces conditions !

En route vers Xcode

Installez-vous confortablement face à votre Mac et lancez Xcode : nous entrons de plain-pied dans le vif du sujet et nous allons découvrir Objective-C et le développement pour iPhone par la pratique ! Avant d'aller plus loin, prêtez-vous à une petite expérience : lancez l'une de vos applications fétiches sur votre iPhone et essayez de "deviner" d'où proviennent les comportements et classes utilisés. Le jeu très populaire *Labyrinth* de Carl Loodberg constitue un bon exemple. Gratuit dans une version comprenant dix niveaux, ce jeu de plateau s'appuie sur l'accéléromètre de l'iPhone pour envoyer une bille d'un point A à un point B, sur un parcours semé de trous.

Figure 2.11

L'écran de jeu de Labyrinth.



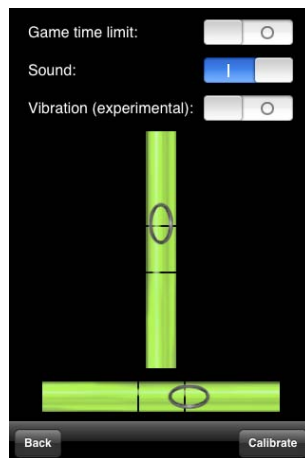
Voici les différents éléments qui le composent :

- **L'aspect visuel.** Une icône de 57 × 57 pixels pour identifier l'application, une série d'arrière-plan de 320 × 480 pixels et des "sprites" à l'effigie de la bille, des trous et de la ligne d'arrivée. Seul le mode Portrait est autorisé et il n'y a pas d'auto-rotation de l'écran.
- **L'aspect audio.** Dans cette version gratuite, il n'y a que quatre "bruitages" employés – le rebond de la bille sur les parois, le bruit de sa chute, le son de la bille sur le "plancher" de bois et le signal sonore qui accompagne la victoire.
- **Le comportement.** Grâce au framework Foundation, le jeu *Labyrinth* tire rapidement parti de l'accéléromètre de l'iPhone. Les coordonnées de la bille sont donc sans cesse réévaluées en fonction de l'inclinaison du terminal. L'application prévoit en plus un rebond sur les parois du jeu. D'un point de vue fonctionnel, les coordonnées de la bille se réinitialisent si elle vient chuter dans un trou (qui correspond à un rectangle dont les coordonnées sont fixes) et on charge le niveau suivant si la bille atteint l'objectif (dont les coordonnées sont également fixes et connues).

- **L'interface.** Au lancement de l'application, un premier écran vient faire patienter l'utilisateur pendant deux secondes le temps de charger ses paramètres (les niveaux déjà réussis apparaissent alors grisés). Une barre de menus permet de basculer entre trois "écrans" : l'aire de jeu, les paramètres et les crédits. Les auteurs optimisent ainsi l'espace tout en offrant un accès rapide aux fonctions essentielles.

Figure 2.12

L'écran des paramètres du jeu, enregistrés dans la sandbox de Labyrinth.



Le concept vous séduit et suscite des vocations ? Passons sans plus tarder à la phase pratique !



3

À la découverte du SDK de l'iPhone

Au sommaire de ce chapitre

- Créer votre premier projet pour iPhone
- Tester et déboguer votre premier projet

Après cette première découverte des éléments fondamentaux du développement pour l'iPhone, vous avez probablement très envie de passer à la phase pratique. Faites chauffer Xcode et Interface Builder, les deux outils indispensables du SDK : nous allons les parcourir sous toutes les coutures jusqu'à mettre en place notre première application iPhone ! Comme vous le constaterez à travers ce premier exercice, ce sont les deux faces d'une même pièce ;

vous effectuerez des allers-retours incessants entre ces deux applications de développement, en vous arrêtant régulièrement au Simulateur d'iPhone afin de valider chaque étape. Sachez par ailleurs qu'il n'est pas nécessaire, à ce stade, de disposer d'un compte de développeur auprès d'Apple : vous pouvez parfaitement vous contenter de manipuler des objets, d'écrire du code source et de tester le résultat sous Mac OS X. Comme nous le verrons par la suite, le précieux sésame octroyé par Apple vous offre essentiellement une option supplémentaire : le dépôt immédiat de votre application compilée sur votre propre iPhone depuis Xcode. Mais à ce stade, il est encore inutile de manipuler votre premier projet dans des conditions réelles ; focalisez-vous essentiellement sur la découverte de l'interface de ces deux logiciels et sur les techniques générales de développement.

Au cours de ce chapitre, vous apprendrez à créer toutes sortes de projets pour iPhone sous Xcode et à gérer l'ensemble des fichiers fondamentaux qui composent une application. Grâce à Interface Builder, vous créerez la "vue" de votre application en profitant d'une bibliothèque riche de composants et vous mettrez en application tous les enseignements généraux à propos d'Objective-C et de Cocoa Touch jusqu'à manipuler votre premier projet sur le Simulateur d'iPhone. Vous profiterez enfin de la large bibliothèque d'applications gratuites disponibles sur le site d'Apple pour parfaire cette première approche et découvrir tous les outils de débogage intégrés à Xcode.

Et si le développement des applications les plus ergonomiques et fonctionnelles de l'App Store ne reposait que sur une judicieuse série de clics à travers Interface Builder ? Retrouvez vos manches : nous allons le vérifier immédiatement !

Créer votre premier projet pour iPhone

Au cours du premier chapitre, vous avez installé le SDK de l'iPhone et vous disposez de tous les outils nécessaires au développement de votre première application. Commencez par lancer Xcode en ouvrant une fenêtre du Finder, puis en déroulant l'arborescence `Developer > Applications` et en effectuant un double-clic sur Xcode. C'est d'ailleurs dans ce répertoire que se trouvent les autres outils liés au SDK de l'iPhone, en particulier Interface Builder et Instruments. En parallèle, profitez-en pour agencer les répertoires de votre disque dur et créer un dossier spécifiquement lié à votre premier projet. Son emplacement de stockage importe peu : vous aurez en revanche besoin de réunir tous les éléments constituant un même projet dans le dossier de votre choix.

Maîtriser Xcode

Xcode vous accueille par un écran de bienvenue qui vous offre un accès direct aux sections emblématiques du centre des développeurs d'Apple : création d'une application, construc-

tion de l'interface utilisateur, stockage des données, optimisation de votre code... Tous ces sujets sont traités à travers une série de vidéos, que vous avez tout intérêt à visionner si vous vous sentez perdus ! De très nombreux articles techniques issus de l'iPhone Dev Center complètent également cette première approche. Si vous vous sentez en confiance, n'hésitez pas à décocher la case Show at launch pour ne plus être importuné par la suite.

Figure 3.1

L'écran de bienvenue de Xcode.



La liaison avec votre propre iPhone

Si vous avez connecté votre iPhone à votre Mac pendant le lancement de Xcode, vous êtes susceptible de rencontrer un message vous demandant si ce périphérique doit être utilisé dans le cadre du développement de votre application. Si vous avez souscrit au programme de développement d'Apple, reportez-vous plutôt à la procédure décrite après le paiement, sur le site des développeurs : elle détaille précisément l'activation de votre propre iPhone et vous bénéficierez ainsi d'une meilleure intégration avec les outils de développement. Pour l'heure, il est inutile de tester le résultat de notre application sur un véritable terminal : cliquez sur le bouton Ignore et fermez la fenêtre.

Déroulez le menu File et cliquez sur New Project. L'assistant de création de projet apparaît à l'écran. Comme nous l'avons vu précédemment, Xcode permet aussi bien de réaliser des applications Cocoa pour Mac OS X que pour iPhone/iPod Touch : ces deux plates-formes figurent ainsi dans le volet gauche de l'assistant. Cliquez sur l'intitulé Application, en dessous de la rubrique iPhone, pour découvrir les six modèles (*templates*) associés à ce type de développement. À l'instar des modèles de documents d'un traitement de texte par exemple, ils représentent six projets types préparés par Xcode. Leur utilisation vous fait ainsi gagner un temps précieux et vous bénéficiez directement de tous les éléments dont votre application a besoin. Voici à quoi ils correspondent (ils sont illustrés Figure 3.2) :

Navigation-Based Application. On utilise ce modèle pour les applications s'articulant autour d'une série de listes et de contrôles de navigation, où les différentes vues s'imbriquent sous la forme d'arborescences. C'est le modèle général des options de réglage de l'iPhone ou de votre liste de contacts, par exemple.

OpenGL ES Application. Ce modèle prépare tous les éléments nécessaires aux applications graphiques, en particulier celles qui lancent des animations ou des effets visuels.

Tab Bar Application. Ce modèle crée rapidement une application dotée d'un système de navigation par onglets avec une vue prédéfinie pour les deux premiers d'entre eux.

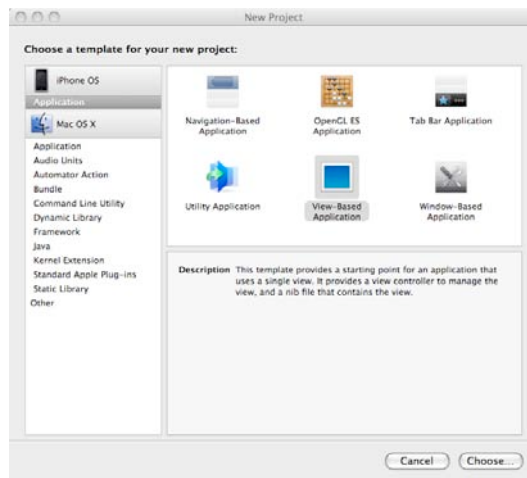
Utility Application. Comme son nom l'indique, ce modèle correspond en particulier aux utilitaires : il met automatiquement en place une vue avec un bouton Info permettant de configurer l'application et un système de navigation offrant un raccourci direct pour revenir à la vue principale du programme. Sans écrire la moindre ligne de code, vous disposez ainsi d'un canevas de base pour votre application : en cliquant sur ce bouton Info, un petit effet visuel à travers lequel l'affichage "se tourne" comme une feuille pour céder la place à un second écran est automatiquement ajouté à votre projet.

View-Based Application. C'est le modèle le plus simple proposé par Xcode : il crée une vue de base sur laquelle vous déposerez tous les éléments depuis Interface Builder.

Window-Based Application. Il s'agit d'un modèle vierge, qui ne dispose que d'une instance de la fenêtre principale de l'iPhone. Privilégiez-le si vous souhaitez démarrer d'un canevas épuré, dans lequel il reste encore tout à faire.

Figure 3.2

Les différents modèles associés à la création d'un nouveau projet pour iPhone sous Xcode.

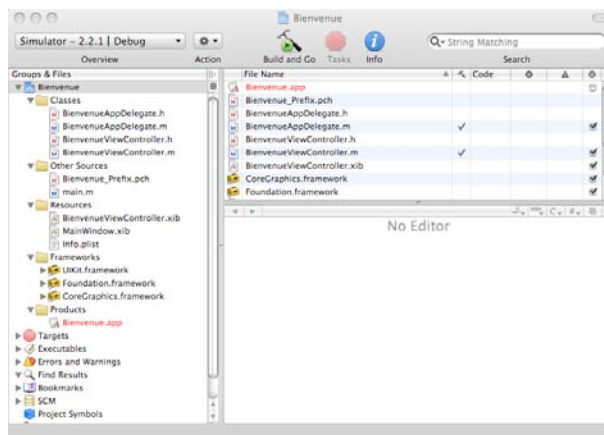


Notre premier projet consiste à afficher un texte sur l'écran de l'iPhone et à déposer une série de contrôleurs : cliquez sur le modèle View-Based Application, puis sur le bouton Choose. Vous êtes ensuite invité à baptiser votre nouveau projet et à choisir un répertoire d'enregistrement. Le nom du projet a une influence majeure sur la création de tous les fichiers de base : choisissez-le avec soin et évitez surtout les doublons si vous enregistrez plusieurs projets dans le même répertoire. Dans notre exemple, nous l'appelons "Bienvenue" et nous le stockons dans le dossier /Documents. Cliquez ensuite sur le bouton Save pour valider votre saisie.

La fenêtre principale de Xcode apparaît à l'écran ; commencez par parcourir les fichiers automatiquement générés et associés à votre projet. Sur le volet gauche de la fenêtre figurent tous les groupes et les fichiers correspondant à votre application. Vous pouvez le confirmer en parcourant le dossier de stockage de votre projet dans le Finder : il s'agit ici d'une vue logique qui ne correspond pas à l'arborescence exacte des fichiers liés à votre application sur votre disque dur. À la manière d'une fenêtre du Finder, vous parcourez une série de groupes et de raccourcis que vous étendez à loisir et dont la racine correspond au nom de votre application – "Bienvenue" dans notre exemple. En cliquant sur l'un des fichiers, son contenu apparaît dans la fenêtre centrale de Xcode : c'est à cet emplacement que vous saisirez directement le code source de votre projet.

Figure 3.3

La fenêtre principale de Xcode après la création de notre premier projet.



Structure générale de tous les projets pour iPhone

Cliquez sur le petit triangle précédant le nom de votre application, dans le volet gauche de Xcode : vous découvrirez les cinq dossiers indispensables à votre projet. Passons leur rôle en revue.

Classes. Il s'agit du groupe essentiel de votre application : il contient toutes les classes en Objective-C que vous avez développées. C'est d'ailleurs le seul groupe de Xcode qui correspond réellement à un dossier physique stocké sur votre disque dur, dans le répertoire associé à votre projet. Vous pouvez librement y créer des sous-dossiers afin d'agencer au mieux votre application et d'y voir plus clair au cours du développement et du débogage.

Other Sources. Ce groupe réunit tous les scripts et codes source qui ne correspondent pas à des classes en Objective-C. Par défaut, Xcode crée automatiquement deux fichiers indispensables :

- **Bienvenue_Prefix.pch.** L'extension de ce fichier désigne un "en-tête précompilé" (*pre-compiled-header*, .pch), soit la liste exhaustive de toutes les bibliothèques et des frameworks sur lesquels s'appuie votre projet. En précompilant ces éléments immuables constituant votre application, Xcode vous fait gagner un temps précieux au moment de la compilation de votre projet : il ne se préoccupe plus que des classes que vous avez créées vous-mêmes. Par défaut, cet en-tête importe les deux frameworks essentiels du développement sous Cocoa Touch : *Foundation* et *UIKit*.
- **main.m.** N'oubliez jamais qu'Objective-C est une surcouche du langage C ; ce fichier définit ainsi la méthode principale `main()` de votre application. Là encore, vous n'avez pas à intervenir sur ce fichier à moins de modifier les arguments de cette méthode. Nous ne nous y attèlerons que dans les projets de plus grande envergure, qui nécessitent un traitement particulier.

Resources. Ce groupe contient tous les éléments indispensables à votre application qui ne correspondent pas à du code source. Vous y grefferez notamment les images, sons, vidéos, textes et fichiers de configuration associés à votre projet, à commencer par l'icône qui lui est propre. Là encore, vous pouvez librement agencer ce groupe comme bon vous semble et créer des sous-dossiers associés à chaque type de fichiers. D'ailleurs, si vous déplacez l'un de ces éléments dans la fenêtre de Xcode, le fichier correspondant stocké sur votre disque dur n'est pas modifié pour autant : Xcode dispose de son propre système de classement interne et vous évite de vous préoccuper de l'emplacement de stockage de toutes vos ressources. Trois éléments indispensables y figurent d'ores et déjà :

- **BienvenueViewController.xib.** Il s'agit du fichier définissant l'interface graphique de votre application (sa "vue"), composé par Interface Builder. Il n'est pas possible de l'éditer manuellement dans Xcode mais vous pouvez en revanche l'ouvrir dans un éditeur de texte, comme TextEdit : vous découvrez alors qu'il s'agit d'un fichier XML décrivant l'interface à travers une série de balises propriétaires. Nous y reviendrons largement dans la suite de ce chapitre.

- **MainWindow.xib.** Dans le même ordre d'idée, ce fichier décrit la vue de la fenêtre principale de votre application. Là encore, il s'agit d'un fichier lié à Interface Builder, que vous modifierez directement à travers cet utilitaire.
- **Info.plist.** Cette extension désigne une liste de propriétés (*Property List*, *.plist*) associée à votre projet. Vous pouvez la modifier manuellement à travers Xcode : elle regroupe tous les paramètres généraux de votre application, comme l'emplacement de son icône, son numéro de version ou sa langue par défaut. Nous personnalisons ce fichier au cours de ce chapitre.

Frameworks. Ce dossier regroupe tous les frameworks sur lesquels s'appuie votre application. Par défaut, trois frameworks sont chargés : *UIKit*, *Foundation* et *CoreGraphics*. Envisagez-les comme de grandes bibliothèques regroupant des éléments spécifiques à certaines facettes du développement (le traitement des effets visuels, les interactions avec l'utilisateur, etc.). Les grands studios créent généralement leurs propres frameworks afin d'optimiser et d'accélérer le développement de tous leurs projets, en reprenant régulièrement des éléments déjà créés auparavant sans avoir à réinventer la roue. Comme nous le verrons par la suite, vous avez également la possibilité d'importer des frameworks libres de droits glanés sur le Web, qui seront ainsi liés à votre application. Dans tous les cas, cette architecture permet de n'importer que les frameworks dont votre application a besoin et ainsi de limiter les ressources qu'elle consomme.

Products. Le dernier groupe réunit tous les fichiers créés au moment de la compilation, en particulier *Bienvenue.app* qui correspond à votre application compilée. Comme vous pouvez vous en rendre compte, ce fichier apparaît en rouge dans l'interface de Xcode ; c'est le signe qu'il manque encore à l'appel. Rien de plus normal : nous n'avons pas compilé le moindre code pour l'instant et Xcode s'est contenté de préparer le terrain.

Comme nous l'avons vu, les fichiers indispensables à votre projet ont été créés automatiquement en reprenant son nom pour les identifier clairement. C'est là tout l'intérêt d'un tel environnement de développement, articulé autour du concept de modèles : vous vous épargnez les tâches rébarbatives et vous entamez directement la création de votre projet. Par ailleurs, ce système de classement propre à Xcode témoigne de la puissance d'une architecture où les vues, les contrôleurs et les modèles sont séparés. Vous pouvez ainsi vous préoccuper de la programmation de votre application tout en important une interface que vous avez déjà préparée, ou au contraire vous consacrer uniquement au design de votre application. C'est d'ailleurs précisément ce que nous nous apprêtons à faire : enfiler votre blouse de designer, nous allons réaliser l'interface de notre premier projet.

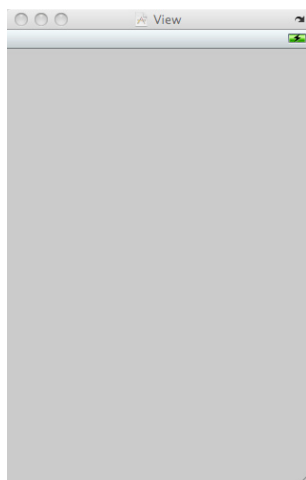
Maîtriser Interface Builder

Dans le volet gauche de Xcode, déroulez l'arborescence en face du groupe Resources et effectuez un double-clic sur le fichier `BienvenueViewController.xib`. Comme nous l'avons vu, il s'agit de la description de l'interface de votre application à travers Interface Builder ; le logiciel s'ouvre d'ailleurs automatiquement en arrière-plan.

Au lancement d'Interface Builder, trois fenêtres prennent place sur votre écran : `BienvenueViewController`, le panneau Library et le canevas View (voir Figure 3.4).

Figure 3.4

L'interface View de notre projet, à l'ouverture d'Interface Builder.



La première fenêtre dresse l'inventaire de tous les éléments présents à l'affichage : chaque icône correspond ainsi à l'instance d'une classe Objective-C. Comme nous le verrons par la suite, toutes ces instances sont triées et laissent ainsi apparaître les dépendances entre les sous-classes. Vous manipulez cette fenêtre à la manière du Finder et vous "déroulez" l'arborescence d'une classe afin de découvrir les éventuelles sous-classes qu'elle héberge. La fenêtre Library contient précisément tous les éléments d'interfaces disponibles dans le SDK de l'iPhone. Par un simple jeu de glisser-déposer vers la fenêtre View, vous créez ainsi des instances de chaque objet correspondant.

Par défaut, vous découvrez pour le moment trois icônes dans la fenêtre `BienvenueViewController` : File's Owner, First Responder et View. Les deux premiers éléments sont indispensables et ne peuvent d'ailleurs pas être supprimés (voir Figure 3.5).

Comme son nom l'indique, File's Owner correspond au "propriétaire" du fichier nib : en modifiant ses paramètres, vous pouvez ainsi lier plusieurs classes ou sous-classes entre elles.

Nous aurons tout le loisir de parcourir plus en détails ces notions de chargement et de liaisons de classes autour d'un exemple concret. Retenez pour l'instant que ce paramètre essentiel

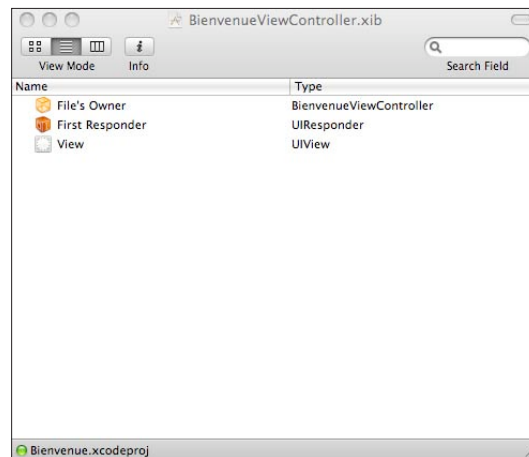
associe un fichier nib spécifique à l'instance d'une classe ou sous-classe. La sous-classe `BienvenueViewController` a été créée automatiquement par Xcode en chargeant ce premier *template* ; son instance recherche immédiatement un fichier `BienvenueViewController.xib`, le charge en mémoire puis en devient naturellement le propriétaire. C'est précisément ce type d'associations que vous pourrez modifier par la suite, à travers l'icône *File's Owner*.

L'icône *First Responder* correspond à l'objet d'interface avec lequel l'utilisateur interagit. Ce "répondeur" évolue sans cesse au cours de ses manipulations : il peut donc s'agir alternativement d'un champ de texte, d'un bouton, d'un onglet et ainsi de suite. L'intérêt d'un tel système est d'offrir un lien direct au programmeur pour communiquer avec cet objet et en récupérer les données, sans prévoir de multiples méthodes "surveillant" chaque contrôleur ou vue d'une application. Là encore, nous aurons largement l'occasion de détailler ce concept autour d'exemples concrets, dès le chapitre suivant.

Au-delà de ces deux icônes indispensables, notre interface ne comprend pour l'instant qu'une seule instance d'objet : `View`. Plus précisément, il s'agit d'une instance de la classe `UIView`, qui définit de manière globale une zone d'affichage à l'écran, avec laquelle l'utilisateur interagit. Vous comprenez ainsi mieux le nom du modèle de base de Xcode pour l'iPhone ("View-Based Application"). Tous les éléments d'interface, comme les boutons, les libellés, les réglettes ou les images correspondent à des instances de sous-classes d'`UIView`. Comme nous le verrons par la suite, il est naturellement possible de définir plusieurs vues au sein d'un même projet ; on parle alors "d'application multivues". C'est d'ailleurs le cas de l'immense majorité des applications pour iPhone, où l'on bascule entre plusieurs vues préparées avec *Interface Builder*, au gré des interactions. Un système de navigation par onglets permet précisément de passer d'une vue à l'autre, par exemple.

Figure 3.5

*La fenêtre
BienvenueViewController
détaille le contenu
du fichier nib.*



Des fichiers nib aux fichiers xib : l'évolution du format d'Interface Builder

Nous l'avons vu au Chapitre 2 : la plupart des outils de développement de Mac OS X, en particulier l'environnement Cocoa, ont été mis en place à l'origine par les ingénieurs de NeXT. Dirigée par Steve Jobs avant son rachat par Apple, cette société a ainsi défini de nombreux standards qui ont persisté à travers les années. Parmi les produits phares de la firme, Interface Builder a été mis en place en 1988. Il a servi d'environnement de prédilection pour développer des applications pour des générations de système, dont NextSTEP, OpenSTEP, Mac OS X et aujourd'hui iPhone OS. Ainsi, les éléments décrivant l'interface graphique d'une application Mac OS étaient stockés à l'origine dans un fichier `.nib` pour "NeXT Interface Builder". Dans la version la plus récente d'Interface Builder, cette extension a été revue et corrigée et on utilise désormais des fichiers `.xib` (*XML Interface Builder*). Sans entrer dans des détails trop techniques, sachez toutefois que ce nouveau format ne supprime pas totalement son auguste prédécesseur : il s'agit essentiellement d'un standard plus clair et plus facile à lire, qui est finalement recompilé en fichier `.nib` lors de l'emballage de votre application. Mais sachez que par coutume et habitude, la plupart des développeurs continuent de parler de "fichiers nib" pour évoquer les éléments d'interface du SDK de l'iPhone. Même la documentation officielle n'a pas envisagé de corriger ce réflexe !

Ajouter des objets avec Interface Builder

Tout l'intérêt d'Interface Builder réside dans son mode de création intuitif et cohérent : vous visualisez simplement les instances de vos classes et vous manipulez une bibliothèque riche d'objets. Nous l'avons vu : tous les éléments d'interface sont des instances de sous-classes d'`UIView` ; si vous avez naturellement la possibilité d'instancier de tels objets par des lignes de code, sous Xcode, il devient vite pénible de personnaliser leurs moindres attributs de cette manière. Imaginez un simple bouton à disposer sur la vue de votre application. En une ligne d'Objective-C, vous pouvez créer une instance de la classe `UIButton`, elle-même sous-classe d'`UIView`. Mais comment parviendrez-vous à placer précisément le bouton sur l'interface, au pixel près, et à régler ses dimensions à travers le code source, sans multiplier les essais successifs ?

La fenêtre View d'Interface Builder résout ce problème : elle correspond à la représentation graphique de l'instance View apparaissant dans le fichier nib (voir Figure 3.6). Par de simples jeux de glisser-déposer depuis la bibliothèque, vous créez ainsi des instances de tous les objets Cocoa Touch supportés par Interface Builder avant de les redimensionner ou de modifier leurs propriétés.

Plus précisément, les objets figurant dans la bibliothèque proviennent tous du framework UIKit, qui est associé à votre projet dès sa création (voir Figure 3.7). Rappelez-vous : ce framework est mentionné dans le volet gauche de Xcode, après la création de notre projet.

Figure 3.6

La fenêtre View correspond à la représentation visuelle de l'instance View. Vous y définissez directement l'interface de votre projet.

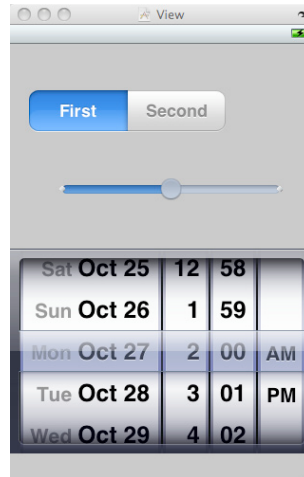


Figure 3.7

La bibliothèque d'objets du framework UIKit de l'iPhone.



Mieux encore : Interface Builder gère de manière naturelle les imbrications et héritages de classes et d'objets. Dans de nombreux cas de figure, il est enfin impossible de créer l'instance d'une sous-classe si vous n'avez pas instancié la classe mère au préalable. Par exemple, vous ne pouvez pas créer un onglet spécifique au sein de votre application sans

avoir déposé une "barre d'onglets" auparavant, prête à l'héberger. Puisque View correspond à la vue de notre interface utilisateur, les objets sous-jacents s'appellent des "sous-vues". À l'inverse de notre exemple, certains objets ne tolèrent pas de sous-vues. Dans tous les cas, il est donc impossible de commettre de telles erreurs avec Interface Builder ; l'application empêche tout simplement de glisser-déposer certains objets si l'opération est interdite. Interface Builder supervise automatiquement l'héritage entre les instances de classes et d'objets.

Figure 3.8

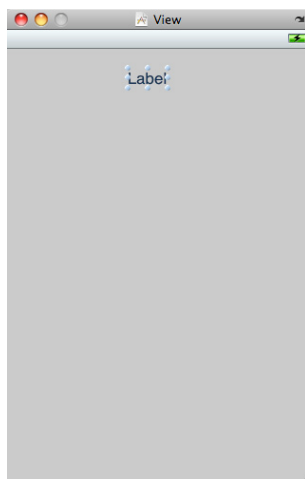
Tous les écrans de l'iPhone sont constitués d'éléments accessibles dans la bibliothèque d'objets d'Interface Builder, à l'image des Réglages généraux, qui réunissent des listes, des libellés et des icônes.



Pour comprendre le fonctionnement général d'Interface Builder, déposons un premier objet sur la fenêtre View. Parcourez la bibliothèque figurant sur la droite de l'écran et recherchez l'objet Label. Il s'agit d'un simple "libellé", soit un texte statique apparaissant sur l'écran de l'iPhone et n'appelant pas la moindre interaction avec l'utilisateur (voir Figure 3.9).

Figure 3.9

On dépose directement notre libellé sur la vue principale de notre application, à travers Interface Builder.



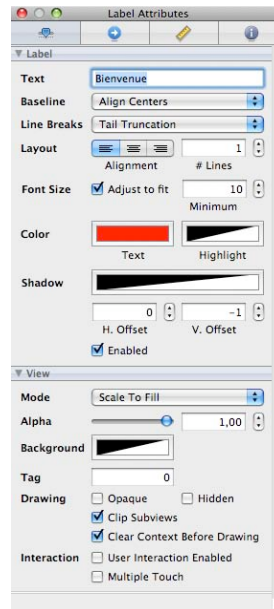
Sélectionnez cet objet, maintenez le bouton gauche de la souris enfoncé et relâchez-le au-dessus de la fenêtre View : un simple champ encadré de poignées apparaît sur l'interface utilisateur. Effectuez un double-clic sur l'objet pour modifier le texte apparaissant par défaut et redimensionnez l'élément ou déplacez-le comme bon vous semble. Pour résumer, nous avons ainsi créé en quelques secondes une instance de la classe `UILabel` ; elle devient une sous-vue de View et vous apercevez son icône dans la fenêtre `BienvenueViewController.xib`, en déroulant l'arborescence View. Bravo : votre premier projet prend forme !

Tout l'intérêt de ce type de design consiste à simplifier au maximum les étapes nécessaires. Vous bénéficiez ainsi d'un éditeur *wysiwyg*¹ et vous visualisez directement l'interface utilisateur, telle qu'elle apparaîtra sur l'iPhone de vos "clients".

Pour vous en persuader, déroulez le menu Tools et cliquez sur Inspector, après avoir sélectionné le libellé. Dans la fenêtre qui apparaît, vous reconnaissez les options traditionnelles d'un éditeur de texte (voir Figure 3.10).

Figure 3.10

On définit simplement les propriétés de notre libellé, à l'aide de l'outil Inspector d'Interface Builder.



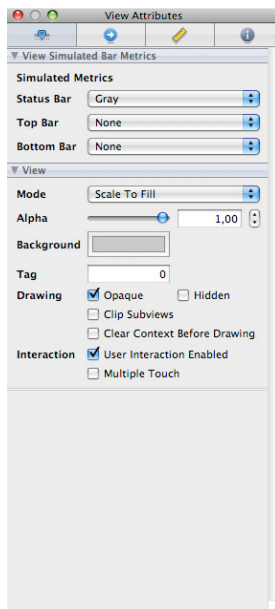
Vous pouvez ainsi changer la couleur et la taille du texte, son alignement, la portée de son ombre ou son opacité par exemple. Multipliez les essais afin d'aboutir au résultat escompté. Vous pouvez invoquer l'outil Inspector pour chaque objet figurant sur l'interface utilisateur,

1. Acronyme de *what you see is what you get* : ce que vous voyez à l'écran correspond au résultat obtenu.

y compris l'instance `View` en elle-même (voir Figure 3.11). Dans ce cas, vous avez essentiellement la possibilité de modifier la couleur d'arrière-plan.

Figure 3.11

L'outil Inspector prend en charge tous les objets de l'interface utilisateur, même la vue principale.



Ce premier essai vous a convaincu de la puissance d'Interface Builder ? Parcourez tous les objets du framework `UIKit` figurant dans la bibliothèque et essayez de les déposer sur la vue principale de votre interface. Vous vérifiez ainsi la faisabilité de votre projet, tel que vous l'aviez esquissé sur papier au Chapitre 1. Selon toutes vraisemblances, de très nombreuses idées nouvelles jailliront spontanément au cours de l'expérience.

Les objets de la bibliothèque d'Interface Builder

On distingue essentiellement trois catégories d'objets `UIKit`, triées dans la bibliothèque d'Interface Builder : les conteneurs de données (*Data Views*), les objets de saisies (*Input & Values*) et les fenêtres et systèmes de navigation (*Windows, Views & Bars*). Voici les principales classes correspondantes :

Les conteneurs de données. Tableaux puisant leur contenu depuis une base de données, images, intégration du navigateur web ou vues défilant automatiquement : tous ces objets servent à agencer au mieux les données de votre application.

- **Table View.** La vue "Table" met en place un tableau visant à accueillir des informations, à la manière du carnet de contacts de l'iPhone par exemple.

- **Table View Cell.** Après avoir créé un tableau, vous ajoutez des cellules individuelles afin de soutenir les données.
- **Image View.** Cet objet sert à charger des images, dans des formats standard. Si l'iPhone supporte de multiples formats, notamment le JPEG, privilégiez dans la mesure du possible les images au format PNG : Xcode parvient à les compresser au maximum au moment de la compilation.
- **Scroll View.** Cette vue crée un large conteneur qui supporte la navigation "au doigt" si son contenu dépasse la hauteur d'un écran. C'est notamment le cas des conditions générales auxquelles vous accédez dans les réglages de votre iPhone : le texte s'étend sur des centaines de lignes, que vous parcourez librement en mimant le comportement d'un ascenseur au doigt.
- **Web View.** Cette vue intègre un navigateur web au sein de votre interface utilisateur : il supporte même les boutons Précédent et Suivant.
- **Text View.** Cette vue permet d'afficher un simple champ de texte multilignes.
- **Picker View.** Cet objet crée une "roulette" que vous complétez par des données qui vous sont propres. C'est notamment le cas de l'horloge ou du réveil de l'iPhone.
- **Date Picker.** Cet objet reprend la vue précédente en l'initialisant avec des dates et des calendriers. Idéal pour mettre rapidement en place une application d'agenda.

La saisie de valeurs. Que vaut une application pour iPhone si vous ne pouvez pas interagir avec elle ? Sur le terminal mobile d'Apple, la saisie de données revêt des formes très diverses, du simple champ de texte faisant apparaître un clavier virtuel à l'originale "roulette" où des valeurs défilent et se sélectionnent du bout des doigts. En créant l'instance de telles classes, vous définissez des objets qui renvoient des données à l'application ; là encore, Interface Builder vous fait gagner un temps précieux en facilitant la configuration de tels mécanismes.

- **Segmented Controls.** Il s'agit d'un simple bouton permettant de basculer entre deux vues. Chaque "côté" du bouton peut contenir du texte ou des images, afin d'indiquer à l'utilisateur les options à sa disposition. En revanche, il n'est pas possible de superposer les deux types de données.
- **Label.** Cet objet est un simple champ de texte statique, généralement très court. Nous l'avons déjà utilisé dans notre premier projet.
- **Round Rect Button.** Cet objet crée un bouton rectangulaire que l'utilisateur peut presser au doigt.
- **TextField.** Cet objet correspond à un champ de texte saisissable par l'utilisateur : après l'avoir sélectionné, un clavier virtuel apparaît à l'écran.

- **Switch.** Cet objet crée un interrupteur visant à activer ou désactiver une option. On en retrouve de nombreuses occurrences dans les réglages de l'iPhone ou de la plupart des applications.
- **Slider.** Il s'agit d'une "réglette" permettant à l'utilisateur de choisir une valeur à travers une barre horizontale, qu'il manipule au doigt. Attention toutefois : n'oubliez pas que l'utilisateur ne peut pas forcément choisir une valeur très précise par son biais. En revanche, c'est un outil idéal pour moduler le niveau d'un paramètre.
- **Progress View.** Cet objet crée un indicateur de progression : il affiche l'état d'avancement d'une opération en cours, comme le chargement ou l'enregistrement de paramètres par exemple.
- **Activity Indicator View.** Dans le même ordre d'idée, cet "indicateur d'activité" renvoie à l'utilisateur un signal lui indiquant qu'une opération est en cours.
- **Page Control.** Cet objet ajoute une série de "points blancs", en bas de l'écran, permettant de basculer d'une "page" à l'autre. C'est l'élément qui figure en bas de l'écran de l'iPhone et qui vous permet de naviguer parmi les applications que vous avez installées.

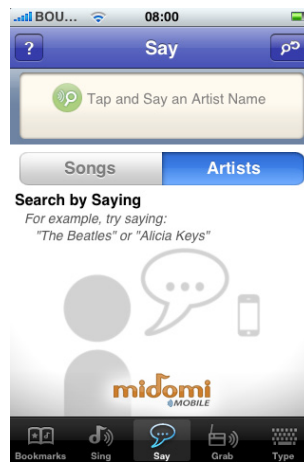
Les systèmes de navigation. Sobre et épuré, l'écran de l'iPhone n'est pourtant jamais dépourvu d'un système de navigation, à commencer par la barre d'informations en haut de l'affichage, qui égrène des détails sur votre opérateur téléphonique, le niveau de batterie ou la puissance du signal WiFi. Barres d'outils, onglets et menus : vous avez la possibilité de reproduire un tel système pour soutenir votre application.

- **Window.** Il s'agit de l'élément de plus grand ensemble parmi les classes d'UIKit. Une telle "fenêtre" regroupe plusieurs vues et votre application peut elle-même compter plusieurs fenêtres.
- **View.** C'est l'élément créé par défaut à l'aide de notre *template* : vous déposez des objets sur une vue, qui deviennent alors des sous-vues. Les applications "multivues" basculent ainsi entre plusieurs vues préparées à l'avance sous Interface Builder.
- **Search Bar.** Spécifique à certaines applications, cette "barre de recherche" apparaît en haut à droite et permet à l'utilisateur d'exécuter des requêtes en faisant apparaître un clavier virtuel.
- **Navigation Bar.** Signifiant littéralement "barre de navigation", cet objet soutient plusieurs sous-éléments qui décrivent ainsi le système de navigation de votre application.
- **Navigation Item.** Cet objet s'imbrique dans une barre de navigation et correspond à l'une des options offertes à l'utilisateur.
- **Toolbar.** La barre d'outils figure en bas de l'écran et comprend plusieurs éléments permettant d'exécuter des comportements spécifiques.
- **Bar Button Item.** Il s'agit de l'un des éléments de la barre d'outils.

- **Flexible Space Bar Button Item.** Lorsque vous associez plusieurs éléments au sein d'une barre d'outils, cet objet correspond à un espace formaté automatiquement entre deux icônes. C'est un objet essentiel pour gérer les modes d'orientation en Portrait ou Paysage : les icônes s'agencent harmonieusement, sans décaler l'affichage pour autant.
- **Fixed Space Bar Button Item.** Similaire à l'élément précédent, cet objet correspond à un espace insécable entre deux options de la barre d'outils. C'est un objet important pour designer une barre d'outils de manière précise : vous pouvez ainsi décaler une option de quelques pixels vers la droite, et la maintenir à cet emplacement quelle que soit l'orientation de l'iPhone.
- **Tab Bar.** Il s'agit du conteneur hébergeant plusieurs onglets, dans le cadre d'un système de navigation.
- **Tab Bar Item.** Associé à l'objet précédent, cet élément correspond à l'un des onglets de la barre.

Figure 3.12

Un système de navigation simple et sobre, à l'aide d'onglets.



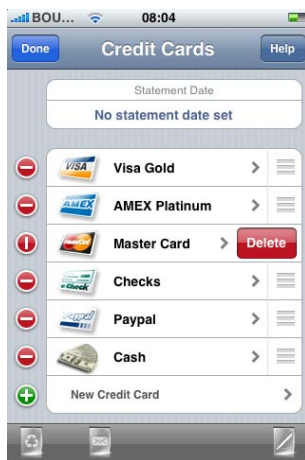
Essayez de déposer sur la fenêtre View un ou plusieurs de ces éléments, puis ouvrez l'outil Inspector afin d'en parcourir les propriétés. Vous créez ainsi très facilement des instances de classes du framework UIKit et vous étoffez simplement l'interface de votre application. Comme nous l'avons vu précédemment, Xcode supporte indifféremment le développement d'applications pour iPhone ou Mac OS X – dans ce dernier cas, ce sont des objets du framework AppKit qui figurent dans la bibliothèque. Le principe est toutefois le même : si l'expérience vous séduit, vous serez en mesure de développer rapidement des applications Cocoa pour Mac OS X... ou tout du moins leur interface graphique, à l'aide d'un logiciel aussi clair et complet qu'Interface Builder.

Figure 3.13

L'interface graphique de notre premier projet, après avoir ajouté d'autres éléments de la bibliothèque.

**Figure 3.14**

Lorsque vous créez des interfaces complexes, votre rôle de designer consiste à veiller à la cohérence de l'interface.



Tester et déboguer votre premier projet

Retournez dans la fenêtre principale de Xcode, à partir de laquelle vous aviez ouvert le fichier `BienvenueViewController.xib`. Vous retrouvez tous les fichiers qui composent votre projet. Jusqu'à présent, nous avons essentiellement réalisé l'interface graphique de notre application en ajoutant un libellé qui fera office d'écran d'ouverture pour un projet de plus grande envergure, que nous compléterons au chapitre suivant. Sachez que dans le jargon des développeurs, on évoque ce type d'écran sous le nom de "Splash Screen" ; il s'agit d'un affichage en plein écran, qui présente rapidement l'application et son intérêt. Nous verrons que

vous pourrez utiliser ce bref message pour faire patienter l'utilisateur pendant le chargement de l'application en elle-même !

Sachez par ailleurs que plusieurs applications pour iPhone ne reposent que sur la conception d'une interface graphique soignée. C'est notamment le cas des nombreuses "lampes de poche", souvent gratuites sur l'App Store. En manipulant Interface Builder, vous avez percé à jour leur grand secret de fabrication : ces applications ne s'articulent qu'autour d'une simple instance de l'objet View dont l'arrière-plan a été réglé en blanc avec une luminosité maximale. Chut : ne le répétez pas !

Il est maintenant l'heure de vérifier la cohérence de notre interface et de nous assurer de son succès grâce au Simulateur d'iPhone. Déroulez le menu File et cliquez sur Save pour enregistrer vos modifications.

Compiler votre projet

Comme nous l'avons vu, Xcode facilite largement la compilation de votre projet, notamment grâce à "l'en-tête précompilé" associé à votre application qui résume toutes les bibliothèques et les frameworks qui lui sont associés. C'est un choix heureux : dans le cadre du développement d'applications plus ambitieuses, où vous programmez des comportements et des événements précis, vous recompileriez inlassablement votre projet des dizaines de fois. C'est en effet absolument nécessaire pour tester la réaction de votre programme dans des conditions réelles !

Déroulez le menu Build de Xcode et cliquez sur Build and Run ("compiler et exécuter").

Figure 3.15

Le résultat après la première compilation de Xcode, avec le Simulateur d'iPhone.



Si vous n'aviez pas enregistré les modifications apportées au fichier `.xib`, Xcode vous le propose à ce moment. Quelques secondes plus tard, le fichier `Bienvenue.app` est prêt et le résultat apparaît dans le Simulateur d'iPhone. Notre libellé figure bien à l'endroit désiré et le rendu est exactement identique à la fenêtre View que nous manipulions sous Interface Builder. Si vous avez créé des champs supplémentaires, notamment des champs de saisie, c'est le moment où jamais de vérifier leur comportement dans le Simulateur. Vous avez ainsi la possibilité de tester le fonctionnement de la plupart des objets issus du framework UIKit.

Modifier les paramètres linguistiques du Simulateur d'iPhone

Par défaut, le Simulateur d'iPhone est en anglais même si ses menus généraux apparaissent en français, dans la barre supérieure de Mac OS X. S'il est tout de même intéressant de visualiser le comportement de votre application pour un public anglophone (nous évoquerons par ailleurs l'intérêt de programmer directement une application dans cette langue, afin d'élargir votre audience potentielle), vous avez tout intérêt à choisir d'autres paramètres linguistiques : vous profiterez ainsi d'un clavier virtuel AZERTY et non QWERTY.

Pour cela, procédez comme si vous manipuliez un véritable iPhone. Cliquez sur le bouton principal afin de revenir à l'écran général de l'iPhone, puis cliquez sur l'icône Settings. Parcourez ensuite les menus et cliquez sur General, puis sur International et enfin Language. Choisissez le français dans la liste proposée (voir Figure 3.16).

Figure 3.16

Le changement des paramètres linguistiques dans le Simulateur d'iPhone.



Sachez au passage qu'il s'agit d'un "tableau de données", comme nous l'avions vu précédemment dans la bibliothèque d'objets UIKit. Cliquez enfin sur le bouton Done, en haut à droite de l'écran, afin de valider vos changements. L'interface du Simulateur est désormais en français. Et si un utilisateur azerty en valait deux ?

Par ailleurs, n'hésitez pas à charger un autre firmware de l'iPhone en déroulant le menu supérieur du Simulateur, dans la barre d'outils de Mac OS X. Vous pouvez ainsi "émuler" plusieurs configurations de l'iPhone et vous assurer du rendu de votre projet dans de multiples cas de figure. Un argument idéal à faire valoir dans la description de votre application sous l'App Store.

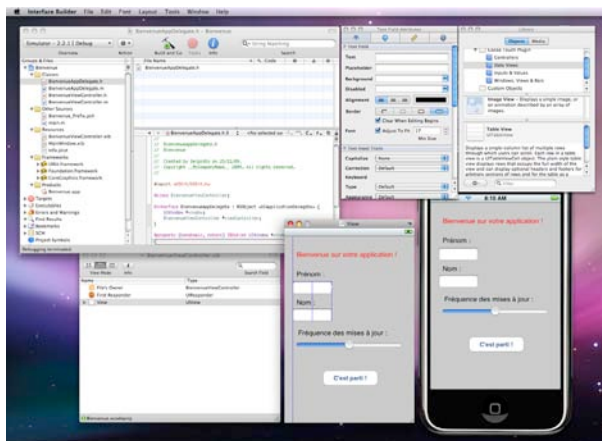
Effectuer un changement, recompiler le projet et vérifier le résultat dans le Simulateur

Afin d'illustrer une réaction qui vous paraîtra rapidement bien naturelle, plaçons-nous dans le contexte suivant : alors que vous étiez si fier de votre interface graphique ou du comportement de votre application, le test à travers le Simulateur d'iPhone a révélé un gros problème. Vous devez le corriger au plus vite. Si le défaut concerne l'interface graphique, ouvrez à nouveau le fichier `BienvenueViewController.xib`, en effectuant un double-clic sur son icône, dans le volet gauche de Xcode. Vous basculez ainsi vers Interface Builder et vous retrouvez la fenêtre View que nous avons mise en place auparavant. Effectuez un changement, en lançant éventuellement l'outil Inspector après avoir cliqué sur l'objet concerné, puis sauvegardez le résultat en déroulant le menu File et en cliquant sur Save.

De retour sous Xcode, déroulez à nouveau le menu Build et cliquez sur Build and Run. Le Simulateur d'iPhone apparaît au premier plan et vous visualisez immédiatement le changement. En fonction de la diagonale de votre écran, il peut être très intéressant d'agencer les fenêtres de Xcode, d'Interface Builder et du Simulateur d'iPhone selon vos propres goûts, afin de basculer rapidement de l'une à l'autre. Comme vous l'avez constaté, il s'agit en effet de trois applications distinctes, dont Xcode représente la "passerelle". Si ces trois outils de développement ne sont pas particulièrement gourmands en ressources dans le cadre de la création d'une application pour iPhone, leur manipulation change du tout au tout entre un iMac 24 pouces et un MacBook Pro 13 pouces. Profitez au mieux du large espace qui vous est accordé sur les modèles hauts de gamme du catalogue d'Apple : vous accélérerez d'autant plus la phase de développement ! Comme nous le verrons par la suite, il sera très intéressant de laisser la fenêtre du débogueur visible pour vérifier les performances et le comportement de votre application, tout en manipulant le Simulateur d'iPhone.

Figure 3.17

N'hésitez pas à agencer au mieux les fenêtres de vos outils de développement, en particulier si vous disposez d'une résolution très large comme sur les iMac ou les MacBook Pro 17 pouces.



Ajouter l'icône de votre application

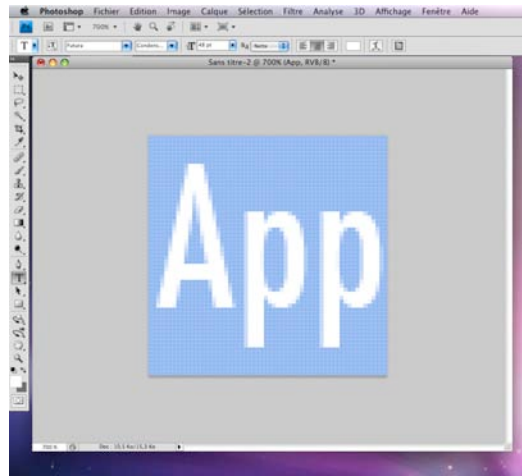
Vous le constatez en manipulant le Simulateur d'iPhone : en cliquant sur le bouton principal afin de revenir au menu général, votre application "Bienvenue" ne dispose que d'une icône blanche. Ce n'est pas très vendeur, n'est-ce pas ? Même si vous n'envisagez pas d'entamer une carrière de développeur professionnel, vous avez tout intérêt à personnaliser cette icône. Non seulement le procédé est simple et confère un certain cachet à votre application, mais il nous permet également d'illustrer la modification de la liste des propriétés associée aux projets pour iPhone.

Dans votre éditeur graphique fétiche, créez un nouveau fichier de 57 × 57 pixels. Contrairement à ce que l'écran principal de l'iPhone laisse à penser, vous n'avez pas besoin de préparer vous-même un rectangle aux bords arrondis avec un fin liseré de lumière en son centre. Ces légères retouches cosmétiques sont directement appliquées à votre icône carrée, afin d'uniformiser les icônes de toutes les applications.

Un guide exhaustif sur la création d'une icône et le message "implicite" qu'elle véhicule dans l'esprit des utilisateurs dépasse largement le cadre de cet ouvrage. Si vous manquez d'inspiration, commencez par parcourir les icônes de vos applications préférées. À quoi tient leur succès ? Le plus souvent, il s'agit d'un subtil jeu de suggestion et d'évocation, en restant le plus sobre possible. L'icône de Facebook, par exemple, s'inscrit sur le fond bleu caractéristique et ne reprend que la première lettre du service : tout le monde comprend de quoi il s'agit. Si vous ne jouissez pas encore de la même popularité, pas de panique ; il vous suffit pour l'instant de rester simple et de réfléchir au message principal que véhicule votre application. Partez d'un fond uni et ajoutez un simple titre dans une police très lisible. Évitez d'importer de trop grands éléments ou de véritables photographies, dont la lecture serait pénalisée à ces faibles dimensions.

Figure 3.18

La création de l'icône de l'application sous Adobe Photoshop.



Vous êtes fin prêt ? Ouvrez une fenêtre du Finder et déplacez directement votre icône vers le répertoire Resources de Xcode correspondant à votre projet. Vous avez également la possibilité de dérouler le menu Project de Xcode puis de cliquer sur Add to project et de sélectionner votre icône dans le navigateur. L'outil vous demande ensuite s'il doit copier le fichier dans le répertoire de votre projet ou s'il doit se contenter d'une référence au fichier d'origine. Cochez la case Copy items into destination group's folder, puis cliquez sur le bouton Add ; vous disposez ainsi d'une copie associée au projet et vous n'avez pas besoin de vérifier si le fichier est toujours présent dans un autre répertoire de votre disque dur. Le nom de votre image importe peu : nous allons la définir dans la liste de propriétés.

Toujours dans le répertoire Resources de votre projet Xcode, sélectionnez le fichier Info.plist. Son contenu apparaît sous la forme d'un tableau, sur la droite de l'écran. Parmi les champs disponibles, sélectionnez Icon file et effectuez un double-clic sur le champ vierge à sa droite. Saisissez alors le nom de votre icône (icone.png par exemple) et pressez la touche Entrée pour valider (voir Figure 3.19).

Nous aurons tout le loisir de compléter les autres champs par la suite : sachez qu'ils définissent entre autres le nom du développeur, la version de l'application ou sa langue d'origine.

L'icône est désormais associée à notre application : sauvegardez le projet, puis déroulez le menu Build et cliquez sur Build and Run. Le Simulateur d'iPhone apparaît à nouveau à l'écran : pressez la touche centrale afin de revenir au menu général ; l'application "Bienvenue" dispose désormais de sa propre icône (voir Figure 3.20). Vous constatez au passage que votre icône carrée revêt maintenant des coins arrondis et qu'un léger effet de surbrillance s'applique sur sa partie supérieure. Il est donc inutile d'essayer de mimer un tel effet avec

vosre logiciel de création graphique ; le compilateur de Xcode s'en charge automatiquement à vosre place.

Figure 3.19

L'ajout de l'icône à nosre projet, sous Xcode.

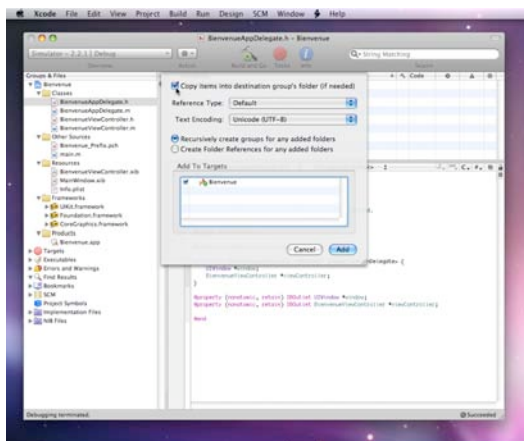


Figure 3.20

Nosre application dispose désormais de sa propre icône et on l'identifie au premier coup d'œil.



Pour aller plus loin : importer des projets officiels d'Apple

Nosre premier exemple est désormais fin prêt et fera office d'écran de bienvenue pour nosre projet de plus grande envergure, que nous compléterons au chapitre suivant. Mais avant d'aller plus loin, que diriez-vous de tester des applications professionnelles, disponibles sur le Dev Center d'Apple ? C'est l'occasion idéale de vous familiariser avec l'agencement de nombreux fichiers au sein de Xcode et surtout de découvrir les outils de débogage à vosre disposition. Pour cela, vous devez disposer d'un compte Apple ; il s'agit généralement du

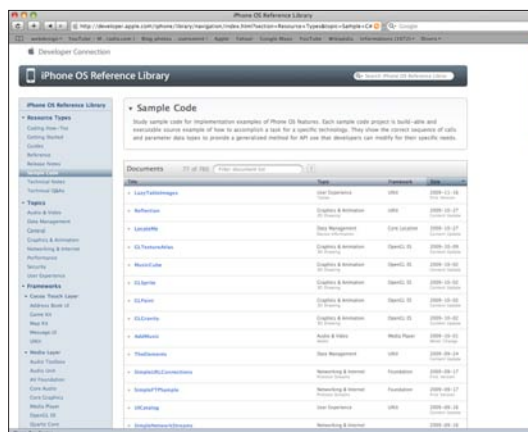
compte que vous utilisez d'ores et déjà sous iTunes ou lors du téléchargement du SDK de l'iPhone.

Avec votre navigateur Internet préféré, rendez-vous à l'adresse <http://developer.apple.com/iphone>, puis cliquez sur le lien Sample Code, à la section Resources for iPhone OS 3.1. Le Centre de développement officiel de l'iPhone comprend de nombreuses vidéos, des articles techniques ainsi que le guide de programmation exhaustif, où toutes les classes sont décortiquées à la loupe. Sur la colonne de gauche, cliquez à nouveau sur le lien Sample Code. Vous êtes alors invité à saisir le précieux sésame associé à votre compte Apple.

Une longue liste comptant plusieurs centaines de projets complets apparaît à l'écran. Parcourez les derniers ajouts et cliquez éventuellement sur les colonnes "Topic" et "Framework" afin de choisir un projet spécifique, relatif aux bibliothèques graphiques ou audio par exemple. Vous pouvez également utiliser le moteur de recherche, en haut de la liste, pour sélectionner une application précise. Cliquez enfin sur l'élément de votre choix. Dans notre exemple, nous choisissons *BubbleLevel*, une application gratuite faisant office de niveau à bulle. Pour chaque projet, le centre de développement vous indique le framework principalement concerné du SDK : ici, il s'agit d'UIKit et cet exemple s'attarde donc essentiellement sur l'interface graphique. Sur la nouvelle page qui apparaît, cliquez sur le lien Download Sample Code. Le téléchargement d'une archive au format .zip démarre. Au passage, vous remarquez la version de l'iPhone OS concernée par cette application. Dans cet exemple, il s'agit de la version 2.0 et vous devrez ainsi modifier les propriétés de Xcode pour épouser un tel système. Dans la fenêtre principale de l'application, déroulez le menu Simulator et choisissez Simulator – iPhone OS 2.0.

Figure 3.21

La liste des projets gratuits sur le Dev Center d'Apple.

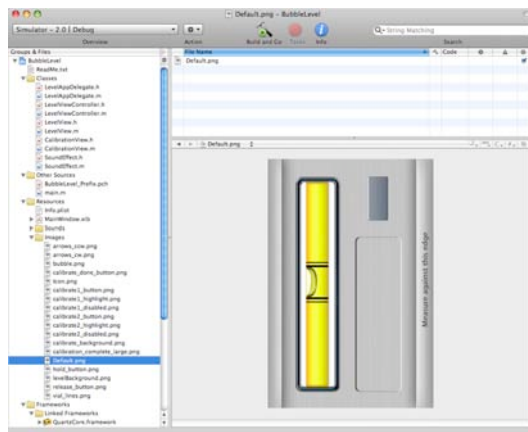


À l'issue du téléchargement, effectuez un double-clic sur l'archive pour l'extraire. Retournez enfin sur Xcode, déroulez le menu File et cliquez sur Open. Vous devez ensuite sélectionner

tionner le fichier associé au projet : il est présent dans l'archive que vous avez téléchargée et s'achève par l'extension `.xcodeproj` (BubbleLevel.xcodeproj dans notre exemple).

Figure 3.22

Le projet de "niveau à bulle" chargé dans Xcode : vous remarquez le nombre important de fichiers et de classes qui lui sont associés.



Vous retrouvez ainsi la disposition habituelle de Xcode et tous les groupes et les fichiers correspondants figurent dans le volet gauche de l'application. Premier constat : le répertoire Resources est beaucoup plus étoffé que dans notre premier exemple et vous constatez ainsi qu'il est possible de créer autant de sous-dossiers que nécessaire. Vous découvrez également que de nombreuses classes personnalisées ont été ajoutées par l'auteur du projet, comme `SoundEffect.h` ou `CalibrationView.h` dans cet exemple de niveau à bulle. Enfin, des frameworks inédits sont importés, en fonction des besoins de chaque application : ici, c'est `AudioToolbox.framework` qui est chargé au-delà des frameworks habituels.

Découvrir les outils de débogage

À la manière d'un projet que vous auriez créé vous-même, vous avez parfaitement la possibilité de compiler cette application d'exemple : déroulez le menu Build et cliquez sur Build and Run afin de lancer le Simulateur d'iPhone. Mieux encore : compte tenu du nombre important de variables, de classes et de ressources exploitées dans ce projet d'exemple, pourquoi n'en profiterions-nous pas pour parcourir les outils de débogage de Xcode ? Déroulez le menu Build, puis cliquez sur Build and Debug afin de lancer cet outil principal. Un "enregistrement" de toutes les interactions et des entrées/sorties de l'utilisateur débute alors. Lorsque vous fermez le Simulateur d'iPhone, un compte-rendu exhaustif vous est présenté. Vous constatez d'ailleurs que la barre d'état de Xcode indique "GDB:Running", ce qui indique que le débogueur est en cours d'enregistrement (voir Figure 3.23).

À l'issue de vos manipulations, déroulez le menu Run de Xcode et cliquez sur Debugger. Une nouvelle fenêtre apparaît au premier plan et dresse la liste de toutes les variables, des

classes et des objets qui ont évolué au fil de l'exécution de l'application. Certes, le projet créé par le Dev Center d'Apple n'exploite pas nécessairement des objets que vous connaissez déjà, mais vous découvrirez ainsi qu'il est possible de valider le comportement de n'importe quelle application à travers ce puissant outil de débogage.

Figure 3.23

L'exécution du programme d'exemple dans le Simulateur d'iPhone.

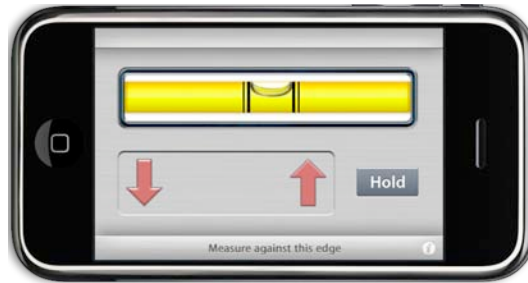
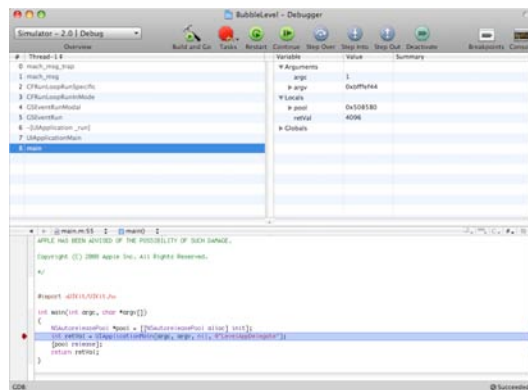


Figure 3.24

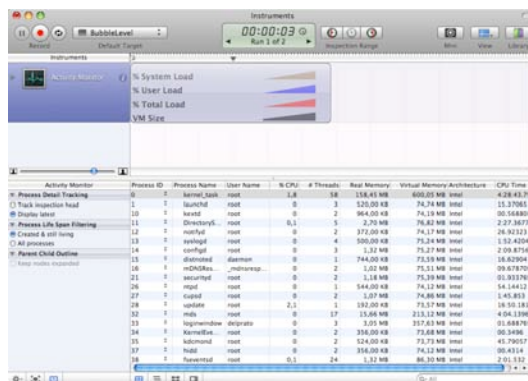
Le débogueur principal de Xcode : les objets chargés figurent en bas à gauche de la fenêtre.



Il existe de nombreux outils supplémentaires visant à valider un aspect spécifique d'une application pour iPhone. Déroulez le menu Run de Xcode puis cliquez sur Start with Performance Tool. Parmi les outils disponibles, essayez en priorité Activity Monitor (moniteur d'activité), Leaks (fuites mémoire) ou Object Allocations (chargement et création des instances de classes et d'objets).

Figure 3.25

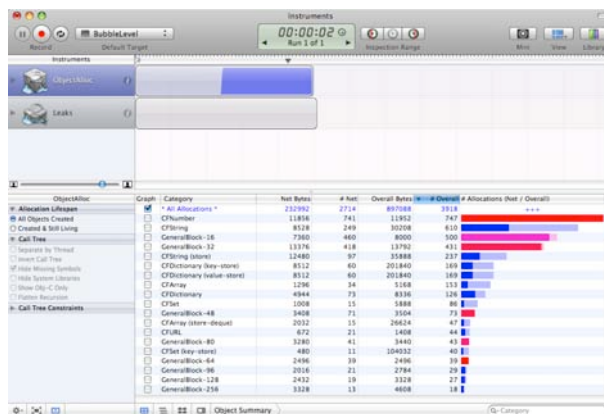
Parmi les outils de débogage, l'Activity Monitor vous renseigne sur la charge du système qu'engendre votre application.



À sa manière, chaque outil vous renseigne sur la complexité de votre programme et sur les éventuelles erreurs que vous avez commises. Nous aurons tout le loisir de les exploiter au maximum à travers nos projets plus ambitieux, dès le chapitre suivant !

Figure 3.26

L'outil Leaks indique clairement les failles de votre application au niveau de la mémoire : rappelez-vous que vous travaillez sur un terminal relativement limité et que vous ne disposez pas nécessairement de ressources infinies !



En route vers le dialogue avec l'utilisateur

Vous maîtrisez désormais les éléments essentiels du SDK et vous connaissez sur le bout des doigts les coulisses de n'importe quelle application pour iPhone. Xcode, Interface Builder et le Simulateur d'iPhone sont les trois pierres angulaires du développement ; vous savez maintenant comment basculer rapidement d'un élément à l'autre, en profitant de toutes les ressources mises à votre disposition. Continuez de tester des projets supplémentaires depuis le Dev Center d'Apple : ils constituent autant d'exercices pour vérifier vos connaissances et mettre en application tous les enseignements de ce chapitre. Testez éventuellement les outils de débogage ; vous constaterez que les résultats diffèrent largement en fonction du

contexte et que les applications les plus ambitieuses consomment des ressources hautement plus importantes que notre modeste exemple.

Inspirons-nous de ces augustes développeurs et augmentons d'un niveau notre parcours d'Objective-C et de Cocoa Touch. Si notre premier projet est bien fonctionnel et constitue une première "vitrine" de vos possibilités, il n'autorise pour l'instant aucune interaction avec l'utilisateur. C'est pourtant l'un des plus grands intérêts des applications pour iPhone et iPod Touch ; conservez les fichiers que vous avez développés jusqu'à présent : ils vont servir d'ouverture à notre première application complète, un quiz visant à tester les connaissances des utilisateurs.



4

Interactions avec vos applications

Au sommaire de ce chapitre

- Le schéma Modèle-Vue-Contrôleur
- Développer la vue et associer des actions aux contrôles
- Les accesseurs et les mutateurs
- Le délégué de l'application
- Finaliser notre projet de quiz
- Pour aller plus loin

À l'issue de notre première exploration d'Objective-C et des outils composant le kit de développement de l'iPhone, nous avons abouti à une application parfaitement fonctionnelle. Son icône figure sur l'écran principal du terminal d'Apple et vous pouvez la lancer pour afficher notre message de bienvenue. Mais voilà : à quoi sert une application aussi statique, qui n'accepte pas la moindre interaction de la part de l'utilisateur ? Vous l'avez constaté à travers votre propre expérience : les applications les plus réussies se manipulent aux doigts et vous offrent un contrôle riche, à travers une série de boutons et d'éléments personnalisés qui déclenchent des actions. L'ergonomie de l'iPhone offre un accès immédiat à l'information et vous permet de plonger littéralement dans tout type d'applications ; l'impressionnante diversité des contrôles multi-touch participe ainsi largement au succès de l'App Store et au charme des applications les plus originales. Mieux encore : c'est un formidable terrain de jeux pour les développeurs, dont la créativité et l'imagination ne sont plus bridés ! Il est de leur ressort d'inventer tout type d'interactions face à un projet, non seulement pour faciliter son utilisation mais aussi pour accroître son intérêt au quotidien.

Reprenons notre exemple de niveau à bulles (*BubbleLevel*), introduit au chapitre précédent. Son principe est simple : afficher à l'écran une valeur, correspondant à l'inclinaison de l'iPhone sur un plan. Pour aboutir à ce résultat, on peut envisager de multiples solutions. Outre le calcul de l'inclinaison à proprement parler, on pourrait imaginer un simple bouton que l'utilisateur presserait pour découvrir cette valeur. Cette méthode n'est toutefois ni très originale ni ergonomique ; elle oblige l'utilisateur à valider un niveau d'arborescence supplémentaire, après avoir lancé l'application, pour obtenir l'information désirée. La solution retenue privilégie donc la simplicité et l'accès immédiat au résultat : la valeur de l'inclinaison s'affiche en permanence dans un libellé et on peut ainsi relever toutes les variations en fonction du déplacement de l'iPhone. Il aura suffi au développeur "d'habiller" ce libellé autour d'une interface qui mime le comportement d'un véritable niveau à bulles pour rendre l'interface plus ergonomique et intuitive.

Figure 4.1

Comparez l'interface simple et épurée du niveau à bulles avec celle, beaucoup plus complexe, de cette liste de tâches personnalisées.



Dans le cadre de vos propres projets, vous devez travailler de la même manière. En revenant au bon vieux papier et crayon, posez-vous sans cesse les questions : "les contrôles de mon application sont-ils simples et immédiatement compréhensibles par tous ?" et "existe-t-il une manière plus rapide d'aboutir au même résultat, en évitant à l'utilisateur de se perdre dans les menus ?". Le dialogue homme-machine et la création d'interfaces graphiques sont deux sujets prépondérants depuis les balbutiements de la programmation informatique. Si les interactions ont évolué et que les contrôles revêtent aujourd'hui d'innombrables aspects, le fond du problème est resté sensiblement le même : vous devez apprendre à servir au mieux l'utilisateur en lui offrant un accès immédiat et intuitif aux mécanismes qui dégagent tout le potentiel de votre application.

Face à l'iPhone, un tel enjeu est particulièrement enthousiasmant. Comme vous l'avez constaté à travers votre première découverte d'Interface Builder, les développeurs disposent d'un très large arsenal de contrôles (boutons, onglets, ascenseurs, roulettes...) permettant d'interagir avec l'utilisateur et l'iPhone offre lui-même un support très souple pour réinventer l'ergonomie de votre application. Certains utilitaires vendus sur l'App Store s'articulent ainsi autour de contrôles jamais vus dans le développement logiciel et vous permettent de valider une action en "secouant" le téléphone, par exemple.

Nous allons explorer toutes ces possibilités autour d'un exemple concret : le développement d'un petit quiz auquel les utilisateurs pourront se prêter entre deux stations de métro.

Figure 4.2

L'interface principale de notre quiz à choix multiple, avec ses libellés et ses quatre boutons de réponse.



Le schéma Modèle-Vue-Contrôleur

Mais avant de passer à la pratique, un petit point théorique s'impose. Le développement pour iPhone s'inscrit dans une logique *Modèle-Vue-Contrôleur* (MVC), un célèbre type d'architecture qui se décline à de très nombreux langages de programmation (C++, Java, Perl, PHP, Python, Ruby...). Il s'agit d'une méthode de conception visant à organiser l'interface homme-machine d'une application. Elle la divise en trois domaines distincts, qui se répondent en permanence :

- Le *modèle* correspond aux données véhiculées par l'application. Il assure leur gestion et offre des méthodes pour les récupérer, les insérer ou les modifier sans se préoccuper de leur présentation. Dans le cas de l'iPhone, il s'agit de l'ensemble des classes contenant les données de votre application.
- La *vue* regroupe tous les éléments graphiques (fenêtres, contrôleurs, etc.) avec lesquels l'utilisateur interagit. Son rôle principal est de présenter les données renvoyées par le modèle, sans effectuer le moindre traitement. La vue met ainsi en place l'interface soutenant les actions de l'utilisateur en les associant à des événements : ces derniers sont alors envoyés au contrôleur.
- Le *contrôleur* correspond à la gestion des événements et des mécanismes qui régissent votre application : il combine le modèle et la vue en les mettant à jour afin de gérer les interactions de l'utilisateur. Si une action implique un changement de données, le contrôleur demande cette modification au modèle et avertit ensuite la vue que les données ont changé pour qu'elle se mette à jour.

L'intérêt d'une telle architecture est de séparer très distinctement ces trois "facettes" du développement, en individualisant les données, la présentation et les traitements. Vous pouvez ainsi plus facilement réexploiter des portions de codes ou mettre à jour votre application. Par exemple, un objet contenant un bouton ne doit pas, en théorie, intégrer le moindre code permettant de gérer les données une fois qu'il a été pressé.

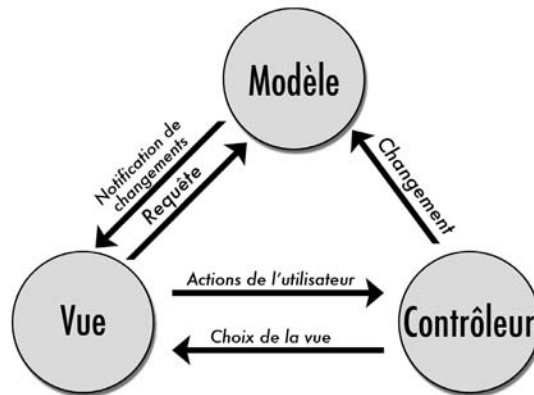
À l'inverse, le code qui gère un enregistrement dans une base de données ne doit pas se préoccuper de l'affichage du résultat. Dans le cas de notre quiz d'exemple, le modèle correspond à l'ensemble des questions et des réponses, la vue contient les boutons que presse l'utilisateur et le contrôleur vérifie la pertinence des réponses en affichant éventuellement la question suivante en cas de succès. Les trois traitements sont nettement séparés et on peut envisager de remettre à jour la série de questions ou la présentation de l'application, sans réécrire entièrement notre code source.

On doit ce type de méthode logique à Trygve Reenskaug¹, qui l'a mise au point en 1979 dans les laboratoires de recherche Xerox PARC. Plus de trente ans plus tard, cette méthode présente toujours le même intérêt dans le cadre du développement avec Cocoa Touch.

Vous créez essentiellement les vues à l'aide d'Interface Builder et vous développez vos modèles à travers les classes Objective-C rédigées dans Xcode. Les contrôleurs correspondent également à des séries de classes : il s'agit le plus souvent de sous-classes de contrôleurs génériques compris dans le framework UIKit, comme `UIViewController` par exemple. Là encore, Apple a pensé aux développeurs et le SDK fourmille de classes prêtes à l'emploi qui vous évitent de réinventer en permanence la roue.

Figure 4.3

Les interactions entre le modèle, la vue et le contrôleur au sein de l'architecture MVC.



Le concept vous paraît encore un peu abstrait ? Pas de panique : nous aurons tout le loisir d'illustrer toutes les ramifications de cette méthode autour d'exemples concrets. Sachez par ailleurs qu'Interface Builder facilite la prise en charge de cette architecture, en vous permettant de relier de manière intuitive votre contrôleur à la vue et au modèle. Puisque le développement d'applications pour iPhone implique des mises à jour régulières afin de contenter votre public, vous avez tout intérêt à souscrire à une telle méthode : non seulement votre code gagne en clarté, mais vous avez aussi la possibilité de mettre à jour plus facilement votre application.

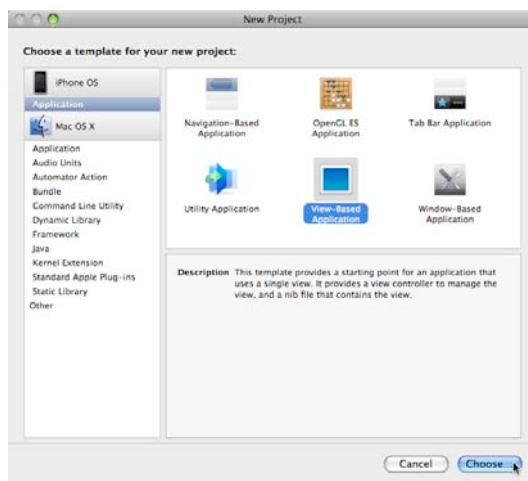
1. Né en 1930, Trygve Reenskaug est l'une des figures de proue de la POO : non seulement il a mis au point le motif de conception MVC, mais il a également développé une série d'outils de développement autour de ce concept et activement participé à l'élaboration du langage de modélisation UML. Sa page personnelle est accessible à l'adresse <http://folk.uio.no/trygver>.

Développer la vue et associer des actions aux contrôles

Même si notre premier essai au chapitre précédent s'était conclu par un succès, nous avons tout intérêt à créer un nouveau projet pour développer notre quiz. Comme vous le savez déjà, la création d'un nouveau projet sous Xcode enregistre automatiquement une série de fichiers autour du nom que vous avez choisi ; vous êtes ainsi assuré d'isoler vos nouveaux éléments dans une structure unique, qui reflète parfaitement votre projet et dont les classes de base sont préparées à votre place.

Figure 4.4

La création de notre projet de Quiz sous Xcode.



Lancez Xcode, déroulez le menu File, puis sélectionnez New Project et choisissez à nouveau le modèle "View-Based Application". Saisissez "Quiz" en guise de nom de projet et validez. Vous découvrez sur le volet gauche de Xcode une série de sous-dossiers et de fichiers prêts à l'emploi. Déroulez en premier lieu le dossier Classes où figurent quatre fichiers essentiels : QuizAppDelegate.h et .m, puis QuizViewController.h et .m. Commencez par cliquer sur QuizViewController.h afin de découvrir son contenu dans la fenêtre principale de Xcode :

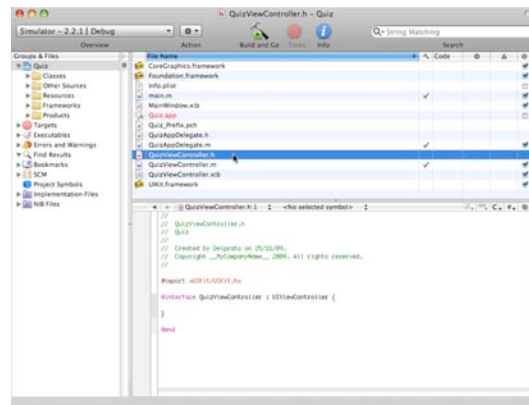
```
# import <UIKit/UIKit.h>
@interface QuizViewController : UIViewController {
}
@end
```

Selon la nomenclature de l'Objective-C, QuizViewController est ici une sous-classe d'UIViewController, l'une des classes génériques visant à définir le comportement du contrôleur de votre application. La sous-classe ne comprend pas pour l'instant le moindre traitement ; il vous reste à définir les actions de votre application entre la série d'accolades.

Réfléchissons plus en détails à notre projet. Dans le cadre d'un quiz, nous souhaitons présenter à l'utilisateur une question et une série de réponses possibles. Toutes ces réponses sont représentées visuellement par quatre boutons, à la manière d'un célèbre jeu télévisé. L'utilisateur indique sa réponse en pressant l'un de ces boutons : notre contrôleur doit donc interagir avec ces éléments, en déduire la réponse du candidat et vérifier si elle est bien correcte. Vous l'avez déjà deviné : on disposera les boutons sur l'interface à l'aide d'Interface Builder, en travaillant sur le fichier nib. La sous-classe `QuizViewController` devra pointer vers ces objets afin de relever le choix de l'utilisateur et de mettre à jour la vue en conséquence. Ce concept de pointeur, absolument essentiel dans la programmation orientée objet, revêt deux aspects en Objective-C : les *outlets* et les *actions*.

Figure 4.5

Les fichiers associés par défaut à notre projet, sous Xcode.



Le concept d'outlets et d'actions

Les *outlets* (littéralement des "issues" ou des "sorties") sont des pointeurs qui servent à désigner un objet figurant dans le nib. Il s'agit d'un simple lien créé entre le contrôleur et l'objet en question. Imaginons que l'on ajoute sous Interface Builder un simple libellé contenant le score du joueur, par exemple. S'il répond correctement à une question, on souhaite mettre à jour ce libellé : en créant un outlet pointant vers cet objet, on est en mesure de modifier depuis le code source la valeur qu'il affiche.

À l'inverse, les éléments composant l'interface du fichier nib peuvent déclencher des méthodes spécifiques dans notre classe contrôleur. On parle alors d'*actions* pour désigner ce type de mécanismes. Par exemple, vous pouvez indiquer sous Interface Builder que lorsque l'utilisateur presse un certain bouton, on exécute une méthode bien précise de notre code source. Les outlets et les actions se répondent donc mutuellement et constituent le principal moyen de développer une application dans le cadre d'une architecture MVC.

Envisageons un exemple de plus grande ampleur, issu du monde de la bureautique, pour synthétiser cette première approche. Un traitement de texte "écoute" en permanence les interactions de l'utilisateur. Il associe une action au bouton Quitter, FermerApplication par exemple. En développant l'interface du traitement de texte, on a créé une liaison entre ce bouton et le contrôleur de l'application. Ce dernier contient la méthode FermerApplication qui effectue une série de traitements successifs : l'enregistrement des paramètres de l'utilisateur puis la fermeture du logiciel. À l'inverse, le code du contrôleur a besoin "d'écouter" les objets présents sur l'interface de l'application, comme une case à cocher ou un champ de texte par exemple. Pour vérifier qu'un champ est rempli, il faut y faire référence depuis le code du contrôleur : on crée ainsi un *outlet* dans le contrôleur, ChampTexte par exemple. Cette fois, la liaison s'effectue du contrôleur vers l'objet d'interface ; à tout moment, on est en mesure de relever la valeur du champ de texte depuis le code du contrôleur. Cet exemple illustre aussi l'intérêt de la méthode MVC : on peut librement modifier l'allure du bouton Quitter sans retoucher les traitements qui lui sont associés... ou l'inverse !

L'application Facebook pour l'iPhone s'inscrit parfaitement dans ce schéma. Les différents boutons de l'interface ont évolué au gré des mises à jour du service web. Dans ses premières versions, l'application ne permettait que de changer le statut de son compte Facebook. Elle s'enrichit aujourd'hui de nombreux contrôleurs supplémentaires, qui visent à ajouter des photos personnelles ou à consulter le statut de ses amis directs (voir Figure 4.6).

Figure 4.6

L'interface de l'application Facebook fourmille de contrôles : ils déclenchent des événements, pris en charge par la logique du contrôleur de l'application.



Le motif de conception MVC a permis aux développeurs d'ajouter progressivement ces contrôleurs et de les lier à de nouvelles actions, sans pour autant réécrire la logique fondamentale de l'application, comme l'authentification auprès des serveurs de Facebook par exemple.

En Objective-C, on définit un outlet de la manière suivante :

```
IBOutlet UIButton *monBouton;  
IBOutlet UILabel *monLibelle;
```

Ici, les variables d'instance `monBouton` et `monLibelle` pointent directement vers des objets appartenant respectivement aux classes `UIButton` et `UILabel`, que l'on aura disposés sur la vue à l'aide d'Interface Builder. Comme vous le verrez par la suite, la création de ce type de liaisons est très intuitive : il vous suffit de manipuler la souris pour voir apparaître toutes les possibilités associées à vos objets dans Interface Builder.

Les actions se définissent dans la classe du contrôleur. Elles visent à "écouter" les interactions de l'utilisateur et à déclencher des mécanismes. Elles correspondent donc à des méthodes que l'on définit à l'aide du mot-clé `IBAction` :

```
- (IBAction)maMethode:(id)sender;
```

Le mot-clé `IBAction` indique que la méthode `maMethode` ne va pas renvoyer la moindre valeur et qu'elle effectue un traitement (à la manière de `void`, dans d'autres langages de programmation). Comme le suggère son préfixe, `IBAction` indique également que la méthode associée est accessible depuis Interface Builder. Il est possible de passer des arguments à la méthode et d'indiquer le nom de l'objet qui a déclenché l'action : c'est le rôle de la seconde portion de cette définition, `(id)sender`. Par exemple, vous pouvez imaginer qu'un bouton spécifique a été pressé et qu'il a lancé une action : cet argument final vous sert à repérer le bouton en question et à personnaliser éventuellement le traitement qui suit dans le code du contrôleur (voir Figure 4.7).

Figure 4.7

Le clavier virtuel de cette application se déclenche en pressant le champ de texte. Une fois la saisie achevée, l'utilisateur renvoie une action vers le contrôleur de l'application.



Si vous n'avez pas besoin de connaître l'objet qui a déclenché l'action, vous pouvez simplifier la déclaration et vous passer de cette portion de code. Ainsi, le code suivant fonctionne parfaitement :

```
- (IBAction)maMethode;
```

Comme vous le constaterez en testant les différents projets gratuits disponibles sur le Centre de développement d'Apple (reportez-vous au chapitre précédent), de nombreux développeurs ont pris l'habitude de mentionner cet argument `sender` sans parfois l'utiliser en retour. Il s'agit d'un lointain héritage de Cocoa, où les méthodes associées aux actions devaient impérativement présenter un tel argument. S'il s'agit de votre première expérience de programmation avec Cocoa Touch, prenez plutôt l'habitude de ne mentionner cet argument que lorsque c'est nécessaire : votre code sera d'autant plus facile à déchiffrer.

Associer les outlets et les actions au projet

Revenons à la fenêtre de Xcode. Dans notre premier exemple, nous allons créer le squelette de notre quiz pour iPhone. Outre un titre et des éléments graphiques visant à habiller l'interface, cette dernière comprend quatre boutons et deux champs de texte. Il s'agit respectivement des quatre possibilités de réponses, de la question et du résultat affiché par l'application ("Bonne réponse" et "Dommage ! Vous vous êtes trompés !"). Les actions sont associées aux boutons et déclenchent une méthode qui vérifie la réponse du joueur. Les *outlets* pointent vers les champs de texte afin de les mettre à jour à partir du code du contrôleur : on affiche ainsi le résultat de chaque question.

Ouvrez à nouveau le fichier `QuizViewController.h` et complétez-le en saisissant les lignes qui apparaissent en gras :

```
#import <UIKit/UIKit.h>

@interface QuizViewController : UIViewController {
    IBOutlet UILabel *reponseTexte; // un simple champ de texte
}
@property (retain, nonatomic) UILabel *reponseTexte;
//déclaration des propriétés

- (IBAction)choixReponse:(id)sender;
// déclaration de l'action
@end
```

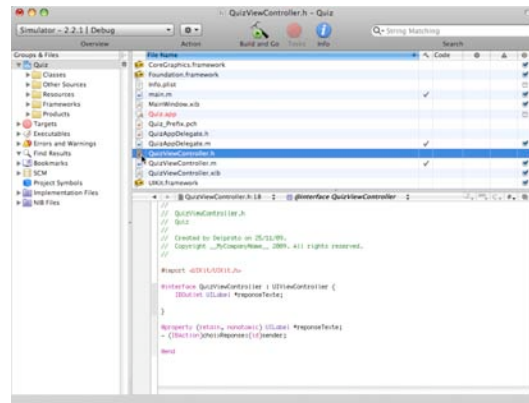
Nous avons tout d'abord défini l'outlet `reponseTexte` pointant vers un simple champ de texte (`UILabel`). La ligne débutant par `@property` déclare les propriétés de cet élément. Concrètement, le compilateur va automatiquement générer plusieurs lignes de code permettant d'accéder à l'objet pointé : les attributs `retain` et `nonatomic` correspondent au type de

lecture de l'objet et sont indirectement liés à la gestion de la mémoire. Nous reviendrons de manière plus détaillée sur ce concept si important dans le cadre limité de l'iPhone, mais retenez qu'il s'agit ici de la nomenclature de base que vous utiliserez dans la déclaration de quasiment toutes vos propriétés. Par défaut, le code généré par le compilateur pour les propriétés est accessible depuis plusieurs tâches à la fois. Le compilateur définit ainsi un mécanisme de synchronisation visant à mettre en permanence à jour les propriétés et à gérer leur accès. L'attribut `nonatomic` indique que vous souhaitez vous passer d'un tel procédé, souvent superflu dans le cadre du développement pour iPhone. Vous améliorez ainsi les performances de votre application. L'attribut `retain` indique que l'objet assigné à la propriété doit être "retenu" : la référence à l'objet n'est ainsi jamais perdue au cours du traitement. Pour des applications plus complexes qui exigent des ressources considérables, il est primordial de "libérer" ce type de références lorsqu'elles deviennent superflues afin de dégager de la mémoire.

À la dernière ligne du code, nous déclarons l'action `choixReponse` que l'on va associer aux différents boutons permettant à l'utilisateur de répondre à la question du quiz (voir Figure 4.8).

Figure 4.8

La modification de l'interface du contrôleur de notre application, sous Xcode.



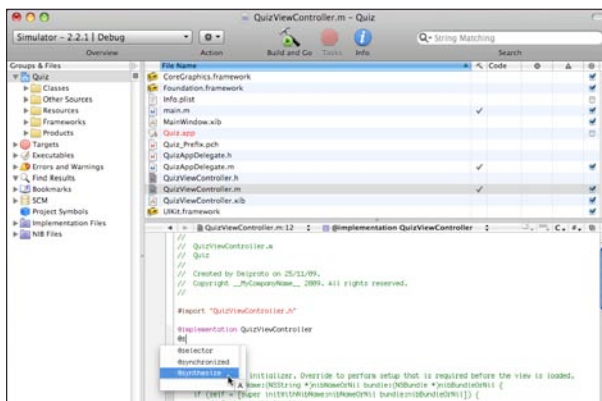
Profitez de la frappe prédictive de Xcode

Il s'agit ici des toutes premières lignes de code en Objective-C que vous saisissez dans la fenêtre de Xcode. Comme vous le constatez, l'éditeur est très souple et intuitif : il complète naturellement les mots-clés, les classes et les attributs que vous saisissez en vous proposant éventuellement des suggestions en cours de frappe. C'est un procédé particulièrement utile lorsque vous multipliez les variables et que vous définissez un très grand nombre d'objets. Dans notre exemple, le nom de notre outlet est automatiquement référencé : lorsque vous devez à nouveau le saisir au moment de déclarer ses propriétés, laissez Xcode le compléter automatiquement. Vous avez ainsi l'assurance de ne pas vous tromper et vous ne commet-

tez pas l'erreur de saisir `ReponseBouton` au lieu de `BoutonReponse`, par exemple. Par ailleurs, n'hésitez pas à profiter de la coloration syntaxique en déroulant le menu View, puis en cliquant sur Syntax Coloring et sur Objective-C++. Vous noterez au passage que Xcode supporte la coloration syntaxique de plusieurs dizaines de langages. Vous pouvez ainsi l'exploiter pour vos prochains scripts PHP, JavaScript ou Perl !

Figure 4.9

Profitez des suggestions de Xcode pour saisir rapidement vos instructions et ne pas vous tromper dans la déclaration de vos outlets ou actions.



Passons à présent au fichier d'implémentation de notre classe, `QuizViewController.m`. Sélectionnez-le dans la fenêtre principale de votre projet et consultez son état par défaut dans Xcode :

```
#import "quizViewController.h"
@implementation QuizViewController
/*
// The designated initializer. Override to perform setup that is required
// before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *
                    nibBundleOrNil) {
    if (self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil])
    {
        // Custom initialization
    }
    return self;
}
*/
/*
// Implement loadView to create a view hierarchy programmatically,
//without //using a nib.
- (void)loadView {
}
*/
/*
// Implement viewDidLoad to do additional setup after loading the view,
```

```
//typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
}
*/

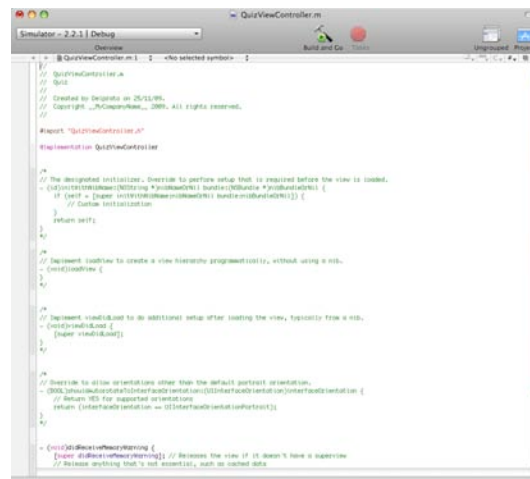
/*
// Override to allow orientations other than the default portrait
// orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation
        interfaceOrientation) {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
*/
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Releases the view if it doesn't
                                    // have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [super dealloc];
}
@end
```

Comme vous le constatez, ce fichier se contente d'importer le contrôleur de notre projet (QuizViewController.h) et définit une série de méthodes commentées, figurant entre les signes `/*` et `*/`. Elles prévoient des traitements spécifiques que l'on retrouve dans la plupart des applications, comme la préparation de certaines actions après avoir chargé la vue principale ou la gestion de l'autorotation de l'affichage.

Figure 4.10

L'implémentation du contrôleur de notre projet, avant sa modification.



Nous n'en avons pas besoin pour le moment : vous pouvez les supprimer. En revanche, nous avons besoin de définir l'action associée aux boutons de réponse, choixReponse. En guise de premier essai, nous allons nous contenter de relever le titre associé à chaque bouton et de l'afficher dans le champ de texte reponseTexte. Nous illustrons ainsi le comportement essentiel de notre application : réagir aux choix de l'utilisateur et entreprendre un traitement en conséquence. Modifiez le fichier QuizViewController.m en ajoutant les lignes figurant en gras afin qu'il contienne le code suivant (nous vous en expliquerons le sens juste après) :

```
#import "QuizViewController.h"

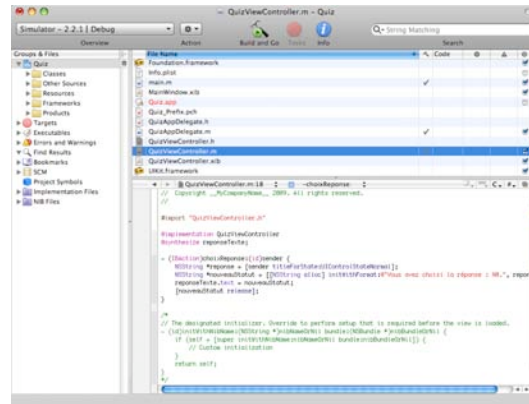
@implementation QuizViewController
@synthesize reponseTexte;
- (IBAction)choixReponse:(id)sender {
    NSString *reponse = [sender titleForState:UIControlStateNormal];
    NSString *nouveauStatut = [[NSString alloc] initWithFormat:@"Vous avez
                                choisi la réponse : %@.", reponse];
    reponseTexte.text = nouveauStatut; [nouveauStatut release];
}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
                                interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Releases the view if it doesn't
                                    // have a superview
    // Release anything that's not essential, such as cached data
}
- (void)dealloc {
    [reponseTexte release];
    [super dealloc];
}
@end
```

Saisir des caractères spéciaux

Que les puristes et les férus de la marque à la pomme nous pardonnent : nous devons prendre le temps d'exposer deux raccourcis-clavier essentiels pour saisir du code dans votre projet. Comme vous le constatez, les déclarations de classes et de méthodes sont toujours encadrées par des accolades à l'instar de nombreux langages de programmation. L'envoi de messages s'effectue à l'aide de crochets. Si vous disposez du clavier standard livré avec l'iMac, ces différents caractères n'y figurent pas ! Pas de panique : vous allez rapidement adopter des raccourcis-clavier. Vous saisissez des accolades en pressant simultanément les touches Alt+(et Alt+). Pour les crochets, appuyez sur les touches Alt+Maj+(et Alt+Maj+). Là encore, profitez de la frappe prédictive de Xcode pour compléter automatiquement vos lignes de code et vérifier que les déclarations sont bien terminées.

Figure 4.11

Nous avons redéfini la logique interne du contrôleur de notre application.



Pour l'heure, notre code est encore relativement simple et accessible. Nous commençons le fichier d'implémentation par la ligne :

```
@synthesize reponseTexte;
```

Cette simple ligne demande au compilateur de créer automatiquement les accesseurs et les mutateurs à notre place dans l'implémentation de la classe QuizViewController. Nous y reviendrons dans un instant, mais retenez qu'il s'agit d'un précieux gain de temps pour accéder en toute simplicité aux propriétés de l'objet `reponseTexte` et interagir avec lui. Deux méthodes "invisibles" sont donc créées en arrière-plan : `reponseTexte` et `setReponseTexte`.

Le code de la méthode associée à l'action `choixReponse` vient ensuite. Il est déclenché dès que l'utilisateur presse le moindre bouton apparaissant dans l'interface :

```
-(IBAction)choixReponse:(id)sender {
    NSString *reponse = [sender titleForState:UIControlStateNormal];
    NSString *nouveauStatut = [[NSString alloc] initWithFormat:@"Vous
        avez choisi la réponse : %@", reponse];
    reponseTexte.text = nouveauStatut;
    [nouveauStatut release];
}
```

En Objective-C, les chaînes de caractères sont encapsulées dans des objets `NSString` – vous remarquez au passage le lien de parenté avec `NeXTStep`, repris dans le préfixe de ces objets ! On définit ici deux objets de ce type : `reponse`, contenant le titre du bouton pressé, et `nouveauStatut` qui correspond au formatage de la réponse à afficher dans l'un des libellés de notre application. Souvenez-vous : le paramètre `sender` associé à l'action `choixReponse` contient systématiquement une référence à l'objet qui a déclenché cette méthode. Ainsi, à chaque fois que l'on exécute `choixReponse`, on dispose d'un accès direct au bouton qui a été pressé. On distingue ainsi le contrôle ayant appelé la méthode et il n'est donc pas nécessaire

de prévoir quatre méthodes différentes pour nos quatre boutons. La méthode d'instance `titleForState` renvoie le titre de l'élément `sender`. Elle exige le paramètre `UIControl-State` qui définit l'état de l'objet (ici, `normal`).

La ligne suivante crée la nouvelle chaîne de caractères `nouveauStatut`, en la formatant selon la nomenclature définie par `initWithFormat`. Le message épouse ainsi la forme "Vous avez choisi la réponse X", où X est remplacé par le titre du bouton pressé. On assigne enfin la propriété `text` de l'objet `reponseTexte` avec le contenu de la chaîne de caractères `nouveauStatut`. Ce dernier objet n'est alors plus utile et on le libère de la mémoire à l'aide de la méthode `release`. Vous notez au passage l'une des particularités de l'Objective-C : il est possible d'accéder aux propriétés d'un objet à l'aide de la notation point (`reponseTexte.text` par exemple) sans faire appel à une méthode spécifique.

En dernier lieu, nous modifions la méthode `dealloc` préparée par Xcode, en libérant la mémoire référencée par l'objet `reponseTexte`.

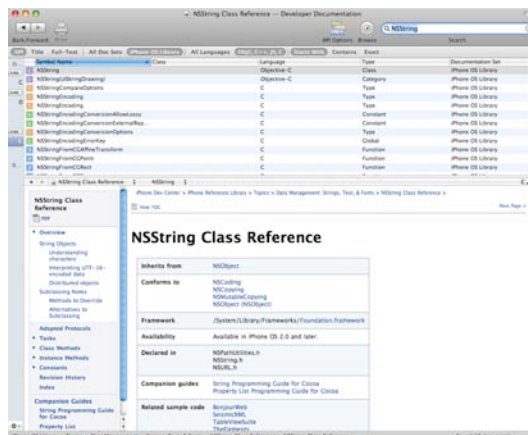
Consulter la documentation en cas de doute

Comme nous l'avons vu, la déclaration de nos méthodes s'appuie sur de nombreux paramètres et attributs liés aux différents types de données. Il n'est pas toujours facile de mémoriser cette nomenclature exacte et de ne rien oublier. Si le compilateur saura vous expliquer la nature de vos erreurs à travers le débogueur, n'hésitez pas à anticiper ces problèmes et à vérifier que vous avez correctement déclaré vos méthodes.

Pour cela, la méthode la plus simple consiste à maintenir la touche Alt du clavier enfoncée : votre pointeur se transforme en une croix. Effectuez alors un double-clic sur un terme, une propriété, un attribut, une méthode ou une classe sur lequel vous avez un doute. Vous accédez automatiquement au guide de référence de l'Objective-C et vous parcourez la définition précise du terme sélectionné (en anglais). Pratique pour ne rien oublier et pour s'instruire en cours de route en parfaissant sa maîtrise du langage !

Figure 4.12

*Un doute ? Une question ?
Plongez dans la
documentation officielle,
accessible à partir de
n'importe quel terme de la
syntaxe d'Objective-C !*



C'est parfait : nous avons envisagé tous les comportements de notre application à ce premier stade et nous avons défini au sein du contrôleur une série d'objets et de méthodes effectuant des références directes à la vue de notre application. Il ne nous reste plus qu'à dessiner cette interface et à réaliser manuellement les connexions entre les objets et le contrôleur, afin de déclencher les bonnes actions !

Dessiner l'interface et associer des méthodes aux objets

Retournez dans la fenêtre principale de Xcode et effectuez à présent un double-clic sur le fichier `QuizViewController.xib`. Comme vous en avez maintenant l'habitude, vous basculez automatiquement vers Interface Builder et la vue principale de votre application apparaît à l'écran. Dans le cadre de notre premier exemple, nous devons disposer les éléments suivants sur l'interface de notre projet :

- Un libellé pointé par l'outlet `reponseTexte`. Il contient la chaîne formatée par notre méthode `choixReponse`. Dans un premier temps, il affiche la phrase "Vous avez choisi la réponse X", mais il peut contenir tout type de chaînes de caractères, comme le score du joueur ou une réaction générale ("Bravo !", "Dommage !", etc.). Il s'agit d'un élément de type `UILabel`, appartenant à la classe `UILabel`.
- Quatre boutons rectangulaires permettant à l'utilisateur d'exprimer un choix. Dans la bibliothèque d'Interface Builder, il s'agit des objets `Round Rect Button`, des instances de la classe `UIButton`.
- D'autres éléments graphiques visant à habiller l'interface. Il peut s'agir d'un champ de texte supplémentaire affichant la question ou d'attributs associés à la vue (couleur d'arrière-plan, opacité du fond, etc.).

Si ce n'est pas déjà fait, ouvrez la bibliothèque d'objets en déroulant le menu `Tools` et en cliquant sur `Library`. Commencez par déposer le libellé, en effectuant un simple glisser-déposer depuis la fenêtre de la bibliothèque vers la fenêtre `View` (voir Figure 4.13). Vous notez au passage que des repères bleus apparaissent sur l'interface. En tant que designer, ils vous sont très utiles pour aligner les éléments au sein de votre application : vous pouvez ainsi facilement vous "caler" sur les autres objets figurant déjà dans votre interface et le rendu est souple et ergonomique. N'hésitez pas à saisir les poignées encadrant votre libellé pour l'étendre sur toute la largeur de l'interface. Ouvrez ensuite la fenêtre de l'inspecteur d'attributs, en déroulant le menu `Tools` et en cliquant sur `Attributes Inspector` ou en pressant simultanément les touches `Cmd+1` (voir Figure 4.14). Face au champ `Layout`, cliquez sur l'icône permettant de centrer le texte. Quelle que soit la longueur de la chaîne de caractères figurant dans ce libellé, vous serez ainsi assuré qu'elle est bien alignée au centre de l'écran. Effectuez enfin un double-clic sur ce libellé afin de supprimer son texte : il sera remplacé automatiquement et nous n'avons pas besoin d'afficher quoi que ce soit dès l'ouverture

de l'application. Toutefois, en auscultant à nouveau l'interface, aucun signe ne semble révéler la présence de ce libellé. Si vous multipliez ces éléments, vous risquez ainsi de les superposer maladroitement et d'en perdre le contrôle. Prenez dès à présent l'habitude de vous référer à la fenêtre `QuizViewController.xib`, qui dresse la liste de tous les éléments imbriqués dans l'objet `View`. En effectuant un double-clic sur chacun d'entre eux, vous les sélectionnez sous Interface Builder et vous êtes en mesure de les redimensionner ou de les adapter librement. Pratique pour ne rien oublier !

Figure 4.13

On dépose un premier libellé sur l'interface de notre application.

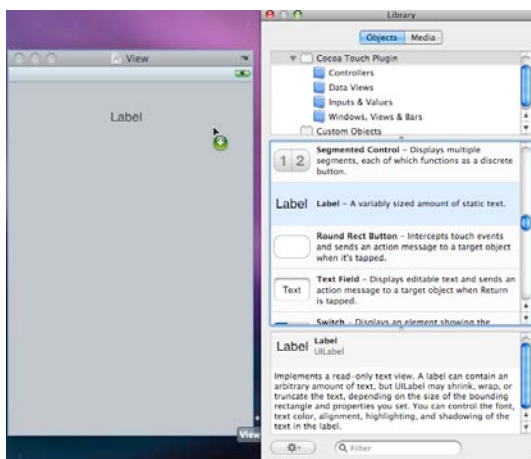
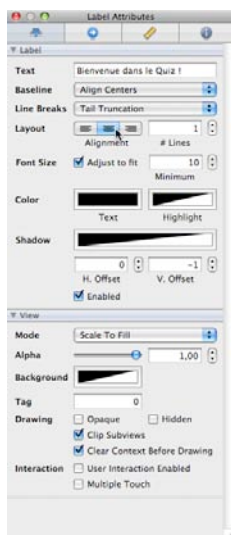


Figure 4.14

Le panneau de l'inspecteur d'attributs vous permet notamment de modifier l'alignement du texte de votre libellé.

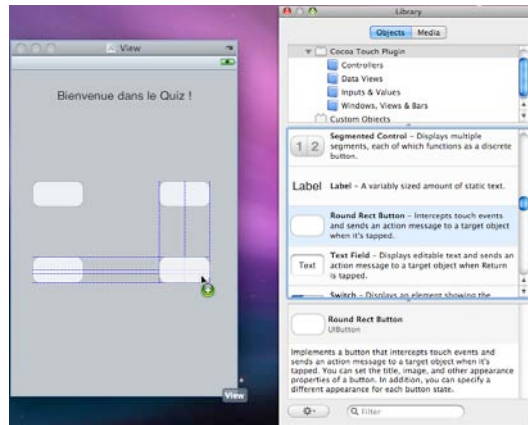


Déplacez ensuite un premier élément `Round Rect Button` depuis la bibliothèque d'objets vers l'interface. Là encore, vous avez la possibilité de redimensionner cet objet et vous

profitez des repères de couleurs pour l'aligner vis-à-vis des autres éléments. Ajoutez trois autres boutons en les positionnant sur la même ligne horizontale et verticale que les autres objets. Les repères bleus se croisent et les boutons se verrouillent dans leur position idéale (voir Figure 4.15). Effectuez un double-clic sur chaque bouton et saisissez les valeurs A, B, C et D : ce sont les titres des boutons.

Figure 4.15

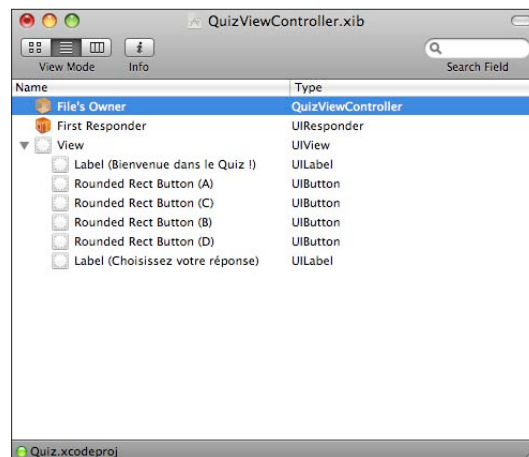
Déposez les quatre boutons aux coins arrondis et profitez des repères pour les aligner correctement les uns par rapport aux autres.



Il ne vous reste plus qu'à associer les outlets et les actions à ces différents éléments d'interface. La manipulation est simple et intuitive, mais exige une certaine dextérité. En arrière-plan, à côté de la fenêtre View, vous retrouvez la fenêtre QuizViewController.xib (voir Figure 4.16).

Figure 4.16

Gardez ouverte la fenêtre QuizViewController.xib afin d'accéder au File's Owner de votre projet.



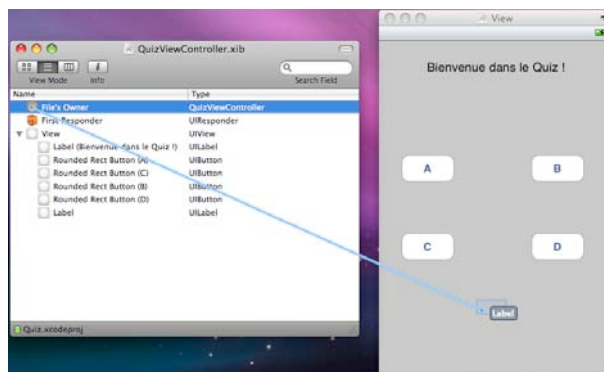
La fenêtre comprend une série d'icônes, dont File's Owner que nous avons déjà vu au chapitre précédent. Pour rappel, il s'agit de la classe responsable du chargement du fichier nib.

Comme nous l'avons vu, une instance d'`UIViewController` ou de ses sous-classes charge automatiquement le fichier nib qui lui est associé. Par exemple, notre classe `QuizViewController` recherche automatiquement le nib `QuizViewController.xib`. Xcode nous a créé un tel fichier et nous venons d'ailleurs de l'éditer : au lancement de notre application, une instance de `QuizViewController` se charge en mémoire, ouvre `QuizViewController.xib` et en devient le "propriétaire" (*File's Owner*). L'icône "File's Owner" symbolise ainsi l'instance du contrôleur de notre application.

Maintenez la touche `Ctrl` enfoncée et cliquez sur l'icône *File's Owner* de la fenêtre `QuizViewController.xib`. Tout en laissant le bouton de la souris enfoncé, déplacez-la vers votre libellé, dans la fenêtre `View`. Une flèche bleue relie les deux fenêtres : relâchez la souris au-dessus du libellé. Un menu contextuel vous présente alors tous les outlets accessibles : sélectionnez `reponseTexte` en cliquant sur son intitulé. Vous venez d'associer l'outlet `reponseTexte` au libellé figurant sur votre interface (voir Figure 4.17).

Figure 4.17

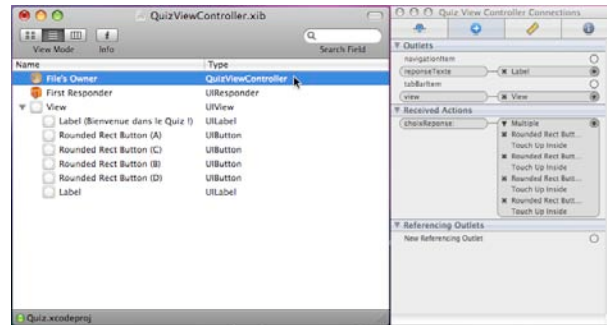
On associe notre outlet au libellé de l'interface.



Vous l'avez compris : ce procédé très intuitif vous permet de connecter en toute simplicité un grand nombre d'outlets à des éléments d'interface très divers. Vous pouvez vérifier les outlets référencés en déroulant le menu `Tools` d'Interface Builder, puis en cliquant sur `Connections Inspector`. Sélectionnez ensuite le libellé : vous avez la confirmation que l'outlet `reponseTexte` est bien associé à *File's Owner*. Vous supprimez éventuellement une telle liaison en cliquant sur la croix, dans le champ `Referencing Outlets` de l'inspecteur de connexions. Très important : n'oubliez surtout pas d'associer l'icône *File's Owner* à l'objet `View` afin que ce dernier soit bien lié au contrôleur de votre application et apparaisse sur l'écran de l'iPhone (voir Figure 4.18). C'est une étape absolument nécessaire dans la composition de toutes les applications, qui prendra tout son sens lorsque vous développerez des projets mettant en scène plusieurs vues. La manipulation est heureusement très simple : pressez la touche `Ctrl` en cliquant sur l'icône *File's Owner* et relâchez la souris au-dessus de l'icône `View` puis sélectionnez l'outlet `view`.

Figure 4.18

Vérifiez les outlets et les actions associées à l'objet *File's Owner* : la vue, le libellé et les boutons doivent figurer dans l'inspecteur de connexions.

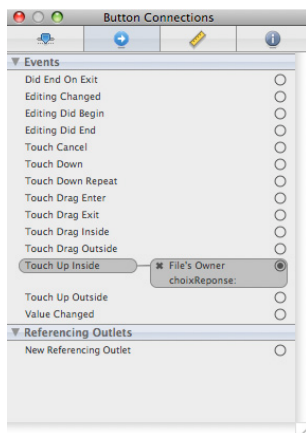


L'association d'actions obéit au principe inverse : vous devez partir des éléments de votre interface pour les lier à l'instance *File's Owner*. Là encore, vous avez parfaitement le droit de maintenir la touche Ctrl enfoncée et de tisser une flèche vers l'icône de *File's Owner*. Mais l'iPhone présente de nombreuses méthodes pour déclencher des actions ! L'utilisateur peut presser un bouton, mais aussi effectuer des rotations avec plusieurs doigts, déplacer un élément en maintenant la pression sur un élément, etc. Pour choisir précisément le type d'interaction qui vous intéresse, retournez à l'inspecteur de connexions. Sélectionnez un premier bouton : une liste de quatorze événements spécifiques figure sous le champ Events. C'est l'événement *Touch Up Inside* qui nous intéresse ici ; il correspond au cas de figure où l'utilisateur a pressé le bouton puis a relâché la pression au-dessus de celui-ci. Il se différencie de l'événement *Touch Up Outside*, qui ne se préoccupe que de la pression de l'utilisateur sur un bouton, quel que soit l'emplacement où il a relâché son doigt. Il est très important de prêter attention à cette différence, en particulier pour une application qui exige des actions mesurées et réfléchies, comme un quiz par exemple. Combien de fois avez-vous pressé un bouton, avant de vous rendre compte que ce n'était pas l'action à entreprendre ? Dans ce type de cas, un réflexe nous pousse à sortir le doigt hors du cadre du bouton puis à relâcher la pression : l'événement *Touch Up Inside* ne se déclenche pas et c'est l'événement *Touch Up Outside* qui est concerné.

Cliquez sur le petit cercle à droite de l'événement *Touch Up Inside* et maintenez le bouton enfoncé, sans presser la touche Ctrl cette fois. Déplacez le pointeur vers l'icône de *File's Owner* et relâchez la souris lorsque la flèche bleue apparaît. Interface Builder vous propose de sélectionner l'action à entreprendre, *choixReponse* dans notre exemple (voir Figure 4.19). Là encore, vous constatez que la liaison s'est bien effectuée et vous pouvez éventuellement la supprimer en cliquant sur la croix dans l'inspecteur de connexions. Par ailleurs, vous pouvez tout à fait enchaîner les événements, en associant plusieurs d'entre eux à cette action, ou au contraire associer plusieurs actions à un même événement.

Figure 4.19

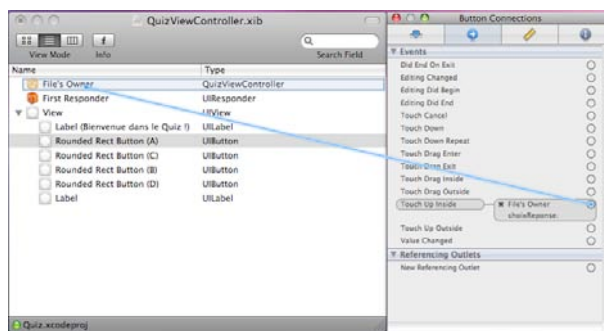
L'inspecteur de connexions dresse la liste de tous les événements associés à vos contrôles.



Tout est opérationnel : l'interface est dessinée, les contrôles sont prêts et les liaisons ont été créées (voir Figure 4.20). Sauvegardez le résultat en déroulant le menu File et en cliquant sur Save, puis retournez à la fenêtre principale de Xcode. Déroulez le menu Build, puis cliquez sur Build and Run. Après quelques secondes de compilation, votre programme apparaît dans le Simulateur d'iPhone. Vous pouvez librement cliquer sur chacun des boutons : le texte du libellé se modifie automatiquement en conséquence. Votre souris mime le comportement des doigts de l'utilisateur – vous avez créé votre première application permettant d'interagir avec lui !

Figure 4.20

On a associé les actions et les outlets de notre projet aux éléments correspondants : des flèches apparaissent sur la fenêtre d'Interface Builder.



Les événements associés à l'iPhone

En tissant un lien vers l'action de notre projet, vous avez remarqué une liste de quatorze événements gérés par l'iPhone. Vous pouvez naturellement remplacer les boutons par n'importe quel autre objet de la bibliothèque. Pour satisfaire pleinement votre curiosité, voici la liste de tous ces événements :

Did End On Exit. On associe traditionnellement cet événement au bouton "retour" associé au clavier virtuel. Il indique alors qu'une opération a été terminée.

Editing Changed. Cet événement surveille la mise à jour d'un champ (par exemple, la saisie d'un mot à l'aide du clavier virtuel) en cours de frappe. On l'utilise notamment pour créer des listes de suggestion à partir des premiers caractères saisis par l'utilisateur.

Editing Did Begin. Un champ vient d'obtenir le focus et l'utilisateur s'apprête à le modifier.

Editing Did En. C'est l'inverse : le champ perd le focus et l'utilisateur a terminé sa saisie.

Touch Cancel. L'utilisateur a annulé sa pression sur l'écran de l'iPhone.

Touch Down. L'utilisateur a pressé l'écran de l'iPhone mais n'a pas encore relevé le doigt. Vous devez absolument distinguer cet événement de "Touch Up", qui correspond à une séquence plus longue et ne se déclenche que lorsque l'utilisateur retire son doigt de l'écran.

Touch Down Repeat. L'utilisateur a pressé (au moins) deux fois d'affilée l'écran, mimant ainsi le comportement d'un double-clic.

Touch Drag Enter. L'utilisateur a pressé l'écran de l'iPhone, puis a déplacé son doigt tout en maintenant la pression. Dans un jeu, cet événement permet de déplacer un objet à l'écran en suivant les coordonnées exprimées par l'utilisateur.

Touch Drag Exit. Il s'agit de l'événement qui achève le précédent : l'utilisateur a finalement relevé son doigt, après l'avoir déplacé sur l'écran de l'iPhone. Le contrôle perd alors le focus.

Touch Drag Inside. L'utilisateur a posé son doigt sur le contrôle considéré puis l'a déplacé sur l'interface en restant au-dessus de la zone définie.

Touch Drag Outside. L'utilisateur a posé son doigt sur le contrôle considéré puis l'a déplacé sur l'interface en sortant du cadre défini à l'origine.

Touch Up Inside. C'est l'un des événements principaux de Cocoa Touch : l'utilisateur a pressé un contrôle puis a relevé le doigt, tout en restant dans la zone considérée. Cet événement confirme généralement le choix de l'utilisateur, lorsqu'il clique sur un bouton par exemple.

Touch Up Outside. À la manière de **Touch Up Inside**, cet événement se déclenche lorsque l'utilisateur presse un contrôle puis relève le doigt. Mais il ne vérifie pas que la pression a été relâchée au-dessus de la zone considérée. Faites attention : en règle générale, lorsqu'un utilisateur se trompe et souhaite annuler sa saisie, il déplace son doigt au-delà du bouton et relâche la pression. C'est ce que définit cet événement.

Value Changed. Un champ ou un élément vient de voir sa valeur changer.

Nous aurons tout le loisir d'exploiter ces événements au cours de nos prochains projets ! Pour l'heure, reprenez comment tisser des liens entre ceux-ci et le File's Owner : le principe reste rigoureusement le même, quel que soit l'événement.

Les accesseurs et les mutateurs

La logique interne de l'Objective-C

À travers ce premier exemple, vous constatez à quel point l'Objective-C est un langage dynamique et souple. Parmi les principales subtilités de la syntaxe de ce langage, vous avez retenu que les interactions entre les objets s'effectuent par l'envoi de messages. Tous ces messages s'écrivent entre crochets ; le premier élément correspond à l'objet auquel il est destiné et le second est le message en lui-même. Ainsi, la ligne :

```
[nouveauStatut release];
```

indique que l'objet `nouveauStatut` reçoit le message `release`, c'est-à-dire qu'on le libère de la mémoire. L'Objective-C est aussi un langage rigoureux : lorsqu'on déclare une classe, on distingue son interface de son implémentation. Le premier définit le comportement de la classe et le second sa logique interne. Le projet automatiquement préparé par Xcode nous en donne un exemple immédiat. La classe `QuizViewController` est déclarée dans deux fichiers que nous avons modifiés précédemment : `QuizViewController.h` (l'interface) et `QuizViewController.m` (l'implémentation). Reportez-vous au code précédent : l'interface débute par le mot-clé `@interface` et s'achève par le mot-clé `@end`. Entre accolades, vous définissez toutes les variables d'instance (notre `outlet reponseTexte`) et avant le mot-clé `@end` se trouvent les méthodes associées (notre `action choixReponse`). L'implémentation n'appelle pas de commentaires particuliers : elle se contente de dresser la liste successive de toute la logique interne de la classe.

Les propriétés en Objective-C

Comme dans bien d'autres langages de programmation, les variables d'instance d'une classe ne sont pas accessibles par les autres classes et lui sont bornées. Traditionnellement, les développeurs devaient définir une série de méthodes permettant de récupérer et d'assigner les valeurs aux variables d'instance d'une classe. On désigne ces méthodes sous le nom d'*accesseur* et de *mutateur*, ou à travers les anglicismes *getters* et *setters*. Pour éviter de surcharger le code, l'Objective-C a introduit le concept des *propriétés*. Celles-ci créent

automatiquement un accesseur aux variables d'instance et se déclarent à l'aide du mot-clé `@property`.

```
@property UILabel *reponseTexte;
```

Cette ligne figure dans la déclaration de l'interface de votre classe principale. Dans la mesure où il s'agit bien d'une méthode, elle n'apparaît pas entre les accolades initiales et se voit ravalée juste avant le mot-clé `@end`. À l'inverse, nous créons automatiquement un mutateur dans l'implémentation de la classe :

```
@synthesize reponseTexte;
```

La variable d'instance `reponseTexte` est donc librement accessible et on a la possibilité de modifier ses valeurs par la simple notation point, comme nous l'avons vu :

```
reponseTexte.text = nouveauStatut;
```

Si l'on devait employer les méthodes traditionnelles d'accesseur et envoyer un message à l'objet, il faudrait alors écrire :

```
[reponseTexte setText:nouveauStatut];
```

en ayant naturellement pris le soin de définir la méthode `setText`. L'Objective-C est ainsi un langage très intuitif, qui vous permet de rédiger du code de manière souple et claire. Vous devez impérativement garder à l'esprit ces concepts de propriétés, de variables d'instance, d'accesseurs déclarés implicitement et de notation point : vous en ferez un usage constant à travers toutes vos applications.

Le délégué de l'application

Lors de la création de notre dernier projet, nous avons constaté que Xcode avait préparé quatre fichiers dans le dossier `Classes`. Nous venons de détailler le concept régissant la classe contrôleur principale, `QuizViewController`. Intéressons-nous à présent au dernier fichier, `QuizAppDelegate`. Il s'agit du délégué de l'application : une classe qui, comme son nom l'indique, prend en charge certaines actions à la place d'un autre objet. Concrètement, le délégué vous permet d'exécuter des traitements spécifiques à des moments-clés du cycle de vie d'une application, définis par la classe principale `UIApplication`. Là encore, la déclaration de la classe `QuizAppDelegate` se décompose en une interface (le fichier `.h`) et une implémentation (le fichier `.m`).

Commencez par ouvrir le fichier `QuizAppDelegate.h` et consultez son contenu :


```
#import <UIKit/UIKit.h>

@class QuizViewController;

@interface QuizAppDelegate : NSObject <UIApplicationDelegate> {
    IBOutlet UIWindow *window;
    IBOutlet QuizViewController *viewController;
}

@property (nonatomic, retain) UIWindow *window;
@property (nonatomic, retain) QuizViewController *viewController;

@end
```

La seule particularité réside dans la déclaration de l'interface :

```
@interface QuizAppDelegate : NSObject <UIApplicationDelegate>
```

Le terme figurant entre les signes < et > correspond à un protocole en Objective-C. Les protocoles regroupent des méthodes qui peuvent être implémentées par n'importe quelle classe. Ici, le délégué s'inscrit dans le protocole `UIApplicationDelegate`, qui définit une série de méthodes s'exécutant à des moments spécifiques du cycle de vie de l'application : son exécution, sa fermeture, des avertissements où la mémoire vient à manquer, une notification adressée par le téléphone (réception d'un SMS ou d'un appel), etc.

Ouvrez le fichier `QuizAppDelegate.m` pour découvrir la déclaration de l'une de ces méthodes :

```
#import "QuizAppDelegate.h"
#import "QuizViewController.h"

@implementation QuizAppDelegate

@synthesize window;
@synthesize viewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [viewController release];
    [window release];
}
```

```

        [super dealloc];
    }

@end

```

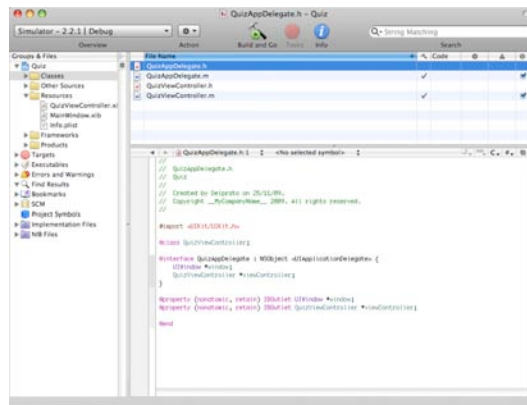
La méthode `applicationDidFinishLaunching` est appelée lorsque la classe `UIApplication` a chargé la vue principale de votre application (voir Figure 4.21). En ajoutant des traitements entre les accolades, vous pouvez ainsi envisager tout type d'actions à gérer avant le démarrage de l'application. Chargement d'un contenu externe grâce à la connexion 3G ou WiFi, ouverture de vues secondaires, lecture d'une base de données : c'est ici que vous définissez tous ces types de traitement. Par défaut, on envoie le message suivant :

```
[window makeKeyAndVisible];
```

L'objet `window` reçoit le message `makeKeyAndVisible`, ce qui affiche la fenêtre de votre application. Dans la mesure du possible, évitez toutefois d'ajouter de trop nombreux traitements à ce stade : le démarrage de votre application s'en trouverait retardé et elle perdrait de l'intérêt aux yeux des utilisateurs. Mais gardez ce concept à l'esprit : il est possible d'effectuer des actions spécifiques à des moments-clés de la vie de votre application, que ce soit pour sauvegarder des réglages juste avant de la clôturer ou pour mettre un traitement en pause lorsque vous recevez un appel. Nous mettrons ce concept en œuvre autour d'un exemple concret au chapitre suivant.

Figure 4.21

Vous pouvez consulter et adapter le code du délégué de l'application à partir de la fenêtre principale de Xcode.



Finaliser notre projet de quiz

Vous bénéficiez désormais d'une meilleure maîtrise des principes fondamentaux de l'Objective-C : il est grand temps de retourner à notre projet et de finaliser le quiz. À l'heure actuelle, notre application regroupe quatre boutons qui effectuent un simple traitement – ils mettent à jour le texte d'un libellé. Nous devons les adapter pour servir notre objectif pre-

mier et poser une réelle question. Par ailleurs, nous devons modifier la logique de notre contrôleur afin qu'il vérifie que la bonne réponse a été donnée.

Cette fois, nous allons procéder de manière inverse en mettant tout d'abord à jour la vue de notre application. Commencez par ajouter un libellé contenant la première question du quiz. Dans le cadre de notre exemple, nous ne mettons pour l'instant en scène qu'une simple question. Si vous souhaitez enchaîner les vues ou préparer un plus grand nombre de questions, reportez-vous au chapitre suivant pour découvrir comment basculer d'une vue à l'autre. En effet, chaque bonne réponse conduira au chargement de la vue suivante, jusqu'à atteindre la dernière question du quiz. Il sera également plus judicieux d'articuler un véritable quiz autour d'une base de données : vous stockerez ainsi toutes les questions et les réponses dans un espace spécifique, facile à mettre à jour. Dans notre exemple, notre nouveau libellé contient la question "En quelle année Mickey Mouse est-il né ?". Effectuez un double-clic sur chaque bouton et remplacez les mentions A, B, C et D par les différentes réponses que vous proposez à l'utilisateur – il s'agit ici de quatre dates parmi lesquelles figure la solution : 1928. Au besoin, étendez les dimensions des boutons ou choisissez l'option Scale To Fill au champ View du panneau des attributs (déroulez le menu Tools et cliquez sur Attributes Inspector).

Il existe plusieurs solutions pour vérifier si l'utilisateur a choisi la bonne réponse :

- Vous pouvez créer deux méthodes, `reponseCorrecte` et `reponseFausse`, et associer manuellement ces actions aux boutons. La première n'est donc liée qu'à un seul bouton (celui correspondant à la bonne réponse !) et la seconde est associée aux trois autres boutons. Cette logique est un peu fastidieuse, dans la mesure où vous devrez répéter plusieurs lignes de code entre les deux méthodes ;
- Vous pouvez aussi vous contenter d'adapter notre méthode `choixReponse` et d'y intégrer une boucle conditionnelle qui compare la réponse exprimée par l'utilisateur à une chaîne de caractères "statique" contenant la solution. Si les deux chaînes sont équivalentes, l'utilisateur a bien répondu. Dans le cas contraire, il s'est trompé. N'oubliez pas que l'attribut *sender* vous permet de repérer précisément le bouton pressé et donc de décliner cette même méthode à l'ensemble des boutons accessibles à l'utilisateur ;

Pour l'heure, nous choisissons la seconde solution. Vous constaterez par la suite qu'il existe des mécanismes plus précis, mais ceux-ci s'articulent autour d'une base de données.

Sous Xcode, ouvrez le fichier `QuizViewController.m` et remplacez son contenu pour qu'il corresponde à la structure suivante :

```
#import "QuizViewController.h"
@implementation QuizViewController
@synthesize reponseTexte;
```

```

-(IBAction)choixReponse:(id)sender {
    NSString *reponse = [sender titleForState:UIControlStateNormal];
    if([reponse isEqualToString:@"1928"]) {
        NSString *nouveauStatut = [[NSString alloc] initWithFormat:@"Vous
            avez trouvé la bonne réponse : %@.", reponse];
        reponseTexte.text = nouveauStatut;
        [nouveauStatut release];
    }
    else {
        NSString *nouveauStatut = [[NSString alloc] initWithFormat:@"Vous
            avez choisi la réponse %@.", reponse];
        reponseTexte.text = nouveauStatut;
        [nouveauStatut release];
    }
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Releases the view if it doesn't
    // have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [reponseTexte release];
    [super dealloc];
}
@end

```

Comme vous le constatez, nous avons opté pour un système extrêmement simple. Après avoir récupéré le titre du bouton pressé, la méthode `choixReponse` ouvre la boucle conditionnelle `if()`. Celle-ci s'articule autour d'un simple envoi de message :

```
[reponse isEqualToString:@"... "]
```

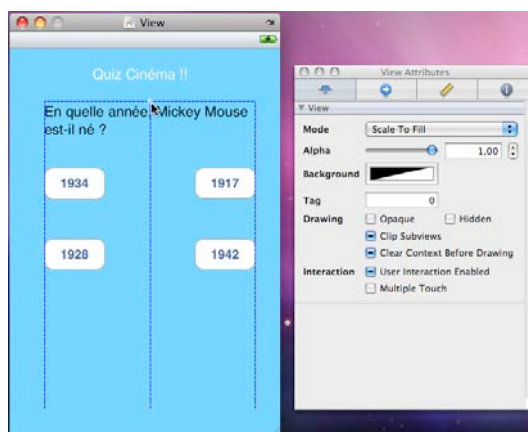
La méthode `isEqualToString` compare deux chaînes de caractères et renvoie un booléen. Si les deux valeurs correspondent, l'utilisateur a donné la bonne réponse et on affiche ainsi un message de félicitations dans le libellé. Dans le cas contraire, on lui indique qu'il s'est trompé à l'aide de la boucle `else()`. Il ne vous reste plus qu'à habiller votre interface pour la rendre la plus séduisante et intuitive possible (voir Figure 4.22).

Sous Xcode, ouvrez à nouveau le fichier `QuizViewController.xib` et sélectionnez chaque élément de l'interface. À l'aide de l'inspecteur de propriétés, vous êtes en mesure d'associer

un arrière-plan à votre objet View (cliquez sur le champ Background) ou de mettre en forme chaque libellé. Là encore, sélectionnez un objet figurant dans la fenêtre View puis utilisez l'inspecteur de propriétés (Cmd+1) pour modifier sa couleur (Color), son ombre portée (Shadow) ou son opacité (Alpha). Vous pouvez également saisir les poignées qui encadrent chaque libellé et les agrandir en hauteur. Dans ce cas-là, n'oubliez pas d'autoriser Interface Builder à tronquer votre texte afin de l'aligner automatiquement sur plusieurs lignes. Saisissez une valeur correspondant au nombre de lignes dont vous avez besoin au champ # Lines et étendez à loisir la hauteur de chaque libellé. Au chapitre suivant, vous découvrirez comment remplacer chaque bouton par une image et comment remplacer la couleur d'arrière-plan par l'élément graphique de votre choix.

Figure 4.22

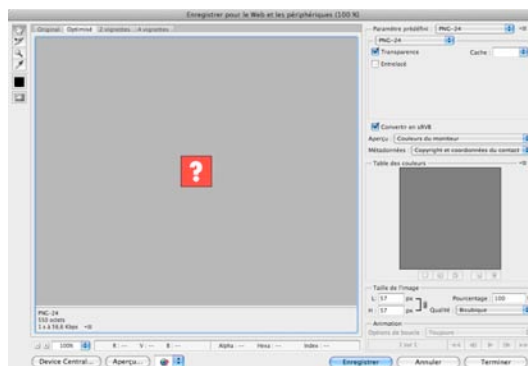
Après quelques retouches cosmétiques et une modification de la logique du contrôleur, notre quiz est pleinement fonctionnel et pose de réelles questions.



Pour l'heure, n'oubliez pas de créer une icône de 57 × 57 pixels à l'aide de votre éditeur graphique préféré et de l'enregistrer sous le nom "icon.png" (voir Figure 4.23).

Figure 4.23

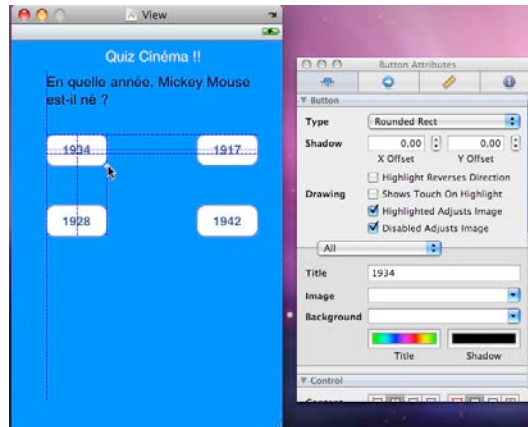
On réalise une icône à l'effigie de notre quiz sous Photoshop, avant de l'enregistrer au format PNG.



Déroulez le menu Project de Xcode, sélectionnez l'option "Add to Project" puis choisissez l'icône que vous venez de créer. Sélectionnez ensuite le fichier Info.plist associé à votre projet et effectuez un double-clic dans le champ Icon file. Saisissez enfin le nom de votre image et enregistrez votre projet pour faire apparaître votre icône dans le Simulateur d'iPhone.

Figure 4.24

Interface Builder permet d'aboutir rapidement à des vues plus dynamiques, avec des éléments d'interface mieux agencés.

**Figure 4.25**

En quelques aménagements simples, notre quiz prend de l'ampleur et s'approche d'une application pleinement fonctionnelle, à diffuser sur l'App Store !



Pour aller plus loin

Notre quiz est désormais en place mais il manque encore de saveur ! La question et les réponses sont purement "statiques" et ont été saisies dans Interface Builder. Pour conférer un plus grand intérêt à votre projet, vous devriez envisager d'enregistrer une grande série de questions et de réponses dans une base de données associée à votre application. Nous nous y consacrerons au Chapitre 7, à travers l'étude précise de l'enregistrement et de la lecture de données.

En attendant, vous pouvez perfectionner ce premier exemple en ajoutant des traitements supplémentaires dans la boucle conditionnelle de la méthode `choixReponse`. Par exemple, vous pouvez mettre à jour un score associé à un nouveau libellé. Vous pouvez également créer une nouvelle vue si l'utilisateur a répondu correctement et lui présenter ainsi une nouvelle question sur le même principe. En enchaînant les vues, vous créez une application très riche qui peut revêtir des aspects très différents !

Il s'agit précisément de la tâche à laquelle nous allons nous atteler au chapitre suivant. Retenez bien les concepts essentiels entrevus au cours de ce chapitre : vous êtes prêt pour développer votre première application multivues.



5

Les vues de votre application

Au sommaire de ce chapitre

- Hiérarchiser les vues de votre application
- Organiser une application multivues
- Modifier l'animation de transition
- MultiConvertisseur, une application multivues composée d'actions et d'outlets
- Le design des vues de votre application
- Pour aller plus loin

Vous savez désormais composer vos propres applications pour iPhone ou iPod Touch en profitant de l'ensemble des contrôles de la bibliothèque pour interagir avec vos utilisateurs. En articulant votre projet autour de l'architecture modèle-vue-contrôleur, vous distinguez la logique interne de votre programme de ses données ou de leur représentation : non seulement votre code gagne en clarté, mais il est également plus facilement réutilisable. Jusqu'à présent, nous n'avons toutefois développé que des applications s'appuyant sur une *vue* unique ; c'est en combinant une multitude d'écrans et d'interfaces que vous aboutirez aux projets les plus ambitieux et les plus ergonomiques pour l'utilisateur final !

Mises à part quelques applications spécifiques qui ne remplissent qu'un rôle très précis (niveau à bulles, lampe-torche, etc.), la plupart des projets pour iPhone s'articulent autour d'une succession d'écrans qui s'adaptent librement au choix de l'utilisateur. Observez le comportement de l'application *Réglages*, installée par défaut sur le terminal (voir Figure 5.1). Après l'avoir lancée, vous accédez à sa vue principale : une série d'actions sont disposées dans un large tableau que vous manipulez au doigt. En cliquant sur l'une des icônes, vous accédez à la rubrique de votre choix (Fond d'écran, Général, Téléphone, Safari, etc.) et vous remplacez cette vue par un écran secondaire qui en détaille le contenu. Une simple pression sur la barre de navigation supérieure et vous revenez vers la vue principale : l'interface de cet utilitaire s'organise de manière hiérarchique autour d'une série de vues prédéfinies, que l'on affiche ou masque selon les *desideratas* de l'utilisateur. Il en va de même pour l'application *Mail* : vous commencez par parcourir la liste de vos comptes de courrier, puis vos différentes boîtes de réception pour accéder enfin aux messages en eux-mêmes. Vous parcourez ainsi une arborescence, qui détaille vos options au fur et à mesure de vos choix.

Figure 5.1

L'application Réglages de l'iPhone s'articule autour d'une série de vues, qui s'imbriquent de manière hiérarchique.



Les systèmes de navigation permettant de basculer d'une vue à l'autre sont très nombreux. Ainsi, l'application *Horloge* s'appuie sur une barre d'onglets pour afficher successivement les heures internationales, l'alarme, le chronomètre ou le minuteur (voir Figure 5.2). Ce système de navigation vous permet de choisir une vue spécifique afin de la rendre active à l'écran. L'application *Météo* opte pour une seconde approche : les bulletins s'enchaînent dans un mouvement de doigt et vous définissez les localités en cliquant sur un bouton "i" disposé en bas à droite de chaque vue. Dans tous les cas, retenez que les applications organisent les vues de manière hiérarchique et qu'une série de contrôleurs les permute en fonction des choix de l'utilisateur.

Figure 5.2

Comparez le système de navigation de l'application Horloge, construit autour d'une série d'onglets, et de l'application Météo dont les vues se manipulent du bout des doigts.



Hiérarchiser les vues de votre application

Ouvrez à nouveau votre dernier projet et accédez au dossier *Resources* dans le volet gauche de Xcode. En consultant le fichier *Info.plist* qui définit les propriétés générales de votre application, vous découvrirez que le fichier nib principal est *MainWindow*. Comme son nom l'indique, il s'agit de la fenêtre principale de votre projet : elle est chargée en premier et définit l'élément racine de la hiérarchie des vues de votre application. Sous Xcode et Interface Builder, elle correspond au fichier *MainWindow.xib* et s'articule autour d'un objet *Window*, qui est une instance de la classe *UIWindow*. Comme nous l'avons vu, le délégué de l'application initialise alors cet objet et déclare le contrôleur principal de votre vue : on charge l'interface *QuizViewController.xib* dans notre exemple précédent. Celle-ci met en place une instance principale de la classe *UIView* qui regroupe elle-même des objets sous-jacents, comme nos libellés (*UILabel*) ou nos boutons (*UIButton*).

Pour schématiser, on peut ainsi conclure qu'une application s'organise autour d'une fenêtre unique *UIWindow* et d'une série d'objets *UIView* imbriqués qui correspondent aux différents écrans que l'utilisateur manipule. En tant que développeur, envisagez votre application et sa classe principale *UIWindow* comme une chaîne de télévision. Elles s'inscrivent dans le

cadre borné d'un téléviseur – l'écran de 320 × 480 pixels de l'iPhone – et obéissent à une "grille des programmes", que l'utilisateur peut librement choisir de suivre ou non. Chacune de vos émissions est une instance de la classe `UIView`. À tout moment, l'utilisateur peut interrompre leur diffusion, reprendre leur lecture ou éteindre le téléviseur. Les émissions font intervenir une série d'acteurs ou de personnalités : il s'agit des éléments d'interface que l'utilisateur manipule. L'élément `UIWindow` constitue ainsi le conteneur principal de votre application. Au sommet de la hiérarchie de votre projet, il regroupe les différents écrans présentés à l'utilisateur (`UIView`) qui mettent en scène tous les éléments visuels (`UILabel`, `UIButton`, etc.).

Plusieurs objets `UIWindow` au sein d'une application

Cette analogie vise à simplifier l'organisation hiérarchique de votre application et correspond à la plupart des projets diffusés sur l'App Store. Mais comme il est possible de posséder plusieurs téléviseurs à la maison, sachez qu'une application peut s'organiser autour de plusieurs instances de la classe `UIWindow`. Retenez toutefois qu'une application ne peut afficher qu'une seule instance de la classe `UIWindow` à la fois, comme on ne peut regarder qu'un seul téléviseur même si on en possède plusieurs à la maison ! C'est la logique de votre programme qui régit le basculement d'un objet `UIWindow` à l'autre. Mais dans la mesure du possible, essayez d'articuler votre projet autour d'un élément `UIWindow` unique à la racine qui organise la navigation au sein d'éléments `UIView` sous-jacents.

Au-delà de leur répercussion sur la programmation, sur laquelle nous reviendrons dans un instant, le design d'interfaces et le choix d'un système de navigation entre les vues définissent l'un des aspects essentiels de votre application : son ergonomie. En offrant aux développeurs une série de contrôles intuitifs et originaux, le SDK de l'iPhone assure une grande homogénéité entre les applications. Pour peu qu'ils aient déjà manipulés les utilitaires de base, tous vos utilisateurs connaissent leur rôle et leur fonctionnement... sur le bout des doigts ! Un système d'onglets leur permet de basculer entre une série de vues qui correspondent à autant d'activités différentes. Une barre de navigation supérieure assure le parcours général entre les fenêtres principales de l'application et leur permet de revenir à l'écran d'accueil. Grâce à une roulette ou une réglette, ils choisissent une valeur précise et modifient les vues en conséquence. Et ils naviguent entre les fenêtres à travers un léger effet d'animation, qui symbolise leur progression au sein d'une application.

Analysez les applications que vous utilisez : elles s'accompagnent rarement d'un manuel d'utilisation et vous devez les prendre en main de manière intuitive, en retrouvant instantanément l'ergonomie de l'iPhone (voir Figure 5.3 pour une exception !). Bien avant de développer vos différentes vues sous Xcode et Interface Builder, vous devez donc réfléchir sur papier à l'agencement de vos éléments d'interface et à votre système de navigation afin

qu'il soit fluide et clair en toutes circonstances. Ne sortez pas envers et contre tout des sentiers battus : on préférera toujours une application à l'interaction riche et ergonomique aux projets qui tentent de tout réinventer et qui s'avèrent complexes à manipuler. Toutefois, ne bridez pas votre imagination pour autant ; c'est en organisant vos vues de manière harmonieuse, autour de contrôles simples et logiques, que vous aboutirez à des résultats originaux et novateurs.

Figure 5.3

En adaptant sur iPhone le concept des jeux d'aventure point'n click, Episode 1112 s'articule autour d'un système de navigation très original et complexe : un manuel d'utilisation est nécessaire pour accompagner l'utilisateur. L'exception qui confirme la règle !



Choisir un système de navigation

Vos utilisateurs sont en droit d'attendre une ergonomie adaptée au type d'application que vous développez. Outre les jeux vidéo, qui constituent une classe à part, on distingue essentiellement deux types d'applications : celles visant à manipuler et à consulter des données et celles faisant office d'utilitaires. En fonction de l'une ou l'autre de ces deux catégories, vous imbriquez vos éléments de manière différente et vous disposez de systèmes de navigation privilégiés.

Consulter des données. La plupart des applications destinées à afficher du contenu s'articulent autour d'un système de listes pour hiérarchiser leurs éléments. En règle générale, leur vue principale se manipule verticalement : l'utilisateur fait défiler une succession d'éléments au doigt jusqu'à atteindre la fin de la liste. Une série de contrôles leur indique que certains éléments constituent des sous-listes imbriquées ou disposent de détails complémentaires. C'est notamment le cas de l'application *Téléphone*, la fonction première de l'iPhone, dont la liste des derniers appels affiche une flèche en regard de chaque entrée : chaque élément de la liste agit comme un simple "bouton" qui renvoie vers une seconde vue, détaillant les coordonnées de chaque appelant (voir Figure 5.4). Songez également aux listes qu'affiche l'application iPod : les artistes ou les albums apparaissent par ordre alphabétique, avec un index sur la droite permettant d'accélérer la navigation. En une pression du doigt sur la pochette d'un album, vous basculez vers la liste des morceaux qu'il contient et vous changez de vue.

Figure 5.4

Le nom des derniers appelants apparaît sous forme de liste : vous parcourez les données et vous accédez à des détails complémentaires, en changeant alors de vue, en touchant la flèche en regard de chaque ligne.



Simple et intuitive, cette disposition offre à vos utilisateurs une grande visibilité sur le contenu de votre application. En amont, vous avez trié les données auxquelles ils accèdent : vous avez donc identifié les principaux éléments qui méritent de figurer sur la vue racine et les utilisateurs ont la possibilité d'affiner leur navigation, en détaillant à chaque fois d'un niveau leur critère de parcours. Autour de ce principe clair et ergonomique, vous pouvez toutefois vous autoriser quelques fioritures. À l'instar de l'écran des réglages de l'iPhone, il est possible de regrouper certains éléments d'une liste en sections. Vous réunissez ainsi les données appartenant au même groupe d'informations et vous organisez le contenu de manière claire et hiérarchisée. Vous pouvez également offrir à vos utilisateurs la possibilité de modifier le contenu d'une liste. C'est notamment le cas de la fenêtre de création d'un nouveau contact : des boutons "+" sont disposés au début de chaque ligne et vous permettent d'accéder au clavier virtuel pour saisir le nom ou le numéro de téléphone d'un ami (voir Figure 5.5).

Figure 5.5

Les longues listes d'éléments ne sont pas nécessairement figées et vouées à la pure navigation : à l'image de la fenêtre d'édition de contacts, elles offrent des possibilités d'interaction et d'édition.



Interagir avec un outil. La seconde grande catégorie d'applications réunit des projets très divers : utilitaires à manipuler au quotidien, flux d'actualité, accès à un service web, outil de création, etc. En offrant plus d'un service à l'utilisateur, ces applications s'articulent autour d'un écran principal et d'une série de vues secondaires, correspondant à chaque fonctionnalité. Safari constitue un parfait exemple d'une telle disposition. Un bandeau de 320×44 pixels occupe le bord inférieur de l'écran (voir Figure 5.6).

Figure 5.6

Le navigateur Safari s'articule autour d'une barre d'outils en bas de l'écran, qui regroupe vos différentes possibilités d'interaction et de navigation.

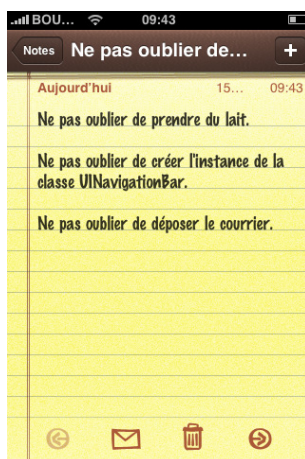


Il s'agit d'une barre d'outils, dont les boutons donnent accès aux fonctions essentielles (page précédente ou suivante, ajout d'un signet, consultation de l'historique, ouverture d'un onglet, etc.). Un tel élément d'interface correspond à une instance de la classe `UIToolbar` et se voit composé d'objets qui sont eux-mêmes des instances d'`UIToolbarItem`.

La plupart des écrans de telles applications arborent également une barre de navigation à leur sommet : on l'utilise généralement pour afficher le titre de la vue actuelle ainsi qu'un bouton permettant de revenir en arrière. Un espace à droite est éventuellement réservé à l'affichage d'un bouton contextuel, vous offrant la possibilité d'ajouter ou d'éditer un élément. L'outil *Notes* dispose d'une telle barre de navigation, qui affiche le titre de la note courante, un bouton pour revenir à l'écran général de sélection et un bouton + afin de créer une nouvelle note. Cette barre de navigation correspond à l'instance de la classe `UINavigationController` et les éléments qui la constituent sont des instances d'`UINavigationControllerItem`. C'est un système d'organisation commun aux applications égrenant du contenu et aux utilitaires : ce type de barre figure généralement en haut de longues listes, pour revenir facilement au niveau supérieur ou ajouter un élément à la liste en cours de lecture (voir Figure 5.7).

Figure 5.7

L'application de prise de notes dispose d'un double système de navigation : une barre de navigation, à son sommet, permet de qualifier la vue actuelle et d'accéder au menu principal tandis que la barre d'outils, en bas de l'écran, offre un accès aux différentes fonctions.



Enfin, de nombreux utilitaires s'appuient sur une navigation par onglets pour renvoyer l'utilisateur vers leurs différentes fonctionnalités. C'est notamment le cas de l'application iPod, dont la classification en Listes, Artistes, Morceaux et Albums correspond à autant d'onglets en bas de l'écran (voir Figure 5.8). Là encore, cette barre emprunte les dimensions traditionnelles de 320 × 44 pixels. Il s'agit d'une instance de la classe `UITabBar` dont les éléments enfants sont des objets `UITabBarItem`.

Vous pouvez ainsi privilégier le système de navigation de votre choix et même les combiner pour offrir une meilleure ergonomie à votre application. Sachez toutefois qu'en affichant simultanément une barre de navigation et une barre d'outils, la zone principale de votre application ne disposera plus que de 320 × 416 pixels d'espace libre (voir Figure 5.9) – ou 416 × 320 pixels en mode paysage.

Figure 5.8

L'application iPod découpe ses différentes vues en autant d'onglets, accessibles à travers une barre en bas de l'écran.

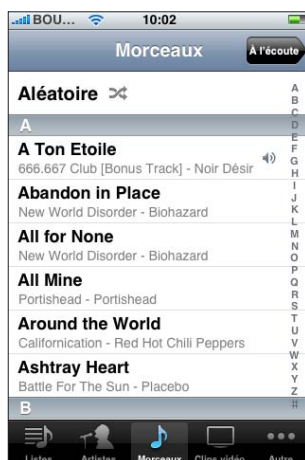


Figure 5.9

L'espace dont dispose le cœur de votre application se réduit si vous multipliez les systèmes de navigation.



Vous devez ainsi parfois ruser : de nombreuses applications qui n'appellent qu'une simple vue supplémentaire de configuration utilisent un bouton "i" logé en bas à droite de l'écran. En cliquant dessus, l'utilisateur fait permuter la vue principale qui semble ainsi se retourner dans un léger effet d'animation pour révéler son verso. Il s'agit bien du chargement d'une nouvelle vue, qui contient une série de réglages permettant de personnaliser le comportement de l'application.

Au-delà du choix du système de navigation, ce dernier point illustre un second aspect que vous devez prendre en compte au moment de hiérarchiser les vues de votre application : les effets d'animation et de transition. Le SDK de l'iPhone vous aide dans cette tâche et propose une série de transitions prêtes à l'emploi, qui se lancent au moment de changer de vue. La solution la plus simple reste la modeste translation d'un écran à l'autre, de manière horizontale. C'est le principe retenu dans le parcours des réglages ou de votre photothèque par exemple. D'autres effets plus ambitieux simulent le "tournement" d'une page ou un fondu enchaîné (voir Figure 5.10).

Figure 5.10

L'effet de transition qui apparaît en cliquant sur le bouton "i" de l'application MiniPiano : la vue principale permute pour révéler son "dos", qui est en réalité une seconde vue.



Organiser une application multivues

Toutes les applications multivues reposent sur le même type de canevas. Lorsque vous créez un nouveau projet sous Xcode, l'éditeur crée automatiquement un fichier `MainWindow.xib` qui correspond à la fenêtre principale de votre application. Le basculement entre une série de vues repose sur la création d'une classe contrôleur ; elle gère les autres vues et les affiche en fonction des choix de l'utilisateur. Ce contrôleur est une instance de la classe `UIViewController` ou plus précisément des classes `UINavigationController` (dans le cas d'une barre de navigation) ou `UITabBarController` (dans le cas d'une barre d'onglets). Ces dernières sont elles-mêmes des sous-classes d'`UIViewController`.

Lorsque vous instaurez un contrôleur de navigation, vous devez bien distinguer le comportement de cet outil de navigation de celui de vos différentes vues. Dans une application comportant une barre d'onglets, par exemple, les pressions sur les onglets interagissent avec le contrôleur de la classe `UITabBarController` : il est responsable de l'affichage des différentes vues. À l'inverse, chaque vue dispose de son propre contrôleur : comme nous l'avons vu précédemment, il est donc possible de déclarer une série d'outlets et d'actions permettant à l'utilisateur d'interagir avec l'application et de déclencher des traitements. Ainsi, lorsque vous manipulez un bouton figurant sur une vue, il va déclencher un événement pris en charge par le contrôleur de la vue actuelle.

Comme nous allons le voir à travers cette série d'exemples, il n'est pas nécessaire d'associer un fichier `.xib` à chaque vue. Vous pouvez tout à fait déposer une série d'objets `UIView` dans le fichier `MainWindow.xib` principal et tisser des liens entre vos boutons et ces vues supplémentaires : votre contrôleur gère alors l'affichage ou la disparition des différentes vues, en ajoutant éventuellement un effet de transition.

De manière générale, le développement d'applications multivues implique une grande rigueur. Vous allez très rapidement vous retrouver avec une série de contrôleurs différents associés à chaque vue, mais aussi avec un contrôleur permettant de gérer leur basculement et éventuellement des fichiers `nib` distincts. La création d'outlets et d'actions entre toutes ces vues sous Interface Builder peut alors sembler pénible ; mais en agissant à pas mesurés et en faisant preuve d'organisation, vous devriez être en mesure de vous y retrouver sans encombre. Pour ne pas vous perdre, retenez ce concept général : lorsque vous créez la classe `MonContrôleur`, Xcode charge automatiquement l'interface `MonContrôleur.xib`. Veillez à soigner tout particulièrement le nom de vos classes pour gagner en clarté.

La navigation par onglets

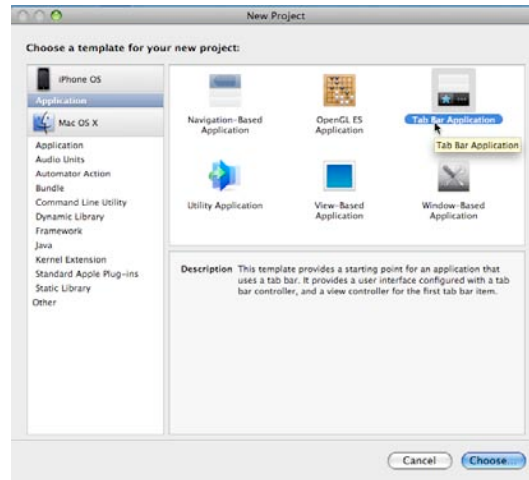
Puisque vous maîtrisez l'aspect théorique de bout en bout, passons à la pratique ! Dans ce premier exemple, nous allons organiser une série de vues autour d'un système de navigation

par onglets. Vous pouvez ainsi l'exploiter pour étoffer le quiz du chapitre précédent : chaque onglet mène vers différents niveaux de difficulté ou vers des questionnaires sur des sujets distincts.

Ouvrez Xcode puis déroulez le menu File et cliquez sur New Project. Parmi la liste des modèles de projets, sélectionnez Tab Bar Application et validez en cliquant sur le bouton Choose (voir Figure 5.11).

Figure 5.11

La création de notre premier système de navigation sous Xcode, à l'aide du modèle Tab Bar Application.



Baptisez le projet MultiQuiz et cliquez sur le bouton Save. Vous constatez d'emblée sous Xcode que le canevas général a changé. En lieu et place du traditionnel fichier MultiQuizViewController.h que vous étiez en droit d'attendre, vous découvrez des fichiers FirstViewController.h et SecondView.xib. Cette disposition doit vous mettre la puce à l'oreille : en organisant votre application autour d'une série d'onglets, vous devez définir chacune des vues de manière distincte. L'interface MainWindow.xib définit le système de navigation et constitue l'élément racine de la hiérarchie de vos vues. À l'aide d'onglets, vous pointez alors vers une seconde vue dont l'interface est définie dans SecondView.xib.

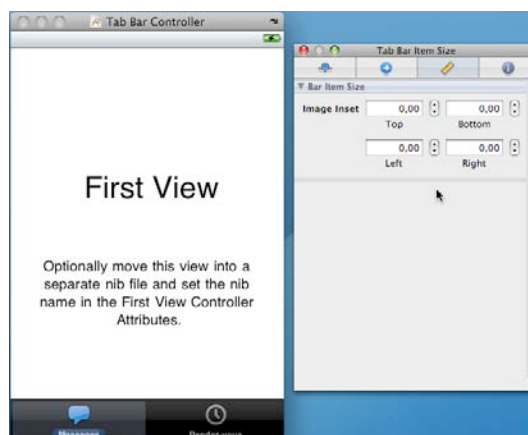
Commencez par ouvrir le fichier MainWindow.xib pour lancer Interface Builder et découvrir le canevas préparé par Xcode. Là encore, les outils de développement de l'iPhone s'avèrent aussi intuitifs que puissants : il vous suffit de cliquer sur les deux onglets figurant en bas de la fenêtre principale pour ouvrir directement la seconde vue, chargée depuis SecondView.xib. Il est donc inutile de revenir systématiquement à la fenêtre de votre projet, sous Xcode, pour dessiner vos différentes vues. Effectuez un double-clic sur chaque onglet figurant en bas de la fenêtre pour les renommer. Vous constatez par ailleurs que la barre soutenant les onglets ne peut pas être redimensionnée : elle occupe systématiquement 44 pixels de haut

et s'étend sur toute la largeur de l'écran. Vous pouvez en revanche l'égayer par une série d'icônes au-dessus du texte, que vous définissez à l'aide de l'inspecteur d'attributs (Ctrl+1).

Revenez sous Xcode, déroulez le menu Project et choisissez Add to Project. Sélectionnez ensuite vos images, qui doivent d'abord être enregistrées au format PNG en niveau de gris à l'aide de votre éditeur préféré. Elles s'intègrent ainsi aux fichiers sources de votre projet. De retour sous Interface Builder, il vous suffit de sélectionner l'un des onglets puis d'ouvrir l'inspecteur d'attributs et de dérouler le menu Image pour découvrir la liste des fichiers que vous avez importés. Basculez vers l'onglet Item Size, en forme de règle, pour définir leur aspect à l'aide des réglages Image Inset. Vous avez ainsi la possibilité d'afficher des onglets dépourvus de texte et n'arborant qu'une image sur toute leur hauteur (voir Figure 5.12).

Figure 5.12

On définit la taille des icônes qui illustrent chaque onglet dans l'inspecteur d'attribut d'Interface Builder.



Cliquez ensuite sur le second onglet, puis sur le lien SecondView.nib qui apparaît au centre de l'interface (voir Figure 5.13).

Figure 5.13

Notre application comprend désormais deux onglets, qui chargent des fichiers nib différents.



Comme nous l'avons vu précédemment, chaque vue est définie par une instance de la classe `UIView`. Par défaut, Interface Builder a disposé un tel objet et l'a complété d'un libellé et d'un champ de texte. Vous pouvez alors les supprimer et ajouter tous les éléments de votre choix. Libellés, boutons de contrôle ou champs de texte : libre à vous de définir l'allure et le rôle de votre seconde vue !

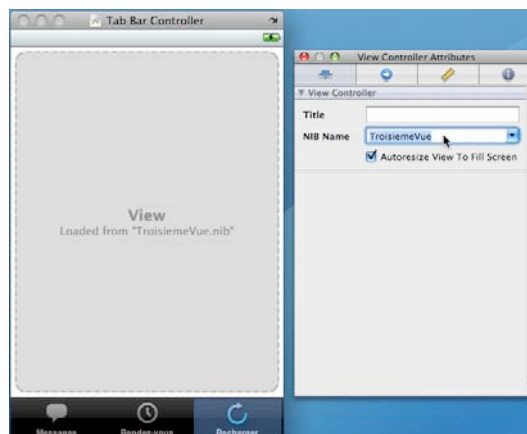
Télécharger des images supplémentaires

La création de vos propres icônes pour enrichir votre barre d'onglets est parfois fastidieuse : non seulement vous devez les enregistrer en niveau de gris, mais elles exigent aussi un certain talent pour les rendre harmonieuses et lisibles ! Si vous ne vous sentez pas en mesure d'affronter un tel exercice, reportez-vous sur les nombreuses bibliothèques libres de droit. Parmi celles-ci, nous vous conseillons en particulier les créations de Glyphish, auxquelles vous accédez à l'adresse www.glyphish.com. La bibliothèque regroupe plus de 130 éléments gratuits et prêts à l'emploi, qui symbolisent de très nombreuses fonctions. Chaque icône représente 30×30 pixels et vous les importez à votre projet sous Xcode pour les placer sous Interface Builder dans un second temps.

Vous souhaitez ajouter un troisième onglet ? Cet exercice illustre parfaitement tous les rouages de Xcode et d'Interface Builder à travers une manipulation que vous effectuerez fréquemment. Commencez par ouvrir `MainWindow.xib`. Dans la fenêtre principale, vous découvrez votre contrôle `Tab Bar Controller` qui définit une série d'onglets. À partir de la bibliothèque, déposez un objet `Tab Bar Item` sur votre barre d'onglet. Les dimensions des boutons précédents s'adaptent automatiquement pour céder la place à ce nouvel élément. Sélectionnez cet onglet et ouvrez l'inspecteur d'attributs. Rendez-vous à l'onglet `View Controller`, figurant en haut à gauche, et saisissez `TroisiemeVue` dans le champ `NIB Name`. La fenêtre principale de l'interface indique alors, à l'onglet sélectionné : `Loaded from TroisiemeVue.nib` (voir Figure 5.14).

Figure 5.14

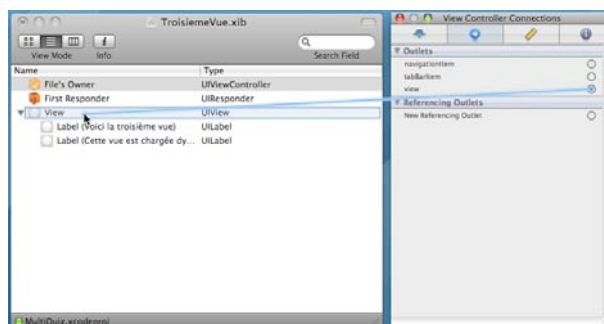
On définit le fichier nib pointé par le troisième onglet.



Retournez sous Xcode, déroulez le menu File et cliquez sur New File. Dans le volet gauche de la fenêtre, sélectionnez la catégorie User Interfaces sous la section iPhone OS puis cliquez sur Window XIB et sur le bouton Next. Saisissez *TroisiemeVue* en guise de nom et validez en cliquant sur Finish. Xcode prépare ainsi un fichier .xib vierge, correspondant à l'interface du troisième onglet. Ouvrez-le sous Interface Builder. Vous constatez qu'il intègre un objet UIWindow dont nous n'avons pas besoin. Supprimez-le et remplacez-le par une instance de la classe UIView, en déplaçant un objet View sur le canevas. Ajoutez ensuite un libellé et des champs de texte éventuels. Avant d'enregistrer et de compiler le projet, vous devez vérifier les dépendances entre les écrans et les classes qui leur sont associées. Par défaut, le File's Owner de *TroisiemeVue.xib* est un objet NSObject. Cliquez sur son intitulé puis sur le bouton "i" de l'inspecteur d'attributs afin d'ouvrir le panneau Object Identity. Dans le champ Class, saisissez *UIViewController*, puis cliquez sur le second onglet de l'inspecteur afin d'ouvrir le panneau View Controller Connections. Cliquez sur le cercle à droite du champ view et déplacez la souris jusqu'à l'élément view de la fenêtre *TroisiemeVue.xib* (voir Figure 5.15).

Figure 5.15

On modifie les propriétés de la troisième vue afin de la relier au canevas principal.



Vous faites ainsi pointer l'outlet vers la vue de votre interface et vous indiquez à Interface Builder que c'est le File's Owner qui est responsable du chargement de votre vue. Enregistrez tous les fichiers puis compilez le projet sous Xcode : vos différents onglets figurent en bas du Simulateur d'iPhone et vous basculez de l'un à l'autre au moindre clic (voir Figure 5.16).

Comme vous l'avez constaté, il est très simple de partir d'un modèle de fichier nib créé par Xcode et de l'intégrer à l'interface de votre application pour l'enrichir sans revoir fondamentalement le code source de votre projet. Vous devez toutefois prêter attention aux connexions et vérifier que les classes correspondantes sont correctement définies : nous aurons tout le loisir de compliquer l'exercice, en ajoutant des contrôleurs distincts, dans la suite de ce chapitre.

Figure 5.16

Après avoir compilé le projet, nos trois onglets figurent dans la barre en bas du Simulateur d'iPhone.



La navigation par boutons

Au-delà de la navigation par onglets entre une série d'écrans, vous pouvez préférer disposer des boutons sur votre interface afin d'accéder à des vues supplémentaires.

C'est l'un des rôles essentiels des utilitaires : vous développez une série de fonctionnalités que l'utilisateur lance individuellement en pressant des boutons. À tout moment, il revient à l'écran d'accueil à l'aide de la barre de navigation. Nous allons entrevoir la création d'une telle application à travers un second exemple.

Sous Xcode, déroulez le menu File et cliquez sur New Project. Choisissez le modèle Window-Based Application et saisissez `MultiVues` en guise de nom de projet. Dans le cadre de cet exemple, nous allons jongler entre deux vues. Vous devez ainsi préparer les classes qui leur sont associées. Déroulez à nouveau le menu File et cliquez sur New File. Dans la fenêtre qui apparaît, sélectionnez la catégorie Cocoa Touch Classes, puis le type de fichier `UIViewController subclass`. Cliquez sur le bouton Next puis saisissez `RacineViewController`. Renouvelez l'opération et créez la classe `VueDeuxController`. Vous disposez ainsi des six fichiers suivants au sein de votre projet :

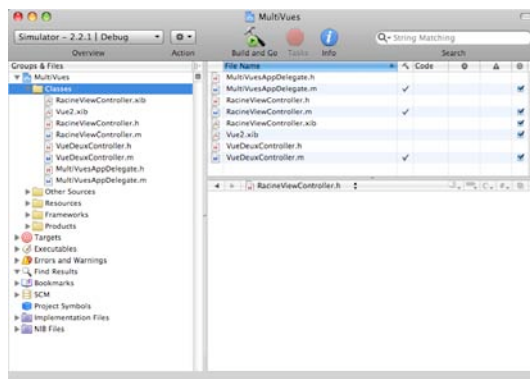
- `MultiVuesAppDelegate.h`, l'interface du délégué de l'application ;
- `MultiVuesAppDelegate.m`, l'implémentation du délégué de l'application ;
- `RacineViewController.h`, l'en-tête du contrôleur de notre première vue ;
- `RacineViewController.m`, l'implémentation du contrôleur de notre première vue ;

- `VueDeuxController.h`, l'en-tête du contrôleur de notre seconde vue ;
- `VueDeuxController.m`, l'implémentation du contrôleur de notre seconde vue.

Nous devons également ajouter deux fichiers d'interface, correspondant à chacune des vues. Sélectionnez le dossier Resources de votre projet, déroulez le menu File et cliquez sur New File. Choisissez la catégorie User Interfaces puis cliquez sur View XIB et sur le bouton Next. Créez successivement les fichiers `RacineViewController.xib` et `Vue2.xib` par ce biais (voir Figure 5.17).

Figure 5.17

L'allure de notre projet à ce stade : on a créé les classes et les fichiers d'interface correspondants aux deux vues.



Ouvrez ensuite `MultiVuesAppDelegate.h` et remplacez son code par le Listing 5.1.

Listing 5.1 : *MultiVuesAppDelegate.h*

```
#import <UIKit/UIKit.h>

@interface MultiVuesAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    UINavigationController *navigationController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UINavigationController
    *navigationController;

@end
```

Cliquez enfin sur `MultiVuesAppDelegate.m` et saisissez le code du Listing 5.2.

Listing 5.2 : *MultiVuesAppDelegate.m*

```
#import "MultiVuesAppDelegate.h"

@implementation MultiVuesAppDelegate

@synthesize window;
@synthesize navigationController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after application launch
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [navigationController release];
    [window release];
    [super dealloc];
}

@end
```

À travers ces quelques lignes, nous avons instancié un objet de la classe `UINavigationController` (une barre de navigation) sous la variable `navigationController` et nous l'avons ajouté à la fenêtre principale de l'application lors de son chargement. Nous devons à présent tisser un lien entre les deux vues et préparer une méthode visant à basculer de l'une à l'autre. Ouvrez le fichier `RacineViewController.h` et saisissez le code qui suit.

Listing 5.3 : *RacineViewController.h*

```
#import <UIKit/UIKit.h>
#import "VueDeuxController.h";

@interface RacineViewController : UIViewController {
    VueDeuxController *vueDeuxController;
}

@property (nonatomic, retain) VueDeuxController *vueDeuxController;

- (IBAction)changerVue:(id)sender;

@end
```

Nous venons ici de créer la variable d'instance de `VueDeuxController`, nous avons défini ses propriétés et nous avons déclaré la méthode `changerVue`. Nous devons à présent définir

la logique de cette méthode, dans le fichier `RacineViewController.m`. Reportez-vous au Listing 5.4 pour adapter le fichier en conséquence.

Listing 5.4 : *RacineViewController.m*

```
#import "RacineViewController.h"
#import "VueDeuxController.h";

@implementation RacineViewController

@synthesize vueDeuxController;

- (IBAction)changerVue:(id)sender {
    if(self.vueDeuxController == nil) {
        VueDeuxController *vueDeux = [[VueDeuxController alloc]
            initWithNibName:@"Vue2" bundle:[NSBundle mainBundle]];
        self.vueDeuxController = vueDeux;
        [vueDeux release];
    }

    [self.navigationController pushViewController:self.vueDeuxController
        animated:YES];
}

@end
```

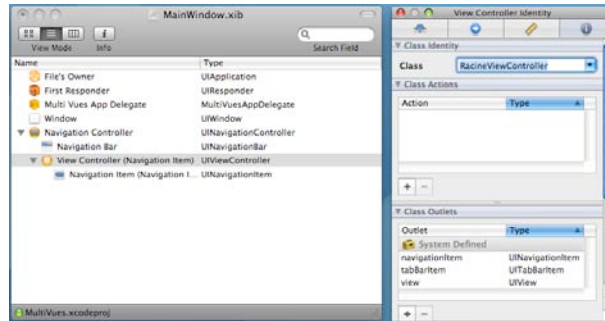
Comme nous l'avons vu précédemment, `RacineViewController` est le contrôleur racine et il est en charge du basculement entre les différentes vues. Nous commençons donc par importer la classe `VueDeuxController.h` et nous profitons des accesseurs et des mutateurs simplifiés d'Objective-C à l'aide des méthodes `@property` et `@synthesize`. Nous définissons ensuite notre méthode `changerVue`, une action qui sera appelée en pressant un simple bouton. La boucle conditionnelle du Listing 5.4 vérifie tout d'abord si la variable d'instance `vueDeuxController` est créée. Si ce n'est pas le cas, nous l'instancions et nous l'associons au fichier `Vue2.xib` – la méthode `initWithNibName` complète automatiquement la valeur "Vue2" par son extension, selon la tradition de l'Objective-C. Nous "poussons" enfin cette nouvelle vue à l'avant-plan de l'application, à l'aide de la méthode `pushViewController`. Nous profitons au passage de l'attribut `animated`, qui attend une valeur booléenne, pour ajouter un effet d'animation en guise de transition entre la vue active et celle contenue dans `Vue2.xib`.

C'est précisément la tâche qui nous incombe à présent : définissons les différentes vues de notre application et ajoutons le bouton permettant de basculer de l'une à l'autre. Commencez par ouvrir le fichier `MainWindow.xib` en effectuant un double-clic sur son intitulé, sous Xcode. Vous accédez à Interface Builder. Vous devez tout d'abord ajouter un contrôleur responsable du changement de vue. Dans cet exemple, déposez l'élément `Navigation`

Controlller dans la fenêtre principale de MainWindow.xib, juste en-dessous de l'objet Window. Déroulez l'arborescence en face de Navigation Controller et sélectionnez l'objet View Controller (voir Figure 5.18).

Figure 5.18

On modifie la classe associée au contrôleur de navigation.



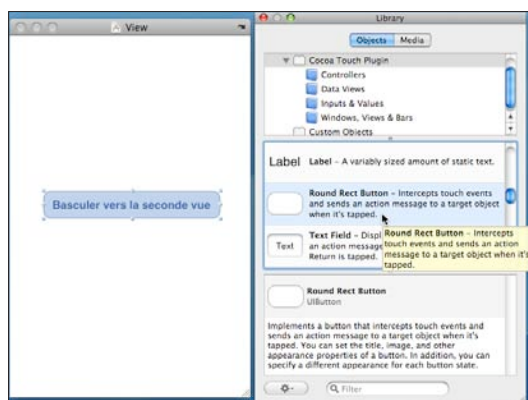
Ouvrez alors l'inspecteur d'identité (Cmd+4) et changez le nom de la classe en saisissant `RacineViewController` à la place d'`UIViewController`. Interface Builder reconnaît automatiquement les classes, les actions et les outlets que vous avez créés et complète le nom que vous saisissez dans le champ. Vérifiez au passage que tous les éléments principaux sont bien reliés entre eux : les outlets delegate de `File's Owner` et `navigationController` de `Navigation Controller` doivent tous deux pointer vers le délégué de l'application. Sélectionnez à nouveau `RacineViewController`, basculez ensuite vers le panneau des attributs (Cmd+1) et saisissez `RacineViewController` dans le champ `NIB Name`. Vous indiquez à Interface Builder qu'il doit charger le fichier `RacineViewController.xib`.

Le résultat ne se fait pas attendre : la fenêtre principale de votre application se voit affublée d'une barre de navigation et son contenu principal est chargé depuis `RacineViewController.xib`. Nous devons précisément le modifier. Enregistrez vos modifications, retournez sous Xcode et effectuez un double-clic sur `RacineViewController.xib`. Il s'agit de la vue principale : modifiez son interface comme bon vous semble, en ajoutant un bouton (Round Rect Button) que l'utilisateur devra presser afin de basculer vers la seconde vue (voir Figure 5.19).

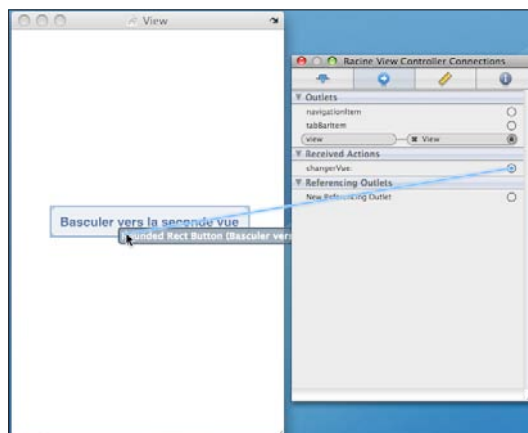
Cliquez ensuite sur l'objet `File's Owner` de la fenêtre `RacineViewController.xib`, ouvrez à nouveau l'inspecteur d'identité et saisissez `RacineViewController` dans le champ `Class`. Pressez enfin la touche `Ctrl` du clavier en cliquant sur `File's Owner` et maintenez la souris enfoncée jusqu'à l'objet `View`. Relâchez la souris et sélectionnez `view` dans le menu contextuel : vous venez d'associer l'outlet correspondant à votre vue (voir Figure 5.20). Ouvrez l'inspecteur de connexions (Cmd+2) et cliquez sur le cercle à droite de l'action `changerVue`. Maintenez le bouton de la souris enfoncé et glissez-la jusqu'au bouton que vous avez créé. Sélectionnez enfin l'événement `Touch Up Inside`.

Figure 5.19

On définit l'allure de notre vue principale, en ajoutant un bouton pour basculer vers la seconde vue.

**Figure 5.20**

On associe notre action `changerVue` au bouton créé sur la vue principale.



Sauvegardez vos modifications, puis retournez sous Xcode et ouvrez à présent le fichier `Vue2.xib`. Il s'agit donc de la seconde vue de votre application. Là encore, commencez par ajouter tous les éléments de votre choix à l'interface. Cliquez ensuite sur l'objet `File's Owner`, ouvrez l'inspecteur d'identité et saisissez `VueDeuxController` dans le champ `Class`. Cliquez en maintenant la touche `Ctrl` enfoncée sur son icône et déplacez la souris jusqu'à l'objet `View`. Enregistrez à nouveau le projet et pressez les touches `Cmd+R` pour compiler le code.

Vous découvrez alors le résultat dans le Simulateur d'iPhone : le bouton sur la vue principale vous permet de charger la seconde vue. Pour revenir à l'écran d'accueil, il suffit de presser l'unique bouton de la barre de navigation, en haut du terminal (voir Figure 5.21). Vous remarquez au passage un léger effet d'animation, à travers lequel les vues basculent de manière horizontale.

Figure 5.21

Nous changeons de vue à volonté, en pressant un simple bouton et en maîtrisant la barre de navigation.



Vous pouvez parfaitement adapter cette application en remplaçant le contrôleur `NavigationController` par l'élément de votre choix, comme un objet `TabBarController` par exemple. Par ailleurs, vous remarquerez que la classe contrôleur `VueDeuxController.h` n'a pas du tout été modifiée. C'est ici que vous définirez la logique interne liée à la seconde vue et que vous disposerez vos méthodes spécifiques, dans la veine du quiz figurant au chapitre précédent, par exemple. Ce sera précisément l'objet de notre prochaine application complète.

La navigation sous forme de listes et de tableaux

Le troisième grand type de système de navigation, basé sur le parcours de tableaux contenant des données, est détaillé au Chapitre 7. À travers un exemple concret, vous découvrirez comment communiquer avec de larges bases de données afin d'en extraire le contenu et de le hiérarchiser autour de sections personnalisées. Sans déflorer prématurément le sujet, sachez qu'un tel système de navigation s'articule autour d'un contrôleur `UITableViewController`, dont le fonctionnement est très proche des deux contrôleurs mis en application à travers les exemples précédents.

Modifier l'animation de transition

Comme nous l'avons vu précédemment, une application multivues charge une série d'écrans dont le basculement est géré par un contrôleur racine, qui les affiche ou les masque en fonction des choix de l'utilisateur. Les transitions entre ces vues bénéficient d'un léger effet d'animation. L'effet le plus simple correspond à l'attribut `animated` de la méthode `pushViewController` : on "pousse" littéralement la vue suivante dans un mouvement hori-

zontal (voir Figure 5.22). Au-delà de son aspect sobre et épuré, cette animation présente un suprême avantage : elle n'alourdit pas votre code source. Vous pouvez en revanche soigner vos transitions et leur consacrer de plus larges blocs de code, en profitant des méthodes `beginAnimations` et `commitAnimations` de la classe `UIView`.

Figure 5.22

Une simple animation de transition où les vues basculent horizontalement.



Le principe demeure simple : vous définissez votre type d'animation en profitant de constantes prédéfinies (transition de gauche à droite, de droite à gauche, transition qui "retourne" une vue, etc.), vous déclarez sa durée en secondes et vous gérez l'apparition ou la disparition des vues.

Partons de l'application précédente et remplaçons la méthode `pushViewController`. Dans la mesure où nous allons définir plusieurs états en fonction de la vue affichée à l'écran, vous devez légèrement retoucher le fichier `RacineViewController.h`, qui correspond au contrôleur racine gérant vos deux vues. Ouvrez à nouveau ce fichier sous Xcode et remplacez son contenu par le code qui figure au Listing 5.5.

Listing 5.5 : *RacineViewController.h*

```
#import <UIKit/UIKit.h>
#import "VueDeuxController.h";

@interface RacineViewController : UIViewController {
    VueDeuxController *vueDeuxController;
    RacineViewController *racineViewController;
}

@property (nonatomic, retain) VueDeuxController *vueDeuxController;
@property (nonatomic, retain) RacineViewController *racineViewController;
```

```
- (IBAction)changerVue:(id)sender;

@end
```

C'est toujours l'action `changerVue` qui est responsable de la permutation entre les deux vues. Nous avons ici simplement déclaré la variable d'instance `racineViewController` en l'associant à notre contrôleur `racine`. Nous avons ensuite défini ses propriétés. À ce titre, n'oubliez surtout pas d'appeler `@synthesize` dans l'en-tête du fichier `RacineViewController.m`, comme suit :

```
@synthesize vueDeuxController, racineViewController;
```

Toujours dans le fichier `RacineViewController.m`, remplacez l'action `changerVue` par la déclaration figurant au Listing 5.6.

Listing 5.6 : Modification de l'action *changerVue* dans *RacineViewController.m*

```
- (IBAction)changerVue:(id)sender {

    [UIView beginAnimations:@"ChangementVue" context:nil];
    [UIView setAnimationDuration:2.0];
    if (vueDeuxController.view.superview == nil) {
        VueDeuxController *vueDeux = [[VueDeuxController alloc]
            initWithNibName:@"Vue2" bundle:[NSBundle mainBundle]];
        self.vueDeuxController = vueDeux;
        [vueDeux release];

        [UIView setAnimationTransition:UIViewAnimationTransitionFlip
            FromRight forView:self.view cache:YES];
        [vueDeuxController viewWillAppear:YES];
        [racineViewController viewWillDisappear:YES];
        [racineViewController.view removeFromSuperview];
        [self.view insertSubview:self.vueDeuxController.view atIndex:0];
        [racineViewController viewDidDisappear:YES];
        [vueDeuxController viewDidAppear:YES];
    }
    else {
        [UIView setAnimationTransition:UIViewAnimationTransitionFlip
            FromLeft forView:self.view cache:YES];
        [racineViewController viewWillAppear:YES];
        [vueDeuxController viewWillDisappear:YES];
        [vueDeuxController.view removeFromSuperview];
        [self.view insertSubview:self.racineViewController.
            view atIndex:0];
        [vueDeuxController viewDidDisappear:YES];
        [racineViewController viewDidAppear:YES];
    }
    [UIView commitAnimations];
}
```

Enregistrez votre projet puis compilez-le. Vous découvrez sans plus attendre le résultat dans le Simulateur d'iPhone : en cliquant sur le bouton de la vue principale, l'écran tourne depuis la droite pour révéler la seconde vue. Renouvelez l'opération et la transition s'effectue dans le sens inverse, jusqu'à revenir à l'écran initial (voir Figure 5.23).

Figure 5.23

Notre nouvelle transition met en scène la rotation des différentes vues.



Malgré son apparente lourdeur, cette approche reste simple et logique. Vous commencez par définir un bloc d'animation à l'aide de l'envoi de message `[UIView beginAnimations]`. Vous précisez ensuite sa durée, exprimée en secondes – dans cet exemple, la transition s'effectue en deux secondes. Vous devez enfin prévoir l'animation en fonction de la vue actuellement affichée à l'écran. C'est précisément là qu'interviennent nos variables d'instance `vueDeuxController` et `racineViewController`, qui correspondent aux deux vues et qui permettent de pointer facilement vers chacune d'elles. Si la seconde vue n'apparaît pas à l'écran, nous l'initialisons et nous définissons une transition de droite à gauche. Les méthodes `viewWillAppear`, `viewWillDisappear` et leurs pendants `viewDidAppear` et `viewDidDisappear` attendent des booléens et vous permettent de gérer précisément l'affichage et la disparition de vos différentes vues. Notez au passage que l'attribut `cache` accélère l'effet de transition en réalisant un "cliché" aplati de vos vues au début de l'animation : on utilise alors ces images pour afficher l'effet au lieu de déplacer d'un seul bloc tous les contrôles présents sur chaque vue.

Les appels aux méthodes `viewWillAppear`, `viewWillDisappear`, `viewDidAppear` et `viewDidDisappear` jouent un rôle prépondérant lorsque vous souhaitez effectuer des traitements juste avant le début ou après la fin d'une animation. C'est souvent le cas lorsque vous traitez des vues qui sont des sous-classes spécifiques d'`UIView`. La seconde portion de la boucle conditionnelle traite le cas contraire : la seconde vue est actuellement affichée à

l'écran et on souhaite retourner à la première. Nous inversons alors l'effet de transition, en appelant la constante `UIViewAnimationTransitionFlipFromLeft`. Il existe à l'heure actuelle quatre constantes différentes :

- `UIViewAnimationTransitionFromLeft`, un basculement depuis la gauche ;
- `UIViewAnimationTransitionFromRight`, un basculement depuis la droite ;
- `UIViewAnimationTransitionCurlUp`, un tournement de page depuis le bas ;
- `UIViewAnimationTransitionCurlDown`, un tournement de page depuis le haut.

N'hésitez pas à tester ces différents effets en remplaçant les constantes au Listing 5.6. Après avoir défini le bloc d'animation, vous exécutez l'effet en appelant `commitAnimations`.

Figure 5.24

Décomposition de l'animation de gauche à droite, avec la constante `UIViewAnimationTransitionFromLeft`.



MultiConvertisseur, une application multivues composée d'actions et d'outlets

Afin de clore notre tour d'horizon des systèmes de navigation et des applications multivues, attelons-nous à un dernier exemple concret. *MultiConvertisseur* (voir Figure 5.25) est une application complète qui vise à illustrer tous les enseignements de ce chapitre. On la manipule à travers une barre d'onglets, qui donne accès à autant de vues : chaque contrôleur prend en charge une série d'opérations et convertit des unités de température, de mesure et de poids.

Figure 5.25

*Notre application
MultiConvertisseur, telle
qu'on souhaite la développer.*



Plaçons-nous dans les conditions réelles et commençons par esquisser le cahier des charges de MultiConvertisseur. Le rôle de l'application est plutôt simple et nous devons réunir les fonctions suivantes :

- Un contrôleur se charge d'afficher les différents onglets.
- La barre d'onglets renvoie automatiquement vers une série de fichiers nib individuels, associés à chaque fonction du convertisseur.
- On distingue alors les fonctions de conversion de températures (degrés Celsius vers Fahrenheit et réciproquement), de longueurs (centimètres en pouces) et de poids (kilogrammes en livres).
- L'utilisateur saisit ses valeurs dans des champs à l'aide du clavier virtuel et lance la conversion en cliquant sur un bouton.
- Avec une telle disposition, il sera facile d'ajouter d'autres unités à convertir, en ne retouchant que les contrôleurs et l'interface associés à la fonction considérée.

Le développement paraît ainsi plus clair et mieux balisé. Pour une application plus ambitieuse, n'hésitez pas à prendre le temps de réfléchir au design des interfaces sur papier et à tracer toutes les interactions du dialogue homme-machine.

Ouvrez Xcode, déroulez le menu File et cliquez sur New Project. Sélectionnez le modèle Tab Bar Application et saisissez MultiConvertisseur en guise de nom de projet. Nous allons commencer par réarranger les fichiers préparés par Xcode et par créer les classes dont nous avons besoin. La barre inférieure se compose de quatre onglets : Infos (écran d'accueil), Températures, Longueurs et Poids. Créez les contrôleurs associés en maintenant la touche Ctrl enfoncée et en cliquant sur le dossier Classes de votre projet. Choisissez ensuite

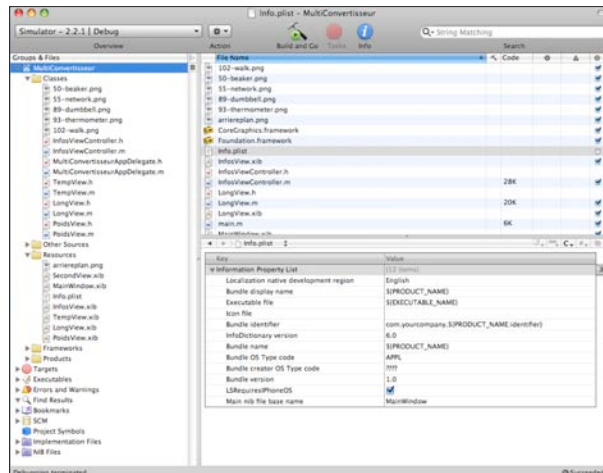
Add > New File et sélectionnez le modèle UIViewController subclass, dans la catégorie Cocoa Touch Classes. Recommencez l'opération afin de créer les fichiers suivants :

- InfosViewController.h et .m.
- TempView.h et .m.
- LongView.h et .m.
- PoidsView.h et .m.

Procédez de même en cliquant sur le dossier Resources afin d'ajouter les fichiers nib correspondants à chaque vue. Cette fois, sélectionnez le modèle View XIB de la catégorie User Interfaces et créez les fichiers InfosView.xib, TempView.xib, LongView.xib et PoidsView.xib (voir Figure 5.26).

Figure 5.26

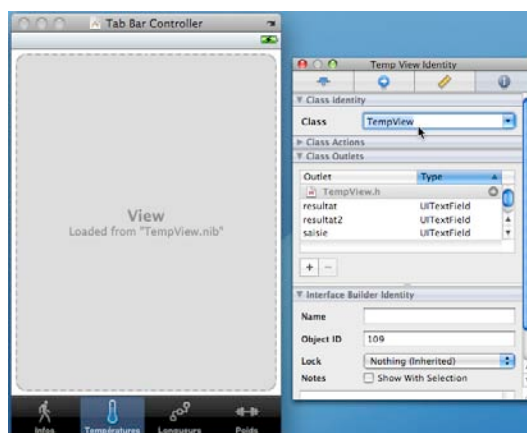
L'organisation initiale de notre projet sous Xcode.



Ouvrez le fichier MainWindow.xib et ajoutez deux autres onglets depuis la bibliothèque, en glissant des objets Tab Bar Item vers la barre en bas de l'écran. Renommez chaque onglet et affichez l'inspecteur d'identité (Cmd+4). Vous devez associer les nouvelles classes à vos onglets : dans le champ Class, saisissez InfosViewController pour le premier onglet et ainsi de suite. Dans l'inspecteur d'attributs (Cmd+1), n'oubliez surtout pas de remplacer le champ NIB Name par le nom du fichier nib associé. Ainsi, l'onglet Température pointe vers la classe et le nib TempView. Sous Interface Builder, vous pouvez basculer d'un onglet à l'autre afin de vérifier que toutes les données ont été correctement liées et que vos quatre vues apparaissent (voir Figure 5.27).

Figure 5.27

Les connexions ont été réalisées au sein de l'interface principale et l'application bascule d'un onglet à l'autre.



Affichez à présent l'interface TempView.xib. Nous vous laissons le soin d'agencer les éléments comme bon vous semble, mais sachez que vous devrez ajouter au moins quatre champs de texte et deux boutons pour lancer les fonctions escomptées. Cliquez sur File's Owner et associez-lui la classe TempView. Vous pourrez ainsi tisser des liens vers les outlets et les actions définis dans TempView.h et .m. Renouvelez l'opération pour les trois autres vues de l'application. L'onglet Infos correspond à l'écran d'accueil, chargé par défaut. Il affiche des informations sur l'application et sur son développeur. Vous pouvez également lui greffer des boutons renvoyant vers les différentes fonctions de conversion, qui agissent alors comme autant de raccourcis pratiques.

Passons sans plus tarder au développement ! Saisissez les codes des listings ci-dessous dans les classes correspondantes.

Listing 5.7 : MultiConvertisseurAppDelegate.h

```
#import <UIKit/UIKit.h>

@interface MultiConvertisseurAppDelegate : NSObject
    <UIApplicationDelegate, UITabBarControllerDelegate> {
    UIWindow *window;
    UITabBarController *tabBarController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UITabBarController
    *tabBarController;

@end
```

Listing 5.8 : *MultiConvertisseurAppDelegate.m*

```
#import "MultiConvertisseurAppDelegate.h"

@implementation MultiConvertisseurAppDelegate

@synthesize window;
@synthesize tabBarController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // Add the tab bar controller's current view as a subview of the window
    [window addSubview:tabBarController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [tabBarController release];
    [window release];

    [super dealloc];
}

@end
```

Listing 5.9 : *InfosViewController.h*

```
#import <UIKit/UIKit.h>

#import "TempView.h";
#import "LongView.h";
#import "PoidsView.h";

@interface InfosViewController : UIViewController {
    TempView *tempView;
    LongView *longView;
    PoidsView *poidsView;
    InfosViewController *infosViewController;
}

@property (nonatomic, retain) TempView *tempView;
@property (nonatomic, retain) LongView *longView;
@property (nonatomic, retain) PoidsView *poidsView;
@property (nonatomic, retain) InfosViewController *infosViewController;

- (IBAction)changerOnglet:(id)sender;

@end
```

Listing 5.10 : *InfosViewController.m*

```
#import "InfosViewController.h"
#import "TempView.h";
#import "LongView.h";
#import "PoidsView.h";

@implementation InfosViewController

@synthesize infosViewController, tempView, longView, poidsView;

- (IBAction)changerOnglet:(id)sender {
    NSString *reponse = [sender titleForState:UIControlStateNormal];
    if([reponse isEqualToString:@"C'est parti !"]) {
        [UIView beginAnimations:@"ChangementVue" context:nil];
        [UIView setAnimationDuration:2.0];
        TempView *tempViewa = [[TempView alloc]
                               initWithNibName:@"TempView" bundle:[NSBundle mainBundle]];
        self.tempView = tempViewa;
        [tempViewa release];

        [UIView setAnimationTransition:UIViewAnimationTransitionFlip
         FromRight forView:self.view cache:YES];
        [tempView viewWillAppear:YES];
        [self.view addSubview:self.tempView.view];
        [tempView viewDidAppear:YES];

        [UIView commitAnimations];
    }
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [infosViewController release];
    [tempView release];
    [longView release];
    [poidsView release];
    [super dealloc];
}

@end
```

Nous avons déposé un bouton intitulé "C'est parti !" sur la vue *InfosView.xib* et nous lui avons associé l'action *changerOnglet*. À travers un petit effet d'animation, l'utilisateur per-

mute ainsi automatiquement vers la première fonction du convertisseur (voir Figure 5.28). Vous pouvez parfaitement ajouter d'autres boutons et exploiter la valeur de sender pour repérer le bouton pressé : vous renvoyez ainsi l'utilisateur vers les différentes fonctions de l'application. À vous de développer ces boucles conditionnelles !

Poursuivons à présent le développement de notre convertisseur en ajoutant la logique des autres contrôleurs. Il s'agit à chaque fois de créer des outlets correspondants aux champs de saisie de l'utilisateur et des actions associées aux boutons de conversion.

Figure 5.28

On clique sur le bouton affiché sur la première vue : on bascule automatiquement vers une seconde vue, dans un effet d'animation.



Listing 5.11 : TempView.h

```
#import <UIKit/UIKit.h>

@interface TempView : UIViewController {
    IBOutlet UITextField *saisie;
    IBOutlet UITextField *resultat;
    IBOutlet UITextField *saisie2;
    IBOutlet UITextField *resultat2;
}

@property (retain, nonatomic) UITextField *saisie;
@property (retain, nonatomic) UITextField *resultat;
@property (retain, nonatomic) UITextField *saisie2;
@property (retain, nonatomic) UITextField *resultat2;

- (IBAction)convertTempFC:(id)sender;
- (IBAction)convertTempCF:(id)sender;

@end
```

Listing 5.12 : TempView.m

```
#import "TempView.h"

@implementation TempView
@synthesize saisie, resultat, saisie2, resultat2;

- (IBAction)convertTempFC: (id) sender {
    float saisieValeur = [[saisie text] floatValue];
    float resultatValeur = (saisieValeur - 32.0f) * 5.0f / 9.0f;
    [resultat setText:[NSString stringWithFormat:@"%3.2f",
        resultatValeur]];
}

- (IBAction) convertTempCF: (id) sender {
    float saisieValeur2 = [[saisie2 text] floatValue];
    float resultatValeur2 = (saisieValeur2 * 1.8f) + 32.0f;
    [resultat2 setText:[NSString stringWithFormat:@"%3.2f",
        resultatValeur2]];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [saisie release];
    [resultat release];
    [saisie2 release];
    [resultat2 release];

    [super dealloc];
}

@end
```

Listing 5.13 : LongView.h

```
#import <UIKit/UIKit.h>

@interface LongView : UIViewController {
    IBOutlet UITextField *saisie;
    IBOutlet UITextField *resultat;
}
@property (retain, nonatomic) UITextField *saisie;
@property (retain, nonatomic) UITextField *resultat;
```

```
- (IBAction)convertLong:(id)sender;

@end
```

Listing 5.14 : LongView.m

```
#import "LongView.h"

@implementation LongView

@synthesize saisie, resultat;

- (IBAction)convertLong: (id) sender {
    float saisieValeur = [[saisie text] floatValue];
    float resultatValeur = (saisieValeur / 2.4f);
    [resultat setText:[NSString stringWithFormat:@"%3.2f",
        resultatValeur]];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [saisie release];
    [resultat release];
    [super dealloc];
}

@end
```

Listing 5.15 : PoidsView.h

```
#import <UIKit/UIKit.h>

@interface PoidsView : UIViewController {
    IBOutlet UITextField *saisie;
    IBOutlet UITextField *resultat;
}
@property (retain, nonatomic) UITextField *saisie;
@property (retain, nonatomic) UITextField *resultat;
```



```
- (IBAction)convertPoids:(id)sender;

@end
```

Listing 5.16 : PoidsView.m

```
#import "PoidsView.h"

@implementation PoidsView

@synthesize saisie, resultat;

- (IBAction)convertPoids: (id) sender {
    float saisieValeur = [[saisie text] floatValue];
    float resultatValeur = (((saisieValeur * 100000.0f) / 0.45359237f)
        /100000.0f);
    [resultat setText:[NSString stringWithFormat:@"%3.2f",
        resultatValeur]];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [saisie release];
    [resultat release];
    [super dealloc];
}

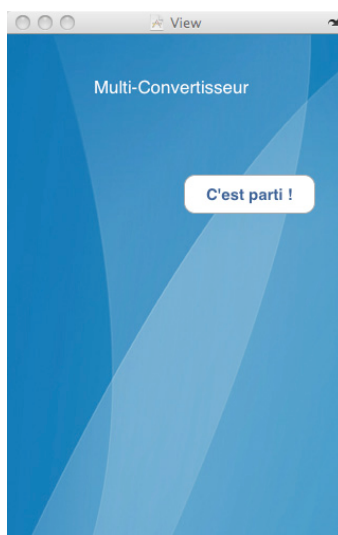
@end
```

Comme vous le constatez, les fonctions demeurent simples : elles récupèrent la valeur saisie par l'utilisateur, procèdent à un calcul de conversion et affichent le résultat dans le second champ. Vous pouvez ainsi librement étoffer le programme de conversion en ajoutant d'autres champs de saisie, comme nous avons commencé à le faire avec la classe TempView.m.

Passons à présent au design des vues. Chargez successivement les fichiers InfosView.xib, TempView.xib, LongView.xib et PoidsView.xib et agencez-les de manière à ce qu'ils ressemblent aux Figures 5.29 à 5.32.

Figure 5.29

La vue `InfosView.xib`, avec son bouton associé à l'action `changerOnglet`.

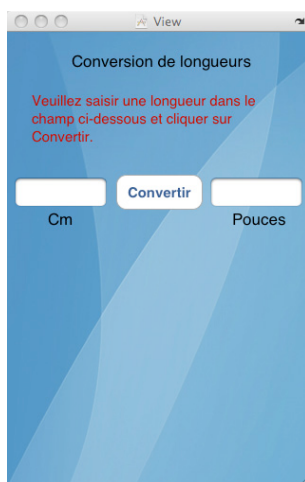
**Figure 5.30**

La vue `TempView.xib` pour la conversion de températures.



Figure 5.31

La vue LongView.xib pour la conversion de longueurs.

**Figure 5.32**

La vue PoidsView.xib pour la conversion de poids.



Nous vous laissons le soin de lier les actions et les outlets de chaque vue, comme au chapitre précédent. Vous devez ainsi associer File's Owner à View, les champs de texte saisie et resultat aux deux objets UITextField et le bouton de conversion au délégué de l'application et à l'action de la classe contrôleur. Une petite particularité que nous n'avons pas encore étudiée : vous devez associer un événement Did End On Exit au champ de saisie afin de faire disparaître le clavier virtuel. Cliquez sur l'un des champs saisie de vos vues puis basculez vers l'inspecteur de connexions (Cmd+2). Cliquez ensuite sur le cercle en face de Did End On Exit et maintenez la souris jusqu'à First Responder. Sélectionnez alors l'action convertPoids par exemple. Ainsi, dès que l'utilisateur ferme le clavier virtuel, la conversion se déclenche : il n'a même plus besoin de presser le bouton Convertir !

Enregistrez toutes vos modifications puis compilez le projet. Le Simulateur d'iPhone affiche vos quatre onglets et vous permet de procéder à toutes les conversions de votre choix (voir Figure 5.33).

Figure 5.33

Notre application MultiConvertisseur au grand complet, en plein calcul de conversions.



Le design des vues de votre application

Alors que l'App Store compte plus de 100 000 applications à l'heure actuelle (<http://148apps.biz/app-store-metrics>), il devient de plus en plus difficile de se démarquer de ses "concurrents" directs et d'offrir une meilleure visibilité à son projet sur la boutique en ligne d'Apple. Bien entendu, la logique de vos contrôleurs et la qualité de votre code jouent un rôle prépondérant dans l'intérêt de votre application : si vous proposez un traitement intelligent et innovant, à travers un outil qui rend de réels services au quotidien, la carrière de votre projet a toutes les chances d'être auréolée de succès. Toutefois, ne négligez pas la phase de design initiale. Vous ne disposez que de quelques secondes pour convaincre vos futurs utilisateurs, à travers la page consacrée à votre projet sur l'App Store. Au-delà de la pure description de votre application, la capture d'écran qui lui est associée doit rapidement convaincre les visiteurs de procéder au téléchargement.

Il faut donc se rendre à l'évidence : notre application MultiConvertisseur n'a rien de "sexy", même si elle remplit le rôle qu'on lui a fixé. Sa série d'écrans blancs tranche trop radicalement avec le design des applications classiques de l'iPhone et des utilitaires intégrés par défaut. En la présentant ainsi sur l'App Store, nous avons toutes les chances d'écoper de commentaires négatifs. Même si vous ne disposez pas de talents graphiques particuliers, il est inutile d'aller ainsi à Canossa : le travail préliminaire est purement gâché par l'aspect

peu flatteur de l'interface. Mais méfiez-vous également de la tendance inverse : de nombreuses applications ne présentent aucun intérêt, si ce n'est leur design tape-à-l'œil qui méprend les utilisateurs. Combien de fois avez-vous téléchargé une application avant de vous rendre compte de l'esbroufe ?

Les applications pour iPhone correspondent à un juste équilibre entre les logiciels bureautiques classiques, dont elles empruntent l'intérêt et la complexité des traitements, et les logiciels des autres plates-formes mobiles, qui partagent la même accessibilité et simplicité d'emploi. Cet équilibre s'exprime également au niveau du design : l'écran de l'iPhone autorise tout type de fioritures (couleurs vives, boutons personnalisés, logos hauts en couleurs, etc.) tout en favorisant les vues sobres et soignées. Après plus de deux ans d'existence de la plateforme, nous disposons d'un certain recul et nous pouvons identifier les bonnes pratiques, les règles d'or et les écueils à éviter en matière de design.

Les tendances du design d'applications

Commençons d'emblée par mettre fin à une idée reçue : non, les applications les plus intéressantes ne remplacent pas nécessairement tous les éléments graphiques officiels du kit de développement de l'iPhone ! C'est devenu une trop grande habitude : de nombreux développeurs troquent les éléments traditionnels (interrupteur, boutons, listes, contrôles de navigation, etc.) contre des images personnalisées, réalisées sous Photoshop. Si l'interface conserve toute sa cohérence et son aspect intuitif, le pari est réussi : vous innovez et vous proposez une nouvelle manière d'interagir avec le terminal d'Apple. Si ces remplacements ne présentent pas d'intérêt, vous alourdissez au contraire l'ergonomie de votre application et vous imposez à vos utilisateurs l'apprentissage d'une série de manipulations (voir Figure 5.34).

Figure 5.34

L'interface de cette application est tellement surchargée d'éléments spécifiques qu'elle en devient difficile à manipuler.



En reprenant tels quels les éléments de base du kit de développement, vous êtes assuré de ne pas dérouter vos utilisateurs. Dans la mesure du possible, essayez donc de conserver les systèmes de navigation standard, utilisez les polices par défaut et ne surchargez pas l'interface de votre application (voir Figure 5.35).

Figure 5.35

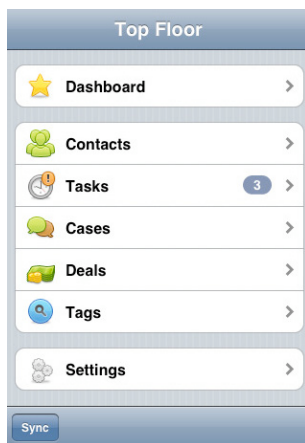
L'interface de l'application Facebook s'inscrit au contraire dans le pur canevas des utilitaires de l'iPhone et présente une manipulation très intuitive.



Les utilitaires ont tout intérêt à arborer un look très classique, avec un arrière-plan gris-bleu et des boutons blancs, clairement identifiés par une série d'icônes. N'y voyez pas nécessairement un nivellement par le bas de la qualité de votre design ; les utilisateurs apprécieront au contraire la sobriété de votre application et la rangeront dès le premier écran de leur iPhone, parmi les utilitaires standard (voir Figure 5.36).

Figure 5.36

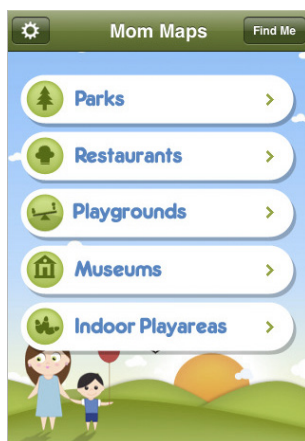
L'interface de Top Floor reprend parfaitement les éléments traditionnels de l'iPhone et offre une vision claire de vos possibilités.



D'autres applications qui se fixent un rôle unique jouent sur le symbolisme de leur sujet. Ainsi, une boîte à outils réunissant des règles, altimètres et niveaux à bulles peut mettre en scène une métaphore visuelle et s'inscrire sur un fond qui évoque le bois ou le métal. Cet écart vis-à-vis du design traditionnel de l'iPhone est parfaitement toléré par les utilisateurs, qui y voient au contraire la marque d'une véritable immersion de l'application dans son sujet. N'y cédez pas coûte que coûte : toutes les applications ne résistent pas nécessairement à l'épreuve d'arrière-plans clinquants et surchargés. Réfléchissez sur papier à l'intérêt d'un tel design et identifiez d'emblée les éléments visuels associés traditionnellement au rôle de votre application (voir Figure 5.37).

Figure 5.37

Avec son interface colorée et dessinée, Mom Maps s'écarte du champ traditionnel des applications tout en imposant subtilement des métaphores visuelles correspondant à son sujet : la petite enfance et les loisirs de nos chères têtes blondes.

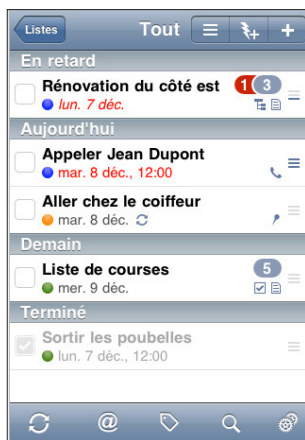


Si votre application repose sur une série de listes réunies autour d'un système de navigation, ne faites jamais preuve d'une originalité démesurée. Le contenu en lui-même prévaut sur tous les autres aspects de votre application. Vos sections doivent être clairement identifiées et la navigation doit demeurer intuitive et accessible en une pression du doigt. Sélectionnez une série de couleurs que vous déployez à l'ensemble de votre application, afin de repérer immédiatement les informations essentielles. Des icônes bien choisies et de petits pictogrammes facilitent la lecture (voir Figure 5.38).

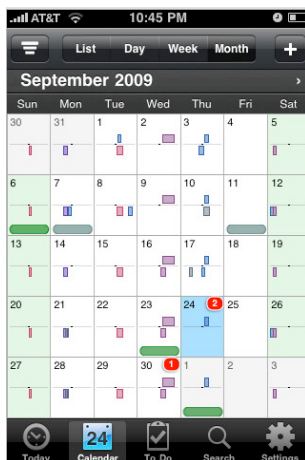
Retenez donc ce principe : il est inutile de remplacer les éléments traditionnels du kit de développement sauf si votre sujet s'y prête parfaitement. Il suffit parfois d'aligner les éléments de manière harmonieuse et de changer une série de couleurs pour conférer un look suffisamment original et moderne à votre application.

Figure 5.38

La liste des tâches à accomplir dans l'application ToDo : la sobriété est de mise et vous visualisez immédiatement le contenu.

**Figure 5.39**

En modifiant les couleurs par défaut des listes et en essayant d'afficher trop d'éléments à l'écran, cette application est confuse et n'est pas facile à piloter au doigt.



Du papier à Photoshop : la refonte de MultiConvertisseur

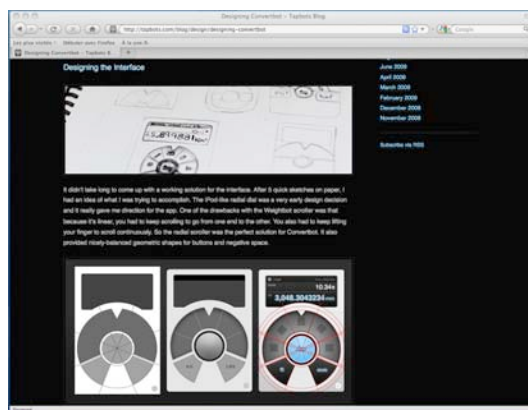
Comme nous l'avons vu, MultiConvertisseur s'inscrit dans l'immense catégorie des utilitaires pour iPhone et essuie une concurrence sévère. Figurant parmi ses plus homologues rivaux, Convertbot offre des fonctions équivalentes tout en arborant un look soigné et très original. Réalisée par Mark Jardine et Paul Haddad, cette application mime l'allure d'un pèse-personne en réunissant toutes ses fonctions de conversion autour d'une "roue", qui déclenche autant d'actions préparées dans le code source. Du propre aveu de ses développeurs, "il aura fallu cinq esquisses sur papier pour aboutir à un tel résultat : la forme ronde au centre de l'écran évoque subtilement le design de l'iPod et nous a offert un canevas de base pour développer l'application et identifier ses rôles principaux". Mark Jardine poursuit son

explication : "notre objectif principal n'était pas d'offrir la meilleure application de conversion d'unités – il existe des dizaines d'utilitaires plus précis, qui vous permettent d'aboutir à un résultat en un seul clic. Notre objectif était de rendre l'opération amusante et ludique. En distinguant plusieurs étapes dans la conversion, les utilisateurs ressentent une plus grande satisfaction lorsqu'ils aboutissent au résultat. Le design est ainsi porteur d'émotion".

Suivez les recommandations de Mark et commencez par travailler sur papier. Une série d'esquisses vous aide à y voir plus clair et à déterminer le rôle essentiel de votre application. Reportez-vous éventuellement à son blog, à l'adresse <http://tapbots.com/blog/design/designing-convertbot>, pour découvrir le fruit de ce premier travail (voir Figure 5.40).

Figure 5.40

Les premières esquisses de Convertbot, jusqu'au résultat final.



Sous Photoshop ou votre logiciel de création graphique préféré, créez un nouveau canevas de 320 × 480 pixels et déposez-y les éléments de votre choix. Vous pouvez télécharger un fichier PSD de 9 Mo contenant la plupart des éléments d'interface à l'adresse <http://www.teehanlax.com/blog/?p=1628>. En alignant tous ces objets et en les personnalisant, vous aboutissez à une première idée du résultat escompté (voir Figure 5.41).

Figure 5.41

La création d'une ébauche de l'interface sous Photoshop CS4.



Vous avez tout intérêt à prendre une capture du Simulateur d'iPhone et à ajouter cette image à un calque, en jouant sur son opacité. Vous pouvez ainsi mieux vous rendre compte de l'emplacement de tous vos contrôles et vous créez vos arrière-plans en conséquence. Photoshop vous permet également de découper votre interface en "tranches" : vous créez ainsi des zones épousant les dimensions de vos boutons et contrôles et vous enregistrez des images destinées à les remplacer intégralement. Lorsque tout est prêt, déroulez le menu Fichier de Photoshop CS4 et choisissez «Enregistrer pour le Web et les périphériques». Sélectionnez le format PNG-24, activez éventuellement la transparence et cliquez sur le bouton Enregistrer.

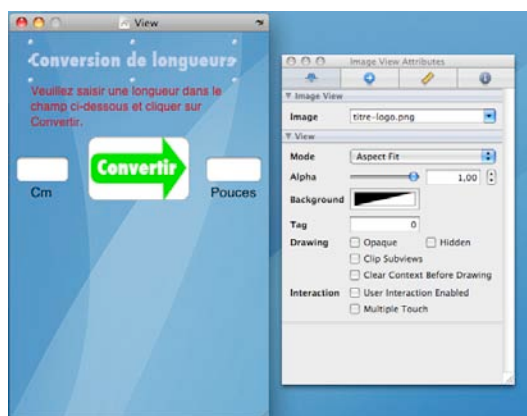
Ouvrez à nouveau votre projet d'application sous Xcode, puis cliquez sur le dossier Resources en maintenant la touche Ctrl du clavier enfoncée. Sélectionnez Add puis Existing Files, pointez vers l'image fraîchement créée et cliquez sur Add. Renouvelez l'opération pour toutes les images que vous souhaitez ajouter à votre projet, notamment les icônes figurant dans la barre d'onglets. Chargez ensuite le fichier nib correspondant à la vue que vous souhaitez éditer. Disposez un objet de type UIImageView sur votre interface et saisissez les poignées qui l'encadrent pour l'adapter aux dimensions de l'image que vous comptez greffer à la vue. Ouvrez ensuite l'inspecteur d'attributs et déroulez le menu en face du champ Image ; le nom de votre fichier importé doit figurer dans la liste. Sélectionnez-le : il apparaît dans le conteneur UIImageView.

Pour disposer une image en arrière-plan, la solution la plus simple consiste à sélectionner l'objet puis à ouvrir l'inspecteur de position (Cmd+3). Saisissez 0 dans les champs X et Y, puis 320 et 480 dans les champs W et H. Même si Interface Builder ne dispose pas d'un système de calques pour maîtriser la superposition de vos éléments, vous avez la possibilité de jouer sur leur empilement. Sélectionnez tous vos autres objets (libellés, boutons, etc.), déroulez le menu Layout et choisissez Send To Front. Si vous avez créé de petites images à l'aide de l'outil Tranche de Photoshop CS4, sélectionnez les boutons qui doivent les arborer, ouvrez l'inspecteur d'attributs et choisissez le fichier correspondant au champ Image. Là encore, vous pouvez jouer sur leur position pour les faire apparaître au-dessus de l'arrière-plan (voir Figure 5.42).

Multipliez les essais jusqu'à aboutir au résultat escompté. N'oubliez pas : vous ne devez pas nécessairement surcharger l'interface. Évitez ainsi de disposer de véritables photographies en guise d'arrière-plan.

Figure 5.42

Nous avons modernisé l'interface de notre application, pour la rendre plus agréable à utiliser.



Précharger la vue de votre application

Lorsque vos utilisateurs cliquent sur l'icône associée à votre application, l'iPhone charge tous les composants nécessaires en mémoire, accède à ses ressources puis ouvre la première vue. Pour les applications les moins complexes, l'opération n'excède pas une seconde et vous avez l'impression que l'utilitaire se charge instantanément à travers une petite animation. Mais pour les applications plus ambitieuses, le délai peut sembler s'éterniser. Une petite astuce consiste à greffer un fichier `Default.png` à la racine des ressources de l'application ; l'iPhone le charge par défaut lors du lancement. En créant une telle image qui reprend l'allure de la première vue de votre application, vous donnez l'impression que votre utilitaire se lance immédiatement. Vous pouvez également utiliser cette image pour créer un splash screen, qui agit comme un écran d'introduction et immerge l'utilisateur dans le cadre de votre application. Les studios professionnels de développement s'appuient généralement sur cet écran pour diffuser leur logo.

Pour aller plus loin

Au cours de ce chapitre, vous avez appris à développer des applications multivues en isolant chaque interface et chaque contrôleur. Vous aboutissez ainsi à des projets plus ambitieux, qui combinent une grande série de fonctionnalités. Au-delà de la pertinence de la logique interne de vos contrôleurs, vous avez également compris à quel point l'ergonomie joue un rôle majeur dans le développement d'applications pour iPhone et iPod Touch. Un système de navigation ne se choisit pas au hasard et doit servir l'intérêt de votre application. N'oubliez jamais que vous vous adressez à des utilisateurs de téléphone mobile, probablement au cours de leurs déplacements ; dans pareille situation, leur concentration n'excède pas quelques secondes et vous ne devez pas leur imposer un système trop complexe à appréhender.

N'hésitez pas à modifier encore les vues de l'application MultiConvertisseur en guise d'exercice et choisissez un autre agencement. Étoffe également le design de chaque vue, en modifiant l'arrière-plan et en chargeant éventuellement des images en guise de titres ou de boutons. Prêtez-vous à un tel jeu avant d'entamer la lecture du chapitre suivant : nous allons désormais nous intéresser à tous les contrôles spécifiques à l'iPhone, qui lui confèrent son ergonomie si particulière. La conception d'applications autour de telles fonctionnalités exige une plus grande rigueur sur le plan du design. Non seulement vos utilisateurs vont presser des boutons et interagir avec votre application à travers toutes sortes de contrôleurs de navigation, mais ils vont surtout la vivre de manière "organique", en fonction de leur prise en main... au sens littéral ! Lorsqu'elle entretient de tels rapports avec ses utilisateurs, votre application ne doit pas souffrir du moindre défaut ergonomique. La clarté de l'interface joue un rôle aussi important que la logique interne de vos classes dans le succès d'une application.



6

Les contrôles spécifiques

Au sommaire de ce chapitre

- Interagir avec des contrôles spécifiques : la création d'un utilitaire
- Interagir avec des réglettes, des alertes, des contrôles segmentés et des sous-vues
- Intégrer une roulette à l'application
- Pour aller plus loin

Si certaines applications très populaires tiennent en une poignée d'éléments intelligemment agencés dans une fenêtre unique, la plupart des projets les plus ambitieux reposent sur une multitude de vues qui s'adaptent aux attentes des utilisateurs. Il n'est pas rare d'imbriquer quatre ou cinq fenêtres différentes, elles-mêmes constellées de boutons, libellés ou champs de texte. Pour préserver un contrôle intuitif malgré la multiplication des objets d'interface, le SDK de l'iPhone dispose d'une bibliothèque très riche : réglettes, interrupteurs, roulettes,

etc. Tous ces éléments autorisent un contrôle minutieux et permettent aux utilisateurs d'interagir avec votre application de manière ergonomique. À l'image du motif de conception MVC, votre tâche ne consiste pas seulement à vous focaliser sur la logique interne de vos programmes ; vous devez également agencer au mieux tous ces éléments, sans pour autant céder à la facilité de déposer pêle-mêle tout type d'objets. Votre application doit toujours pleinement justifier l'utilisation de tous ces contrôles avancés.

Nous l'avons vu au chapitre précédent : si l'écran de l'iPhone présente nécessairement un affichage limité, les systèmes de navigation (onglets, boutons, barres d'outils, etc.) vous aident à regrouper correctement vos éléments et à ne pas alourdir l'interface. L'immense famille des utilitaires gagne à s'articuler autour de tels systèmes, afin de préserver la lisibilité de l'application. Au cours de ce chapitre, nous verrons comment aboutir à de tels projets autour de ces contrôles avancés ; ces derniers vous font gagner un temps précieux et élargissent considérablement le champ de vos possibilités tout en présentant une interface conviviale aux utilisateurs.

Face à de tels objectifs, les roulettes jouent un rôle prépondérant. Elles constituent l'une des interfaces-clés de l'iPhone : on navigue verticalement parmi leurs éléments, en les faisant glisser au doigt. Numéros de téléphone, paramètres d'une application ou contenu extrait d'un service web : il existe de nombreuses manières d'agencer les éléments qu'elles contiennent et de les rendre interactives. Elles deviennent alors la colonne vertébrale de toute votre application et donnent accès à une immense variété de services et d'informations. Au cours de ce chapitre, nous étudierons leur intérêt et nous apprendrons à les disposer sur l'écran de l'iPhone à travers une série d'exemples concrets.

Interagir avec des contrôles spécifiques : la création d'un utilitaire

Le modèle Utility Application

Tel que le conçoit Xcode, les utilitaires sont des applications mono-tâches qui servent un but précis : ce sont des outils simples que les utilisateurs déclenchent pour répondre à un besoin spécifique (voir Figure 6.1). Bulletin météo, calcul d'itinéraires, outil de bricolage, etc., tous ces projets s'articulent autour d'une vue principale, qui offre un accès direct à l'information, et d'une vue secondaire accessible à l'aide d'un petit bouton "i". Vous l'avez probablement déjà repéré sur vos applications préférées, à commencer par le programme Bourse qui est préinstallé sur l'iPhone : ce bouton déclenche une animation de transition qui fait pivoter la vue principale vers cette vue secondaire, à travers laquelle on configure sommairement

l'utilitaire ou l'on accède à des informations complémentaires. Vous pouvez rapidement développer une application de ce type grâce à l'un des modèles intégrés à Xcode.

Figure 6.1

L'application GuitarToolkit vous permet d'accorder à une guitare et d'accéder à une page d'information consacrée au développeur.



Nous allons créer une application de ce type en ajoutant une série de contrôles avancés à la vue principale (interrupteurs, boutons, champs de texte, etc.) et en décrivant le projet dans la vue secondaire. Sous Xcode, déroulez le menu File > New Project, puis choisissez le modèle "Utility Application". Cliquez ensuite sur le bouton Choose et saisissez "Motdepasse" en guise de nom de projet. Notre utilitaire est un générateur de mot de passe : en fonction de critères sélectionnés par l'utilisateur (emploi de majuscules, minuscules, chiffres et/ou signes de ponctuation, longueur du mot de passe, etc.), l'application propose aléatoirement une chaîne de caractères qu'on pourra employer lors de la création d'un compte auprès d'un service web, par exemple. Contrairement à notre exemple de quiz, où l'on devait presser directement le seul bouton correspondant à notre réponse, il s'agit ici de réunir tous les critères définis par l'utilisateur et de les traiter d'un seul tenant à l'aide d'une action.

Après avoir créé votre nouveau projet, essayez de le compiler en pressant simultanément les touches Cmd+R. Le Simulateur d'iPhone présente cette première ébauche : votre utilitaire s'articule autour de deux vues vides que l'on permute en pressant un petit bouton "i" (voir Figure 6.2). Depuis la seconde vue, on revient vers la vue principale en cliquant sur un bouton figurant dans la barre de titre – Xcode a donc mis en place une application autour d'un système de navigation sommaire, qui est une instance de la classe UINavigationController.

Figure 6.2

Le comportement initial d'un utilitaire créé sous Xcode.



Auscultez de plus près les fichiers créés automatiquement. Leur agencement obéit au principe suivant :

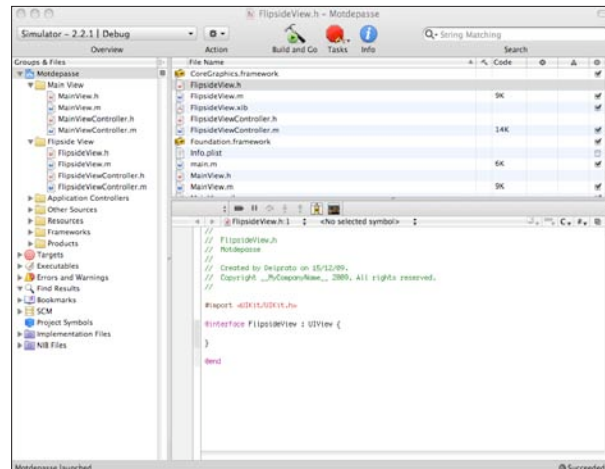
- Comme l'indique la liste des propriétés `Info.plist`, le fichier nib par défaut est `Main-Window`. Il est ainsi chargé en premier.
- En observant ce fichier, on constate qu'il intègre une instance de la classe `RootViewController`. Il s'agit d'une sous-classe d'`UIView` qui ne dispose que d'un outlet, `info-Button`, et d'une action, `toggleView`. Le premier est un bouton affiché par défaut sur l'interface qui déclenche l'action : la vue principale permute vers la vue secondaire, selon le principe général que nous avons exposé au chapitre précédent.
- La classe `RootViewController` gère donc le changement de vue. On lui associe par défaut deux contrôleurs supplémentaires : `MainViewController` et `FlipSideViewController`, qui correspondent respectivement à la vue principale et secondaire. En parallèle, cette classe met en place une instance de la classe `UINavigationController` que l'on utilise à travers l'action `toggleView` pour gérer le changement de vue. Comme vous le constatez en ouvrant le fichier `RootViewController.m`, c'est la vue principale qui est chargée par défaut : `initWithNibName:@"MainView"` indique que la vue `MainView.xib` s'affiche dès l'ouverture de l'application.
- Cette vue est pour l'instant vide. Elle ne comprend qu'une instance de la classe `MainView`, elle-même sous-classe d'`UIView`. Son contrôleur `MainViewController` est lui aussi vide et s'apprête à recueillir tous les traitements que vous souhaitez faire figurer sur la vue principale de votre utilitaire.

- L'action `toggleView` de la classe `RootViewController` gère le chargement de la vue secondaire. Elle commence par vérifier la vue qui figure actuellement sur l'écran de l'iPhone. S'il ne s'agit pas de la vue secondaire, on appelle la méthode `loadFlipsideViewController` et on charge le second fichier nib, `FlipSideView.xib`. Cette méthode exploite ensuite la barre de navigation instanciée dans la fenêtre principale de l'application, `MainWindow.xib`, pour ajouter un titre à la seconde vue et permettre le retour à la vue principale.
- L'interface de cette seconde vue est donc définie par le fichier `FlipsideView.xib`. Là encore, la fenêtre est vide et ne comprend qu'une instance de la classe `FlipSideView`. Son contrôleur `FlipsideViewController` est également vide et s'apprête à héberger toutes les actions déclenchées sur la seconde vue de votre utilitaire.

Si tout ceci vous paraît encore abstrait, parcourez en détail le code source et retenez ce concept essentiel : pour personnaliser rapidement votre utilitaire, vous devez retoucher les deux vues successives créées par Xcode (`MainView` et `FlipsideView`), adapter leur classe contrôleur et éventuellement personnaliser la barre de titre qui soutient leur basculement. Vous avez également la possibilité d'ajouter des traitements personnalisés au lancement de votre utilitaire grâce au délégué de l'application, `MotdepasseAppDelegate`. Même s'il apparaît en anglais, le nom des classes est suffisamment explicite pour mettre rapidement le pied à l'étrier : vous profitez bel et bien du canevas d'une application multivues en quelques clics de souris (voir Figure 6.3).

Figure 6.3

L'agencement initial des classes créées par Xcode dans notre projet d'utilitaire.



Le cahier des charges et l'ébauche préliminaire

Plaçons-nous dans le contexte d'un projet réel. Avant de mettre les mains dans le cambouis et d'ajouter nos éléments d'interface et nos actions, nous devons rédiger un bref cahier des charges et définir les interactions sur papier. Les mots de passe trop évidents sont aujourd'hui devenus la cible préférée des pirates, qui en profitent pour accéder à la quasi-totalité de nos activités. Il n'est en effet pas rare de décliner le sempiternel même sésame à l'ensemble de nos services web (courrier électronique, réseaux sociaux, services en ligne, etc.), ce qui les expose dangereusement. Notre utilitaire vise à générer aléatoirement des mots de passe beaucoup plus complexes, qui mêlent éventuellement des majuscules et des minuscules mais aussi des chiffres et des signes de ponctuation. Lorsqu'un utilisateur s'apprête à créer un compte sur un forum ou un service en ligne, il lui suffira de dégainer son iPhone et d'utiliser notre application pour s'assurer de choisir un mot de passe inviolable.

En dégageant ainsi le rôle essentiel de notre application, nous avons identifié les principaux contrôles que nous devons disposer sur l'interface :

- un champ de saisie pour indiquer le nombre de caractères composant le mot de passe ;
- des contrôles permettant de complexifier le mot de passe et d'y faire figurer des caractères majuscules et minuscules, des chiffres et des signes de ponctuation ;
- un bouton pour déclencher la création du mot de passe ;
- un champ affichant le résultat ;
- un bouton "i" pour basculer vers la vue secondaire ;
- une seconde vue où l'on met en garde l'utilisateur contre le risque d'employer des mots de passe trop évidents et à travers laquelle on réalise la "publicité" de notre gamme de produits.

Dans ce type de projet, le dialogue homme-machine est réduit à sa plus simple expression : l'utilisateur personnalise une série de valeurs et clique sur un bouton unique afin de lancer le traitement. En retour, notre application affiche le résultat escompté. Des pressions supplémentaires sur le bouton génèrent de nouveaux mots de passe : l'utilisateur peut ainsi rapidement parcourir une série de propositions et choisir celle qui lui paraît la plus efficace.

Parmi les éléments à notre disposition, il nous apparaît clairement que les interrupteurs (classe `UISwitch`) constituent la solution idéale pour choisir les éléments devant figurer dans les mots de passe. On active ou non l'ajout de signes de ponctuation, par exemple. Il s'agit d'un contrôle emblématique de l'iPhone. Les utilisateurs s'y sont habitués dès l'écran des réglages principaux et ils comprendront d'emblée son rôle et son fonctionnement. Sur papier, nous essayons d'agencer au mieux tous ces éléments qui figureront dans la vue principale de l'application. Le premier résultat apparaît à la Figure 6.4.

Figure 6.4

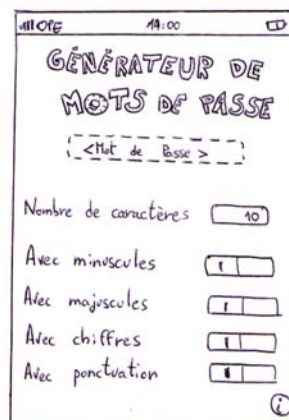
Une première ébauche de l'interface.



Cette première esquisse révèle quelques défauts. En figurant par défaut sous les contrôles, le champ contenant le mot de passe généré risque d'alourdir l'affichage. Il s'agit de l'information principale renvoyée par notre application : elle doit figurer au premier plan. Afin d'éviter que l'utilisateur tente d'y saisir quoi que ce soit, nous réglerons sa couche alpha à 0 (le champ de texte deviendra ainsi totalement transparent) et nous le ferons apparaître progressivement au moment de générer un mot de passe. Les différents interrupteurs doivent être remisés au second plan ; si les utilisateurs vont essayer de combiner des critères distincts lors de leur première découverte de l'application, il y a fort à parier qu'ils vont maintenir la même nomenclature par la suite. Nous aboutissons ainsi à une seconde esquisse sur papier, plus proche du résultat escompté (voir Figure 6.5).

Figure 6.5

On modifie l'agencement des contrôles, pour gagner en clarté et en lisibilité.



Dans le cadre d'applications complexes, il n'est pas futile de réfléchir sur papier aux algorithmes à mettre en place. Notre générateur de mots de passe ne compte qu'une action :

celle-ci récupère la valeur des interrupteurs, crée une longue chaîne de caractères épousant les souhaits de l'utilisateur, puis sélectionne aléatoirement une série de caractères issus de cette chaîne. Le résultat final est ensuite présenté à l'utilisateur, qui le découvre à travers un champ de texte apparaissant progressivement. Nous avons ainsi besoin de quatre interrupteurs, correspondant aux différents critères, de deux champs de texte pour saisir la longueur du mot de passe et afficher le résultat, et d'un bouton pour lancer l'opération.

La logique interne des contrôleurs

Comme nous l'avons vu, il n'est pas nécessaire de retoucher la classe `RootViewController` pour aboutir à un utilitaire fonctionnel : Xcode a préparé tous les fichiers nécessaires et a tissé les liaisons entre les différents éléments de base. Pour personnaliser la barre de titre figurant au-dessus de la vue secondaire, nous devons toutefois retoucher un aspect de l'implémentation de cette classe. Modifiez le fichier `RootViewController.m` en ajoutant les deux lignes apparaissant en gras au Listing 6.1.

Listing 6.1 : *RootViewController.m*

```
#import "RootViewController.h"
#import "MainViewController.h"
#import "FlipsideViewController.h"

@implementation RootViewController

@synthesize infoButton;
@synthesize flipsideNavigationBar;
@synthesize mainViewController;
@synthesize flipsideViewController;

- (void)viewDidLoad {

    [super viewDidLoad];
    MainViewController *viewController = [[MainViewController alloc]
                                           initWithNibName:@"MainView" bundle:nil];
    self.mainViewController = viewController;
    [viewController release];

    [self.view addSubview:mainViewController.view
     belowSubview:infoButton];
}

- (void)loadFlipsideViewController {

    FlipsideViewController *viewController = [[FlipsideViewController
                                                alloc] initWithNibName:@"FlipsideView" bundle:nil];
    self.flipsideViewController = viewController;
    [viewController release];
}
```

```

// Set up the navigation bar
UINavigationController *aNavigationBar = [[UINavigationController alloc]
initWithFrame:CGRectMake(0.0, 0.0, 320.0, 44.0)];
aNavigationBar.barStyle = UIBarStyleBlackOpaque;
self.flipsideNavigationBar = aNavigationBar;
[aNavigationBar release];

UIBarButtonItem *buttonItem = [[UIBarButtonItem alloc]
initWithTitle:@"retour" style:UIBarButtonItemStylePlain
target:self action:@selector(toggleView)];
UINavigationController *navigationItem = [[UINavigationController alloc]
initWithTitle:@"À propos"];
navigationItem.rightBarButtonItem = buttonItem;
[flipsideNavigationBar pushViewController:navigationItem
animated:NO];
[navigationItem release];
[buttonItem release];
}

- (IBAction)toggleView {
/*
This method is called when the info or Done button is pressed.
It flips the displayed view from the main view to the flipside view
and vice-versa.
*/
if (flipsideViewController == nil) {
[self loadFlipsideViewController];
}

UIView *mainView = mainViewController.view;
UIView *flipsideView = flipsideViewController.view;

[UIView beginAnimations:nil context:NULL];
[UIView setAnimationDuration:1];
[UIView setAnimationTransition:([mainView superview] ?
UIViewAnimationTransitionFlipFromRight :
UIViewAnimationTransitionFlipFromLeft) forView:self.
view cache:YES];

if ([mainView superview] != nil) {
[flipsideViewController viewWillAppear:YES];
[mainViewController viewWillDisappear:YES];
[mainView removeFromSuperview];
[infoButton removeFromSuperview];
[self.view addSubview:flipsideView];
[self.view insertSubview:flipsideNavigationBar
aboveSubview:flipsideView];
[mainViewController viewDidDisappear:YES];
[flipsideViewController viewDidAppear:YES];
} else {

```

```
[mainViewController viewWillAppear:YES];
[flipsideViewController viewWillDisappear:YES];
[flipsideView removeFromSuperview];
[flipsideNavigationBar removeFromSuperview];
[self.view addSubview:mainView];
[self.view insertSubview:infoButton
    aboveSubview:mainViewController.view];
[flipsideViewController viewDidDisappear:YES];
[mainViewController viewDidAppear:YES];
}
[UIView commitAnimations];
}

/*
// Override to allow orientations other than the default portrait
// orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
*/

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [infoButton release];
    [flipsideNavigationBar release];
    [mainViewController release];
    [flipsideViewController release];
    [super dealloc];
}

@end
```

Comme vous pouvez le constater, le contrôleur racine commence par charger la vue "Main-View" (soit MainView.xib) dans l'action viewDidLoad qui s'exécute automatiquement au lancement de l'application. L'action loadFlipsideViewController charge la seconde vue lorsque l'on presse le bouton "i" figurant en bas à droite de l'écran. Nous sommes essentiellement intervenus lors de l'initialisation de la barre de navigation, en modifiant le titre de la seconde vue et le libellé figurant sur le bouton de retour :

```
UIBarButtonItem *buttonItem = [[UIBarButtonItem alloc]
    initWithTitle:@"retour" style:UIBarButtonItemStylePlain target:self
    action:@selector(toggleView)];
```

```

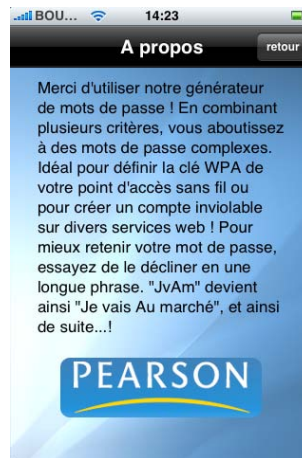
UINavigationController *navigationItem = [[UINavigationController alloc]
initWithTitle:@"À propos"];
navigationItem.rightBarButtonItem = buttonItem;

```

La première ligne crée un bouton de type `UIBarButtonItem` que l'on positionne en haut à droite. Nous changeons son titre : il contient désormais le mot "retour" (et non "Done" et ses différentes versions localisées). La ligne suivante définit le titre de la seconde vue. Par défaut, Xcode lui donne le nom du fichier nib qui définit son interface. Nous remplaçons cette occurrence par le titre "À propos" – notre seconde vue contient des éléments de présentation générale de notre gamme de produits. À ce titre, nous ne modifierons pas précisément cette seconde vue ; nous nous contentons de charger une image en arrière-plan à l'aide de classe `UIImageView` et d'ajouter un libellé de type `UILabel`. Pour le fond d'écran, n'oubliez pas de dessiner une image au format PNG épousant les dimensions de 320 × 460 pixels. La barre de titre occupe en effet 20 pixels de hauteur (voir Figure 6.6).

Figure 6.6

Notre seconde vue présente des informations d'ordre général.



Intéressons-nous à présent à la vue principale. Par défaut, sa classe contrôleur est `MainViewController` et son interface se base sur une instance de la classe `MainView`. Commencez par retoucher le contrôleur principal en y définissant l'outlet essentiel de l'application : le bouton déclenchant le calcul d'un nouveau mot de passe. Modifiez le fichier `MainViewController.h` en saisissant le Listing 6.2.

Listing 6.2 : *MainViewController.h*

```

#import <UIKit/UIKit.h>

@interface MainViewController : UIViewController {
    IBOutlet UIButton *genererMdp;
}

```



```
}  
@property (nonatomic, retain) UIButton *genererMdp;  
@end
```

Vous en avez désormais l'habitude : modifiez également l'implémentation de cette classe, comme au Listing 6.3.

Listing 6.3 : *MainViewController.m*

```
#import "MainViewController.h"  
#import "MainView.h"  
  
@implementation MainViewController  
  
@synthesize genererMdp;  
  
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)  
    nibBundleOrNil {  
    if (self = [super initWithNibName:nibNameOrNil  
        bundle:nibBundleOrNil]) {  
        // Custom initialization  
    }  
    return self;  
}  
  
/*  
// Implement viewDidLoad to do additional setup after loading the view,  
// typically from a nib.  
- (void)viewDidLoad {  
    [super viewDidLoad];  
}  
*/  
  
/*  
// Override to allow orientations other than the default portrait  
// orientation.  
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)  
    interfaceOrientation {  
    // Return YES for supported orientations  
    return (interfaceOrientation == UIInterfaceOrientationPortrait);  
}  
*/  
  
- (void)didReceiveMemoryWarning {  
    [super didReceiveMemoryWarning];  
    // Releases the view if it doesn't have a superview  
    // Release anything that's not essential, such as cached data  
}  
  
- (void)dealloc {
```

```

        [genererMdp release];
        [super dealloc];
    }

@end

```

Nous devons à présent modifier la classe `MainView`, qui correspond aux différents traitements s'opérant sur la vue principale. Saisissez successivement les Listings 6.4 et 6.5 pour définir l'interface et l'implémentation de cette classe.

Listing 6.4 : *MainView.h*

```

#import <UIKit/UIKit.h>

@interface MainView : UIView {
    // On définit les outlets correspondant aux contrôles graphiques
    IBOutlet UITextField *mdpLongueur;
    IBOutlet UISwitch *ajoutMin;
    IBOutlet UISwitch *ajoutMaj;
    IBOutlet UISwitch *ajoutChiffres;
    IBOutlet UISwitch *ajoutPonctuation;
    IBOutlet UITextField *mdp;
}

@property (nonatomic, retain) UITextField *mdpLongueur;

// On définit l'action responsable de la génération du mot de passe
- (IBAction)genererMdp;

@end

```

Dans ce code, nous avons essentiellement défini les différents outlets correspondant aux interrupteurs (les quatre critères évoqués précédemment) et aux champs de texte (la longueur du mot de passe et l'affichage du résultat). On définit également l'action responsable de la création du mot de passe.

Listing 6.5 : *MainView.m*

```

#import "MainView.h"

#define RANDOM_SEED() srandom(time(NULL))
#define RANDOM_INT(__MIN__, __MAX__) ((__MIN__) + random() %
    ((__MAX__+1) - (__MIN__)))

@implementation MainView

@synthesize mdpLongueur;

```

```
- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame]) {
        // Initialization code
    }
    return self;
}

- (IBAction)genererMdp {

    // Effet d'animation du champ contenant le mot de passe
    // On réinitialise son opacité
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:1];
    [mdp setAlpha:0.0];
    [UIView commitAnimations];

    // On le fait apparaître progressivement
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:1];
    [mdp setAlpha:1.00];
    [UIView commitAnimations];

    NSInteger valMdpLongueur = [mdpLongueur.text intValue];
    BOOL boutonMin = ajoutMin.on;
    BOOL boutonMaj = ajoutMaj.on;
    BOOL boutonChiffres = ajoutChiffres.on;
    BOOL boutonPonctuation = ajoutPonctuation.on;

    NSString* contenuMdp = @"";

    // Définition des chaînes de caractères
    // correspondant aux différents critères
    NSString* dicoMin = @"abcdefghijklmnopqrstuvwxyz";
    NSString* dicoMaj = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    NSString* dicoChiffres = @"0123456789";
    NSString* dicoPonctuation = @"~!/?#%^&*()";

    RANDOM_SEED();

    // Cas où aucun critère n'est sélectionné
    if(!boutonMin && !boutonMaj && !boutonChiffres && !boutonPonctuation)
    {
        UIAlertView *alerte = [[UIAlertView alloc] initWithTitle:@"Aucune
            règle" message:@"Sélectionnez au moins l'un des critères à
            l'aide des interrupteurs"
            delegate:self cancelButtonTitle:@"OK" otherButtonTitles: nil];
        [alerte show];
        [alerte release];
    }

    return;
}
```

```

NSString* dicoMdp = @"";

// On fusionne les différentes chaînes de caractères
// afin d'en créer une, dicoMDP, qui correspond
// exactement aux critères de l'utilisateur
if(boutonMin)
    dicoMdp = [NSString stringWithFormat:@"%%%%", dicoMdp, dicoMin];

if(boutonMaj)
    dicoMdp = [NSString stringWithFormat:@"%%%%", dicoMdp, dicoMaj];

if(boutonChiffres)
    dicoMdp = [NSString stringWithFormat:@"%%%%", dicoMdp,
        dicoChiffres];

if(boutonPonctuation)
    dicoMdp = [NSString stringWithFormat:@"%%%%", dicoMdp,
        dicoPonctuation];

// On sélectionne aléatoirement l'un des caractères
// de cette longue chaîne, jusqu'à créer une
// nouvelle chaîne de caractères épousant la longueur désirée.
for(NSInteger i=0; i<valMdpLongueur; i++) {

    int index = RANDOM_INT(0, [dicoMdp length]-1);

    NSRange range = NSMakeRange(index, 1);
    NSString *caractereMdp = [dicoMdp substringWithRange:range];

    contenuMdp = [NSString stringWithFormat:@"%%%%", contenuMdp,
        caractereMdp];

    NSLog(caractereMdp);

}

// contenuMdp contient cette chaîne de caractères
// On met à jour le champ de texte mdp.
mdp.text = @"";
mdp.text = contenuMdp;
}

- (void)drawRect:(CGRect)rect {
    // Code du dessin
}

- (void)dealloc {
    [super dealloc];
}

@end

```

Le principe de l'action `genererMdp` est plutôt simple. Nous commençons par relever la valeur du champ de texte `mdpLongueur`, contenant la longueur désirée du mot de passe. Nous créons ensuite quatre booléens correspondant aux quatre interrupteurs : on relève ainsi les critères choisis par l'utilisateur. En parallèle, nous définissons quatre chaînes de caractères représentant les différentes possibilités (alphabet en minuscules et majuscules, chiffres de 0 à 9, signes de ponctuation, etc.). Nous devons également prendre en compte le cas de figure où l'utilisateur n'a pas choisi le moindre critère. La création du mot de passe est alors impossible et nous l'avertissons à travers une boîte d'alerte :

```
UIAlertView *alerte = [[UIAlertView alloc] initWithTitle:@"Aucune
règle" message:@"Sélectionnez au moins l'un des critères à l'aide des
interrupteurs" delegate:self cancelButtonTitle:@"OK" otherButtonTitles:
nil];
[alerte show];
[alerte release];
```

Ces trois lignes créent une instance `alerte` de la classe `UIAlertView`. On l'initialise avec un titre ("Aucune règle"), un message d'erreur et un bouton de validation permettant de revenir à la vue principale (voir Figure 6.7). Nous envoyons ensuite un message pour faire apparaître cette boîte de dialogue à l'écran.

Figure 6.7

La boîte d'alerte apparaît lorsque l'utilisateur n'a sélectionné aucun critère.



Notre action `genererMdp` crée ensuite une longue chaîne de caractères, en cumulant les "dictionnaires" correspondant aux différents critères sélectionnés par l'utilisateur. Ainsi, s'il n'a choisi que l'emploi des majuscules et minuscules, la chaîne `dicoMdp` comprendra 52 caractères (a-zA-Z). Une boucle `for` extrait alors aléatoirement l'un de ces caractères de la chaîne pour créer une nouvelle chaîne `contenuMdp`. La boucle se poursuit tant que cette

dernière variable ne comprend pas le nombre suffisant de caractères, tel que l'utilisateur l'a décidé en premier lieu à l'aide du champ de texte `mdpLongueur`. On affiche en dernier lieu le résultat dans le champ de texte `mdp` à travers un léger effet d'animation. La couche alpha du champ de texte est tout d'abord initialisée à 0 avant d'atteindre sa valeur maximale au bout d'une seconde. Le champ apparaît ainsi progressivement à l'affichage.

L'habillage de l'interface et la liaison des outlets et actions

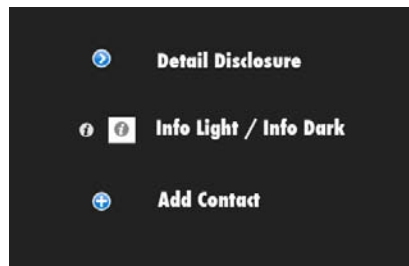
Il ne nous reste plus qu'à définir l'allure de notre interface principale pour achever notre application. Sous Xcode, sélectionnez tout d'abord le fichier `MainWindow.xib`. Vous basculez automatiquement vers Interface Builder. Déroulez l'arborescence face à l'instance de la classe `RootViewController` jusqu'à découvrir l'objet `UIButton`. Il s'agit de l'un des quatre boutons prédéfinis par le SDK de l'iPhone, `Info Light`. C'est cet élément qui déclenche l'action `toggleView` responsable du basculement entre nos deux vues. Dans l'inspecteur d'attributs (`Cmd+1`), vous pouvez modifier son type et choisir une autre apparence, comme `Info Dark` par exemple. Le bouton apparaît alors sur un fond sombre et non bleu clair.

Les boutons prédéfinis du SDK de l'iPhone

Afin d'homogénéiser l'interface de vos applications et de reprendre les éléments typiques de l'iPhone, n'hésitez pas à utiliser les quatre boutons prédéfinis par le SDK. Lorsque vous ajoutez un objet de type `UIButton`, ouvrez l'inspecteur d'attributs et déroulez le menu `Type`. La première option, `Custom`, vous laisse le champ libre pour définir l'allure de votre bouton. L'option `Rounded Rect` correspond à un type de boutons que nous avons déjà largement utilisé et qui figure de manière individuelle dans la bibliothèque d'objets. Le type `Detail Disclosure` représente une flèche pointant vers la droite, enchâssée dans un bouton circulaire. C'est le modèle utilisé dans la liste des contacts de l'iPhone afin d'accéder aux fiches détaillées de tous vos interlocuteurs. Les déclinaisons `Info Light` et `Info Dark` correspondent aux boutons "i" que nous utilisons dans le cadre de notre application. Enfin, le type `Add Contact` est un simple signe +, comme l'élément figurant en haut de la liste des contacts. Tous ces objets prédéfinis vous font gagner un temps précieux et vous permettent de développer une application plus intuitive, qui reprend des contrôles auxquels les utilisateurs sont largement habitués (voir Figure 6.8).

Figure 6.8

*Les différents boutons
prédéfinis du SDK
de l'iPhone.*



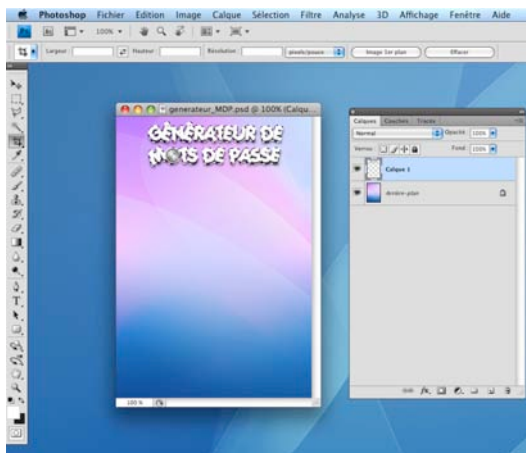
Sur cette vue, déplacez éventuellement le bouton d'informations. Retenez ce concept essentiel : il est intégré au contrôleur racine de votre utilitaire et figure ainsi en surimpression sur la vue principale. Vous devez veiller à ce que l'interface de cette dernière ne vienne pas empiéter sur son emplacement.

Ouvrez à présent l'interface `MainView.xib` correspondant à la vue principale. Déposez tous les interrupteurs, libellés, champs de texte et boutons dont vous avez besoin. Dans notre exemple, nous essayons de reproduire au mieux notre première esquisse sur papier. Nous avons ainsi tout intérêt à ajouter des objets `UIImageView` pour soutenir un arrière-plan et un logo. Ouvrez votre éditeur graphique préféré pour préparer un canevas global, dont vous extrairez chacun des éléments.

Sous Photoshop CS4, la manipulation n'est pas fondamentalement éloignée de la création d'une interface web. Commencez par créer un nouveau document de 320 × 460 pixels (la barre de titres occupant 20 pixels de hauteur sur l'interface), puis déposez votre arrière-plan sur un nouveau calque. Notre logo correspond à un petit montage intégrant le symbole d'un verrou à la place de la lettre "O" dans l'expression "mots de passe". Là encore, nous créons un nouveau calque pour accueillir ce logo et vérifier son intégration sur l'arrière-plan. Dans la palette des calques de Photoshop, cliquez sur cet élément en pressant la touche `Cmd` : le logo est automatiquement détourné (voir Figure 6.9). Effectuez ensuite un copier-coller pour le déposer sur un nouveau document. Déroulez enfin le menu `Fichier` et cliquez sur "Enregistrer pour le Web et les périphériques". Sélectionnez le format PNG-24 et activez la transparence en cochant la case associée. Cliquez sur le bouton `Enregistrer` : vous disposez ainsi d'un logo correctement découpé, sur fond transparent.

Figure 6.9

Le design de notre vue principale sous Photoshop. On isole ici le logo de l'application.



Sous Xcode, déroulez le menu `Project` et cliquez sur `Add to Project`. Sélectionnez vos images et validez, en cochant éventuellement la case "Copy items into destination group's folder".

Sous Interface Builder, ajoutez un nouvel objet UIImageView épousant les dimensions de votre logo. Ouvrez l'inspecteur d'attributs, puis sélectionnez votre image dans le champ Image. Procédez de la même manière pour l'arrière-plan, sans oublier de faire figurer à l'avant-plan tous vos éléments d'interface (voir Figure 6.10). Pour cela, cliquez sur leur intitulé dans la fenêtre MainView.xib, déroulez le menu Layout et sélectionnez Send To Front (voir Figure 6.11).

Figure 6.10

On dispose tous nos contrôles graphiques sur la vue principale.

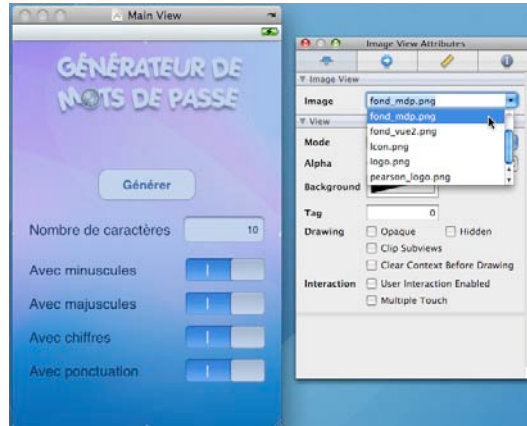
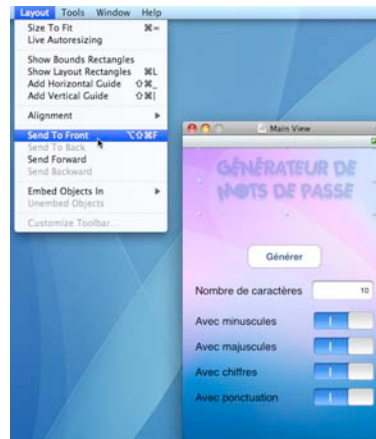


Figure 6.11

Avec Interface Builder, on place correctement le logo défini précédemment.

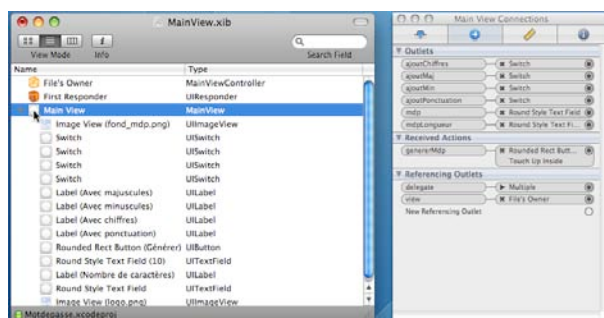


Vous êtes satisfait du résultat ? Il ne vous reste plus qu'à lier tous les outlets et actions aux différents contrôles graphiques. Cliquez tout d'abord sur File's Owner, maintenez la touche Ctrl du clavier enfoncée et déplacez le curseur jusqu'au bouton qui déclenche l'action. Sélectionnez l'élément `genererMdp` du menu contextuel qui apparaît à l'écran. Cliquez ensuite sur Main View, puis effectuez la même opération pour chaque interrupteur et les deux champs de texte (voir Figure 6.12). Pour ces derniers, vérifiez que vous avez désactivé la

saisie de l'utilisateur dans le champ affichant le mot de passe (en décochant la case Enabled à la section Control de l'inspecteur d'attributs) et que vous avez permis la disparition du clavier virtuel dans le champ correspondant à la longueur désirée du mot de passe. Rappelez-vous : vous devez sélectionner ce champ, ouvrir l'inspecteur de connexions (Cmd+2) et relier l'événement Did End On Exit au First Responder. En dernier lieu, sélectionnez le bouton et associez l'événement Touch Up Inside à l'action genererMdp en cliquant sur le petit cercle figurant à droite de l'inspecteur de connexions et en déplaçant votre curseur jusqu'à Main View. L'inspecteur de connexions doit ainsi correspondre à la Figure 6.12.

Figure 6.12

Les connexions de notre vue principale.



Enregistrez tous vos fichiers et compilez une première fois votre projet. Le résultat correspond à nos attentes : l'utilisateur manipule une série d'interrupteurs afin de définir ses critères, puis lance la création d'un mot de passe en cliquant sur un bouton au centre de l'écran (voir Figure 6.13). Le champ contenant le résultat apparaît progressivement et le bouton "i" permet de basculer rapidement vers la seconde vue.

Figure 6.13

Le résultat final dans le Simulateur d'iPhone.



L'intérêt de la vue secondaire

Dans notre exemple, nous nous sommes essentiellement concentrés sur la vue principale. Vous avez toutefois constaté que Xcode prépare un projet complet et gère automatiquement le basculement entre les deux vues à l'aide du bouton d'informations. En guise d'exercice, vous avez ainsi tout intérêt à disposer les interrupteurs sur cette seconde vue. Elle agit alors comme une page de configuration, à travers laquelle l'utilisateur sélectionne l'ensemble de ces critères. Comme nous l'avons vu précédemment, il est peu probable que ces critères évoluent au cours du temps : l'utilisateur sélectionne une série de règles une fois pour toutes et l'ensemble des mots de passe générés ultérieurement obéiront à la même nomenclature. Vous dégagez ainsi un espace significatif sur la vue principale et vous conférez un tout autre intérêt à la vue secondaire.

Interagir avec des réglettes, des alertes, des contrôles segmentés et des sous-vues

La bibliothèque des éléments du SDK est particulièrement riche et il existe de nombreux objets dérivés de la classe `UIKit` qui autorisent un contrôle plus minutieux de l'information. Parmi ceux-ci, les réglettes jouent un rôle important : elles permettent aux utilisateurs de manipuler minutieusement un élément d'interface jusqu'à choisir une valeur extrêmement précise. Leur implémentation nécessite une grande rigueur ; dans la mesure où elles se contrôlent au doigt, les réglettes ne doivent souffrir d'aucun défaut d'ergonomie, sous peine de multiplier les essais et de rendre l'interface laborieuse.

Les contrôles segmentés sont des instances de la classe `UISegmentedControl`. Ils contiennent par défaut deux états qui agissent comme autant de boutons. Ils illustrent parfaitement le principe des contrôles actifs et passifs : une valeur est actuellement sélectionnée et seule l'intervention de l'utilisateur modifie leur état.

Les sous-vues correspondent à des objets de type `UIView` imbriqués dans une vue plus grande. À la manière des objets `UIImageView`, vous pouvez ainsi modifier leur état, leur visibilité et leurs propriétés sans pour autant redessiner la vue principale dans son ensemble.

Nous allons passer en revue l'ensemble de ces concepts autour d'un exemple concret.

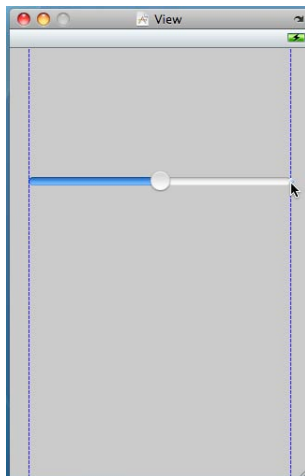
Ajouter une réglette

Créez un nouveau projet sous Xcode de type "View-Based Application". Saisissez "Slider" en tant que nom de projet, puis validez en cliquant sur le bouton Save. Dans cet exemple, nous allons essentiellement nous concentrer sur l'intégration d'une réglette et sur les possibilités d'interaction qu'elle nous offre. Effectuez un double-clic sur le fichier `SliderView-`

Controlller.xib et ajoutez un élément de type `Slider` sur l'interface principale de votre application. N'hésitez pas à saisir les poignées encadrant ce nouvel objet afin de l'étendre sur toute la largeur de l'écran de l'iPhone : il est très important que la manipulation soit intuitive et vous devez éviter de disposer un élément trop étriqué (voir Figure 6.14).

Figure 6.14

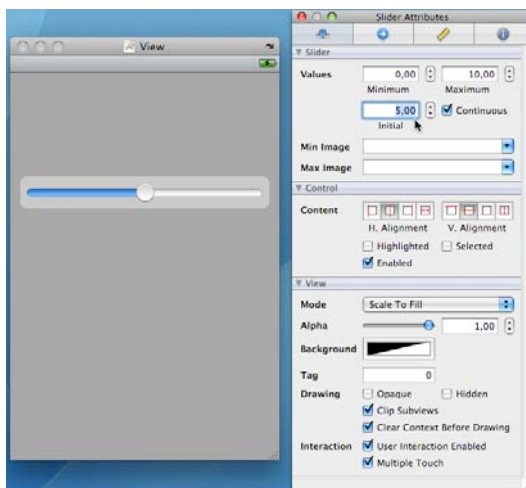
L'ajout d'une réglette au centre de l'application.



Sélectionnez votre réglette et ouvrez l'inspecteur d'attributs. Comme vous le constatez au champ `Slider`, vous devez choisir les valeurs bornant la réglette ainsi que la valeur affichée par défaut. Dans ce premier exemple, saisissez `0,00` dans le champ `Minimum`, `10,00` dans le champ `Maximum` et `5,00` dans le champ `Initial` (voir Figure 6.15).

Figure 6.15

L'inspecteur d'attributs permet de définir les valeurs symbolisées par la réglette.



Dans un second temps, ajoutez un simple champ de texte en bas de l'écran. Nous l'utiliserons pour afficher la valeur sélectionnée par l'utilisateur à l'aide de la réglette. À ce stade, nous avons besoin de définir deux outlets (la réglette et le champ de texte) et une action qui s'exécute lorsque l'utilisateur manipule la réglette. Ouvrez le fichier `SliderViewController.h` sous Xcode et saisissez le code figurant au Listing 6.6.

Listing 6.6 : *SliderViewController.h*

```
#import <UIKit/UIKit.h>

@interface SliderViewController : UIViewController {
    IBOutlet UITextField *champReponse;
    IBOutlet UISlider *reglette;
}
@property (nonatomic, retain) UITextField *champReponse;
@property (nonatomic, retain) UISlider *reglette;
- (IBAction)sliderClic:(id)sender;
@end
```

Modifiez à présent l'implémentation de cette classe en saisissant le code suivant dans le fichier `SliderViewController.m`.

Listing 6.7 : *SliderViewController.m*

```
#import "SliderViewController.h"

@implementation SliderViewController

@synthesize champReponse;
@synthesize reglette;

- (IBAction)sliderClic:(id)sender {
    NSString *nouveauTexte = [[NSString alloc] initWithFormat:@"%f",
        reglette.value];

    champReponse.text = nouveauTexte;
    [nouveauTexte release];
}

/*
// The designated initializer. Override to perform setup that is required
// before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
    nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
        bundle:nibBundleOrNil]) {
        // Custom initialization
    }
}
```

```
        return self;
    }
    */

    /*
    // Implement loadView to create a view hierarchy programmatically,
    // without using a nib.
    - (void)loadView {
    }
    */

    /*
    // Implement viewDidLoad to do additional setup after loading the view,
    // typically from a nib.
    - (void)viewDidLoad {
        [super viewDidLoad];
    }
    */

    /*
    // Override to allow orientations other than the default portrait
    // orientation.
    - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
        interfaceOrientation {
        // Return YES for supported orientations
        return (interfaceOrientation == UIInterfaceOrientationPortrait);
    }
    */

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
        // Releases the view if it doesn't have a superview
        // Release anything that's not essential, such as cached data
    }

    - (void)dealloc {
        [champReponse dealloc];
        [reglette dealloc];
        [super dealloc];
    }

@end
```

Le principe de notre action est extrêmement simple : lorsque l'utilisateur manipule la réglette, on relève sa valeur à l'aide de la notation point (`reglette.value`) et on la convertit en une chaîne de caractères. On modifie alors le champ de texte en conséquence, en affichant ainsi la valeur sélectionnée par l'utilisateur. Nous avons précédemment fixé les bornes de la réglette sous Interface Builder : le champ de texte renvoie donc une valeur comprise entre 0 et 10 en fonction de la position du curseur.

Il ne nous reste plus qu'à lier les outlets et l'action sous Interface Builder. Ouvrez le fichier `SliderViewController.xib`, sélectionnez `File's Owner` en pressant la touche `Ctrl` du clavier, puis relâchez la souris au-dessus du champ de texte : sélectionnez l'élément `champ-Reponse`.

Renouvelez l'opération jusqu'à la réglette, en sélectionnant cette fois l'outlet `reglette`. Cliquez enfin sur la réglette, ouvrez l'inspecteur de connexions et cliquez sur le petit cercle à droite de l'événement `Value Changed`. Glissez la souris jusqu'à l'icône `First Responder`, puis sélectionnez l'action `sliderClic`. Enregistrez votre interface et compilez le projet : le champ de texte reflète la valeur de la réglette (voir Figure 6.16).

Figure 6.16

La réglette renvoie ici une valeur comprise entre 0 et 10, qui apparaît dans le champ de texte.



Afficher une alerte à l'utilisateur

Nous avons entrevu cette technique au cours de notre premier exemple, lorsque l'utilisateur n'avait pas sélectionné le moindre critère pour générer un mot de passe : il est possible de lui présenter toutes sortes de boîtes de dialogue à l'écran, afin d'afficher une valeur relevée ou un message d'erreur. La solution la plus simple consiste à créer une instance de la classe `UIAlertView`. Parmi ses attributs, vous avez la possibilité d'indiquer un titre à la boîte d'alerte, d'afficher un message personnalisé et d'ajouter un bouton permettant de la faire disparaître.

Modifions notre exemple afin d'afficher la valeur de la réglette dans une telle boîte d'alerte. Pour cela, nous devons commencer par ajouter une action à l'interface de notre classe prin-

ci-pale. Mettez à jour le fichier `SliderViewController.h` à l'aide du code figurant au Listing 6.8.

Listing 6.8 : *SliderViewController.h*

```
#import <UIKit/UIKit.h>

@interface SliderViewController : UIViewController {
    IBOutlet UITextField *champReponse;
    IBOutlet UISlider *reglette;
}
@property (nonatomic, retain) UITextField *champReponse;
@property (nonatomic, retain) UISlider *reglette;
- (IBAction)sliderClic:(id)sender;
- (IBAction)alerteClic:(id)sender;
@end
```

Modifiez à présent l'implémentation de cette classe à travers le fichier `SliderViewController.m`.

Listing 6.9 : *SliderViewController.m*

```
#import "SliderViewController.h"

@implementation SliderViewController

@synthesize champReponse;
@synthesize reglette;

- (IBAction)sliderClic:(id)sender {
    NSString *nouveauTexte = [[NSString alloc] initWithFormat:@"%f",
        reglette.value];

    champReponse.text = nouveauTexte;
    [nouveauTexte release];
}

- (IBAction)alerteClic:(id)sender {
    NSString *msg = nil;
    msg = [[NSString alloc] initWithFormat:@"Vous avez sélectionné la
        valeur %f", slider.value];
    UIAlertView *alerte = [[UIAlertView alloc] initWithTitle:@"Valeur
        sélectionnée" message:msg delegate:self cancelButtonTitle:@"Je
        recommence !" otherButtonTitles:nil];
    [alerte show];
    [alerte release];
}
```

```
[msg release];  
}  
  
[...]  
  
@end
```

Sous Interface Builder, déposez un bouton sur la vue principale. Sélectionnez-le puis ouvrez l'inspecteur de connexions. Cliquez sur le cercle à droite de l'événement Touch Up Inside et reliez-le à l'élément First Responder. Choisissez alors l'action alerteClic. Enregistrez votre travail et compilez à nouveau le projet. La valeur de la réglette apparaît toujours dans le champ de texte, mais lorsque vous cliquez sur le bouton au centre de l'écran vous affichez désormais une alerte contenant cette même valeur. Il vous suffit de cliquer sur le bouton "Je recommence !" pour faire disparaître l'alerte et revenir à la vue principale (voir Figure 6.17).

Figure 6.17

L'alerte de base récupère la valeur de la réglette et l'affiche dans une boîte de dialogue.



Ajouter un contrôle segmenté et une sous-vue

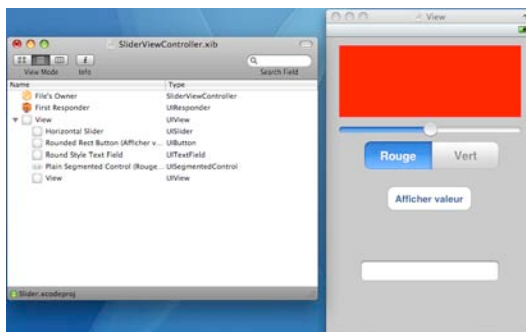
Mais en l'absence du moindre élément à contrôler, la réglette ne présente qu'un intérêt très limité. Certes, l'utilisateur choisit une valeur que notre application relève mais celle-ci n'est pour l'instant d'aucune utilité. Afin d'illustrer le comportement d'une réglette, ajoutons une sous-vue à notre interface : la réglette jouera sur l'opacité de ce nouvel objet.

En parallèle, nous allons disposer un contrôle segmenté sur notre vue principale. Celui-ci permet de basculer entre deux positions : dans notre exemple, l'utilisateur va ainsi choisir la couleur d'arrière-plan de la sous-vue avant de jouer sur sa transparence.

Sous Interface Builder, déposez un objet de type View sur votre vue principale. Saisissez les poignées qui l'encadrent afin de le redimensionner sur l'écran. Dans notre exemple, nous souhaitons ajouter un simple rectangle dont la couleur et l'opacité se modifient en fonction des choix de l'utilisateur. En auscultant la fenêtre `SliderViewController.xib`, vous constatez que ce nouvel objet `UIView` est bien imbriqué dans l'élément View parent, à l'instar du champ de texte, du bouton ou de la réglette (voir Figure 6.18). Déposez ensuite un objet de type `Segmented Control` sur l'interface. Effectuez un double-clic sur chaque "versant" de ce contrôle afin d'en changer les libellés : la zone de gauche, active par défaut, permettra d'afficher un rectangle rouge et la zone de droite un rectangle vert. Inscrivez clairement ces deux états sur le contrôle afin que l'utilisateur comprenne au premier coup d'œil son rôle et son intérêt.

Figure 6.18

L'interface de notre application comprend désormais un contrôle segmenté et une sous-vue imbriquée dans la vue principale.



Nous devons mettre à jour notre classe principale en ajoutant un outlet, correspondant à la sous-vue que l'on va manipuler, et une action permettant de modifier son opacité. Mettez à jour le fichier `SliderViewController.h` avec le code ci-dessous.

Listing 6.10 : *SliderViewController.h*

```
#import <UIKit/UIKit.h>

@interface SliderViewController : UIViewController {
    IBOutlet UITextField *champReponse;
    IBOutlet UISlider *reglette;
    IBOutlet UIView *sousVue;
}
@property (nonatomic, retain) UITextField *champReponse;
@property (nonatomic, retain) UISlider *reglette;
@property (nonatomic, retain) UIView *sousVue;
- (IBAction)sliderClic:(id)sender;
- (IBAction)alerteClic:(id)sender;
- (IBAction)changeCouleur:(id)sender;
@end
```

Définissons à présent l'action `changeCouleur`. Nous devons également modifier l'action `sliderClic` afin de retoucher dynamiquement l'opacité de la sous-vue dès que l'utilisateur manipule la réglette. Modifiez le fichier `SliderViewController.m` avec le code figurant au Listing 6.11.

Listing 6.11 : *SliderViewController.m*

```
#import "SliderViewController.h"

@implementation SliderViewController

@synthesize champReponse;
@synthesize reglette;
@synthesize sousVue;

-(IBAction)sliderClic:(id)sender {
    NSString *nouveauTexte = [[NSString alloc] initWithFormat:@"%f",
                                reglette.value];
    champReponse.text = nouveauTexte;
    [sousVue setAlpha:reglette.value];
    [nouveauTexte release];
}

-(IBAction)changeCouleur:(id)sender {
    UISegmentedControl *segmentedControl = (UISegmentedControl *)sender;
    NSInteger segment = segmentedControl.selectedSegmentIndex;

    if (segment == 0) [sousVue setBackgroundColor:[UIColor redColor]];
    else [sousVue setBackgroundColor:[UIColor greenColor]];
}

-(IBAction)alerteClic:(id)sender {
    NSString *msg = nil;
    msg = [[NSString alloc] initWithFormat:@"Vous avez sélectionné la
    valeur %f", reglette.value];
    UIAlertView *alerte = [[UIAlertView alloc] initWithTitle:@"Valeur
    sélectionnée" message:msg delegate:self cancelButtonTitle:@"Je
    recommence !" otherButtonTitles:nil];
    [alerte show];
    [alerte release];
    [msg release];
}

[...]
```

```
- (void)dealloc {
    [champReponse dealloc];
    [reglette dealloc];
    [sousVue dealloc];
}
```

```

        [super dealloc];
    }

@end

```

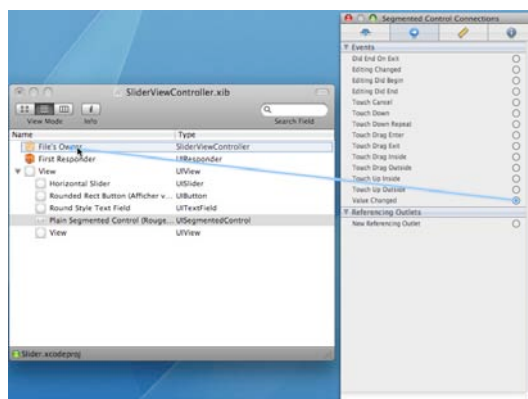
Là encore, la notation d'Objective-C nous permet d'aboutir rapidement au résultat. Le contrôle segmenté déclenche l'action `changeCouleur`. On crée alors un objet de type `UISegmentedControl` et on relève l'état actuellement sélectionné. Les deux versants de ce contrôle sont identifiés par des index : 0 pour la partie de gauche et 1 pour la partie de droite. À l'aide d'une simple boucle conditionnelle, on modifie la couleur d'arrière-plan de l'objet sous-vue en fonction de l'élément sélectionné. Ainsi, lorsque l'utilisateur presse la partie gauche du contrôle, l'objet sous-vue revêt la couleur rouge. Il devient vert lorsque l'on clique sur la partie droite du contrôle. Là encore, nous utilisons les constantes `redColor` et `greenColor` avec la méthode `setBackgroundColor` afin de faciliter le changement de couleur.

En parallèle, l'action `sliderClic` compte une opération supplémentaire : en fonction de la valeur de la réglette, on modifie l'opacité de l'objet sous-vue grâce à la méthode `setAlpha`.

Vous devez toutefois modifier l'interface en associant les outlets et actions. Sélectionnez `File's Owner`, pressez la touche `Ctrl` du clavier et glissez la souris jusqu'à la sous-vue. Choisissez alors l'option sous-vue du menu contextuel qui apparaît à l'écran. Sélectionnez enfin le contrôle segmenté, ouvrez l'inspecteur de connexions et cliquez sur le cercle à droite de l'événement `Value Changed`. Déplacez la souris jusqu'à l'icône `File's Owner` et choisissez l'action `changeCouleur` (voir Figure 6.19).

Figure 6.19

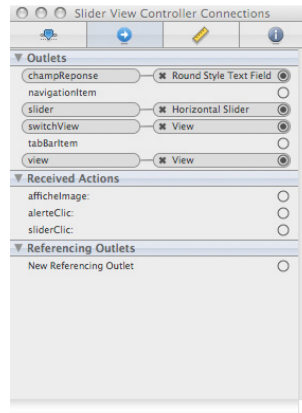
On associe l'événement `Value Changed` au contrôle segmenté.



Lorsque l'utilisateur cliquera sur l'un des états de ce contrôle, il déclenchera ainsi cette action. Dernière opération à ne pas oublier : vous devez modifier les valeurs qui bornent la réglette. En effet, l'opacité d'un objet d'interface varie de 0 à 1. Cliquez sur la réglette, ouvrez l'inspecteur d'attributs et sélectionnez `0,00` dans le champ `Minimum` ; `1,00` dans le champ `Maximum` et `0,50` dans le champ `Initial` (voir Figure 6.20).

Figure 6.20

Vérifiez dans l'inspecteur de connexions que toutes vos associations sont bien identiques à cet exemple.



Enregistrez le résultat et compilez votre projet. Vous avez mis en avant tous les contrôles avancés du SDK de l'iPhone. La réglette définit une valeur comprise entre 0 et 1. Celle-ci apparaît aussi bien dans un champ de texte situé sur la vue principale qu'à travers une boîte d'alerte qui apparaît en surimpression à l'écran. En parallèle, un contrôle segmenté permet de modifier la couleur d'arrière-plan d'une sous-vue. Quelle que soit la couleur actuellement choisie, la réglette joue sur l'opacité de cette sous-vue : les teintes rouge et vert s'estompent progressivement ou deviennent totalement opaques (voir Figure 6.21).

Figure 6.21

Notre application complète modifie l'opacité d'une sous-vue à l'aide d'une réglette.



Vous pouvez naturellement effectuer le même traitement sur n'importe quelle instance d'une sous-classe d'`UIView` et ainsi jouer sur la transparence d'une image, d'un libellé ou d'un champ de texte par exemple.

Intégrer une roulette à l'application

En manipulant le module des réglages de l'iPhone ou le calendrier, vous avez probablement déjà remarqué les "roulettes" qui défilent sur l'écran du mobile et qui vous aident à choisir une valeur précise (voir Figure 6.22).

Figure 6.22

Le minuteur de l'application Horloge est une roulette d'un type particulier, timer.



On désigne souvent ce type de contrôle par leur patronyme anglais : il s'agit de pickers, des contrôles avancés qui formatent un certain volume de données sous la forme de tableaux dont les cellules se manipulent du bout des doigts. Ces roues peuvent revêtir de nombreuses apparences et se diviser en plusieurs sous-modules avec lesquels on interagit de manière individuelle. Ainsi, la sélection d'une donnée sur la première roue peut modifier les informations figurant sur la seconde roue et ainsi de suite.

L'implémentation de tels contrôles implique donc une grande rigueur. Non seulement vous devez veiller à l'homogénéité des données que vous faites figurer sur les roulettes, mais vous devez également gérer au mieux les interactions avec l'utilisateur. Leur intégration à une application soulève en parallèle de nombreux problèmes de design : leurs dimensions ne sont pas modifiables et ils occupent généralement l'espace principal sur l'écran, quel que soit le nombre d'informations que vous y faites figurer. Pour toutes ces raisons, les développeurs préfèrent en règle générale les disposer sur une seconde vue dédiée aux réglages d'une application. Redoublez de concentration : disposées harmonieusement, les roulettes sont des contrôles très agréables à utiliser qui participent largement à l'ergonomie d'une application.

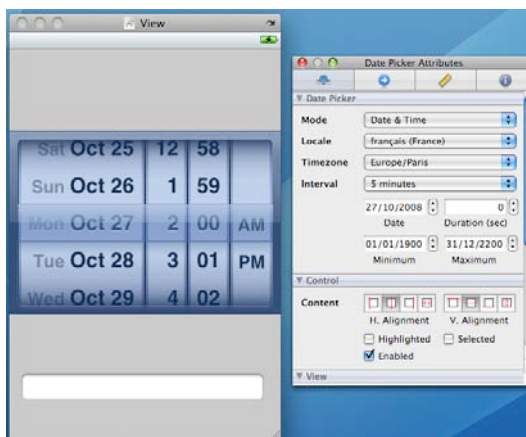
Les roulettes de temps

L'un des rôles essentiels des roulettes consiste à sélectionner une date précise. Le SDK de l'iPhone intègre de nombreux modèles prédéfinis qui vous permettent de solliciter l'utilisateur et de l'enjoindre à repérer une date du calendrier. Ces contrôles prêts à l'emploi intègrent jusqu'à quatre roues et relèvent ainsi le jour, le mois, l'année et l'heure précisés par l'utilisateur. Même s'il s'agit de pickers traditionnels, ces modèles disposent de leur propre classe en Cocoa Touch : `UIDatePicker`.

Sous Xcode, déroulez le menu File, cliquez sur New Project et sélectionnez le modèle "View-Based Application". Saisissez `Roue01` en guise de nom de projet et validez en cliquant sur le bouton Save. De retour dans la fenêtre principale du développement, ouvrez le fichier `Roue01ViewController.xib`. Déposez ensuite un contrôle de type `Date Picker` depuis la bibliothèque d'Interface Builder vers votre vue principale (voir Figure 6.23).

Figure 6.23

L'ajout d'un objet `Date Picker` et la modification de ses attributs.



Dans un second temps, ajoutez un simple champ de texte et étendez-le sur toute la largeur de la vue. Nous l'utiliserons pour afficher la date sélectionnée par l'utilisateur. Sélectionnez votre roulette puis ouvrez l'inspecteur d'attributs. Comme vous le constatez à la section `Date Picker`, le SDK de l'iPhone prévoit quatre modèles différents : `Date & Time` (quatre roues), `Time` (trois roues), `Date` (trois roues) et `Timer` (deux roues). Mieux encore : les champs `Locale` et `Timezone` vous permettent de définir la localisation des informations de date et d'heure. En choisissant "français" dans le premier champ, puis "Europe/Paris" dans le second, vous utilisez automatiquement des règles françaises de formatage.

Interface Builder n'adapte pas toujours le contrôle à votre sélection et vous ne verrez le résultat final que sur un véritable iPhone ou sur le Simulateur. N'hésitez pas non plus à indiquer la date de départ et de fin à travers l'inspecteur d'attributs de la roulette.

Enregistrez votre interface et revenez à Xcode : nous associerons les outlets et actions dans un second temps. Commencez par ouvrir le fichier `Roue01ViewController.h` et ajoutez les lignes figurant en gras au listing ci-dessous.

Listing 6.12 : *Roue01ViewController.h*

```
#import <UIKit/UIKit.h>

@interface Roue01ViewController : UIViewController {
    IBOutlet UIDatePicker *roueDate;
    IBOutlet UITextField *champTexte;
}
@property (nonatomic, retain) UIDatePicker *roueDate;
@property (nonatomic, retain) UITextField *champTexte;
-(IBAction)afficheDate;

@end
```

Nous avons ici mentionné nos deux outlets, `roueDate` et `champTexte`, et introduit une action, `afficheDate` qui relève la date sélectionnée par l'utilisateur et l'affiche dans le champ de texte. Modifiez à présent l'implémentation de cette classe, en ajoutant les lignes suivantes au fichier `Roue01ViewController.m`.

Listing 6.13 : *Roue01ViewController.m*

```
#import "Roue01ViewController.h"

@implementation Roue01ViewController
@synthesize roueDate;
@synthesize champTexte;

-(IBAction)afficheDate {
    NSDate *dateSelectionnee = [roueDate date];
    NSString *message = [[NSString alloc] initWithFormat:@"Date : %@",
        dateSelectionnee];
    champTexte.text = message;
    [message release];
}

/*
// The designated initializer. Override to perform setup that is required
// before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
    nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
        bundle:nibBundleOrNil]) {
        // Custom initialization
    }
}
```

```

        return self;
    }
    */

    /*
    // Implement loadView to create a view hierarchy programmatically,
    // without using a nib.
    - (void)loadView {
    }
    */

    // Implement viewDidLoad to do additional setup after loading the view,
    // typically from a nib.
    - (void)viewDidLoad {
        NSDate *maintenant = [[NSDate alloc] init];
        [roueDate setDate:maintenant animated:YES];
        [maintenant release];
        [super viewDidLoad];
    }

    /*
    // Override to allow orientations other than the default portrait
    // orientation.
    - (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
        interfaceOrientation {
        // Return YES for supported orientations
        return (interfaceOrientation == UIInterfaceOrientationPortrait);
    }
    */

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
        // Releases the view if it doesn't have a superview
        // Release anything that's not essential, such as cached data
    }

    - (void)dealloc {
        [roueDate release];
        [champTexte release];
        [super dealloc];
    }

@end

```

Notez tout d'abord que nous avons modifié la méthode `viewDidLoad` qui s'exécute lorsque la vue est chargée par l'iPhone. Nous avons défini un objet `NSDate` correspondant à la date courante et nous l'avons assigné à la roulette en tant que valeur de départ, grâce à la méthode `setDate`. L'action `afficheDate` est très simple : on repère tout d'abord la date sélectionnée par l'utilisateur avant de formater une chaîne de caractères et de l'afficher dans le champ de texte.

Ouvrez votre vue principale sous Interface Builder, cliquez sur l'icône File's Owner en maintenant la touche Ctrl enfoncée et déplacez le curseur de la souris vers la roulette. Sélectionnez alors l'élément `roueDate` du menu contextuel. Procédez de la même manière avec le champ de texte. Cliquez ensuite sur la roulette, ouvrez l'inspecteur de connexions et reliez l'événement `ValueChanged` au File's Owner, en sélectionnant l'action `afficheDate`. Enregistrez votre interface puis compilez votre projet. La roulette apparaît au centre de l'écran et la moindre pression met le champ de texte à jour (voir Figure 6.24).

Figure 6.24

La roulette permet de sélectionner une date et de l'afficher dans le champ de texte.



Le formatage de la date révèle toutefois des valeurs pénibles à afficher, comme "2010-11-14 09:05:09 +0100". Il s'agit de la nomenclature par défaut d'Objective-C, que vous pouvez heureusement adapter à l'aide de la classe `NSDateFormatter`. Là encore, Cocoa Touch prévoit une série de constantes pour vous aider à formater simplement des dates. Nous allons les utiliser dans un premier temps. Ouvrez à nouveau le fichier `Roue01ViewController.m` et modifiez l'action `afficheDate` comme ci-dessous.

Listing 6.14 : *afficheDate*

```
-(IBAction)afficheDate {
    NSDate *dateSelectionnee = [roueDate date];
    NSDateFormatter *formatageDate = [[[NSDateFormatter alloc] init]
    autorelease];
    [formatageDate setDateStyle:NSDateFormatterShortStyle];
    [formatageDate setTimeStyle:NSDateFormatterShortStyle];
    NSString *dateFormatee = [formatageDate stringFromDate:dateSelectionnee];
    champTexte.text = dateFormatee;
```

```
[formatageDate release];  
[dateFormatee release];  
}
```

On crée ici une instance `formatageDate` de la classe `NSDateFormatter`. À l'aide des méthodes `setDateStyle` et `setTimeStyle`, on choisit un formatage spécifique, défini par la constante `NSDateFormatterShortStyle`. On applique ensuite cette règle à la date relevée à l'aide de la roue. Le champ de texte se met à jour en conséquence et affiche désormais des valeurs du type "11/14/10 9:10 AM" (voir Figure 6.25).

Figure 6.25

On modifie le formatage de la date.



Les autres constantes de formatage

Le SDK de l'iPhone reconnaît quatre constantes principales pour formater les dates, dans le même style que `NSDateFormatterShortStyle`. `NSDateFormatterMediumStyle` renvoie des dates du type "Nov 14, 2009 9:45:58 AM". `NSDateFormatterLongStyle` formate les données selon la nomenclature "November 14, 2009 9:45:51 AM CEST". Enfin, `NSDateFormatterFullStyle` renvoie "Saturday, November 14, 2009 9:45:33 AM CET". Sélectionnez le formatage qui vous convient et appliquez les règles de localisation des chaînes de caractères pour renvoyer des valeurs selon la notation française.

Heureusement, vous avez également la possibilité de créer votre propre règle de formatage de toutes pièces et de lui appliquer une notation française. On parvient à un tel résultat grâce aux méthodes `setDateFormat` et `setLocale`, toutes deux liées à la classe `NSDateFormatter`. Ouvrez à nouveau le fichier `Roue01ViewController.m` et adaptez l'action `afficheDate` comme au Listing 6.15.

Listing 6.15 : *afficheDate*

```
-(IBAction)afficheDate {
    NSDate *dateSelectionnee = [roueDate date];
    NSDateFormatter *formatageDate = [[[NSDateFormatter alloc] init]
    autorelease];
    [formatageDate setLocale:[[[NSLocale alloc]
        initWithLocaleIdentifier:@"fr_FR"] autorelease]];
    [formatageDate setDateFormat:@"EEEE d MMMM YYYY, HH:mm"];
    NSString *dateFormatee = [formatageDate stringFromDate:
        dateSelectionnee];
    champTexte.text = dateFormatee;
    [dateFormatee release];
}
```

Enregistrez vos modifications et compilez à nouveau votre projet. Votre roulette de temps renvoie désormais des valeurs du type "samedi 14 novembre 2009, 09:50". Nous avons commencé par forcer la localisation en français à l'aide de la méthode `setLocale` et de l'identifiant `fr_FR`. Nous avons ensuite défini la règle de formatage grâce à la syntaxe "EEEE d MMMM YYYY, HH:mm" ; le nom du jour et du mois figurent au complet et la date est renvoyée en bout de chaîne, séparée par une virgule. Adaptez librement cette syntaxe aux besoins de vos applications.

Intégrer une roulette personnalisée

Les instances de la classe `UIDatePicker` correspondent à des roulettes d'un type prédéfini, dont les données sont automatiquement remplacées par un calendrier. Vous avez naturellement la possibilité de charger tout type de contenu dans vos roulettes, en créant des instances de la classe `UIPickerView`. Ces derniers ne demeurent que des conteneurs visuels, qui attendent deux protocoles : la source des données (`UIPickerViewDataSource`) et leur délégué (`UIPickerViewDelegate`). Le premier élément répond à la question "quelles sont les données à afficher ?" et le second à "comment traiter ces données ?". À travers ce type de relations, vous comprenez immédiatement l'intérêt de ces roulettes personnalisées : vous pouvez charger leur contenu depuis n'importe quelle source, qu'il s'agisse d'un simple tableau, d'une base de données, d'un fichier de propriétés `plist` ou d'une URL externe.

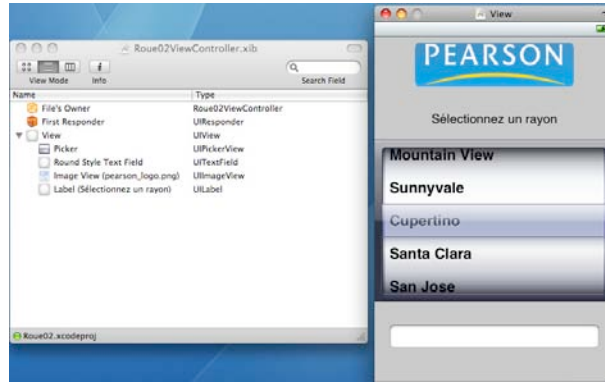
Ajouter une roulette simple

Dans le cadre de notre premier exemple, nous évoluerons dans un contexte simple et nous créerons les données dans un tableau intégré à la classe principale. Ouvrez Xcode et créez un nouveau projet de type View-Based Application que vous nommerez "Roue02". Effectuez un double-clic sur la vue `Roue02ViewController.xib` et déposez un objet `Picker View` sur la vue principale. Comme précédemment, ajoutez également un champ de texte, en l'étendant sur toute la largeur de l'interface. Telle que nous nous apprêtons à la définir, la

roulette ne présentera jamais les "vraies" données sous Interface Builder et renverra son allure par défaut, où la ville de Cupertino est sélectionnée (voir Figure 6.26).

Figure 6.26

L'intégration d'une roulette personnalisée dans notre application.



Nous souhaitons ici créer l'application officielle des éditions Pearson. La roulette représente ainsi les différents "rayons" de la librairie et l'utilisateur est invité à sélectionner le domaine littéraire de son choix (sciences humaines, apprentissage de l'anglais, informatique, multimédia, etc.). Sous Interface Builder, sélectionnez la roulette puis ouvrez l'inspecteur de connexions. Reliez les outlets dataSource et delegate à l'icône du File's Owner. Nous indiquons ainsi que la source des données et leur traitement sont définis dans la classe contrôleur principale. Enregistrez votre fichier xib et revenez à Xcode.

Dans l'interface de notre classe principale, nous avons besoin de définir deux outlets (la roulette et le champ de texte) ainsi qu'un tableau contenant les données. Commencez par éditer le fichier `Roue02ViewController.h` en saisissant le code figurant au listing ci-dessous.

Listing 6.16 : *Roue02ViewController.h*

```
#import <UIKit/UIKit.h>

@interface Roue02ViewController : UIViewController <UIPickerViewDelegate,
    UIPickerViewDataSource> {
    IBOutlet UIPickerView *roueSimple;
    IBOutlet UITextField *champTexte;
    NSArray *roueDonnees;
}

@property (nonatomic, retain) UIPickerView *roueSimple;
@property (nonatomic, retain) UITextField *champTexte;
@property (nonatomic, retain) NSArray *roueDonnees;

@end
```

Rien de bien inhabituel pour l'instant : nous définissons les deux outlets ainsi que le tableau de données, qui est une instance de la classe `NSArray`. C'est également dans cette interface que vous déclarerez vos actions, si vous ajoutez des contrôles supplémentaires comme des boutons par exemple. L'implémentation de la classe est toutefois plus complexe. Dans la mesure où nous avons choisi de déclarer la source et le délégué de la roulette dans la classe principale, vous devez ajouter une série de méthodes. Ouvrez le fichier `Roue02ViewController.m` et adaptez son code en conséquence.

Listing 6.17 : *Roue02ViewController.m*

```
#import "Roue02ViewController.h"

@implementation Roue02ViewController
@synthesize roueSimple;
@synthesize roueDonnees;
@synthesize champTexte;

- (void)viewDidLoad {
    NSArray *donnees = [[NSArray alloc] initWithObjects:@"Economie",
        @"Sciences", @"Histoire", @"Apprentissage", @"Classes prépas",
        @"Informatique", @"Multimédia", @"Vie pratique", nil];
    self.roueDonnees = donnees;
    [donnees release];
    [super viewDidLoad];
}

/*
// The designated initializer. Override to perform setup that is required
// before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
    nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
        bundle:nibBundleOrNil]) {
        // Custom initialization
    }
    return self;
}
*/

/*
// Implement loadView to create a view hierarchy programmatically,
// without using a nib.
- (void)loadView {
}
*/

/*
// Override to allow orientations other than the default portrait
// orientation.
```

```

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
*/

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [roueSimple release];
    [roueDonnees release];
    [champTexte release];
    [super dealloc];
}

// Nombre de composantes de la roulette
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)roueSimple {
    return 1;
}

// Nombre de lignes à afficher dans la roulette
- (NSInteger)pickerView:(UIPickerView *)roueSimple
    numberOfRowsInComponent:(NSInteger)component {
    return [roueDonnees count];
}

// Délégué appelé x fois, où x est la valeur retournée par
// numberOfRowsInComponent
- (NSString *)pickerView:(UIPickerView *)roueSimple
    titleForRow:(NSInteger)cellule
    forComponent:(NSInteger)component {
    return [roueDonnees objectAtIndex:cellule];
}

// Gestionnaire d'événements déclenché lorsqu'on
// sélectionne une ligne (didSelectRow)
- (void)pickerView:(UIPickerView *)roueSimple didSelectRow:(NSInteger)
    cellule inComponent:(NSInteger)component {
    NSString *choix = [roueDonnees objectAtIndex:cellule];
    NSString *message = [[NSString alloc] initWithFormat:@"Choix : %@",
        choix];
    champTexte.text = message;
}

@end

```

L'affichage de la roulette commence par la création d'un tableau `donnees`, que l'on initialise à l'aide d'une série de mots-clés au sein de la méthode `viewDidLoad`. Dans notre exemple, il s'agit des différents rayons des éditions Pearson. On affecte ensuite le contenu de ce tableau à la variable `rouesDonnees`. Nous devons alors prendre le temps de définir correctement la source de la roulette. On doit tout d'abord indiquer le nombre de composantes qui la constituent, c'est-à-dire le nombre de "roues" qu'elle doit présenter :

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)roueSimple {  
    return 1;  
}
```

Ici, notre roulette ne comporte qu'un seul élément sélectionnable à la fois. On doit maintenant définir le nombre de lignes à afficher sur cette unique composante :

```
- (NSInteger)pickerView:(UIPickerView *)  
    roueSimple numberOfRowsInComponent:(NSInteger)component {  
    return [roueDonnees count];  
}
```

Notre tableau contient huit cellules et la roulette est ainsi instanciée avec autant de lignes sélectionnables. La méthode suivante est donc appelée huit fois afin de remplir les lignes de la roulette avec chacune des cellules du tableau :

```
- (NSString *)pickerView:(UIPickerView *)roueSimple  
    titleForRow:(NSInteger)cellule  
    forComponent:(NSInteger)component {  
    return [roueDonnees objectAtIndex:cellule];  
}
```

Enfin, nous devons gérer les événements associés à la roulette, en particulier la sélection d'une ligne spécifique. La méthode suivante est appelée à chaque fois que l'utilisateur choisit l'une des entrées :

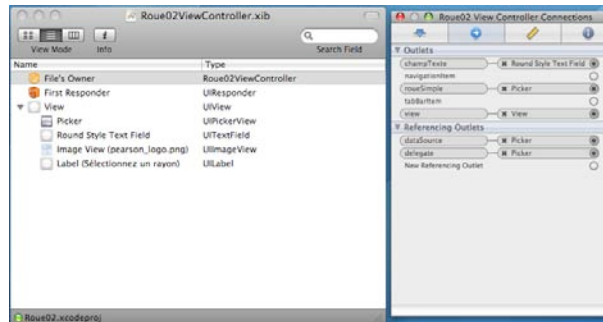
```
- (void)pickerView:(UIPickerView *)roueSimple didSelectRow:(NSInteger)  
    cellule inComponent:(NSInteger)component {  
    NSString *choix = [roueDonnees objectAtIndex:cellule];  
    NSString *message = [[NSString alloc] initWithFormat:@"%@",  
        choix];  
    champTexte.text = message;  
}
```

On repère ainsi la cellule sélectionnée et on la convertit en une chaîne de caractères que l'on affiche dans le champ de texte de notre application. Ouvrez à nouveau votre vue principale sous Interface Builder, cliquez sur File's Owner en maintenant la touche Ctrl enfoncée et dé-

finissez vos deux outlets : `roueSimple` et `champTexte`. Vérifiez également que les éléments `dataSource` et `delegate` de la roulette sont bien associés à `File's Owner` (voir Figure 6.27).

Figure 6.27

Les connexions de notre interface principale.



Enregistrez votre projet et compilez-le : vous sélectionnez ensuite votre rayon dans le Simulateur d'iPhone et la valeur relevée figure dans le champ de texte (voir Figure 6.28).

En lieu et place du champ de texte, vous pouvez envisager tout type de traitement pour la sélection de l'utilisateur. Un bouton permettrait alors de charger une seconde vue en fonction du choix initial, par exemple.

Figure 6.28

On sélectionne facilement un élément à l'aide de la roulette, pour l'afficher dans le champ de texte en bas de l'écran.



Ajouter une roulette multiple

Comme vous venez de le voir, la définition d'une roulette nécessite d'indiquer le nombre de "composantes" qui la constituent. Il s'agit du nombre d'éléments que l'utilisateur est susceptible de sélectionner individuellement. À l'affichage, cela se traduit par l'ajout de

roues supplémentaires sur votre contrôle, à la manière des roulettes de date que nous avons développées au cours de ce chapitre. Consacrons-nous à un nouvel exemple pour illustrer les différences fondamentales avec la technique précédente. Sous Xcode, créez le nouveau projet "Roue03" et ajoutez directement un élément UIPickerView et un champ de texte à la vue principale. Dans notre exemple, nous allons demander à l'utilisateur de choisir une ville française, puis un service qui lui est associé (météo, infos pratiques, circulation, etc.). Modifiez ensuite le code du fichier `Roue03ViewController.h`, en saisissant le listing ci-dessous.

Listing 6.18 : *Roue03ViewController.h*

```
#import <UIKit/UIKit.h>

#define indexVille 0
#define indexService 1

@interface Roue03ViewController : UIViewController <UIPickerViewDelegate,
    UIPickerViewDataSource> {
    IBOutlet UIPickerView *roueDouble;
    IBOutlet UITextField *champTexte;
    NSArray *villesContenu;
    NSArray *servicesContenu;
}

@property (nonatomic, retain) UIPickerView *roueDouble;
@property (nonatomic, retain) UITextField *champTexte;
@property (nonatomic, retain) NSArray *villesContenu;
@property (nonatomic, retain) NSArray *servicesContenu;

@end
```

Nous ouvrons l'en-tête de ce fichier par la définition de deux constantes, `indexVille` et `indexService`, qui nous permettront d'identifier chacune des deux composantes de la roulette. Nous créons ensuite deux outlets, `roueDouble` et `champTexte` qui correspondent aux deux éléments d'interface. Enfin, nous ajoutons deux tableaux de type `NSArray` qui contiendront les données affichées dans les deux roulettes. Nous terminons cette interface par la déclaration de toutes les propriétés. Une fois encore, nous n'ajoutons pas directement d'action : c'est l'une des méthodes associées à la classe `UIPickerView` qui sera responsable du traitement des données.

Modifiez à présent le fichier `Roue03ViewController.m` en ajoutant la déclaration des données et le délégué de la roulette.

Listing 6.19 : Roue03ViewController.m

```

#import "Roue03ViewController.h"

@implementation Roue03ViewController
@synthesize roueDouble;
@synthesize champTexte;
@synthesize villesContenu;
@synthesize servicesContenu;

- (void)viewDidLoad {
    NSArray *villesArray = [[NSArray alloc] initWithObjects:@"Paris",
        @"Marseille", @"Lyon", @"Bordeaux", @"Lille", nil];
    self.villesContenu = villesArray;
    [villesArray release];

    NSArray *servicesArray = [[NSArray alloc] initWithObjects:@"La
        météo", @"Les infos", @"La circulation", @"Les loisirs", @"Les
        écoles", nil];
    self.servicesContenu = servicesArray;
    [servicesArray release];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [roueDouble release];
    [villesContenu release];
    [servicesContenu release];
    [super dealloc];
}

- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)roueDouble {
    return 2;
}

- (NSInteger)pickerView:(UIPickerView *)roueDouble
    numberOfRowsInComponent:(NSInteger)component {
    if (component == indexVille)
        return [self.villesContenu count];
    return [self.servicesContenu count];
}

- (NSString *)pickerView:(UIPickerView *)roueDouble
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component {
    if (component == indexVille)
        return [self.villesContenu objectAtIndex:row];
}

```

```
        return [self.servicesContenu objectAtIndex:row];
    }

    - (void)pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger)
        cellule inComponent:(NSInteger)component {
        NSInteger villeChoisie = [pickerView selectedRowInComponent:
                                indexVille];
        NSInteger serviceChoisi = [pickerView selectedRowInComponent:
                                indexService];

        NSString *ville= [villesContenu objectAtIndex:villeChoisie];
        NSString *service= [servicesContenu objectAtIndex:serviceChoisi];

        NSString *message = [[NSString alloc] initWithFormat:@"%@" à %@ ?
                                C'est parti !", service, ville];

        champTexte.text = message;
    }

@end
```

Ce contrôleur illustre parfaitement les possibilités d'extension de notre exemple précédent. Au sein de la méthode `viewDidLoad`, nous définissons cette fois deux tableaux contenant le nom de quelques villes françaises et les services associés que notre application compte afficher. Nous indiquons ensuite que la roulette doit compter deux composantes différentes, à travers la méthode `numberOfComponentsInPickerView`. Pour chaque composante, la méthode `numberOfRowsInComponent` construit des roulettes contenant le nombre de lignes approprié : il s'agit du nombre d'éléments définis dans chacun des deux tableaux. Grâce au délégué, nous remplissons enfin les roulettes avec les éléments des tableaux. Notez comment les constantes déclarées dans l'en-tête nous permettent de valider rapidement chacune des deux roulettes :

```
    if (component == indexVille) {
        ...
    }
```

Cette boucle conditionnelle vérifie que l'on parcourt actuellement la composante de gauche, dédiée aux villes. Enfin, la méthode `didSelectRow` récupère à chaque sélection de ligne l'élément de la roulette de gauche et de droite. On concatène une chaîne de caractères dans la variable `message`, que l'on affiche en dernier lieu dans le champ de texte en bas de l'écran.

Sous Interface Builder, cliquez sur l'icône *File's Owner* tout en maintenant la touche `Ctrl` du clavier enfoncée, puis désignez les deux outlets : la roulette et le champ de texte. À l'inverse, cliquez sur la roulette et ouvrez l'inspecteur de connexions ; assignez les éléments `dataSource` et `delegate` à *File's Owner*. Enregistrez votre travail puis compilez le projet. Le Simulateur de l'iPhone vous permet désormais de sélectionner individuellement deux infor-

mations, qui sont ensuite combinées dans une seule chaîne de caractères (voir Figure 6.29). Vous pouvez envisager tout type de traitement à partir de cette sélection initiale. Dans notre exemple, le traitement se poursuivra alors par l’affichage de la météo d’une localité spécifique ou par l’état actuel du trafic.

Figure 6.29

On sélectionne individuellement deux roulettes, dont le contenu est extrait de deux tableaux.



Définir des éléments dépendants dans une liste de propriétés

Même si elle offre de bons résultats, la technique précédente manque indéniablement de souplesse. Il est en effet difficile de saisir d’interminables tableaux dans le code de votre contrôleur principal, sans compter que les deux roulettes ne dépendent pas l’une de l’autre. Pour peupler rapidement le contenu de l’une des roulettes, vous avez ainsi tout intérêt à utiliser l’éditeur de listes de propriétés : à la manière de votre fichier `Infos.plist`, vous créez un fichier XML dont le contenu est directement intégré à la roulette. En imbriquant plusieurs éléments les uns dans les autres, vous créez rapidement une liste hiérarchique d’objets : vous avez ainsi la possibilité de faire figurer les éléments enfants dans la roulette de droite. On imagine tout de suite l’intérêt d’un tel dispositif – non seulement vous allégez votre code source en chargeant un fichier de données externes, mais vous avez aussi la possibilité d’affiner les choix de l’utilisateur, qui sélectionne un élément précis après avoir choisi une catégorie générale.

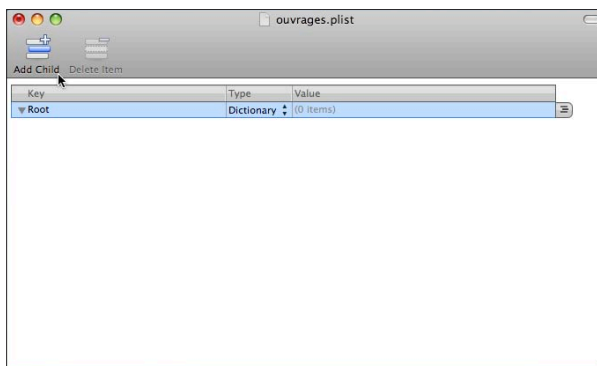
Reprenons notre exemple d’application officielle des éditions Pearson. Lors de notre première étude, nous nous sommes contentés d’afficher les "rayons" de la librairie. Avec un tel système, il nous est possible de définir l’ensemble du catalogue de l’éditeur dans une liste de propriétés, en imbriquant les titres des ouvrages dans les rubriques qui leur sont associées. La roulette de gauche reprend ainsi les catégories définies précédemment et la

moindre sélection modifie le contenu de la roulette de droite, qui n'affiche que les ouvrages correspondant à la requête.

Ouvrez une fenêtre du Finder et rendez-vous dans le dossier /Developer/Applications/Utilities. Effectuez un double clic sur "Property List Editor" : un fichier XML vierge apparaît à l'écran. Il est défini par un élément racine, Root, qui obéit au type Dictionary. Sélectionnez cette première ligne, puis cliquez sur le bouton Add Child de la barre d'outils (voir Figure 6.30).

Figure 6.30

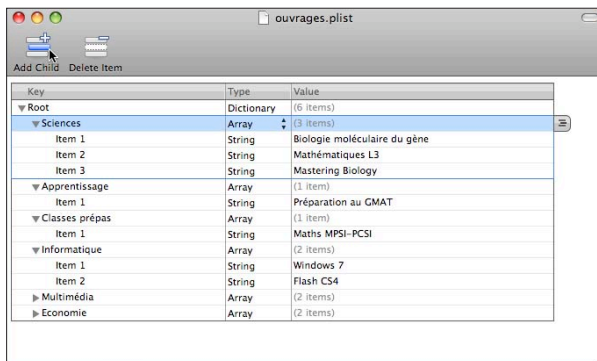
On définit le contenu de la roulette à travers une liste de propriétés.



Saisissez alors le nom d'un premier élément parent. Il s'agit ici d'une catégorie d'ouvrages, "Informatique" par exemple. Déroulez le menu au champ Type et sélectionnez Array. Vous créez ainsi un premier tableau, capable d'héberger des éléments enfants. Cliquez sur la flèche à gauche de cette première entrée afin de dérouler l'arborescence. Cliquez à nouveau sur le bouton Add Child de la barre d'outils afin d'ajouter une entrée imbriquée dans ce premier tableau ; il s'agit ici d'une chaîne de caractères, de type String donc, qui contient le nom d'un ouvrage du rayon informatique. Renouvelez la procédure pour tous les éléments que vous souhaitez faire figurer dans votre roulette (voir Figure 6.31). Vos éléments parents devront ainsi être définis en tant qu'array et vos éléments enfants en tant que string.

Figure 6.31

On complète la liste des propriétés en saisissant toutes les données devant figurer dans la roulette et en les hiérarchisant.



Déroulez enfin le menu File de Property List Editor > Save as et saisissez "ouvrages.plist" avant de cliquer sur le bouton Save. Vous avez ainsi rapidement créé le contenu de votre roulette, en triant l'ensemble des données en éléments parents et éléments enfants.

De retour sous Xcode, créez un nouveau projet de type View-Based Application que vous nommerez Roue04. Déroulez ensuite le menu Project, cliquez sur Add to Project et sélectionnez le fichier ouvrages.plist. Cochez la case "Copy items into destination group's folder" afin de copier l'élément dans le dossier de votre projet. Ouvrez ensuite le fichier Roue04ViewController.xib et ajoutez un objet Picker View et un champ de texte comme vous en avez maintenant largement l'habitude. Affichez l'inspecteur de connexions, sélectionnez votre roulette et associez les éléments dataSource et delegate au File's Owner. Enregistrez votre interface puis revenez sous Xcode.

Comme nous l'avons vu à travers l'outil Property List Editor, la création d'un tel index dont les éléments sont hiérarchisés correspond à un *dictionnaire*. En Objective-C, vous devez ainsi déclarer une instance de la classe NSDictionary qui va contenir l'ensemble de votre liste de propriétés. En parallèle, vous déclarez deux tableaux de type NSArray qui correspondent à chacune des deux roulettes. Modifiez le fichier Roue04ViewController.h en conséquence.

Listing 6.20 : Roue04ViewController.h

```
#import <UIKit/UIKit.h>

#define indexRayon 0
#define indexOuvrage 1

@interface Roue04ViewController : UIViewController <UIPickerViewDelegate,
    UIPickerViewDataSource> {
    IBOutlet UIPickerView *roulette;
    IBOutlet UITextField *champTexte;

    NSDictionary *ouvragesPearson;
    NSArray *rayons;
    NSArray *ouvrages;
}

@property (retain, nonatomic) UIPickerView *roulette;
@property (retain, nonatomic) UITextField *champTexte;
@property (retain, nonatomic) NSDictionary *ouvragesPearson;
@property (retain, nonatomic) NSArray *rayons;
@property (retain, nonatomic) NSArray *ouvrages;

@end
```

Si l'implémentation de cette classe est rigoureuse, elle n'est pas fondamentalement différente de notre exemple précédent. La lecture des données définies dans la liste des propriétés vous épargne du reste la création manuelle des tableaux. Adaptez le fichier `Roue04ViewController.m` à l'aide du listing ci-dessous.

Listing 6.21 : *Roue04ViewController.m*

```
#import "Roue04ViewController.h"

@implementation Roue04ViewController
@synthesize roulette;
@synthesize champTexte;
@synthesize ouvragesPearson;
@synthesize rayons;
@synthesize ouvrages;

- (void)viewDidLoad {
    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistChemin = [bundle pathForResource:@"ouvrages"
                                                ofType:@"plist"];
    NSDictionary *dictionnaire = [[NSDictionary alloc]
                                   initWithContentsOfFile:plistChemin];
    self.ouvragesPearson = dictionnaire;
    [dictionnaire release];
    NSArray *composantes = [self.ouvragesPearson allKeys];
    NSArray *tri = [composantes sortedArrayUsingSelector:@selector(compare)];

    self.rayons = tri;
    NSString *rayonSelec = [self.rayons objectAtIndex:0];
    NSArray *array = [ouvragesPearson objectForKey:rayonSelec];
    self.ouvrages = array;

    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [roulette release];
    [ouvragesPearson release];
    [ouvrages release];
    [rayons release];
    [champTexte release];
    [super dealloc];
}
```

```

- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 2;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component {
    if (component == indexRayon)
        return [self.rayons count];
    return [self.ouvrages count];
}

- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component {
    if (component == indexRayon)
        return [self.rayons objectAtIndex:row];
    return [self.ouvrages objectAtIndex:row];
}

- (void)pickerView:(UIPickerView *)pickerView
    didSelectRow:(NSInteger)row
    inComponent:(NSInteger)component
{
    if (component == indexRayon)
    {
        NSString *rayonSelec = [self.rayons objectAtIndex:row];
        NSArray *array = [ouvragesPearson objectForKey:rayonSelec];
        self.ouvrages = array;
        [roulette selectRow:0 inComponent:indexOuvrage animated:YES];
        [roulette reloadComponent:indexOuvrage];
    }

    NSInteger rayonLigne = [roulette selectedRowInComponent:indexRayon];
    NSInteger ouvrageLigne = [roulette selectedRowInComponent:
        indexOuvrage];

    NSString *rayonChoisi = [self.rayons objectAtIndex:rayonLigne];
    NSString *ouvrageChoisi = [self.ouvrages objectAtIndex:ouvrageLigne];

    NSString *message = [[NSString alloc] initWithFormat:
        @"%@ > %@", rayonChoisi, ouvrageChoisi];

    champTexte.text = message;

    [message release];
}

@end

```


Tout commence par la méthode `viewDidLoad`, dans laquelle vous créez une instance de la classe `NSBundle`. Ce type d'objet permet de charger de manière dynamique des ressources externes, comme votre liste de propriétés par exemple. Jusqu'à présent, nous avons essentiellement manipulé ce type de ressources à travers Interface Builder, en sélectionnant les fichiers correspondants dans l'inspecteur d'attributs. Sachez toutefois que si vous souhaitez importer des images dans vos applications en quelques lignes de code, vous devez créer des instances de la classe `NSBundle` de ce type. Vous indiquez alors le chemin d'accès à votre fichier "ouvrages.plist" puis vous initialisez un dictionnaire avec son contenu :

```
NSString *plistChemin = [bundle pathForResource:@"ouvrages"
                                             ofType:@"plist"];
NSDictionary *dictionnaire = [[NSDictionary alloc]
                              initWithContentsOfFile:plistChemin];
self.ouvragesPearson = dictionnaire;
```

Souvenez-vous de votre liste de propriétés : les rayons correspondent aux éléments-clés et les ouvrages représentent toutes les valeurs qui leur sont associées. Nous commençons par remplir la roulette de gauche, qui ne contient donc que les clés de votre liste de propriétés. On récupère de telles valeurs à l'aide de la méthode `allKeys`.

```
NSArray *composantes = [self.ouvragesPearson allKeys];
NSArray *tri = [composantes sortedArrayUsingSelector:
                @selector(compare:)];
self.rayons = tri;
```

N'oubliez pas que nous sommes ici en train de modifier la déclaration de la méthode `viewDidLoad` : nous prévoyons ainsi le traitement initial de notre application. Nous partons du principe que le premier rayon est actuellement sélectionné : il correspond à l'index 0 du tableau `rayons`. Chargeons en conséquence le nom de tous les ouvrages associés à ce rayon dans la roulette de droite, afin de finaliser notre vue de départ :

```
NSString *rayonSelec = [self.rayons objectAtIndex:0];
NSArray *array = [ouvragesPearson objectForKey:rayonSelec];
self.ouvrages = array;
```

La méthode déléguée introduit une nouvelle boucle conditionnelle afin de relever la valeur sélectionnée dans la première roulette. On recharge alors la seconde composante en fonction de ce choix : les données évoluent dans la roulette de droite.

```
- (void)pickerView:(UIPickerView *)pickerView
didSelectRow:(NSInteger)row
inComponent:(NSInteger)component {
    if (component == indexRayon)
    {
        NSString *rayonSelec = [self.rayons objectAtIndex:row];
        NSArray *array = [ouvragesPearson objectForKey:rayonSelec];
```

```
self.ouvrages = array;  
[roulette selectRow:0 inComponent:indexOuvrage animated:YES];  
[roulette reloadComponent:indexOuvrage];  
}  
}
```

Rappelez-vous : cette méthode est déclenchée à chaque fois que l'utilisateur exerce une sélection sur la roulette. La composante concernée apparaît au paramètre `component`. On vérifie donc si la modification porte sur la roulette de gauche. Si c'est le cas, l'utilisateur a changé de rayon : il faut lui présenter une nouvelle liste d'ouvrages. On récupère alors le nom du rayon concerné à l'aide du paramètre `row` et on crée un nouveau tableau contenant les éléments enfants du rayon sélectionné. On réactualise enfin la roulette de droite à l'aide de la méthode `reloadComponent`. La suite du traitement se poursuit comme à l'exemple précédent : on récupère les valeurs sélectionnées par l'utilisateur et on les affiche dans le champ de texte figurant à l'écran.

Sous Xcode, vérifiez toutes les associations de vos outlets. Le File's Owner doit pointer vers le champ de texte et la roulette ; cette dernière voit ses outlets `dataSource` et `delegate` connectés au File's Owner. Enregistrez votre interface et compilez votre projet. Vous manipulez ainsi les deux roulettes dans le Simulateur d'iPhone et le moindre changement sur la composante de gauche recharge automatiquement la composante de droite (voir Figure 6.32).

Figure 6.32

Le contenu de la roulette de droite dépend de la sélection sur la roulette de gauche.

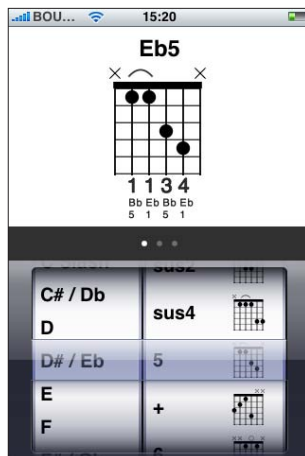


Pour aller plus loin

Au cours de ce chapitre, nous avons examiné en détails les contrôles avancés du kit de développement de l'iPhone. Contrairement aux simples boutons et aux champs de texte, ils occupent un espace significatif sur l'interface de votre application et votre rôle de designer se complique en conséquence : vous devez pleinement justifier l'utilisation de roulettes ou d'interrupteurs, sous peine d'alourdir votre interface et de la rendre confuse. Mais ils offrent en contrepartie des interactions plus riches et plus intuitives. En les agencant intelligemment autour d'une série d'onglets ou de vues secondaires, vous créez rapidement des écrans de réglages simples et précis sans dénaturer pour autant l'allure de votre application (voir Figure 6.33).

Figure 6.33

L'application 7 Chords exploite des roulettes dépendantes pour afficher la grille de tous les accords de guitare.



Par ailleurs, la lecture de données externes comme notre liste de propriétés ouvre un champ immense et facilite l'intégration de contenus supplémentaires. Nous allons précisément aborder cet aspect au cours du chapitre suivant, où nous apprendrons à lire et à enregistrer des données. C'est un passage obligé pour développer des applications plus ambitieuses encore, que les utilisateurs vont exploiter quotidiennement et mettre à jour en fonction de leurs besoins.



Lire et écrire des données

Au sommaire de ce chapitre

- Lire des données
- Enregistrer des données
- Pour aller plus loin

Au cours du chapitre précédent, vous avez appris à manipuler tout type de données et à agencer le contenu de votre application autour de contrôles élaborés. Parmi ceux-ci, nous avons exposé en détails la structure des roulettes, en leur associant des tableaux ou des listes de propriétés constellées d'informations.

Informations. Le mot est lâché. C'est bien l'épine dorsale du développement d'applications pour iPhone. Mise à part une série d'utilitaires dont l'intérêt tient en une simple vue et un contrôleur unique, la plupart des applications que vous développerez visent à soutenir une

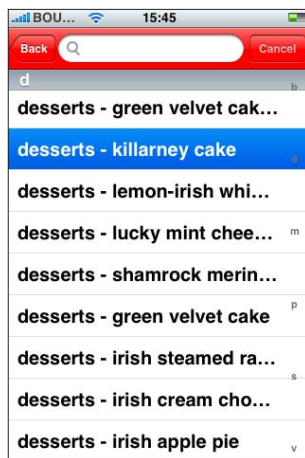
grande quantité d'informations. Lecteurs de livres électroniques, guides touristiques ou recettes de cuisine : toutes ces applications reposent sur de volumineuses bases de données, dont le contenu s'agence au mieux pour interagir avec l'utilisateur. Le contrôle privilégié de ce type de structure est la liste, qui prend l'apparence d'un tableau que l'on manipule du bout des doigts et dont les cellules sont cliquables (voir Figure 7.1). L'iPhone les utilise à tous les niveaux, en particulier dans sa fonction de téléphone où vous sélectionnez et trieze vos contacts à partir d'un large tableau. Les listes ne sont pas fondamentalement différentes des roulettes que nous avons développées précédemment. Elles puisent leur contenu à partir de tableaux ou de listes de propriétés et prévoient de nombreuses méthodes pour gérer la sélection de l'utilisateur ou la réaction au clic. Ce type de contrôle s'inscrit parfaitement dans un système de navigation de grande envergure.

À l'inverse, le contenu est parfois généré par l'utilisateur et vous devez l'enregistrer afin qu'il le retrouve au prochain lancement de l'application. Le kit de développement de l'iPhone dispose de nombreuses solutions pour enregistrer les préférences de l'utilisateur ou les données qu'il a compulsées, à travers plusieurs types de bases de données ou de fichier systèmes. Dans le même ordre d'idée, vous avez également la possibilité d'interroger un service à distance afin de récupérer des données formatées ; c'est notamment le cas des parseurs XML, particulièrement adaptés à la création d'un lecteur de flux RSS.

Ne négligez donc pas la lecture et l'écriture de données : ces deux opérations sont essentielles pour l'intérêt de vos applications et vous devez les maîtriser ... sur le bout des doigts.

Figure 7.1

L'application Recipes! s'articule autour d'une large base de données qui présente des recettes de cuisine sous forme de liste.



Lire des données

La lecture du contenu associé à votre application ne constitue qu'un versant de cette opération essentielle : vous devez également apprendre à l'agencer de manière élégante, afin de favoriser sa lisibilité et d'autoriser l'utilisateur à le manipuler. S'il est possible d'afficher des données dans des champs de texte, des libellés ou des roulettes, comme nous l'avons vu au cours des chapitres précédents, vous avez tout intérêt à maîtriser les listes. Illimitées en longueur, elles permettent d'afficher un grand volume de données tout en esquissant un système de navigation simple et intuitif.

En fonction du volume de données que vous souhaitez présenter à l'utilisateur, vous pouvez envisager tout type d'implémentation. La solution la plus simple consiste à trier le contenu avant de l'afficher dans des cellules sélectionnables. En cliquant sur un terme de premier niveau, l'utilisateur affine son parcours et il navigue alors parmi une seconde liste constituée d'éléments plus précis. Vous déclinez ainsi le principe de la liste hiérarchisée, réalisée à l'aide de l'éditeur de listes de propriétés comme précédemment, et vous le transformez en un véritable système de navigation. Mais les listes revêtent des aspects très divers et vous pouvez largement personnaliser leur apparence en leur greffant une barre de recherche, un index alphabétique ou une série d'icônes.

À l'instar des roulettes, leur création est toutefois très rigoureuse. Vous devez spécifier la provenance des données, mais aussi définir une série de délégués qui prendront en charge les interactions avec l'utilisateur. Si la tâche n'est pas désarmante au vu de nos progrès en Objective-C, vous devrez redoubler de vigilance afin d'éviter des séances de débogage aussi longues que fastidieuses !

Créer une liste simple à partir de contenu externe

S'il est naturellement possible d'initialiser le contenu d'une liste à partir d'un tableau (un objet de type `NSArray`) défini de manière statique dans le contrôleur de votre application, vous avez tout intérêt à réunir vos données dans un fichier externe de type XML. Non seulement la mise à jour des informations de votre application sera plus simple, mais vous différenciez aussi de manière nette le contenu de votre projet de son aspect visuel et de ses traitements. Par ailleurs, vous créez ainsi une structure souple, capable de recueillir tout type de contenu. N'oubliez jamais que l'iPhone dispose d'un accès WiFi et 3G : selon toutes vraisemblances, vos utilisateurs pourront prétendre à une connexion Internet et rapatrier ainsi des données depuis l'extérieur. Si ces données sont formatées en XML, vous gagnez sur toute la ligne et vous étendez largement le giron de votre application, en vous ouvrant les portes des innombrables API de services web (Flickr, Twitter, Facebook, Google, Amazon, etc.). Nous y reviendrons.

Pour l'heure, contentons-nous d'un premier exemple s'inscrivant dans la continuité du chapitre précédent. Nous créons toujours l'application de référence des éditions Pearson et nous souhaitons présenter ses différents "rayons" à travers une liste ordonnée. Comme vous le verrez par la suite, ce mode de présentation est beaucoup plus adapté à ce type de tâche que les roulettes que nous avons mises en place jusqu'à présent.

Votre première opération consiste à créer une liste de propriétés à l'aide de l'éditeur intégré aux outils de développement de Mac OS X. Ouvrez une fenêtre du Finder, puis rendez-vous dans Developer/Applications/Utilities et effectuez un double-clic sur Property List Editor. Vous pouvez parfaitement partir du fichier ouvrages.plist réalisé précédemment. Dans notre exemple, nous l'étoffons largement en créant des sous-rubriques liées à tous les rayons principaux de l'éditeur. Définissez ainsi une série d'éléments parents et enfants en cliquant sur les boutons Add Item et Add Child de la barre d'outils. Ce dernier n'apparaît que lorsqu'un élément parent est défini selon le type Array ou Dictionary. Au final, vous devez ainsi faire apparaître des éléments parents de niveau un et des éléments enfants qui en dépendent (voir Figure 7.2).

Figure 7.2

La création de notre liste de propriétés, liste_Pearson.plist.

Key	Type	Value
Root	Dictionary	(8 items)
↳ Économie Gestion	Array	(11 items)
Item 1	String	Economie, Société
Item 2	String	Efficacité professionnelle
Item 3	String	Entrepreneuriat
Item 4	String	Environnement
Item 5	String	Finance, Comptabilité
Item 6	String	Management
Item 7	String	Marketing
Item 8	String	Sciences appliquées à la gestion
Item 9	String	Stratégie
Item 10	String	Systèmes d'informations
Item 11	String	Tourisme
↳ Sciences, techniques et médecine	Array	(11 items)
↳ Sciences humaines	Array	(3 items)
↳ Apprentissage de l'anglais	Array	(4 items)
↳ Classes préparas	Array	(2 items)
↳ Informatique et Internet	Array	(12 items)
↳ Photo, multimédia	Array	(4 items)
↳ Vie pratique, jeux	Array	(7 items)

Déroulez enfin le menu File, cliquez sur Save As et saisissez liste_Pearson.plist dans le champ prévu à cet effet. Ouvrez Xcode, déroulez le menu File et cliquez sur New Project. Sélectionnez le modèle View-Based Application afin de commencer d'emblée avec des contrôleurs prêts à l'emploi. Validez en cliquant sur le bouton Choose, puis saisissez "Table" en guise de nom de projet.

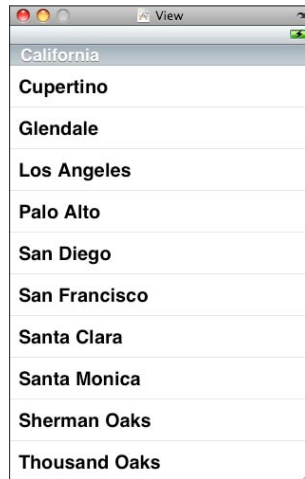
Créer la vue

Commencez par ouvrir le fichier TableViewController.xib associé à votre projet. Sous Interface Builder, déposez un élément Table View sur la vue principale de votre application. Là encore, cet élément ne révélera jamais sa "véritable" apparence sous Interface Builder et se voit constitué d'une série de villes américaines. N'y prenez pas garde : nous le

remplacerons par le contenu de notre fichier `.plist`. Par défaut, l'instance de la classe `UITableView` occupe tout l'espace de l'écran de l'iPhone (voir Figure 7.3). Contrairement aux roulettes, vous pouvez toutefois saisir les poignées qui l'encadrent et modifier sa hauteur ou sa largeur. Si le contenu dépasse de l'écran de l'iPhone, la liste se poursuit indéfiniment à la manière d'un "ascenseur" d'une application bureautique.

Figure 7.3

La vue principale de notre application comprend une instance d'`UITableView`.



Sélectionnez ensuite l'élément `Table View` dans la fenêtre `TableViewController.xib` et ouvrez l'inspecteur de connexions. Vous retrouvez les deux outlets associés aux roulettes : `dataSource` et `delegate`. Comme pour ces contrôles, vous devez les relier à l'icône du `File's Owner`. Cliquez sur les petits cercles figurant à droite de chaque entrée et maintenez la souris enfoncée jusqu'à cette icône. Cliquez éventuellement sur cette dernière icône afin de vérifier que la vue est bien connectée. Enregistrez votre travail et revenez sous Xcode.

Inscrire la liste dans une interface plus complexe

Dans notre exemple, l'objet `Table View` occupe l'ensemble de l'écran de l'iPhone et devient ainsi le système de navigation privilégié de notre application. Mais bien souvent, il ne constitue que l'un des éléments devant figurer à l'affichage : vous pouvez alors réduire ses dimensions ou l'intégrer dans une vue secondaire. Dans ce dernier cas, vous avez tout intérêt à créer un projet de type `Navigation-Based Application` ou `Tab Bar Application`. Vous disposerez ainsi d'un contrôleur racine, responsable de la navigation entre les différentes vues, et vous chargerez votre seconde vue à partir d'un fichier `xib` distinct.

Dans ce cas, vous gagnez un temps précieux en déroulant le menu File puis en cliquant sur New File et en créant de nouvelles instances de la classe UITableViewController et de l'interface View XIB. Notre exemple de catalogue des éditions Pearson s'inscrit précisément dans ce type de canevas ; *a priori*, nous avons tout intérêt à créer une série d'onglets ("Dernières parutions", "Consulter le catalogue", "Nous contacter", etc.) dont chaque vue correspond à une instance de la classe UITableViewController distincte.

Charger les données

Intéressons-nous à présent à la logique des contrôleurs. Dans un premier temps, ajoutez la liste des propriétés à votre projet. Maintenez la touche Ctrl enfoncée et cliquez sur le dossier Resources. Sélectionnez Add > Existing Files et sélectionnez le fichier liste_Pearson.plist créé précédemment. Cochez la case "Copy items into destination group's folder" afin de copier le fichier dans le dossier associé à votre projet. Il vous suffira de le sélectionner dans le Finder, puis de le retoucher à l'aide de l'utilitaire dédié pour mettre à jour les données qu'affiche votre application.

Nous allons charger les données qu'il contient dans un tableau de type NSArray. Pour cela, vous devez déclarer cette variable dans le contrôleur de votre vue, TableViewController.h. Vous devez également étendre cette classe aux protocoles UITableViewDelegate et UITableViewDataSource, puisque vous avez relié les deux outlets de TableView au File's Owner. Ouvrez ce fichier sous Xcode et modifiez son code à l'aide du listing ci-dessous.

Listing 7.1 : TableViewController.h

```
#import <UIKit/UIKit.h>
@interface TableViewController : UIViewController
    <UITableViewDelegate, UITableViewDataSource> {
    NSArray *donnees;
}

@property (nonatomic, retain) NSArray *donnees;

@end
```

L'implémentation de cette classe s'inscrit dans la même structure que celle des roulettes : vous chargez le contenu dès que la vue apparaît à l'écran, grâce à la méthode viewDidLoad et vous conformez la liste à travers une série de méthodes standardisées comme numberOfRowsInSection (nombre de lignes par section de la liste) et cellForRowAtIndexPathIndexPath (nom de la cellule pour chaque ligne considérée). La méthode didSelectRowAtIndexPath permettra enfin de gérer la réaction de l'application au contact de l'utilisateur. Ouvrez le fichier TableViewController.m et complétez-le à l'aide du Listing 7.2.

Listing 7.2 : *TableViewController.m*

```

#import "TableViewController.h"

@implementation TableViewController

@synthesize donnees;

/*
// The designated initializer. Override to perform setup that is required
// before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)
    nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil]) {
        // Custom initialization
    }
    return self;
}
*/
// Chargement des données de notre liste
- (void)viewDidLoad {
    NSBundle *bundle = [NSBundle mainBundle];
    NSString *plistChemin = [bundle pathForResource:@"liste_pearson"
        ofType:@"plist"];
    NSDictionary *dictionnaire = [[NSDictionary alloc] initWithContentsOfFile:
        plistChemin];
    NSArray *niveauUn = [dictionnaire allKeys];
    // Tri alphabétique du tableau
    NSArray *tri = [niveauUn sortedArrayUsingSelector:@selector(compare:)];
    self.donnees = tri;
    [dictionnaire release];
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [donnees release];
    [super dealloc];
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
    (NSInteger)section {
    return [self.donnees count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath
    Path:(NSIndexPath *)indexPath {

```

```
static NSString *cellule = @"Cellule";
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
                        cellule];

if (cell == nil) {
    cell = [[[UITableViewCell alloc]
             initWithFrame:CGRectZero reuseIdentifier: cellule] autorelease];
}
NSUInteger row = [indexPath row];
cell.text = [donnees objectAtIndex:row];
return cell;
}

@end
```

Dans la méthode `viewDidLoad`, nous créons un objet de type `NSBundle` responsable du chargement de données externes. Nous lui indiquons alors le chemin d'accès à notre fichier `.plist` contenant les données à afficher dans la liste. Dans ce premier exemple, nous extrayons les éléments parents à l'aide de la méthode `allKeys`, puis nous les trions par ordre alphabétique avant de les intégrer au tableau `donnees` défini dans l'interface de la classe contrôleur. Si vous préférez conserver l'ordre défini manuellement dans la liste des propriétés, supprimez la ligne de tri et assignez le contenu de la variable `niveauUn` au tableau `donnees`.

```
NSBundle *bundle = [NSBundle mainBundle];
NSString *plistChemin = [bundle pathForResource:@"liste_pearson"
                                                ofType:@"plist"];
NSDictionary *dictionnaire = [[NSDictionary alloc] initWithContentsOfFile:
                             :plistChemin];
NSArray *niveauUn = [dictionnaire allKeys];
// Tri alphabétique du tableau
NSArray *tri = [niveauUn sortedArrayUsingSelector:@selector(compare)];
self.donnees = tri;
```

À ce stade, les données à afficher dans la liste sont contenues dans le tableau `donnees`. Nous commençons par indiquer au contrôleur le nombre de lignes à préparer dans la liste. On y parvient à l'aide de la méthode `numberOfRowsInSection` :

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSectionSection:
(NSInteger)section {
    return [self.donnees count];
}
```

Nous devons enfin présenter le titre des cellules de la liste, qui correspondent à autant d'instances de la classe `UITableViewCell`. Pour cela, nous utilisons l'attribut `indexPath` qui permet de pointer vers une cellule spécifique. Nous parcourons ainsi toutes les cellules de la liste en remplaçant successivement leur texte par les éléments de notre tableau `donnees`.

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath
    Path:(NSIndexPath *)indexPath {
    static NSString *cellule = @"Cellule";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        cellule];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithFrame:CGRectZero reuseIdentifier: cellule]
            autorelease];
    }
    NSUInteger row = [indexPath row];
    cell.text = [donnees objectAtIndex:row];
    return cell;
}
```

Enregistrez le code et compilez le programme. L'application se lance dans le Simulateur d'iPhone et vous découvrez votre liste constituée d'éléments puisés dans le fichier .plist (voir Figure 7.4).

Figure 7.4

La liste d'éléments apparaît sur l'écran de l'iPhone et charge un contenu externe.



Mettre en forme les cellules

Comme vous l'avez vu précédemment, la déclaration du contenu des cellules n'occupe qu'une seule méthode de notre contrôleur. Il est heureusement possible d'ajouter des traitements spécifiques à chaque cellule et de modifier leur apparence du tout au tout. Nous allons ici apprendre à ajouter une image à côté de chaque entrée et à choisir un autre style typographique.

Dans un premier temps, créez une icône au format PNG. Ses dimensions importent peu : vous pourrez librement adapter la hauteur des cellules par la suite. Toutefois, privilégiez de petits pictogrammes et essayez de composer un document sur fond transparent de 35 × 35 pixels environ. Comme nous le verrons par la suite, il est possible d’adapter la hauteur des cellules de votre liste.

Lorsque votre image est prête, retournez sous Xcode, maintenez la touche Ctrl enfoncée et cliquez sur le dossier Resources. Sélectionnez Add > Existing Files et choisissez l’image que vous avez créée. Ajoutez cet élément au dossier de votre projet. Dans notre exemple il s’agit de l’image `livre.png`. Nous souhaitons la faire apparaître à gauche de chaque cellule, afin d’identifier précisément qu’il s’agit de catégories d’ouvrages. On y parvient en modifiant la méthode `cellForRowAtIndexPath`, responsable de l’affichage de chaque cellule. Complétez-la à l’aide du listing ci-dessous.

Listing 7.3 : `cellForRowAtIndexPath`

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
    NSIndexPath:(NSIndexPath)section {
    static NSString *cellule = @"Cellule";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        cellule];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithFrame:CGRectZero reuseIdentifier: cellule] autorelease];
    }
    NSUInteger row = [indexPath row];
    cell.text = [niveauUn objectAtIndex:row];
    UIImage *image = [UIImage imageNamed:@"livre.png"];
    cell.image = image;
    return cell;
}
```

Nous avons ici créé une instance de la classe `UIImage` à travers la variable `image` et nous avons chargé notre petit pictogramme. Nous affectons ensuite cette instance à la variable `cell.image` à l’aide de la notation point : on affiche ainsi l’image dans chaque cellule construite à la volée. Enregistrez et compilez le programme : le résultat ne se fait pas attendre et chaque ligne arbore désormais le pictogramme. La transparence de l’image PNG est préservée (voir Figure 7.5).

Dans le même ordre d’idée, vous avez la possibilité d’appliquer quelques règles typographiques aux titres de vos cellules. Là encore, tous les changements se répercutent sur la méthode `cellForRowAtIndexPath` et sur un attribut de la classe `UITableViewCell` : `font`. Contentez-vous d’ajouter la ligne suivante à cette méthode, à la ligne qui suit la déclaration de notre pictogramme.

Figure 7.5

Les cellules de notre liste intègrent une image personnalisée.

**Listing 7.4 : *cellForRowAtIndexPath***

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
    Path:(NSInteger)section {
    static NSString *cellule = @"Cellule";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        cellule];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithFrame:CGRectZero reuseIdentifier: cellule] autorelease];
    }
    NSUInteger row = [indexPath row];
    cell.text = [niveauUn objectAtIndex:row];
    UIImage *image = [UIImage imageNamed:@"livre.png"];
    cell.image = image;
    cell.font = [UIFont systemFontOfSize:60];
    return cell;
}
```

Enregistrez le code et compilez à nouveau le programme : les titres des cellules s'affichent désormais dans un corps de police normal, avec une taille de caractères de 60 points (voir Figure 7.6). Remplacez "systemFontOfSize" par "boldSystemFontOfSize" ou "italicSystemFontOfSize" pour obtenir un corps gras ou italique.

Figure 7.6

Nous avons modifié le style typographique du titre des cellules : elles débordent désormais de l’affichage.



Comme vous pouvez le constater, la taille choisie est trop large et le titre des cellules est tronqué. Pour y remédier, vous devez forcer votre programme à adopter une hauteur spécifique pour toutes les cellules de la liste. Là encore, une méthode spécifiquement liée à la classe `UITableView` vient à votre rescousse et vous permet d’appliquer les dimensions de votre choix. Il s’agit de la méthode `heightForRowAtIndexPath` qui, comme son nom l’indique, définit la hauteur type de chacune de vos cellules. Déclarez-la dans le code principal de votre contrôleur, juste après la fonction précédente.

Listing 7.5 : `heightForRowAtIndexPath`

```
- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:
    (NSIndexPath *)indexPath {
    return 65;
}
```

Heureuse nouvelle : la taille retournée par cette méthode est également exprimée en points. Dans la mesure où nous avons défini des titres avec une police de 60 points, veillez à laisser une petite marge confortable afin que les cellules ne se superposent pas. Vous pouvez rapidement consulter le résultat à l’aide du Simulateur d’iPhone (voir Figure 7.7).

Figure 7.7

Les cellules d'une liste peuvent librement s'étendre en hauteur et s'adapter ainsi à vos choix typographiques.



Pour définir les règles typographiques associées aux titres des cellules, nous avons créé une instance de la classe `UIFont`. Celle-ci vous permet notamment de choisir une autre police système. En la couplant à une instance de la classe `UIColor`, vous sélectionnez également la couleur des caractères. Modifiez une dernière fois la méthode `cellForRowAtIndexPath`, en appliquant les changements figurant en gras au listing ci-dessous.

Listing 7.6 : `cellForRowAtIndexPath`

```
(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath
    Path:(NSIndexPath)section {
    static NSString *cellule = @"Cellule";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        cellule];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithFrame:CGRectZero reuseIdentifier: cellule] autorelease];
    }
    NSUInteger row = [indexPath row];
    cell.text = [niveauUn objectAtIndex:row];
    UIImage *image = [UIImage imageNamed:@"livre.png"];
    cell.image = image;
    NSString *police = @"Courier";
    UIColor *couleur = [UIColor redColor];
    cell.font = [UIFont fontWithName:police size:60];
    cell.textColor = couleur;

    [image release];
    [police release];
    return cell;
}
```


Le nom des polices système s'inscrit sous la forme d'une chaîne de caractères. On commence donc par créer une instance de la classe `NSString` pour gagner en clarté, en l'initialisant avec le nom de la police "Courier". Libre à vous de modifier cet exemple par la police de votre choix ! Reportez-vous à l'adresse <http://daringfireball.net/misc/2007/07/iphone-osx-fonts> pour en découvrir la liste complète, avec des exemples précis de rendu. Vous devez scrupuleusement respecter le nom de chaque famille de police, y compris leurs déclinaisons les plus complexes (comme "Arial Rounded MT Bold" par exemple). Nous déclarons ensuite la police de nos cellules à l'aide de la méthode `fontWithName:<UIFont>size:<taille>`. Dans le même ordre d'idée, nous utilisons une couleur prédéfinie (`redColor`) pour l'appliquer au texte des cellules. Cette déclaration passe par l'attribut `textColor`. Compilez le projet pour voir le résultat dans le Simulateur d'iPhone (voir Figure 7.8).

Figure 7.8

Nous avons modifié la police, la taille et la couleur des titres de cellules.



Personnaliser vos cellules avec parcimonie

S'il paraît tentant de choisir des polices et des couleurs personnalisées pour les cellules de votre liste, procédez avec justesse et minutie. Vous n'êtes pas en train de composer un site web, sur lequel vous exprimez pleinement votre personnalité et vos goûts : vos utilisateurs devront rapidement apprivoiser votre application, sans qu'elle ne vienne heurter leurs habitudes. En choisissant une police et une couleur particulières, vous courez le risque de les rebuter et de marquer trop profondément votre différence vis-à-vis des applications concurrentes. À moins de composer l'application d'une marque, qui dispose d'ores et déjà d'une charte graphique bien établie, essayez plutôt de vous fondre dans le moule... et de vous démarquer par le caractère singulier de vos contrôleurs ! Si le style de votre liste tranche radicalement avec les habitudes des utilisateurs, vous devez le justifier en déclinant à fond cette charte graphique à tous les éléments de l'application.

Réagir au contact de l'utilisateur

Si les longues listes constituent en soi un élément prépondérant de votre application, en soutenant des informations très riches que vous compulsez dans un fichier externe, elles ne révèlent tout leur intérêt qu'en se montrant interactives. À l'instar de n'importe quel contrôle du kit de développement de l'iPhone, elles autorisent en effet des manipulations de la part de l'utilisateur, qui déclenchent alors des traitements secondaires. L'exemple typique de la liste de contacts est parfaitement adapté : en touchant l'un des noms figurant dans votre carnet, vous déclenchez automatiquement un appel téléphonique.

Votre application peut réagir de la même manière et générer une multitude de sous traitements. Affichage d'une seconde liste, basculement vers une vue auxiliaire, apparition d'une alerte... Toutes les idées sont possibles ! Votre liste devient alors un système de navigation à part entière. En prenant soin de préparer au mieux cet aspect de votre application, vous inventez une nouvelle manière de la mettre à jour : il vous suffira de modifier la liste des propriétés (ou plus généralement le fichier externe chargé par votre programme) pour offrir de nouvelles interactions à vos utilisateurs. Comme pour les roulettes, la gestion des interactions passe par la méthode `didSelectRowAtIndexPath`. Cette méthode récupère précisément le numéro de la cellule pointée par l'utilisateur : souple et pratique, elle vous permet d'envisager rapidement des traitements personnalisés pour toutes les entrées de vos listes.

Modifiez à nouveau le fichier `TableViewController.m` en intégrant cette nouvelle méthode à la fin du code source, comme au Listing 7.7. Dans notre exemple, nous restons dans un cadre simple et borné : la pression sur une cellule de la liste affiche une alerte, qui correspond à une instance de la classe `UIAlertView`.

Listing 7.7 : *didSelectRowAtIndexPath*

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
    (NSIndexPath *)indexPath {
    NSInteger cellule = [indexPath row];
    NSString *valeurCellule = [niveauUn objectAtIndex:cellule];
    NSString *message = [[NSString alloc] initWithFormat: @"Vous avez
        choisi le rayon %@", valeurCellule];
    UIAlertView *alerte = [[UIAlertView alloc]
        initWithTitle:@"Choix d'un rayon"
        message:message
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];

    [alerte show];
    [message release];
    [alerte release];
}
```

Enregistrez le code source et compilez à nouveau votre application. La moindre pression sur une cellule de la liste fait jaillir un message d’alerte à l’utilisateur, en reprenant le titre du rayon sélectionné (voir Figure 7.9). Vous pouvez facilement adapter cet exemple à vos propres besoins, en intégrant d’autres traitements spécifiques. La première instruction de cette méthode récupère le numéro de la cellule pointée :

```
NSInteger cellule = [indexPath row];
```

Figure 7.9

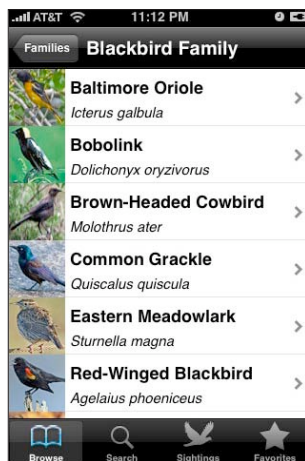
Nous avons sélectionné une cellule et une alerte apparaît automatiquement.



N’hésitez pas à vous reporter aux exemples du Chapitre 5 afin de vous remémorer le chargement d’une vue secondaire. En préparant un second contrôleur associé à une nouvelle vue contenant elle-même une liste, vous définissez ainsi un niveau hiérarchique. L’élément parent pointé dans la première vue charge une seconde liste constituée de ses éléments enfants. Il est nécessaire de définir cette seconde vue dans un contrôleur distinct, et non de recharger intégralement le contenu de la liste principale, afin d’offrir un bouton de navigation aux utilisateurs. Ces derniers pourront ainsi cliquer sur l’un des titres pour consulter les éléments sous-jacents, avant de revenir à la liste globale et repartir éventuellement dans une autre direction. Ce second contrôleur obéit strictement au même principe que notre exemple : on parcourt à nouveau le fichier `liste_Pearson.plist` en partant cette fois de l’élément sélectionné en guise de point de départ. On charge alors dynamiquement un tableau contenant de nouvelles valeurs, que l’on dispose dans une liste à l’aide de la méthode `cellForRowAtIndexPath` (voir Figure 7.10).

Figure 7.10

Ce dictionnaire des oiseaux s'appuie sur une série de listes et de fichiers externes pour charger son contenu.



Créer un lecteur de flux RSS

Jusqu'à présent, nous avons chargé du contenu externe à notre application sous la forme d'une liste de propriétés. Facile à mettre à jour, cette base est toutefois créée et stockée sur votre propre poste local. Il y a fort à parier qu'en marge de votre passion grandissante pour l'iPhone, vous disposiez d'un espace personnel sur la Toile. Votre site web ou votre blog s'articule très probablement autour d'un flux RSS généré automatiquement ou préparé par vos soins à l'aide d'un script PHP de votre crû. Dans tous les cas, vous connaissez la souplesse et la polyvalence d'un tel format : il est interprété par une grande quantité de logiciels, de nos navigateurs web aux petits gadgets du dock.

Proposer un lecteur de flux RSS avec l'adresse de votre blog en guise d'application iPhone peut ainsi largement contribuer à élargir votre audience. En diffusant gratuitement cette application, vous attisez inévitablement la curiosité de tous les férus de l'App Store : ils pourront ainsi consulter vos derniers billets dans le creux de leur mobile et vous n'aurez plus réellement à vous préoccuper des mises à jour de votre application. En effet, les changements apportés à votre blog ou votre site web se répercuteront automatiquement sur votre application iPhone, qui charge directement votre flux RSS (voir Figure 7.11). Bien entendu, c'est une instance de la classe `UITableView` qui soutiendra le titre des différents billets !

Le kit de développement de l'iPhone supporte de nombreux parseurs XML concurrents. Nous ne participerons pas à cette querelle de chapelles, qui vise à en élire un parmi tant d'autres. L'une des bibliothèques les plus utilisées actuellement est *TouchXML*. Elle se base sur le projet open-source *libxml2* et constitue une solution simple et rapide pour lire des fichiers XML. Sachez par ailleurs que Cocoa Touch dispose d'une classe `NSXMLParser` dont

vous pourrez prendre connaissance à travers la documentation officielle de Xcode. Saisissez ce terme dans l'éditeur, puis pressez la touche Alt et sélectionnez-le pour découvrir son implémentation. Nous lui préférons TouchXML dans le cadre de notre exemple pour sa plus grande souplesse et sa facilité d'utilisation.

Figure 7.11

De nombreux sites web célèbres disposent désormais d'une application pour iPhone : il s'agit généralement d'un lecteur de flux RSS préconfiguré.



Intégrer la bibliothèque TouchXML

Avant d'aller plus loin, vous devez tout d'abord vous procurer la dernière version de la bibliothèque TouchXML. On doit ce projet à l'équipe TouchCode, qui dispose d'un espace collaboratif à l'adresse <http://code.google.com/p/touchcode>. Le projet bénéficie d'un suivi professionnel et une large communauté suit son évolution à travers le Wiki officiel. Pour télécharger la bibliothèque, rendez-vous sur la page officielle du projet puis cliquez sur l'onglet Downloads. Sélectionnez TouchXML dans la liste et décompressez l'archive dans un dossier de votre disque dur, au sein de Deve1oper par exemple (voir Figure 7.12).

Ouvrez à nouveau Xcode et créez un nouveau projet en déroulant le menu File, puis en cliquant sur New Project. Choisissez le modèle Navigation-Based Application et cliquez sur le bouton Choose. Saisissez LecteurRSS en guise de nom de projet. Vous en avez désormais largement l'habitude : tous les fichiers nécessaires sont prêts à l'emploi et votre application s'articule d'ores et déjà autour d'une liste... vide. En revanche, nous n'avons pas encore expérimenté l'ajout d'une bibliothèque complémentaire, venant élargir les fonctions du kit de développement de l'iPhone. Déroulez le menu Project et cliquez sur l'option Edit Project Settings (voir Figure 7.13). Une fenêtre comprenant quatre onglets apparaît à l'écran. Cliquez sur l'onglet Build.

Figure 7.12

On télécharge la dernière version de la bibliothèque TouchXML.

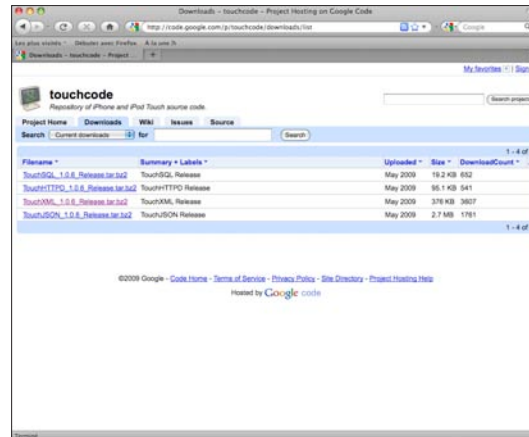
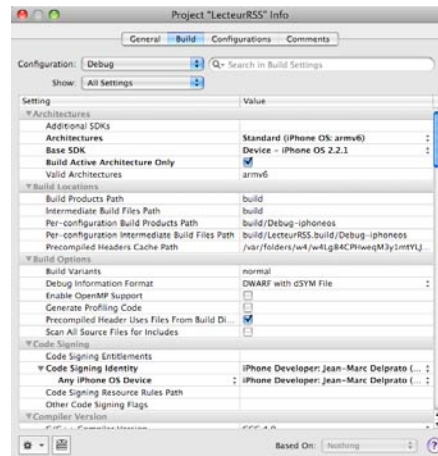


Figure 7.13

La fenêtre des propriétés du projet.



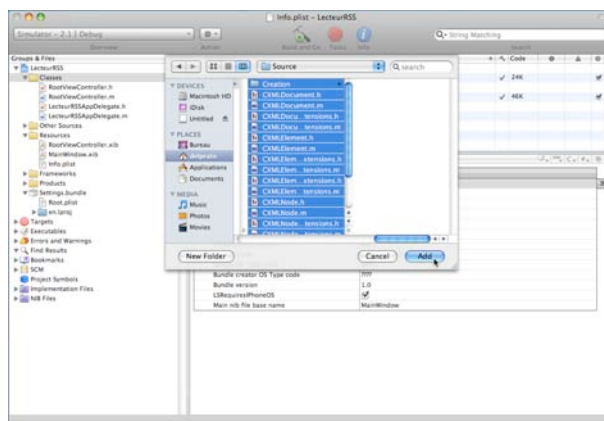
Dans le champ de recherche en haut à droite de la fenêtre, saisissez Header Search Paths. Vous filtrez les éléments : effectuez un double-clic sur la seule ligne apparaissant à l'écran, cliquez sur le bouton + et saisissez /usr/include/libxml2 dans le champ Path. Cliquez ensuite sur OK. Dans le champ de recherche, saisissez à présent Other Linker Flags et sélectionnez la ligne. Ajoutez -lxml2 dans le champ considéré. Vous venez de charger les bibliothèques libxml2 de Cocoa sur lesquelles s'appuie TouchXML. Fermez la fenêtre des propriétés de votre projet.

Finalisez cette opération initiale en ajoutant les classes de TouchXML que vous venez de télécharger. Pour cela, pressez la touche Ctrl de votre clavier et cliquez sur le dossier Classes de votre projet. Sélectionnez Add > Existing Files dans le menu contextuel et pointez l'explorateur vers le dossier contenant la bibliothèque TouchXML. Choisissez plus précisément

le répertoire TouchXML/Common/Source, sélectionnez l'ensemble des fichiers (Cmd+A) sauf le dossier Tidy et validez en cliquant sur Add (voir Figure 7.14). Conservez les paramètres par défaut et cliquez à nouveau sur Add.

Figure 7.14

L'ajout de la bibliothèque TouchXML à notre projet.



Sur le volet gauche de votre projet, dans le dossier Classes, vous découvrez une vingtaine de fichiers supplémentaires, dont TouchXML.h et une série de bibliothèques commençant par CXML. Votre application peut immédiatement les mettre à profit pour parcourir un flux RSS !

Implémenter le parseur XML

Dans le cadre de notre exemple, nous souhaitons intégrer le flux RSS typique d'un blog courant, hébergé sur le service FeedBurner par exemple. Chaque billet est identifié par le nœud <item>...</item> et dispose d'une série d'éléments enfants : title, link, description, pubDate, etc. L'implémentation de la classe TouchXML correspond ainsi aux étapes suivantes :

1. On charge le flux depuis son URL.
2. On crée une instance de la classe CXMLElement, visant à soutenir un document XML complet, que l'on initialise avec le contenu du flux téléchargé.
3. On crée un tableau de type NSArray : chacun de ces éléments correspond à un nœud <item> du flux.
4. On crée un dictionnaire de type NSMutableDictionary dans lequel on stocke tous les éléments de chaque nœud. Le nom des éléments correspond aux clés du dictionnaire. Ainsi, on pourra identifier la date de publication, le titre ou la description de chaque billet figurant dans le flux RSS.

5. On doit prévoir un tableau global pour manipuler aisément le contenu du flux RSS à travers toutes les méthodes du contrôleur de la vue. Nous serons ainsi en mesure d'afficher des informations supplémentaires, comme la description d'un billet par exemple, lorsque l'utilisateur sélectionnera un titre.

Commencez par ouvrir le fichier `RootViewController.h` sous Xcode et ajoutez les éléments figurant au Listing 7.8.

Listing 7.8 : *RootViewController.h*

```
#import <UIKit/UIKit.h>
#import "TouchXML.h"

@interface RootViewController : UITableViewController {
    IBOutlet UITableView *listeBillets;
    NSMutableArray *billetsFlux;
}

@property(retain, nonatomic) NSMutableArray *billetsFlux;
@end
```

Vous noterez qu'on importe la bibliothèque TouchXML associée à notre projet puis qu'on définit un outlet correspondant à l'objet `UITableView`. Nous ne devons pas oublier de l'associer au File's Owner, sous Interface Builder. Nous déclarons ensuite notre tableau global, `billetsFlux`.

Saisissez à présent le code suivant dans l'implémentation du contrôleur :

Listing 7.9 : *RootViewController.m*

```
#import "RootViewController.h"
#import "LecteurRSSAppDelegate.h"

@implementation RootViewController
@synthesize billetsFlux;

- (void)viewDidLoad {
    [super viewDidLoad];

    // On indique l'URL du flux
    NSString *adresseFlux = @"http://feeds2.feedburner.com/BilletsCocoafr";

    // On initialise le NSMutableArray qui contiendra tous les billets
    billetsFlux = [[NSMutableArray alloc] init];

    // On crée un objet NSURL pour accéder à l'adresse du flux
    NSURL *url = [NSURL URLWithString: adresseFlux];
```



```
// On crée notre parseur à l'aide de la classe CMXDocument de
// TouchXML
CMXMLDocument *parseurXML = [[[CMXMLDocument alloc]
    initWithContentsOfURL:url options:0 error:nil] autorelease];

// On crée un tableau contenant tous les billets du flux
NSArray *listeItems = NULL;

// On remplit le tableau en identifiant les noeuds "<item>" du flux RSS
listeItems = [parseurXML nodesForXPath:@"//item" error:nil];

// On accède aux données de chaque élément individuel
for (CXMLElement *billetIndividuel in listeItems) {

    // On stocke temporairement les éléments de chaque billet
    NSMutableDictionary *elementBlog = [[NSMutableDictionary alloc]
        init];

    int compteur;

    // On parcourt tous les éléments du noeud <item> courant
    for(compteur = 0; compteur < [billetIndividuel childCount];
        compteur++) {

        // On ajoute le nom et la valeur de chaque élément au
        // dictionnaire elementBlog
        [elementBlog setObject:[billetIndividuel
            childAtIndex:compteur] stringValue] forKey:[billetIndividuel
            childAtIndex:compteur] name];
    }

    // On ajoute le contenu de chaque noeud au tableau global
    [billetsFlux addObject:[elementBlog copy]];
}

[adresseFlux release];
[url release];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

// Nombre de cellules de la liste
```

```

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
    (NSInteger)section {
    return [billetsFlux count];
}

// On modifie l'apparence des cellules
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath
    Path:(NSIndexPath *)indexPath {

    static NSString *cellule = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        cellule];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithFrame:CGRectZero reuseIdentifier:cellule] autorelease];
    }

    int indexBillet = [indexPath indexAtPosition: [indexPath length] -1];
    // On affiche le titre de chaque billet
    [cell setText:[billetsFlux objectAtIndex: indexBillet] objectForKey:
        @"title"]];
    return cell;
}

- (void)dealloc {
    [super dealloc];
    [billetsFlux release];
}

@end

```

Ouvrez le fichier `RootViewController.xib`, sélectionnez `File's Owner` en pressant la touche `Ctrl` de votre clavier et pointez la flèche vers `Table View`. Sélectionnez `listeBillets` dans le menu contextuel qui apparaît. Enregistrez votre interface et votre code, puis compilez le programme (voir Figure 7.15).

Et si nous mettions à profit l'exemple précédent en réagissant aux manipulations de l'utilisateur ? Une fois encore, tout passe par la méthode `didSelectRowAtIndexPath`. Dans la mesure où nous disposons de tout le contenu du flux dans un tableau global, `billetsFlux`, nous pouvons librement en extraire une information précise. Il s'agit ici d'afficher une alerte contenant le texte du billet. En une pression sur un bouton, on revient à la liste complète des billets. Ajoutez la méthode suivante au fichier `RootViewController.m`.

Figure 7.15

*Le lecteur de flux RSS
affiche les derniers
billets du blog choisi.*

**Listing 7.10 : didSelectRowAtIndexPath**

```
// On gère la sélection d'un billet
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
    (NSIndexPath *)indexPath {
    NSUInteger indexBillet = [indexPath indexAtPosition:
        [indexPath length] - 1];
    NSString *description = [[billetsFlux objectAtIndex: indexBillet]
        objectForKey: @"description"];

    NSString *message = [[NSString alloc] initWithFormat: @"%@",
        description];
    UIAlertView *alerte = [[UIAlertView alloc]
        initWithTitle:@"Lecture du billet"
        message:message
        delegate:nil
        cancelButtonTitle:@"Revenir à la liste"
        otherButtonTitles:nil];

    [alerte show];
    [message release];
    [alerte release];
}
```

Les fenêtres d'alerte se manipulent elles-mêmes du bout des doigts : quelle que soit la longueur de la chaîne de caractères ainsi récupérée dans la variable `description`, l'utilisateur pourra lire le texte en entier. Il suffit de cliquer sur le bouton d'annulation, arborant le libellé "Revenir à la liste" pour masquer la fenêtre (voir Figure 7.16). Simple et efficace !

Adapter cet exemple à votre propre blog

Vous avez probablement envie de personnaliser cet exemple autour de votre propre blog. N'hésitez pas à ajouter un titre, voire une série d'onglets pour accéder à des fonctions complémentaires. Si la liste reste désespérément vide, votre blog utilise probablement un autre format de flux. Plusieurs standards cohabitent en effet sur le marché et ils n'identifient pas tous les billets à l'aide de nœuds `<item>`, même si la norme RSS, qui est la plus courante, le prévoit. Vérifiez la structure de votre propre flux

Par ailleurs, vous pouvez adapter cet exemple à bien d'autres projets : si vous êtes développeur web, n'hésitez pas à créer des scripts PHP qui génèrent quantité de flux XML personnalisés. En modifiant la propriété de la méthode `nodesForXPath`, vous pouvez rapidement rapatrier d'autres types d'informations, que vous encapsulez selon l'arborescence de votre choix. On peut ainsi imaginer un guide de restaurants, dont les éléments sont isolés dans des nœuds `<restos>...</restos>` par exemple. En passant à votre script PHP des paramètres dans l'URL, vous pouvez ainsi facilement extraire des enregistrements d'une base MySQL et les conformer en XML. Ajoutez des champs de texte ou des contrôles à votre interface afin de laisser l'utilisateur choisir la fonction qu'il souhaite appeler : l'URL soumise au parseur de TouchXML s'adapte ainsi en conséquence.

Figure 7.16

En sélectionnant l'un des billets, on affiche son contenu dans une boîte d'alerte.



Enregistrer des données

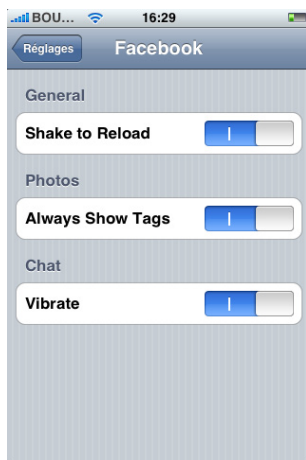
À l'origine, les premiers développeurs pour iPhone ont essentiellement eu pour vocation de présenter des informations et du contenu à l'utilisateur, afin de lui prêter immédiatement assistance face à un besoin spécifique. Songez aux premières applications que vous avez installées : l'App Store était constellé de purs utilitaires, comme des lampes de poche ou

des niveaux à bulles par exemple. Très rapidement, une seconde génération d'utilitaires a vu le jour et a proposé une expérience plus poussée ; l'utilisateur était invité à saisir les informations qu'il devait conserver au fil du temps. C'est notamment le cas des gestionnaires de budget ou des pense-bêtes.

Que vous sollicitiez largement vos utilisateurs ou non, vous devez impérativement vous intéresser aux paramètres externes de votre application. Votre iPhone en main, parcourez l'écran des Réglages. Très vraisemblablement, vous y verrez des entrées dédiées à des applications que vous avez installées, comme Facebook, Finetune, LinkedIn ou Shazam par exemple (voir Figure 7.17). Tout comme Mac OS X, iPhone OS centralise les "Préférences Système" dans un espace du mobile et laisse tous les développeurs y consigner les paramètres liés à leurs applications. Grâce à la classe `NSUserDefaults`, vous chargez ces préférences lors de l'exécution de votre programme. Vous avez ainsi la possibilité d'enregistrer les réglages personnalisés de l'utilisateur et de lui offrir une plus grande souplesse.

Figure 7.17

L'écran des Préférences Système de l'application Facebook.



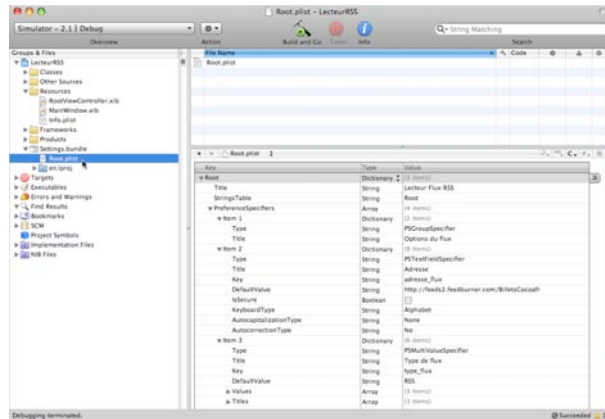
Sauvegarder les Préférences Système

Le kit de développement de l'iPhone facilite réellement la tâche des développeurs qui souhaiteraient s'inscrire dans un tel système. Les Préférences Système sont en effet définies dans un bloc spécifique (on parle de *bundle de préférences*) qui contient au moins une liste de propriétés, `Root.plist`. Celui-ci décrit une série de réglages proposés à l'utilisateur, qui prennent la forme d'un champ de texte à remplir, d'une réglette à déplacer ou d'un élément de liste à sélectionner par exemple. Tous ces champs et contrôles sont réunis dans une vue unique, située dans l'écran des Réglages de l'iPhone.

Conservez le projet `LecteurRSS.xcodeproj` ouvert. Sous Xcode, déroulez le menu `File > New File`. Dans le volet gauche de la fenêtre, sélectionnez `Settings` sous la section `iPhone OS`, puis choisissez `Settings Bundle` et validez en cliquant sur le bouton `Next`, puis `Finish`. Sur le volet `Groups & Files` de Xcode, déroulez l'élément `Settings.bundle` et sélectionnez `Root.plist`. Vous accédez à la liste des propriétés des préférences de votre application (voir Figure 7.18).

Figure 7.18

On crée la liste des Préférences Système de notre lecteur de flux RSS.



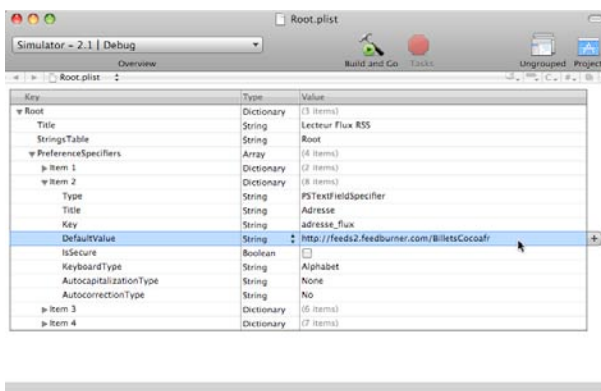
Par défaut, la liste des propriétés comprend quatre éléments d'exemple. Commencez par effectuer un double-clic en face du champ `Title` et saisissez le nom de votre application. Il s'agit du titre qui apparaît à l'écran des Réglages de l'iPhone. Vous découvrez ensuite un nœud `PreferenceSpecifiers` : il s'agit des différents types de contrôles permettant à l'utilisateur de définir ses préférences. Le premier élément correspond au nom du groupe des préférences : son type est `PSGroupSpecifier`. Saisissez "Options du flux" dans le champ `Title`. Les trois autres éléments présentent l'intégration de contrôles spécifiques : un champ de texte (`PSTextFieldSpecifier`), un interrupteur pour choisir entre deux valeurs (`PSToggleSwitchSpecifier`) et une réglette (`PSSliderSpecifier`). Vous êtes parfaitement libre d'ajouter les éléments de votre choix, en réduisant l'arborescence d'un nœud et en cliquant sur le bouton `+` figurant à la fin de chaque ligne. Vous avez également la possibilité de copier/coller un élément pour le dupliquer rapidement.

Dans le cadre de notre exemple, nous souhaitons essentiellement laisser l'utilisateur saisir son adresse de flux RSS préféré. Nous lui soumettons ainsi un champ de texte, dont la valeur contient par défaut l'adresse du blog précédent. S'il ne touche pas aux Préférences Système, il ne se rendra même pas compte de l'existence d'une telle option et le lecteur de flux affichera les billets prévus par le développeur. Conservez le deuxième élément et adaptez le champ `Title` et `Key`. Ce dernier définit la constante qui sera reprise par le contrôleur de votre programme. Dans le champ `DefaultValue`, saisissez l'adresse par défaut du lecteur

RSS (voir Figure 7.19). Vous pouvez également choisir le type de clavier virtuel et activer la correction automatique en cours de saisie. Vérifiez à chaque fois le type d'objet attendu : nous utilisons ici des chaînes de caractères.

Figure 7.19

Nous offrons aux utilisateurs la possibilité de saisir leur propre adresse de flux.



Poursuivez éventuellement avec un second champ de texte, visant à définir le nombre de billets à afficher par exemple. Vous pouvez aussi adapter la règle à ce type de valeurs. Enregistrez le fichier `Root.plist` et compilez le projet. Cliquez sur le bouton central du Simulateur d'iPhone et parcourez l'écran Réglages. Vous découvrirez la section consacrée à votre application, avec les contrôles que vous avez préparés. Les valeurs par défaut y figurent et un clavier virtuel apparaît lorsque vous cliquez sur un champ de texte (voir Figure 7.20).

Figure 7.20

Les contrôles définis apparaissent à l'écran Réglages.



Vous pouvez très facilement enregistrer tout type de valeur. Peu importe l'ordre dans lequel vous avez défini les éléments du fichier `Root.plist` : vous définissez vous-même leur rôle au sein de votre application. Essayez au passage de modifier la liste de préférences et d'ajouter un élément de type `PSMultiValueSpecifier` : il s'agit d'une liste à choix multiple. Là encore, définissez un champ `Title`, `Key` et `DefaultValue`, mais ajoutez également deux objets de type `Array`. Vous les nommerez respectivement `Titles` et `Values` et ils correspondront aux choix et aux valeurs que l'utilisateur peut sélectionner (voir Figures 7.21 et 7.22). Dans le cadre de notre application, nous pouvons laisser l'utilisateur choisir le type de flux XML considéré (ATOM, RSS, RDF, etc.).

Figure 7.21

On modifie la liste des préférences pour ajouter un choix supplémentaire, parmi plusieurs valeurs prédéfinies.

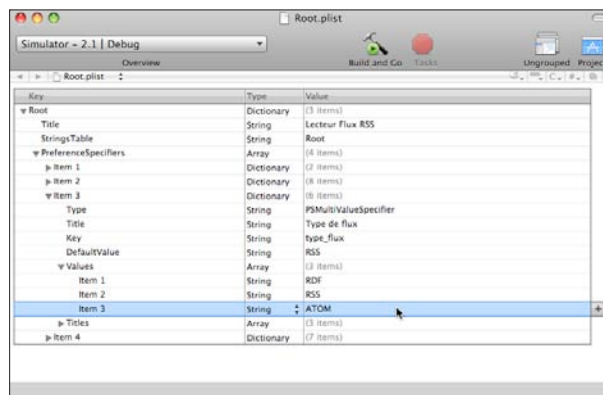


Figure 7.22

Ce contrôle se traduit à l'écran par une liste sélectionnable.



Il est maintenant l'heure d'adapter le contrôleur de notre vue. Ouvrez le fichier `RootViewController.h` et ajoutez les lignes suivantes au début du code :


```
#define kAdresseFlux @"adresse_flux"  
#define kNombreBillets @"nbr_billets"  
#define kTypeFlux @"type_flux"
```

Il s'agit ici de trois constantes, que l'on associe aux valeurs du champ Key de chaque préférence système.

Ouvrez à présent l'implémentation `RootViewController.m` et adaptez la méthode `viewDidLoad`.

Listing 7.11 : `viewDidLoad`

```
- (void)viewDidLoad {  
    [super viewDidLoad];  
  
    // On charge les Préférences Système  
    NSUserDefaults *preferences = [NSUserDefaults standardUserDefaults];  
    NSString *adresseFlux = [NSString alloc];  
    adresseFlux = [preferences objectForKey:kAdresseFlux];  
  
    [...]  
}
```

On crée ici une instance de la classe `NSUserDefaults` et on charge les Préférences Système. On récupère ensuite la valeur associée à la constante `kAdresseFlux`, c'est-à-dire à la clé `adresse_flux` de la liste des propriétés. Il n'y a plus la moindre mention d'une URL au sein du contrôleur et pourtant l'application affiche bien un flux RSS lorsque vous la compilez à nouveau (voir Figure 7.23).

Figure 7.23

On utilise les Préférences Système pour charger dynamiquement un nouveau flux RSS.



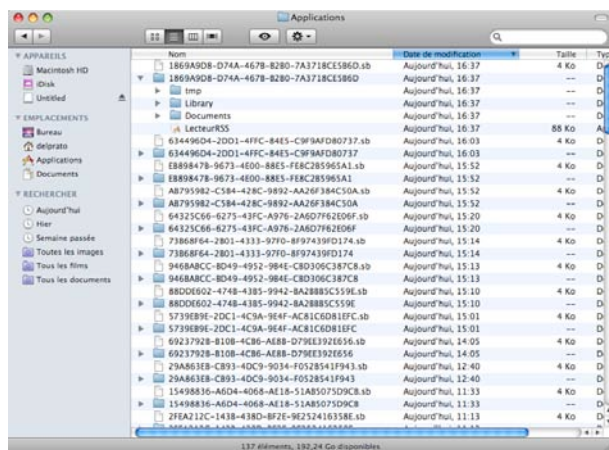
Vous pouvez librement adapter ce principe à tous vos besoins et limiter ainsi le nombre de billets affichés ou choisir un autre paramètre à passer à la méthode `nodesForXPath`, en fonction du type de flux. Même après avoir fermé le Simulateur d'iPhone, vous retrouvez le paramètre que vous avez défini manuellement en lançant à nouveau votre application. Vous avez également la possibilité d'intégrer des champs de texte à votre vue principale, dont les valeurs de base correspondent à ces Préférences Système. L'utilisateur pourra ainsi adapter ces valeurs aussi bien dans l'écran Réglages de l'iPhone qu'au sein de votre application. Vous connaissez la marche à suivre : les champs de texte doivent correspondre à des outlets que vous reliez au File's Owner du contrôleur de la vue.

Écrire un fichier

Lorsque votre application doit lire et enregistrer des données qui ne relèvent pas des Préférences Système, l'une des solutions les plus simples consiste à écrire un fichier. Le Simulateur de l'iPhone rend compte de l'organisation des fichiers sur le mobile d'Apple. Ouvrez une fenêtre du Finder, accédez à la racine de votre compte puis ouvrez Bibliothèque/Application Support/iPhone Simulator/User. Vous découvrez ainsi l'agencement exact des dossiers et des fichiers sur iPhone OS. Parmi les cinq dossiers présents par défaut, vous remarquez le répertoire Applications. Il contient une série de fichiers `.sb` et de sous-répertoires portant le même nom (voir Figure 7.24).

Figure 7.24

Le Simulateur reproduit l'arborescence des fichiers de l'iPhone.



L'extension `.sb` correspond aux "sandbox" (littéralement, des bacs à sable) allouées à chaque application. C'est une zone qu'elles peuvent librement agencer et utiliser pour stocker leurs propres données. Afin d'éviter les conflits, les noms des dossiers sont uniques et sont générés par le système. Vous pouvez heureusement récupérer le chemin d'accès complet à la sandbox d'une application à l'aide de la méthode `NSSearchPathForDirectories-`

InDomains, comme nous allons le voir par la suite. Si votre application doit stocker temporairement le résultat de traitements, vous profitez d'un accès plus rapide encore au dossier /tmp qui lui est dédié, à travers la méthode NSTemporaryDirectory().

La solution la plus simple et efficace pour écrire un fichier consiste à *sérialiser* une classe, c'est-à-dire à convertir l'ensemble de ses éléments en octets et à les enregistrer dans une liste de propriété. Certes, tous les types de classes ne sont pas supportées, mais le panel est suffisamment large et vous pouvez ainsi enregistrer le contenu de variables définies en tant que NSString, NSDate, NSNumber, NSArray, NSDictionary, NSMutableArray et NSMutableDictionary. L'opération s'effectue en une seule ligne :

```
[<classe instanciée> writeToFile:<chemin_accès> atomically:YES];
```

Bien entendu, n'oubliez jamais que toutes les fonctions traditionnelles du langage C sont également accessibles, comme fwrite() ou fopen(). Ces solutions dépassent toutefois le simple giron de notre ouvrage et nous vous invitons à vous reporter au livre *Le langage C++* de Jesse Liberty et Bradley Jones, paru aux éditions Pearson, pour en savoir davantage.

Dans notre ultime exemple, nous allons modifier le lecteur de flux RSS en lui ajoutant deux fonctions :

- Un bouton "Supprimer", en haut à droite, permettra de réduire à loisir les billets affichés par le lecteur. Vous "personnalisez" ainsi son affichage en effaçant les articles qui vous intéressent le moins.
- Un bouton "Enregistrer", en haut à gauche, va alors sauvegarder la sélection de l'utilisateur sous la forme d'un fichier .plist.

Ce fichier pourra ainsi être archivé ou au contraire rechargé afin de faire office de cache.

Commencez par modifier le fichier RootViewController.h. On y définit une constante supplémentaire (le nom du fichier à sauvegarder) et on déclare deux méthodes : une pour récupérer le chemin d'accès au dossier Documents associé à notre application et une seconde pour supprimer l'un des billets affichés à l'écran.

Listing 7.12 : RootViewController.h

```
#define kAdresseFlux          @"adresse_flux"
#define kNomFichier           @"selection_blog.plist"

#import <UIKit/UIKit.h>
#import "TouchXML.h"

@interface RootViewController : UITableViewController {
    IBOutlet UITableView *listeBillets;
    NSMutableArray *billetsFlux;
```

```

    }

    @property(retain, nonatomic) NSMutableArray *billetsFlux;
    -(IBAction)supprimerBillet:(id)sender;
    -(NSString *)cheminAcces;

    @end

```

Modifiez ensuite le fichier `RootViewController.m` comme suit :

Listing 7.13 : *RootViewController.m*

```

#import "RootViewController.h"
#import "LecteurRSSAppDelegate.h"

@implementation RootViewController
@synthesize billetsFlux;

- (NSString *)cheminAcces {
    NSArray *chemins = NSSearchPathForDirectoriesInDomains
        (NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *dossierDocuments = [chemins objectAtIndex:0];
    return [dossierDocuments stringByAppendingPathComponent:kNomFichier];
}

-(IBAction)supprimerBillet:(id)sender {
    [self.tableView setEditing:!self.tableView.editing animated:YES];
}

-(void)enregFichier:(id)sender {
    [billetsFlux writeToFile:[self cheminAcces] atomically:YES];
}

- (void)viewDidLoad {
    [super viewDidLoad];

    // On indique l'URL du flux

    NSUserDefaults *preferences = [NSUserDefaults standardUserDefaults];
    NSString *adresseFlux = [NSString alloc];

    adresseFlux = [preferences objectForKey:kAdresseFlux];

    // On initialise le NSMutableArray contenant tous les billets
    billetsFlux = [[NSMutableArray alloc] init];

    // On crée un objet NSURL pour accéder à l'adresse du flux
    NSURL *url = [NSURL URLWithString: adresseFlux];

    // On crée notre parseur à l'aide de la classe CMXDocument de
    // TouchXML
    CXMLElement *parseurXML = [[[CXMLElement alloc]

```

```
initWithContentsOfURL:url options:0 error:nil] autorelease];

// On crée un tableau contenant tous les billets du flux
NSArray *listeItems = NULL;

// On remplit le tableau en identifiant les noeuds "<item>" du flux
// RSS
listeItems = [parseurXML nodesForXPath:@"//item" error:nil];

// On accède aux données de chaque élément individuel
for (CXMLElement *billetIndividuel in listeItems) {

    // On stocke temporairement les éléments de chaque billet
    NSMutableDictionary *elementBlog = [[NSMutableDictionary alloc]
                                         init];

    int compteur;

    // On parcourt tous les éléments du noeud <item> courant
    for(compteur = 0; compteur < [billetIndividuel childCount];
        compteur++) {

        // On ajoute le nom et la valeur de chaque élément au
        // dictionnaire elementBlog
        [elementBlog setObject:[billetIndividuel
                                childAtIndex:compteur] stringValue]
        forKey:[billetIndividuel childAtIndex:compteur] name];
    }

    // On ajoute le contenu de chaque noeud au tableau global
    [billetsFlux addObject:[elementBlog copy]];
}

UIBarButtonItem *supprBouton = [[UIBarButtonItem alloc]
                                initWithTitle:@"Supprimer"
                                style:UIBarButtonItemStyleBordered
                                target:self
                                action:@selector(supprimerBillet:)];
self.navigationItem.rightBarButtonItem = supprBouton;

UIBarButtonItem *enregBouton = [[UIBarButtonItem alloc]
                                initWithTitle:@"Enregistrer"
                                style:UIBarButtonItemStyleBordered
                                target:self
                                action:@selector(enregFichier:)];
self.navigationItem.leftBarButtonItem = enregBouton;
[enregBouton release];
[supprBouton release];

[adresseFlux release];
[url release];
}
```

```

/*
// Override to allow orientations other than the default portrait
// orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
*/

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

// Nombre de cellules de la liste
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
    (NSInteger)section {
    return [billetsFlux count];
}

// On modifie l'apparence des cellules
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath
    Path:(NSIndexPath *)indexPath {

    static NSString *cellule = @"DeleteMeCellIdentifier";
    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:
        cellule];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithFrame:CGRectZero reuseIdentifier:cellule] autorelease];
    }

    int indexBillet = [indexPath indexAtPosition: [indexPath length] -1];
    // On affiche le titre de chaque billet
    [cell setText:[billetsFlux objectAtIndex: indexBillet] objectForKey:
        @"title"]];

    return cell;
}

- (void) tableView:(UITableView *) tableView commitEditingStyle:
    (UITableViewCellEditingStyle) editingStyle
    forRowAtIndexPath:(NSIndexPath *) indexPath {
    NSUInteger row = [indexPath row];
    [self.billetsFlux removeObjectAtIndex:row];
    [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
        withRowAnimation:

```

```

        UITableViewRowAnimationFade];
    }

    // On gère la sélection d'un billet
    - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:
        (NSIndexPath *)indexPath {
        NSUInteger indexBillet = [indexPath indexAtPosition: [indexPath
                                                                length] - 1];
        NSString *description = [[billetsFlux objectAtIndex: indexBillet]
                                objectForKey: @"description"];
        NSString *message = [[NSString alloc] initWithFormat: @"%@",
                                                                description];
        UIAlertView *alerte = [[UIAlertView alloc]
                                initWithTitle:@"Lecture du billet"
                                message:message
                                delegate:nil
                                cancelButtonTitle:@"Revenir à la liste"
                                otherButtonTitles:nil];

        [alerte show];
        [message release];
        [alerte release];
    }

    - (void)dealloc {
        [super dealloc];
        [billetsFlux release];
    }

@end

```

Enregistrez le projet et compilez-le : vous avez désormais la possibilité de personnaliser la liste en cours d'affichage en cliquant sur le bouton Supprimer, puis en sélectionnant l'une des cellules (voir Figures 7.25 et 7.26).

Figure 7.25

On commence par charger le flux à l'aide d'une adresse spécifique, indiquée dans les Préférences Système.



C'est la méthode `commitEditingStyle` qui en est responsable, en rechargeant dynamiquement le contenu du tableau. Le second bouton, Enregistrer, lance la procédure de sauvegarde (voir Figure 7.27). On commence tout d'abord par repérer le chemin d'accès à la sandbox de l'application, puis on enregistre le contenu du tableau `billetsFlux` sous la forme d'un fichier `.plist`.

Figure 7.26

On supprime ensuite tous les billets qui ne nous intéressent pas.



Figure 7.27

On enregistre enfin le contenu du flux sélectionné sous la forme d'une liste de propriétés.



Pour aller plus loin

La boucle est désormais bouclée : vous avez appris à lire, puis à écrire des listes de propriétés. Vous avez donc parfaitement la possibilité de vérifier la présence d'un fichier dans la sandbox de votre application et de le charger : vous créez ainsi rapidement un système de cache et vous offrez tout de même du contenu à vos utilisateurs malgré l'absence d'une connexion Internet, par exemple. En disposant l'enregistrement du fichier dans la méthode `applicationWillTerminate` prédéfinie dans les projets de Xcode, vous mettez en place un système souple de sauvegarde des données. Cette méthode s'exécute en effet lorsque l'application est sur le point de se quitter, lorsque vous répondez à un coup de fil inopiné par exemple.

Cocoa Touch dispose également d'un système de gestion de bases de données très léger, *sqlite3*. S'articulant autour du langage SQL, il vous permet d'enregistrer et de lire tout type de données. Son intégration dépasse toutefois le cadre de notre ouvrage et vous oblige à maîtriser un langage complémentaire. Pour en savoir plus sur son intégration et ses usages, reportez-vous à l'adresse <http://tinyurl.com/yo9n95>.

Puisque nous savons désormais lire et écrire des fichiers, que diriez-vous d'une petite halte du côté du multimédia ? Il s'agit de l'un des dossiers associés à la sandbox de votre application (Media). En y stockant tout type de contenu supplémentaire, vous enrichissez votre projet et vous offrez aux utilisateurs un environnement plus immersif et plus souple.



8

Le multimédia

Au sommaire de ce chapitre

- Les images et les photographies
- L'audio et la vidéo
- Le dessin et les animations
- Pour aller plus loin

Si l'iPhone est un grand communicant, autant à l'aise dans les fonctions téléphoniques de base que dans l'interaction avec des services extérieurs ou des données associées à votre application, c'est aussi un mobile à l'incroyable potentiel multimédia. Appareil photo, baladeur dans la veine de l'iPod, lecture d'images sur lesquelles on zoome en écartant deux doigts, affichage de séquences vidéos, animations en 3D, etc. Les développeurs les plus ambitieux disposent d'une palette très large de fonctions pour étendre le champ de leurs

applications et offrir une expérience plus riche à leurs utilisateurs. À l'instar de tous les composants du kit de développement de l'iPhone, la gestion des données multimédia obéit à un principe simple et vous permet de manipuler rapidement ce type de ressources. Comme au chapitre précédent, il est possible de stocker l'ensemble de ces données dans la *sandbox* de votre application afin de compacter toutes les ressources dont elle a besoin.

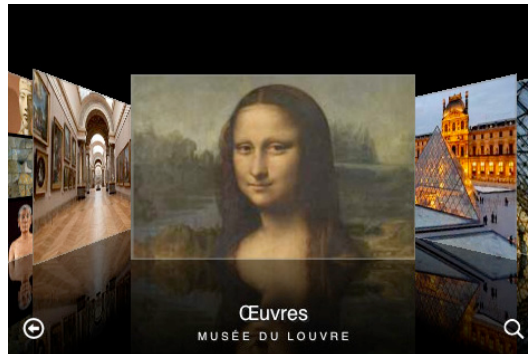
Mais avant de foncer tête baissée parmi ces possibilités et de parsemer vos projets de fichiers MP3 ou d'images en grand format, réfléchissez à deux fois. Comme nous l'avons vu, les applications pour iPhone visent avant tout à servir un rôle simple et précis. Les utilisateurs doivent être capables d'interroger immédiatement votre application et d'obtenir un résultat en moins de vingt secondes. Ne cédez donc pas envers et contre tout aux sirènes du multimédia, en bardant votre interface de contrôles superflus ou d'une bande son à l'arrière-plan ! Mal exploité, ce type de données ne se contente pas d'accaparer inutilement l'attention de l'utilisateur ; il consomme également des ressources considérables jusqu'à faire planter le mobile d'Apple dans le pire des cas.

Vous voilà mis en garde. L'iPhone peut d'ores et déjà rendre des services considérables avec les éléments de base du kit de développement : on ne reprochera jamais sa trop grande sobriété à une application, si elle remplit parfaitement le rôle qu'on lui a fixé. Effectuez en revanche un usage intensif de ce type de ressources si votre projet s'articule précisément autour de la photographie ou de la musique. Lecteurs multimédias, retouches photographiques, création audio et gestion d'albums : les utilisateurs qui téléchargent ce type d'applications savent à quoi s'attendre et ils seront déçus de ne pas profiter d'un environnement riche ou de ne pas plonger dans de saisissants clichés à manipuler du bout des doigts.

À titre d'exemple, l'application du Musée du Louvre fait figure de modèle et combine intelligemment tous les atouts de l'iPhone (voir Figure 8.1). Le contenu éditorial est riche et s'appuie sur des contrôles standard, en particulier des libellés, des champs de texte et des boutons. Toutes les vues s'enrichissent harmonieusement d'images ou de séquences vidéo et vous parcourez les collections en zoomant sur les détails d'une sélection d'œuvres. Si vous envisagez d'enrichir votre propre application de ce type d'éléments, prenez cet exemple en modèle. L'usage du multimédia y est constant, mais il se justifie sans cesse : aucune image n'est là par hasard et ces ressources soutiennent un contenu éditorial très riche.

Figure 8.1

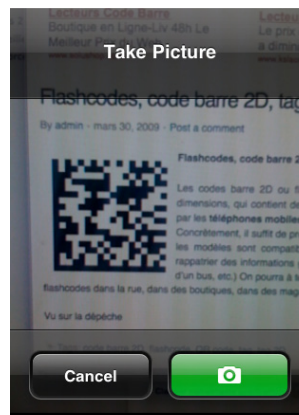
L'application du Musée du Louvre.



L'utilisation de l'appareil photo de l'iPhone fait figure de cas particulier. Il ne s'agit pas d'imposer des images à l'utilisateur, mais bien de lui proposer d'utiliser le capteur de l'iPhone pour prendre le sujet de son choix en flagrant déclin, avant de manipuler éventuellement le résultat avec votre application. La tâche est clairement identifiée et il est le seul à décider des éléments figurant sur l'interface de son téléphone. L'intérêt artistique de ces éléments visuels ne vous incombe donc pas ! De nombreuses applications mettent ainsi en place un bouton déclenchant une photo ; c'est notamment le cas de la liste de tâches *ToDo*, qui vous permet d'archiver les factures liées à vos achats courants ou d'*UpCode* grâce auquel vous photographiez et convertissez les codes FlashCode présents sur certaines publicités (voir Figure 8.2).

Figure 8.2

Grâce à UpCode, vous prenez en photo les codes barre "FlashCodes" qui sont automatiquement convertis.

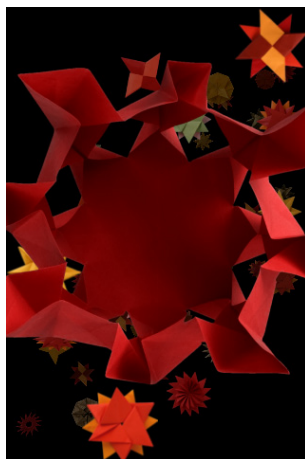


Les éléments sonores sont moins envahissants et plus discrets. On les réservera essentiellement aux jeux vidéo et éventuellement aux messages d'alertes, afin d'attirer l'attention de l'utilisateur comme sur une application bureautique. Si vous envisagez de programmer un lecteur multimédia, inspirez-vous de l'application iPod installée par défaut – s'il pa-

raît prétentieux de rivaliser avec cet auguste compétiteur, si complet et ergonomique, vous avez en revanche la possibilité de spécialiser votre lecteur dans certains formats ou genres musicaux. Là encore, vous pouvez profiter de l'immense variété des contrôles de l'iPhone pour offrir une nouvelle manière d'écouter sa musique préférée. C'est notamment le pari du groupe britannique *Snow Patrol*, qui propose une application officielle très déroutante dans laquelle vous écoutez des extraits des derniers albums tout en dépliant des origamis (voir Figure 8.3). Un vrai concept ! Par ailleurs, sachez que vous pouvez faire appel aux fonctions internes de l'iPhone et accéder à son microphone afin d'enregistrer l'environnement extérieur. De nombreux dictaphones numériques de poche déferlent ainsi sur l'App Store d'iTunes.

Figure 8.3

L'application officielle du groupe Snow Patrol déclenche des sons en fonction des pressions de l'utilisateur.



Passons sans plus tarder à la pratique et voyons ensemble comment les ressources multimédias contribuent à enrichir votre application.

Les images et les photographies

Afficher des images

Vous avez appris au Chapitre 6 à manipuler de petites illustrations sous Interface Builder et à les charger en guise de logo ou d'arrière-plan. Le chargement puis le positionnement d'images en Objective-C obéissent au même principe général : vous créez une vue d'image à travers une instance de la classe `UIImageView` et vous chargez les données correspondantes avec une instance de la classe `UIImage`. Il vous suffit ensuite d'ajouter la vue d'image à la vue principale pour faire apparaître l'illustration à l'écran. L'image en elle-même peut être

stockée au sein de votre application, chargée depuis la photothèque de l'utilisateur ou téléchargée depuis une source externe.

Sous Xcode, déroulez le menu File, cliquez sur New Project et choisissez le modèle View-Based Application. Baptisez votre application Photo et validez la création de votre nouveau projet. En guise de premier exemple, nous allons volontairement charger une image dont les dimensions dépassent celles de l'écran de l'iPhone. Préparez un fichier `photo.jpg` puis cliquez sur le dossier Resources en maintenant la touche Ctrl du clavier enfoncée. Choisissez Add, puis Existing Files et sélectionnez votre nouveau fichier. Il s'ajoute au projet. Nous allons créer une instance de la classe `UIImageView` et lui faire soutenir l'image que nous venons d'ajouter. Très simple, ce code ne comprend que quelques lignes. Ouvrez le fichier `PhotoViewController.m` et modifiez la déclaration de la méthode `loadView` comme au listing ci-dessous.

Listing 8.1 : *PhotoViewController.m*

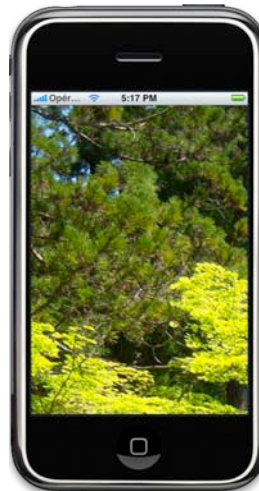
```
- (void)loadView {
    UIImageView *vueImage = [[UIImageView alloc] initWithImage:
                               [UIImage imageNamed:@"photo.jpg"]];

    self.view = vueImage;
    [vueImage release];
}
```

Enregistrez votre programme et compilez le projet (voir Figure 8.4).

Figure 8.4

L'image chargée dépasse de l'écran de l'iPhone.



Dans le Simulateur d'iPhone, vous constatez que l'image dépasse largement de l'écran : elle s'affiche en taille réelle et seule sa partie supérieure gauche (sur 320×480 pixels) apparaît sur l'interface. La barre de statut recouvre même une partie de l'image, sur 20 pixels de hauteur. Pour y remédier, vous devez définir le cadre dans lequel s'inscrit votre vue d'image. Il existe plusieurs solutions pour aboutir à un tel résultat et redimensionner à la volée le cliché : remplacez la déclaration de `loadView` comme au Listing 8.2.

Listing 8.2 : Redimensionner la zone d'affichage

```
- (void)loadView {  
    UIImageView *vueImage = [[UIImageView alloc]  
        initWithFrame:[UIScreen mainScreen] applicationFrame];  
    [vueImage setImage:[UIImage imageNamed:@"photo.jpg"]];  
    self.view = vueImage;  
    [vueImage release];  
}
```

Relancez le Simulateur d'iPhone et constatez le résultat : quelles que soient ses dimensions, votre image apparaît désormais en entier sur l'écran (voir Figure 8.5).

Figure 8.5

L'image chargée s'adapte désormais aux dimensions de l'écran de l'iPhone.



La classe `UIScreen` vous permet de délimiter le cadre de votre application. Sa méthode `mainScreen` retourne l'objet représenté par l'écran de l'iPhone et la propriété `applicationFrame` correspond exactement au cadre de votre application, soit l'écran entier du mobile sans sa barre de statut (320×460 pixels). Dans notre exemple, nous avons ainsi réduit la vue d'image à ces dimensions et la photographie s'adapte au canevas en conséquence. Mais voilà : si votre image n'obéit pas au même ratio que l'iPhone, elle va paraître déformée. Grâce à la propriété `contentMode`, vous définissez le type de redimensionnement du contenu

de la vue. Il existe treize constantes différentes ; dans notre exemple, nous allons utiliser `UIViewContentModeScaleAspectFit` afin de préserver l’aspect de l’image. Modifiez une nouvelle fois la déclaration de `loadView` avec le code figurant au Listing 8.3.

Listing 8.3 : Préserver l’aspect de l’image

```
- (void)loadView {
    UIImageView *vueImage = [[UIImageView alloc]
        initWithFrame:[UIScreen mainScreen] applicationFrame]];
    [vueImage setImage:[UIImage imageNamed:@"photo.jpg"]];
    vueImage.contentMode = UIViewContentModeScaleAspectFit;
    self.view = vueImage;
    [vueImage release];
}
```

Cette fois, l’image conserve ses propriétés d’origine et se voit simplement contrainte dans le conteneur de 320 × 460 pixels. Si elle est au format portrait, sa hauteur est redimensionnée à 460 pixels, et si elle est au format paysage, c’est sa largeur qui est réduite à 320 pixels. Son aspect est préservé et des zones transparentes apparaissent éventuellement autour de l’image afin de meubler l’espace vacant. Reportez-vous au Tableau 8.1 pour découvrir les autres constantes de la propriété `contentMode` et leur rôle. Essayez de modifier l’exemple précédent afin de tester toutes vos possibilités (voir Figure 8.6).

Tableau 8.1 : Les constantes de la propriété `contentMode`

Constante	Rôle
UIViewContentModeScaleToFill	L’image occupe tout l’espace de la vue et son aspect est éventuellement modifié (mode par défaut).
UIViewContentModeScaleAspectFit	L’image adapte ses dimensions en préservant son aspect.
UIViewContentModeScaleAspectFill	L’image occupe tout l’espace de la vue en préservant son aspect : certaines zones peuvent ainsi être rognées.
UIViewContentModeRedraw	Redessine l’image lorsque les dimensions de la vue changent.
UIViewContentModeCenter	L’image est centrée au milieu de la vue, sans adapter ses dimensions.
UIViewContentModeTop	L’image est centrée en haut de la vue, sans adapter ses dimensions.
UIViewContentModeBottom	L’image est centrée en bas de la vue, sans adapter ses dimensions.

Tableau 8.1 : Les constantes de la propriété `contentMode` (suite)

Constante	Rôle
<code>UIViewContentModeLeft</code>	L'image s'aligne à gauche de la vue, sans adapter ses dimensions.
<code>UIViewContentModeRight</code>	L'image s'aligne à droite de la vue, sans adapter ses dimensions.
<code>UIViewContentModeTopLeft</code>	L'image est chargée à partir de son bord supérieur gauche, sans redimensionnement.
<code>UIViewContentModeTopRight</code>	L'image est chargée à partir de son bord supérieur droit, sans redimensionnement.
<code>UIViewContentModeBottomLeft</code>	L'image est chargée à partir de son bord inférieur gauche, sans redimensionnement.
<code>UIViewContentModeBottomRight</code>	L'image est chargée à partir de son bord inférieur droit, sans redimensionnement.

Figure 8.6

La constante `UIViewContentModeScaleAspectFill` oblige parfois à rogner l'image d'origine.



L'intérêt des propriétés de redimensionnement

Ce premier exemple illustre la nécessité de préparer soigneusement les images que vous affichez à travers votre application. À l'aide de votre logiciel de création graphique préféré, n'hésitez pas à rogner ou à changer l'échelle de vos illustrations de telle sorte à ce qu'elles épousent le ratio de l'écran de l'iPhone : vous limiterez ainsi l'apparition de zones transparentes. Vous réduirez au passage la taille du fichier correspondant. À moins de prévoir l'utilisation de zooms sur l'image, évitez en effet d'intégrer des fichiers trop volumineux au

sein de votre projet. Par ailleurs, les propriétés de `contentMode` présentent un grand intérêt lors de la rotation de l'écran de l'iPhone ; lorsque l'utilisateur pivote son mobile en mode paysage, la vue contenant l'image est redessinée et vous vous assurez ainsi que celle-ci n'est pas tronquée à l'écran. L'application *Photos* préinstallée sur l'iPhone exploite précisément cette solution et utilise la propriété `UIViewContentModeScaleAspectFit` pour préserver l'aspect de vos clichés et les adapter quel que soit le mode d'affichage choisi. Nous y reviendrons largement au cours du chapitre suivant.

Adapter les dimensions de la vue d'image

Au cours des chapitres précédents, nous avons essentiellement défini les dimensions des instances de la classe `UIImageView` à la souris, à travers Interface Builder. Si la manipulation est simple et intuitive, elle peut parfois s'avérer rébarbative dans le cas du chargement d'images. Vous devez en effet définir des outlets dans le contrôleur de la vue puis les associer à vos éléments d'interface. Vous effectuez ainsi un incessant va-et-vient entre Xcode et Interface Builder. En considérant l'expérience dont vous disposez aujourd'hui, vous constatez qu'il est parfois plus rapide de se passer d'Interface Builder et de modifier la méthode `loadView` afin de définir directement vos éléments d'interface en Objective-C, comme nous l'avons fait à travers notre premier exemple. Attention toutefois : en procédant de la sorte, vous ne vous inscrivez plus pleinement dans le schéma MVC dans la mesure où la vue ne s'isole pas dans un fichier nib. Mais pour des programmes simples, vous aboutissez rapidement à un résultat.

Nous avons initialisé notre instance de la classe `UIImageView` à travers la méthode `initWithFrame` en adaptant ses dimensions à celles de l'écran de l'iPhone. Dans le même ordre d'idée, vous avez la possibilité de positionner très précisément vos vues d'images à l'aide de la structure `CGRect`. Comme son nom l'indique, cet objet appartient au framework *Core Graphics* dont *Quartz* est le moteur graphique. On retrouve naturellement ce framework sous Mac OS X : il est responsable de l'affichage de l'interface Aqua du système. Le framework *Core Graphics* fait partie de la couche *Media* de *Cocoa Touch*, au même titre que *Core Audio* et *Core Animation* sur lesquels nous reviendrons dans la suite de ce chapitre.

La structure `CGRect` définit un rectangle aux dimensions et au positionnement précis. En utilisant la propriété `frame` de tous les types de vues, vous modifiez ainsi l'interface de votre application en quelques lignes d'Objective-C. Vous devez impérativement vous familiariser avec cette technique afin de gérer au mieux l'auto-rotation de l'iPhone et le repositionnement d'éléments d'interface, comme nous le verrons par la suite. Par ailleurs, cette méthode vous oblige à maîtriser au mieux le système de coordonnées du kit de développement – un atout précieux lorsque vous souhaitez disposer des onglets, des boutons ou une barre de navigation sur une interface très complexe !

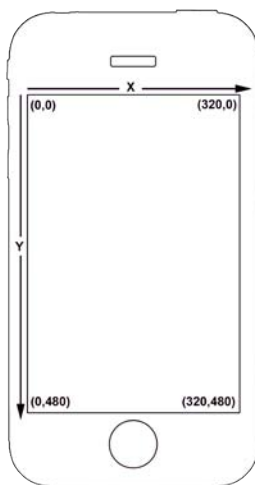
Sur l'écran de l'iPhone, les coordonnées s'expriment en pixels et l'origine correspond au coin supérieur gauche. L'axe des abscisses (x) est horizontal et l'axe des ordonnées (y) est vertical, que l'on tienne le mobile en mode portrait ou paysage. En considérant que l'écran de l'iPhone fait 320×480 pixels dans le mode normal d'affichage, on comprend ainsi le système de coordonnées :

- le coin supérieur droit correspond au point (320, 0) ;
- le coin inférieur gauche correspond au point (0, 480) ;
- le coin inférieur droit correspond au point (320, 480).

Vous pouvez ainsi vous repérer dans le plan et disposer précisément toutes les vues de votre application (voir Figure 8.7). À ce titre, passons à la pratique en adaptant notre dernier exemple à un cas concret. Nous souhaitons disposer une légende en-dessous du cliché chargé précédemment : nous devons ainsi prévoir une zone pour abriter un libellé. La vue d'images se réduit en hauteur et nous ajoutons une instance de la classe `UIView` pour soutenir la légende. Elle contient elle-même une instance de la classe `UILabel` correspondant au libellé. Celle-ci dérive d'`UIView` et on peut donc également préciser ses coordonnées à l'aide de la structure `CGRect`.

Figure 8.7

Le système de coordonnées de l'iPhone.



Afin d'illustrer le principe de chargement des vues, nous allons modifier cette fois la méthode `viewDidLoad` de notre contrôleur. Elle intervient juste après l'exécution de `loadView` : la vue est chargée mais elle n'est pas encore affichée. La méthode `viewWillAppear` vient ensuite. La vue est sur le point d'apparaître à l'écran : vous procédez éventuellement à vos ultimes réglages. À l'inverse, la méthode `viewDidAppear` est appelée après son affichage final. Il est intéressant d'utiliser `viewDidLoad` lorsque vous souhaitez changer les proprié-

tés d'objets instanciés par la vue d'un fichier nib. Dans ce cas, la méthode `loadView` a en effet chargé tous les objets de la vue : on les initialise avec du code inclus dans la méthode `viewDidLoad`. Si les traitements sont particulièrement lourds, vous limitez ainsi la consommation mémoire en distinguant le chargement des objets de leur code d'initialisation. Dans notre exemple, nous souhaitons ajouter des sous-vues à la vue principale : nous pouvons ainsi la désigner par `self.view` dans la mesure où elle a été chargée. Recopiez le code du Listing 8.4.

Listing 8.4 : Chargement d'une image avec une légende

```
- (void)viewDidLoad {

    // Préparation de la vue pour l'image
    UIImageView *vueImage =
        [[UIImageView alloc] initWithFrame:CGRectMake(0, 0, 320, 420)];
    [vueImage setImage:[UIImage imageNamed:@"photo.jpg"]];
    vueImage.contentMode = UIViewContentModeScaleAspectFill;

    // Préparation de la vue pour la légende
    UIView *vueLegende = [[UIView alloc] initWithFrame:CGRectMake
        (0, 420, 320, 40)];
    vueLegende.backgroundColor = [UIColor darkGrayColor];

    // Déclaration de la légende
    UILabel *legende = [[UILabel alloc] initWithFrame:CGRectMake
        (20, 5, 280, 30)];
    legende.text = @"Temple à proximité de Tokyo";
    legende.textAlignment = NSTextAlignmentCenter;
    legende.textColor = [UIColor whiteColor];
    legende.backgroundColor = [UIColor clearColor];

    // Ajout de la vue de la légende
    [vueLegende addSubview:legende];
    [self.view addSubview:vueLegende];

    // Ajout de la vue de l'image
    [self.view addSubview:vueImage];

    [vueLegende release];
    [vueImage release];
    [legende release];
}
```

Enregistrez le projet puis compilez-le : l'interface principale de votre application affiche bien deux vues, l'image et la légende, avec un libellé positionné en bas de l'écran (voir Figure 8.8).

Figure 8.8

Notre interface affiche la photo et une légende, positionnées grâce au système de coordonnées.



Nous avons d'abord conservé l'instance de la classe `UIImageView`, destinée à soutenir la photo. Cette vue est délimitée par un rectangle de 320×420 pixels, positionné en haut à gauche. Nous chargeons l'image et nous préservons son aspect comme précédemment. Nous déclarons également l'instance `vueLegende` de la classe `UIView` afin de positionner la légende. Elle correspond à un rectangle de 320×40 pixels, placé juste en-dessous de la photographie. Son origine est ainsi définie au point (0,420). Petite subtilité à des fins d'illustration du code : nous souhaitons que le libellé soit une sous-vue de `vueLegende` et non de la vue principale. Ses coordonnées sont ainsi définies par rapport à cette vue parente ; l'origine correspond donc en réalité au point (0, 420) de la vue globale (voir Figure 8.9).

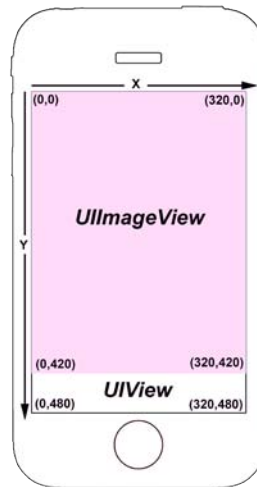
Nous aurions parfaitement pu nous dispenser de l'instance `vueLegende` et placer directement le libellé aux coordonnées attendues, en fonction de la vue principale. L'ajout de la sous-vue aurait alors été :

```
UILabel *legende = [[UILabel alloc] initWithFrame:CGRectMake(
                                                                    (20, 425, 280, 30));
legende.text = @"Temple à proximité de Tokyo";
legende.textAlignment = NSTextAlignmentCenter;
legende.textColor = [UIColor whiteColor];
legende.backgroundColor = [UIColor clearColor];
[self.view addSubview:legende];
```

Le résultat est strictement identique en ce qui concerne le libellé ; l'instance `vueLegende` nous permet toutefois d'ajouter une couleur en arrière-plan. En guise d'exercice, essayez de vous passer totalement de cette instance `vueLegende` en modifiant les dimensions du libellé et sa propriété `backgroundColor` afin de reproduire le même effet visuel.

Figure 8.9

Les coordonnées de nos différents objets d'interface.



Un portfolio de photographe

Si vous êtes photographe, combinez cet exemple avec le diaporama manipulable au doigt, présenté au Chapitre 9, afin de présenter votre travail dans une application dynamique (voir Figure 8.10). Essayez alors d'intégrer la légende à la vue `vueImage` afin de la "déplacer" en même temps que l'image à laquelle elle est associée.

Figure 8.10

En définissant soigneusement l'interface de votre application, vous pouvez aboutir à un portfolio haut en couleur, comme celui du photographe Eric Lafforgue !



Charger une image depuis une source externe

Dans cette série d'exemples, nous avons ajouté directement l'image à charger à notre projet, sous Xcode. Essayons à présent d'importer l'image depuis une source externe. Le service [ParisLive.net](http://parislive.net) propose une webcam centrée sur la Tour Eiffel. Elle dépose à intervalles réguliers une image JPG, toujours du même nom, sur un serveur web. Nous pouvons ainsi créer une application qui charge cette image afin de relayer le service sur iPhone.

On y parvient à l'aide de la classe `NSURL`, qui vous permet de manipuler des URL et les ressources auxquelles elles font référence. Utilisé conjointement avec la classe `NSData`, cet objet récupère ainsi le contenu d'une image et la charge dans une instance de la classe `UIImage`.

Sous Xcode, créez un nouveau projet de type "View-Based Application". Modifiez alors la méthode `loadView` du contrôleur de la vue, comme au Listing 8.5.

Listing 8.5 : Chargement de la webcam de la Tour Eiffel

```
- (void)loadView {
    NSURL *url = [NSURL URLWithString:@"http://www.parislive.net/
                                     eiffelwebcam1.jpg"];

    UIImage *image = [UIImage imageWithData:
                      [NSData dataWithContentsOfURL:url]];
    UIImageView *vueWebcam = [[UIImageView alloc]
                              initWithFrame:CGRectMake(16, 64, 288, 352)];
    [vueWebcam setImage:image];
    self.view = vueWebcam;
    [vueWebcam release];
}
```

Le principe est simple. On commence par définir l'adresse de l'image de la webcam. Le nom du fichier est toujours le même, ce qui facilite largement l'accès à la ressource. On crée ensuite une instance de la classe `UIImage`, en l'initialisant avec le contenu de l'URL. Une instance de la classe `UIImageView` la présente à l'écran. Ses coordonnées ont été définies de telle sorte à ce que l'image apparaisse centrée. La webcam renvoie en effet une image de 288 × 352 pixels et nous définissons l'origine de la structure `CGRect` afin de laisser des marges équivalentes de part et d'autre de la photo. Nous aurions pu aboutir au même résultat en étendant l'instance d'`UIImageView` à l'ensemble de la vue principale et en utilisant l'une des constantes vues précédemment pour centrer l'image.

Enregistrez le code et compilez le projet : l'image de la webcam de la Tour Eiffel apparaît bien au centre de l'écran (voir Figure 8.11). Le serveur ajoute automatiquement la date et

l'heure courantes à l'image, ce qui vous permet de vérifier qu'elle est bien chargée en temps réel.

Figure 8.11

L'image de la webcam est chargée depuis un serveur web et apparaît sur l'écran de l'iPhone.



Ajouter un contrôle afin de recharger la vue

Mais bien souvent, l'élément graphique chargé depuis une source extérieure ne constitue que l'un des objets que vous souhaitez afficher à l'écran. Vous avez besoin de mettre à jour la vue d'image mais aussi d'ajouter des libellés et des contrôles permettant de déclencher des méthodes personnalisées. Cet exemple de webcam affichée en temps réel s'y prête tout particulièrement : il ne récupère pour l'heure que la dernière image de la Tour Eiffel. Si on laisse tourner infiniment l'application, l'image n'est pas mise à jour et la webcam perd de son intérêt.

Modifions le code de notre projet afin d'ajouter un bouton "Recharger" qui déclenchera le téléchargement d'une nouvelle image. L'interface sera remise à jour en conséquence et l'utilisateur pourra ainsi surveiller la Tour Eiffel à toute heure du jour ou de la nuit ! Pour cela, la variable d'instance de la classe `UIImageView` doit être une variable globale, accessible à de multiples endroits de votre programme. Nous avons baptisé notre projet "Webcam" – ouvrez le fichier `WebcamViewController.h` et modifiez l'interface du contrôleur comme suit :

```
@interface WebcamViewController : UIViewController {UIImageView *vueWebcam;
}
@end
```

Modifiez ensuite le code de l'implémentation du contrôleur comme au Listing 8.6.

Listing 8.6 : WebcamViewController.m

```
#import "WebcamViewController.h"

@implementation WebcamViewController

- (void)changeWebcam:(id)sender {
    NSURL *url = [NSURL URLWithString:@"http://www.parislive.net/
                                     eiffelwebcam1.jpg"];

    UIImage *image = [UIImage imageWithData:
                      [NSData dataWithContentsOfURL:url]];
    vueWebcam.contentMode = UIViewContentModeScaleAspectFit;
    [vueWebcam setImage:image];
}

- (void)viewDidLoad {
    [super viewDidLoad];
    NSURL *url = [NSURL URLWithString:@"http://www.parislive.net/
                                     eiffelwebcam1.jpg"];

    UIImage *image = [UIImage imageWithData:
                      [NSData dataWithContentsOfURL:url]];
    vueWebcam = [[UIImageView alloc] initWithFrame:CGRectMake(16, 24, 288, 352)];
    [vueWebcam setImage:image];

    UIButton *bouton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [bouton setFrame:CGRectMake(110, 395, 100, 30)];
    [bouton setTitle:@"Recharger" forState:UIControlStateNormal];
    [bouton addTarget:self action:@selector(changeWebcam:)
                     forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:bouton];

    [self.view addSubview:vueWebcam];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [super dealloc];
    [vueWebcam release];
}

@end
```

Enregistrez le code et compilez le projet. Cette fois, un bouton "Recharger" apparaît sous la première image de la webcam et vous permet de mettre à jour l'affichage, en téléchargeant la dernière vue réactualisée sur la Tour Eiffel (voir Figure 8.12).

Figure 8.12.

Le bouton déclenchant le rafraîchissement de l'image est ajouté dans le code du contrôleur.



Nous avons en effet créé une instance de la classe `UIButton`, de type `UIButtonTypeRoundedRect` et nous lui avons assigné les coordonnées de la légende de photo, définies à l'exemple précédent. Le bouton et la vue de la webcam sont des sous-vues de la vue principale : leurs coordonnées s'inscrivent ainsi dans le gabarit global de l'écran de l'iPhone. La ligne suivante correspond au choix de l'événement qui déclenche une action :

```
[bouton addTarget:self action:@selector(changeWebcam:)
      forControlEvents:UIControlEventTouchUpInside];
```

Cette ligne est donc équivalente à notre procédé habituel, qui consistait à définir une action dans l'interface de notre contrôleur puis à modifier le fichier nib en associant cette méthode à l'événement `TouchUpInside` d'un bouton aux coins arrondis. Nous n'avons jamais invoqué Interface Builder au cours de la programmation de cette application et nous aboutissons ainsi à un résultat rapide et efficace.

Charger des images et des PDF avec la classe `UIWebView`

L'iPhone dispose d'un navigateur web très réactif et complet, qui supporte sans problème le chargement de fichiers JPEG, PNG, TIF ou PDF très volumineux. Tous ces types de fichiers sont reconnus immédiatement, sans intervention de votre part, et le navigateur offre un cadre convivial et prêt à l'emploi pour manipuler ces ressources externes. Vous l'implémentez dans vos applications à l'aide de la classe `UIWebView`. Vous créez ainsi une vue corres-

pendant au navigateur web, expurgé du moindre contrôle, et l'utilisateur peut librement le manipuler (zooms avec deux doigts, défilement de la page avec un doigt, etc.).

Par ailleurs, cette classe offre d'immenses possibilités à tous les développeurs web. Si vous êtes déjà rompu à la création de larges bases de données, de scripts PHP ou ASP et d'arbres XML, vous avez ainsi l'opportunité de créer des applications très riches en quelques lignes d'Objective-C. Dans votre code, il vous suffira de disposer une vue `UIWebView` puis d'ajouter une série de contrôles déclenchant l'accès à d'autres URL (ou à des adresses formatées avec un paramètre, comme `script.php?id=4&rubrique=2` par exemple). Bien entendu, le serveur distant devra être accessible sous peine de réduire l'intérêt de votre application à peau de chagrin.

Nous allons illustrer la puissance de cette classe en chargeant à distance le PDF du guide de l'utilisateur de l'iPhone. Il pèse 6,4 Mo mais son chargement et sa consultation ne feront absolument pas souffrir votre iPhone.

Dans un nouveau projet de type "View-Based Application", surchargez la méthode `loadView` de votre contrôleur, comme au Listing 8.7.

Listing 8.7 : L'implémentation de la classe `UIWebView`

```
- (void)loadView {
    NSURL *url = [NSURL URLWithString:@"http://manuals.info.apple.com/fr/
                                   Guide_de_l_utilisateur_de_l_iphone.pdf"];
    NSURLRequest *requete = [NSURLRequest requestWithURL:url];

    UIWebView *vueWeb = [[UIWebView alloc] initWithFrame:
                        [[UIScreen mainScreen] applicationFrame]];
    [vueWeb setScalesPageToFit:YES];

    [vueWeb loadRequest:requete];
    self.view = vueWeb;

    [vueWeb release];
}
```

Nous avons tout d'abord défini l'URL du document PDF à charger. Celle-ci s'inscrit alors dans une requête soutenue par la classe `NSURLRequest`. Nous créons ensuite l'instance de la classe `UIWebView` en l'étendant à l'ensemble de l'écran de l'iPhone. La propriété `setScalesPageToFit` permet à l'utilisateur de modifier l'échelle du document par de simples zooms en écartant deux doigts. Nous lançons enfin la requête à l'aide de la méthode d'instance `loadRequest`. Nous ajoutons alors la vue à l'interface de notre application. Enregistrez le projet et compilez-le : le navigateur de l'iPhone occupe tout l'écran et le document PDF

est chargé en quelques secondes en situation réelle, en fonction du débit de la connexion WiFi ou 3G (voir Figure 8.13).

Figure 8.13

L'objet `UIWebView` soutient le navigateur et vous permet de charger rapidement de nombreuses ressources.



Vous pouvez librement adapter cet exemple à tout type de ressources ou de fichiers PDF. En inscrivant cette vue dans une barre de navigation, constellée d'onglets par exemple, ou dans une liste d'éléments pointant vers des fichiers associés à votre application, vous créez en quelques lignes un lecteur de livres numériques ou un lecteur multimédia, capable d'accéder à une grande variété de contenus (voir Figure 8.14).

Figure 8.14

L'application du magazine Premiere utilise ce type de représentation pour afficher les horaires de salles de cinéma.



Gare aux avertissements mémoire !

Si vous essayez de charger des PDF définitivement trop copieux pour l'iPhone, la méthode `applicationDidReceiveMemoryWarning` du délégué de votre application risque de se lancer. Si la plupart du temps, vous pouvez ignorer ces messages d'avertissement (la classe `UIWebView` gère automatiquement les ressources mémoires et parvient à limiter sa consommation de ressources de manière drastique), vous devez toutefois prendre le temps d'examiner plus en détails votre application à l'aide des instruments de Xcode. Pour cela, déroulez le menu Run, sélectionnez Start with Performance Tool puis Leaks. Enregistrez une trame de l'exécution de votre application et surveillez ainsi la consommation des ressources. Renouvelez l'opération avec l'instrument Activity Monitor. Comparez ces valeurs avec le chargement du guide de l'utilisateur en PDF, comme à l'exemple précédent.

Charger la photothèque de l'utilisateur

Plébiscité par les utilisateurs, l'appareil photo intégré à l'iPhone peut être manipulé par votre application. Dans le même ordre d'idée, vous avez accès à la photothèque personnelle de chaque utilisateur. Nous avons vu jusqu'à présent que l'application dispose de sa propre "sandbox" qui réunit tous les éléments dont elle a besoin ; elle est heureusement en mesure de charger les images communes de l'iPhone et de retoucher ainsi les clichés pris en-dehors de son cadre. L'exercice s'articule autour de la classe `UIImagePickerController`, qui met en place un large tableau constellé de vignettes des clichés et gère la sélection individuelle de photos et le parcours de la photothèque (voir Figure 8.15).

Figure 8.15

Le contrôleur charge automatiquement une vue avec la photothèque de l'utilisateur.



Le principe général ne diffère pas fondamentalement de l'implémentation des autres types de contrôleurs. Vous créez ainsi une instance de la classe `UIImagePickerController`, vous

indiquez sa source (la photothèque ou le lancement du capteur de l'iPhone) et vous l'implémentez dans votre application. Lorsque l'utilisateur sélectionne une image ou capture un cliché, le résultat est transmis au délégué de votre application. Vous pouvez ainsi surcharger la logique de ce délégué afin de procéder à quelques réglages et offrir des fonctions d'édition basique. Mais il existe désormais une grande variété de modèles d'iPhone et d'iPod Touch, qui ne disposent pas tous du même capteur. L'iPod Touch en est même dépourvu et seuls les derniers iPhone 3GS intègrent la possibilité de capturer des séquences vidéo. Vous devez ainsi vous assurer, à travers quelques lignes de code, que l'utilisateur dispose bien du matériel adéquat avant de lui proposer de telles fonctions. On y parvient à l'aide d'une simple boucle conditionnelle :

```
if ([UIImagePickerController isSourceTypeAvailable:
    UIImagePickerControllerSourceTypePhotoLibrary]); {
    // La photothèque est disponible sur cet iPhone
}
```

On distingue trois constantes pour la méthode `isSourceTypeAvailable`. Elles vous permettent également d'initialiser la classe `UIImagePickerController` :

- `UIImagePickerControllerSourceTypePhotoLibrary`. On utilise l'ensemble de la photothèque de cet iPhone : elle comprend à la fois les photos capturées avec le mobile et les images synchronisées à travers iTunes.
- `UIImagePickerControllerSourceTypeSavedPhotosAlbums`. On se sert uniquement des photos capturées avec l'iPhone.
- `UIImagePickerControllerSourceTypeCamera`. On utilise directement la vue de la caméra, afin de prendre un nouveau cliché en flagrant déclin ; il sera retourné au délégué du contrôleur.

On implémente le parcours de la photothèque très simplement : en déclarant une instance de la classe `UIImagePickerController` dans la méthode `viewDidLoad` de votre contrôleur, par exemple.

Créez un nouveau projet sous Xcode de type "View-Based Application" et baptisez-le "Phototheque". Remplacez alors la déclaration du contrôleur par les Listings 8.8 et 8.9.

Listing 8.8 : *PhotothequeViewController.h*

```
#import <UIKit/UIKit.h>

@interface PhotothequeViewController : UIImagePickerController
    <UIImagePickerControllerDelegate, UINavigationControllerDelegate> {

}
@end
```

Comme vous le constatez, notre contrôleur devient une instance de la classe `UIImagePickerControllerController` et il implémente les protocoles `UIImagePickerControllerControllerDelegate` et `UINavigationControllerDelegate`. En effet, le parcours des photos s'effectue dans une interface de navigation, avec un bouton permettant de revenir à l'écran général de sélection des albums.

Listing 8.9 : *PhotothequeViewController.m*

```
- (void)viewDidLoad {
    if([UIImagePickerController
isSourceTypeAvailable:UIImagePickerControllerSourceTypePhotoLibrary]) {
        UIImagePickerController *vuePhotos =
            [[UIImagePickerController alloc] init];
        vuePhotos.delegate = self;
        vuePhotos.sourceType =
            UIImagePickerControllerSourceTypePhotoLibrary;
        [self presentViewController:vuePhotos animated:YES];
        [vuePhotos release];
    }
}
```

Dans la méthode `viewDidLoad`, nous commençons par vérifier que la photothèque est bien disponible sur l'iPhone de l'utilisateur. On y parvient à l'aide d'une constante de la méthode `isSourceTypeAvailable`, comme évoqué précédemment. On initialise alors notre instance de la classe `UIImagePickerController` et on rattache son délégué à notre contrôleur principal. Nous indiquons ensuite la source des données, en utilisant la même constante que lors du test initial. Il ne nous reste plus qu'à afficher l'interface de navigation sur la vue, à l'aide de la méthode `presentModalViewController`. Il s'agit d'une méthode d'instance de la classe `UIViewController`, qui fait apparaître à l'écran l'objet `UIImagePickerController` défini précédemment dans un effet de "volet" vertical.

Enregistrez et compilez votre projet. Le Simulateur d'iPhone dispose par défaut de trois albums, contenant trente photos d'exemple (Hawaii, Slovénie et remise de diplôme). Parcourez les albums et utilisez les boutons figurant automatiquement dans la barre de titre pour afficher toutes les vignettes (voir Figure 8.16).

Figure 8.16

Notre application affiche les photos d'exemple du Simulateur d'iPhone.



Charger des photos individuelles et sauvegarder des retouches

Dans l'exemple précédent, nous affichons la photothèque de l'utilisateur au sein de notre application. Mais voilà : lorsque vous cliquez sur l'une des vignettes, il ne se produit rien de particulier. Nous avons en effet défini le délégué de l'application, mais nous n'avons implémenté aucune méthode spécifique. Vous devez ajouter deux méthodes liées au contrôleur `UIImagePickerController` :

- `didFinishPickingImage` s'exécute lorsque l'utilisateur a sélectionné une image individuelle.
- `imagePickerControllerDidCancel` s'exécute lorsque l'utilisateur annule la sélection d'une image ou la prise de vue avec l'appareil photo intégré.

La méthode `didFinishPickingMediaWithInfo`

Apparue avec le SDK de l'iPhone OS 3.0, la méthode `didFinishPickingMediaWithInfo` vise à compléter `didFinishPickingImage` lorsqu'il s'agit de capturer des séquences vidéo. Son giron est donc plus large et vous permet de récupérer de nombreuses informations associées au "média" capturé par l'utilisateur, à travers un objet `NSDictionary`. Implémentez plutôt cette méthode lorsque votre application s'articule autour de médias au sens large et que vous souhaitez diffuser un projet compatible exclusivement avec l'iPhone OS 3.0. Vous accédez à l'image spécifique sélectionnée par l'utilisateur en procédant comme suit :

```
- (void) imagePickerController:(UIImagePickerController *)picker  
    didFinishPickingMediaWithInfo:(NSDictionary *)details {
```



```

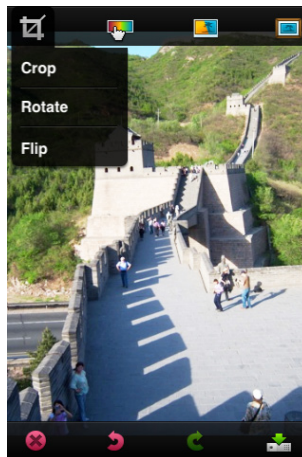
        imageView.image = [details objectForKey:UIImagePickerController
        EditedImage];
    [...]
}

```

En définissant ces deux méthodes, vous offrez un réel contrôle à l'utilisateur et la navigation au sein de la photothèque sert un objectif précis. Créons un nouveau projet : nous souhaitons que l'utilisateur sélectionne l'un de ses clichés puis le redimensionne du bout des doigts. La photo retouchée sera ensuite sauvegardée dans la sandbox de l'application. Il s'agit du comportement par défaut du gestionnaire de fond d'écran, situé dans les réglages de l'iPhone. De nombreuses applications s'appuient sur ce type de code pour enrichir des clichés personnels de bordures ou de pictogrammes (voir Figure 8.17).

Figure 8.17

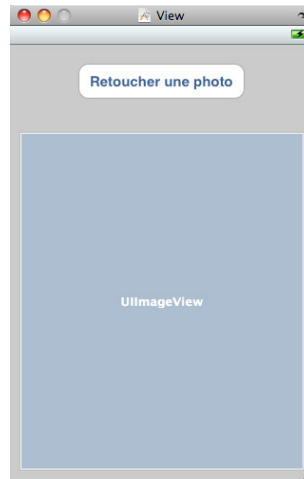
Les applications permettant de retoucher des photos personnelles comptent parmi les plus populaires d'iTunes.



Dans la mesure où l'application a une vocation graphique, nous allons réaliser sa vue principale avec Interface Builder. Créez un nouveau projet sous Xcode de type View-Based Application et baptisez-le RetouchePhoto. Ouvrez directement RetouchePhotoViewController.xib afin de lancer la vue dans Interface Builder. Au strict minimum, nous devons prévoir un objet UIImageView pour afficher l'image finale après son redimensionnement et un bouton pour déclencher le lancement de l'opération. Vous intégrerez ainsi facilement cette première fonction à une plus large application de retouche photographique. Déposez donc un élément Image View en bas de l'écran, en l'élargissant au cadre de l'iPhone. Ajoutez ensuite un objet Round Rect Button et effectuez un double-clic dessus afin de saisir son libellé, "Retoucher une photo". Enrichissez éventuellement l'interface des éléments visuels de votre choix. La vue globale devrait au moins ressembler à celle présentée à la Figure 8.18.

Figure 8.18

On prépare l'interface de notre application de retouche photo.



N'oubliez pas qu'en appelant les méthodes `presentModalViewController` et `dismissModalViewController`, vous ajoutez automatiquement un effet d'animation vertical : tenez-en compte dans le placement de vos objets. Sauvegardez votre interface, puis saisissez le code du Listing 8.10 dans le fichier `RetouchePhotoViewController.h` :

Listing 8.10 : *RetouchePhotoViewController.h*

```
#import <UIKit/UIKit.h>

@interface RetouchePhotoViewController : UIViewController
<UIImagePickerControllerDelegate, UINavigationControllerDelegate> {
    IBOutlet UIImageView *vuePhoto;
    IBOutlet UIButton *bouton;
    UIImagePickerController *imagePicker;
}

- (IBAction)retouchePhoto;

@property (retain, nonatomic) UIImagePickerController *imagePicker;

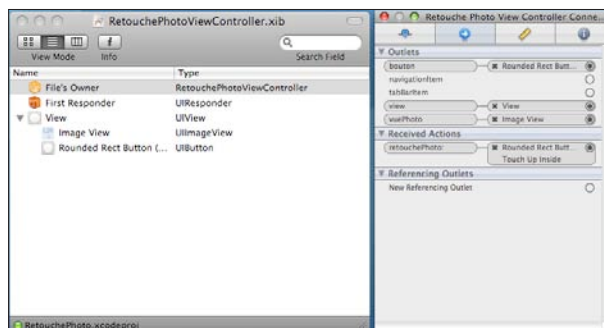
@end
```

Nous avons ici reproduit le code de l'exemple précédent, en définissant nos deux outlets : `vuePhoto` désigne l'instance de la classe `UIImageView` et `bouton` celle de la classe `UIButton`. La variable globale `imagePicker` nous permettra de désigner le contrôleur de la photothèque dans la logique de notre application. Nous déclarons également l'action `retouchePhoto`, qui déclenche la navigation dans la photothèque. Ouvrez à nouveau votre vue sous Interface

Builder et créez les liaisons. Pressez la touche Ctrl en cliquant sur File's Owner et sélectionnez l'instance d'UIImageView (vuePhoto) et celle d'UIButton (bouton). Cliquez ensuite sur ce bouton, ouvrez l'inspecteur de connexions (Cmd+2) et associez l'événement "Touch Up Inside" à File's Owner, en sélectionnant retouchePhoto. Vos connexions doivent correspondre à celles de la Figure 8.19. Enregistrez l'interface et retournez sous Xcode. Saisissez alors le code du Listing 8.11 dans RetouchePhotoViewController.m.

Figure 8.19

Les connexions de l'interface de notre application.



Listing 8.11 : RetouchePhotoViewController.m

```
#import "RetouchePhotoViewController.h"

@implementation RetouchePhotoViewController

#define DOSSIER_DOC [NSHomeDirectory() stringByAppendingPathComponent:
                    @"Documents"]
@synthesize imagePicker;

- (IBAction)retouchePhoto {
    [self presentViewController:self.imagePicker animated:YES];
}

- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingImage:(UIImage *)img
    editingInfo:(NSDictionary *)editInfo {

    vuePhoto.image = img;

    int i = 0;

    // Création du chemin d'accès unique pour enregistrer l'image retouchée
    // Chaque fichier prendra la forme "retouche_x.png"

    NSString *cheminAcces = [DOSSIER_DOC stringByAppendingPathComponent:
                             @"retouche.png"];
```

```

        while ([[NSFileManager defaultManager] fileExistsAtPath:cheminAcces])
            cheminAcces = [NSString stringWithFormat:@"%s@/%s@_%d.%s",
                          DOSSIER_DOC, @"retouche", ++i, @"png"];

        // Affichage du chemin d'accès dans le débogueur, utile pour le
        // Finder
        NSLog(@"%@", cheminAcces);

    // Enregistrement du fichier au format PNG
    [UIImagePNGRepresentation(img) writeToFile:cheminAcces atomically:YES];

    // On fait disparaître la vue de la photothèque
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
}

// Implement viewDidLoad to do additional setup after loading the view,
// typically from a nib.
- (void)viewDidLoad {
    // Test de la présence d'une photothèque
    // Permission d'éditer l'image choisie
    if([UIImagePickerController
isSourceTypeAvailable:UIImagePickerControllerSourceTypePhotoLibrary]) {
        self.imagePicker = [[UIImagePickerController alloc] init];
        self.imagePicker.delegate = self;
        self.imagePicker.allowsImageEditing = YES;
    }
}

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker
{
    [[picker parentViewController] dismissModalViewControllerAnimated:YES];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [super dealloc];
    [imagePicker release];
}

@end

```

Sauvegardez le projet et compilez-le. Dans le Simulateur d'iPhone, vous retrouvez l'interface de votre application de retouche photo. En cliquant sur le bouton, vous faites apparaître la photothèque de l'utilisateur. En sélectionnant un cliché, vous avez alors la possibilité de le redimensionner ou de le rogner. Validez votre choix : le résultat apparaît au centre de l'écran et un fichier est enregistré dans la sandbox de l'application (voir Figure 8.20). Sous Xcode, cliquez l'icône "gdb" au-dessus de la fenêtre de code et découvrez le chemin d'accès complet au répertoire de votre application. Accédez-y avec Finder : vous y retrouvez toutes vos images modifiées, enregistrées sous le nom `retouche_1.png`, `retouche_2.png`, etc.

Figure 8.20

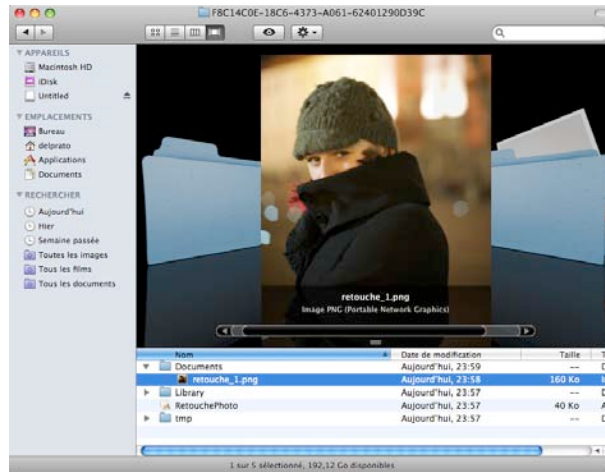
On sélectionne une photo de son album personnel avant de le redimensionner librement.



Le code est relativement clair. Au lancement de la vue, nous initialisons notre instance de la classe `UIImagePickerController`. Nous vérifions que l'iPhone dispose bien d'une photothèque, nous assignons le délégué du contrôleur et nous autorisons l'édition d'images. En cliquant sur le bouton central, on déclenche l'action `retouchePhoto`. Celle-ci fait alors apparaître la photothèque. Lorsque l'utilisateur sélectionne un cliché, il peut l'étendre ou le réduire : la propriété `allowsImageEditing` du contrôleur de photo est en effet activée. S'il valide son choix, la méthode `didFinishPickingImage` est appelée. On remplace alors la vue d'images avec le résultat et on crée un chemin d'accès unique. Celui-ci est composé du chemin au dossier `/Documents` de la sandbox de l'application (défini dans la constante `DOSSIER_DOC`, dans l'en-tête du fichier), suivi de `retouche_i.png`, où `i` est un entier que l'on incrémente. On procède enfin à l'enregistrement de l'image PNG et on fait disparaître la photothèque pour revenir à la vue principale de l'application (voir Figure 8.21).

Figure 8.21

Les modifications sont sauvegardées dans un fichier unique, que vous pouvez visualiser dans une fenêtre du Finder.

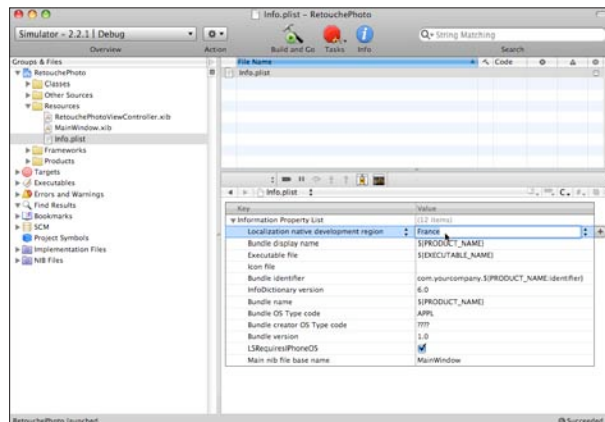


Traduire en français les contrôles prédéfinis

Par défaut, votre application est localisée en langue anglaise et les menus ou contrôles générés automatiquement ("Move and scale", "Cancel", etc.) s'expriment dans la langue de Shakespeare. Pour forcer totalement l'usage de la langue française, ouvrez le fichier `Info.plist` associé à votre projet et saisissez France dans le champ "Localization native development region" (voir Figure 8.22).

Figure 8.22

Modifiez les informations associées à votre projet pour forcer la traduction française des menus génériques.



Utiliser l'appareil photo de l'iPhone dans une application

Le Simulateur d'iPhone ne nous permet hélas pas de tester le comportement de l'appareil photo. Sachez toutefois que vous pouvez très facilement modifier le code précédent afin de démarrer le contrôleur d'images sur ce type de vue. Il vous suffit en effet de définir la propriété `sourceType` de l'instance de la classe `UIImagePickerControllerController`, comme au Listing 8.9 :

```
- (void)viewDidLoad {  
    if([UIImagePickerController isSourceTypeAvailable:UIImagePickerContro  
llerSourceTypeCamera]) {  
        UIImagePickerController *vuePhotos = [[UIImagePickerController  
alloc] init];  
        vuePhotos.delegate = self;  
        vuePhotos.sourceType = UIImagePickerControllerSourceTypeCamera;  
        [self presentViewController:vuePhotos animated:YES];  
        [vuePhotos release];  
    }  
}
```

Vous ne constaterez donc aucun changement dans le Simulateur d'iPhone, mais cette méthode lance la vue typique de la capture de photos (voir Figure 8.23). Sachez par ailleurs que vous pouvez ajouter vos propres photos dans le simulateur afin de tester votre application en situation réelle. Pour cela, copiez le dossier `/100APPLE` présent à la racine d'un vrai iPhone dans le répertoire `~/Bibliothèque/Application Support/iPhone Simulator/User/Media/DCIM`. Pratique pour enrichir rapidement la photothèque du simulateur !

Figure 8.23

Contrôlez l'appareil photo de l'iPhone de vos utilisateurs et enregistrez leurs images au sein de votre application.



L'audio et la vidéo

Déclencher un son

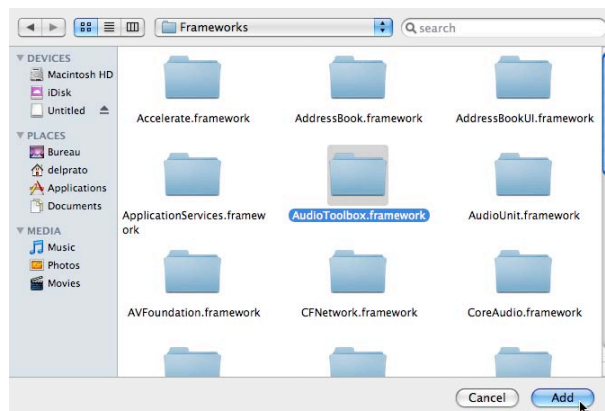
L'iPhone fait la part belle au son et il existe de nombreux frameworks capables de gérer l'aspect audio de votre application. On en repère cinq principaux, associés à des rôles bien spécifiques :

- *Media Player* délivre un lecteur audio/vidéo prêt à l'emploi. C'est un framework idéal pour soutenir la lecture de fichiers MP3, de podcasts ou de livres audio.
- *AVFoundation* permet essentiellement de lire et d'enregistrer une source audio en quelques lignes d'initialisation.
- *Audio Toolbox* est un framework très puissant pour déclencher des sons et les synchroniser à des actions, mais aussi pour les convertir à la volée et les enregistrer.
- *Audio Unit* gère essentiellement les plug-ins de traitement audio.
- *OpenAL* est un framework avant tout destiné aux jeux vidéo.

Tous les frameworks présentent leurs avantages et leurs inconvénients. Il n'est pas nécessaire de les connaître tous sur le bout des doigts : si l'aspect audio joue un rôle important sur l'iPhone, notamment pour avertir l'utilisateur ou enrichir l'ergonomie d'une application, vous ne devez pas y céder en permanence. Tâchez de justifier l'intégration du son à vos projets en lui conférant un objectif précis. Pour ajouter un framework à votre projet, cliquez sur le dossier /Frameworks dans le volet gauche de Xcode tout en maintenant la touche Ctrl du clavier. Choisissez Add puis Existing Frameworks. Sélectionnez le dossier /Frameworks, puis importez l'un des frameworks, AudioToolbox.framework par exemple (voir Figure 8.24).

Figure 8.24

*L'ajout du Framework
AudioToolbox à notre projet.*



Vous devez ensuite importer le fichier de classes dans votre contrôleur, comme par exemple :

```
# import <AudioToolbox/AudioServices.h>
```

Afin de tester l'initialisation d'une classe audio du kit de développement, nous allons bâtir une application autour du chant des oiseaux. Il s'agit d'une vaste encyclopédie du monde animal, qui présente à l'utilisateur une sélection de spécimens en associant à chaque fiche descriptive un bouton déclenchant son "cri". C'est le modèle de l'application *Cui-Cui! Chants d'oiseaux d'Europe*, d'iSpiny, ou de l'*AudioZoo* de Bas Meijer (voir Figure 8.25).

Figure 8.25

Comme son nom l'indique,
l'application AudioZoo
fait la part belle aux
sons en tout genre !



Dans cet exemple, le son correspond à un fichier WAV que l'on importe au sein de notre projet, comme nous l'avons fait précédemment pour des images ou des listes de propriétés. L'iPhone supporte les formats suivants :

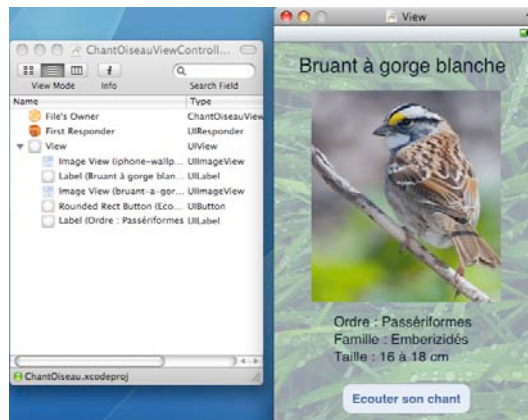
- MP3, le format audio le plus répandu ;
- AAC (*Advanced Audio Coding*), format audio offrant la qualité du MP3 à un débit inférieur ;
- ALAC (*Apple Lossless Audio Codec*), format audio sans perte de qualité ;
- HE-AAC (*High Efficiency Advanced Audio Coding*) ;
- AMR (*Adaptive Multi-Rate*) ;
- iLBC (*Internet Low Bitrate Codec*), un format adapté à la simple voix ;
- IMA4, un format compressé décodé par le processeur de l'iPhone ;
- Linear PCM, le format audio non-compressé (WAV et AIFF) ;
- μ -law et a-law.

Vous vous contenterez très probablement de sources en WAV, MP3, AAC et AIFF. Le chargement de fichiers s'articule systématiquement autour du même canevas, quel que soit le format choisi.

Sous Xcode, créez un nouveau projet de type "View-Based Application" et baptisez-le ChantOiseau. Commencez par importer le framework *AudioToolbox*, en suivant la procédure décrite précédemment. En parallèle, intégrez un fichier WAV correspondant au chant d'un oiseau. Dans notre exemple, il s'agit d'un bref extrait de trois secondes, que nous avons rapatrié depuis l'adresse www.randonneur.net. Nous vous laissons développer l'interface selon vos goûts. *A priori*, elle contiendra une image de l'animal considéré ainsi qu'une fiche descriptive. Nous allons ici ajouter un bouton aux contours arrondis permettant de déclencher le son (voir Figure 8.26).

Figure 8.26

Nous créons la fiche descriptive du bruant à gorge blanche, en prévoyant un bouton pour déclencher son chant.



Modifiez tout d'abord le fichier `ChantOiseauViewController.h` comme au Listing 8.12.

Listing 8.12 : *ChantOiseauViewController.h*

```
#import <UIKit/UIKit.h>
#import <AudioToolbox/AudioServices.h>

@interface ChantOiseauViewController : UIViewController {
    UIImageView *vueOiseau;
    SystemSoundID bruantGorgeBlanche;
}

@end
```

Nous avons ici défini un objet `UIView` visant à soutenir la fiche descriptive d'un "bruant à gorge blanche". Après avoir importé la classe du framework `AudioTool`, nous définissons un objet de type `SystemSoundID`. Il s'agit de l'élément de base du service audio : il renvoie vers un fichier audio. Modifiez à présent l'implémentation de votre contrôleur, en saisissant le code du Listing 8.13 dans le fichier `ChantOiseauViewController.m`.

Listing 8.13 : *ChantOiseauViewController.m*

```
#import "ChantOiseauViewController.h"

@implementation ChantOiseauViewController

- (void) chantOiseau {
    AudioServicesPlaySystemSound (bruantGorgeBlanche);
}

- (void)loadView {
    vueOiseau = [[UIView alloc] initWithFrame:[[UIScreen mainScreen]
                                                applicationFrame]];
    self.view = vueOiseau;
    [vueOiseau release];

    // Bouton déclenchant la lecture du fichier audio
    UIButton *bouton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    [bouton setFrame:CGRectMake(110, 395, 150, 30)];
    [bouton setTitle:@"Cri de l'oiseau" forState:UIControlStateNormal];
    [bouton addTarget:self action:@selector(chantOiseau) forControlEvents
        :UIControlEventsTouchUpInside];
    [self.view addSubview:bouton];

    // On définit le chemin d'accès au fichier audio
    id fichierAudio = [[NSBundle mainBundle] pathForResource:@"bruant-
        gorge_blanche" ofType:@"wav"];
    CFURLRef cheminAcces = (CFURLRef)[[NSURL alloc] initWithFileURLWithPath:
        fichierAudio];
    AudioServicesCreateSystemSoundID (cheminAcces, &bruantGorgeBlanche);
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning]; // Releases the view if it doesn't
    have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
```

```
        if(bruantGorgeBlanche) AudioServicesDisposeSystemSoundID  
                                (bruantGorgeBlanche);  
        [vueOiseau release];  
        [super dealloc];  
    }  
  
@end
```

La lecture du fichier audio n'occupe que quelques lignes de code tout au plus. Nous indiquons tout d'abord le chemin d'accès à notre extrait au format WAV :

```
id fichierAudio = [[NSBundle mainBundle] pathForResource:  
                  @"bruant-gorge_blanche" ofType:@"wav"];  
CFURLRef cheminAcces = (CFURLRef)[[NSURL alloc] initWithPath:  
                                   fichierAudio];
```

Nous créons ensuite l'objet correspondant au son à l'aide de la fonction `AudioServicesCreateSystemSoundID`. Le second paramètre correspond à l'instance d'un objet `SystemSoundID`. Nous associons l'appel de la méthode `chantOiseau` au bouton rectangulaire : lorsque l'utilisateur le presse, on déclenche la lecture du son à l'aide de la fonction `AudioServicesPlaySystemSound` (voir Figure 8.27). Simple, n'est-ce pas ?

Figure 8.27

En cliquant sur le bouton correspondant, l'utilisateur déclenche le chant de l'oiseau.



Par ailleurs, ce framework gère très simplement les vibrations de l'iPhone. Remplacez son initialisation par la ligne suivante, afin de faire trembler le téléphone de l'utilisateur :

```
AudioServicesPlaySystemSound (kSystemSoundID_vibrate);
```

Vous ne pouvez toutefois pas faire varier l'intensité de la vibration : elle correspond à une secousse brève.

Vous pouvez facilement adapter cet exemple à un autre framework lié à la gestion de l'audio, comme AVFoundation par exemple. Dans ce cas, importez le framework correspondant puis remplacez l'implémentation de votre contrôleur avec le code suivant :

```
# import <AVFoundation/AVFoundation.h>
[...]
NSString *fichierAudio = [[NSBundle mainBundle] pathForResource:@"bruant-
                                gorge_blanche" ofType:@"wav"];
AVAudioPlayer *lecteurAudio = [[AVAudioPlayer alloc]
    initWithContentsOfURL:[NSURL URLWithString:fichierAudio] error:nil];
[lecteurAudio play];
```

Comme vous le constatez, l'instance de la classe AVAudioPlayer accepte directement un objet NSURL en entrée. Vous pouvez ainsi facilement charger un son à distance, à travers une requête HTTP par exemple.

Lire une vidéo

La lecture de séquences vidéo est intimement liée aux mêmes frameworks que précédemment et la procédure est toute aussi simple et rapide. On privilégie toutefois le framework *MediaPlayer*, compatible avec une large sélection de formats et de codecs (H.264, MP4, MOV, MPV...). Il crée en quelques lignes de code un lecteur vidéo qui occupe l'ensemble de l'écran de l'iPhone. Il dispose par défaut de contrôles permettant de régler le volume, d'avancer dans la séquence ou de mettre la lecture en pause (voir Figure 8.28).

Figure 8.28

Les contrôles par défaut du lecteur implémenté à partir de la classe MediaPlayer.



Commencez par importer le framework MediaPlayer comme précédemment, en cliquant sur le dossier Frameworks de votre projet tout en maintenant la touche Ctrl du clavier enfoncée. L'initialisation du lecteur vidéo et le chargement du fichier *sequence.mp4* associé à votre projet s'effectuent de la manière suivante :

```
# import <AVFoundation/AVFoundation.h>

NSString *fichierVideo = [[NSBundle mainBundle]
                           pathForResource:@"sequence" ofType:@"mp4"];
NSURL *lienVideo = [NSURL URLWithString:fichierVideo];
MPMoviePlayerController lecteurVideo = [[MPMoviePlayerController alloc]
                                          initWithContentURL:lienVideo];

[lecteurVideo play];
```

N'oubliez pas d'associer le fichier `sequence.mp4` à votre projet avant de le compiler et de saisir ces quelques lignes de code au sein de la méthode `loadView` ou `viewDidLoad` par exemple. Le résultat ne se fait pas attendre : le lecteur vidéo occupe l'ensemble de l'écran et l'utilisateur peut contrôler la diffusion de la séquence. Là encore, vous avez la possibilité de lire en *streaming* un flux vidéo récupéré à partir d'une URL. À l'instar des méthodes associées au parcours de la photothèque, vous disposez d'une méthode pour enchaîner par un second traitement lorsque la lecture de la vidéo s'arrête. Pour cela, vous devez tout d'abord définir un objet `NSNotificationCenter` correspondant au centre des notifications du système.

```
[[NSNotificationCenter defaultCenter]
 addObserver:self
 selector:@selector(moviePlaybackDidFinish:)
 name:MPMoviePlayerPlaybackDidFinishNotification
 object:lecteurVideo];
```

Lorsque la notification `MPMoviePlayerPlaybackDidFinishNotification` est envoyée, le lecteur vidéo a fini de lire la séquence chargée : on appelle alors la méthode `moviePlaybackDidFinish` :

```
- (void) moviePlaybackDidFinish:(NSNotification*)notification {
    // Insérer ici le code post-lecture
}
```

Pour une plus ample découverte des options liées au lecteur vidéo initialisé à partir du framework `MediaPlayer`, reportez-vous aux exemples de codes sources offerts par Apple, à l'adresse http://developer.apple.com/iphone/library/samplecode/MoviePlayer_iPhone. Cet exemple complet articule le lecteur vidéo autour d'un système de navigation par onglets, tout en autorisant la lecture de séquences déposées sur le Web ou associées au projet (un fichier `Movie.m4v` est inclus dans l'archive `.zip` à télécharger). Vous y découvrez notamment les options de mise à l'échelle du contenu et l'utilisation du système de notifications (voir Figure 8.29).

Figure 8.29

Le lecteur vidéo peut aussi bien charger une séquence depuis une source externe qu'un fichier associé à votre projet.



Le dessin et les animations

Nous achevons notre tour d'horizon des éléments multimédias de l'iPhone par l'une des plus grandes couches de l'iPhone OS, *Core Graphics*. Tracé de lignes droites, dessin de formes géométriques, ajout de points : ses classes vous permettent de dessiner toutes sortes d'éléments graphiques à l'écran. C'est le moteur graphique *Quartz* qui est responsable du traitement de ces instructions et qui présente le résultat. Il fonctionne selon le principe d'un *canevas virtuel* : vous définissez tous les éléments apparaissant à l'écran et il restitue une image entière, composée des objets que vous avez déclarés en indiquant leurs coordonnées. On parle alors du *contexte graphique* : lorsque vous effectuez des traitements graphiques, vous récupérez les informations liées à la vue en cours, vous procédez à des traitements et des ajouts, puis vous mettez à jour le contexte de la vue.

Nous allons mettre en place un exemple très simple, que vous pourrez librement personnaliser avec l'ensemble des fonctions de dessin de *Core Graphics*. Dans un nouveau projet de type "View-Based Application" baptisé *Peinture*, modifiez le fichier *xib* en ajoutant une large instance de la classe *UIImageView*. Elle correspond au canevas sur lequel l'application va dessiner. Modifiez ensuite l'interface *PeintureViewController.h* comme au Listing 8.14.

Listing 8.14 : *PeintureViewController.h*

```
#import <UIKit/UIKit.h>

@class VueDessin;

@interface PeintureViewController : UIViewController {
    IBOutlet VueDessin *vueDessin;
}
@end
```

Comme vous pouvez le voir, nous avons défini la classe `VueDessin`. On lui associe un outlet : il correspondra à l'instance de la classe `UIImageView` défini dans le fichier nib. Déroulez le menu File, cliquez sur New File et choisissez `UIView` subclass à la section Cocoa Touch Classes. Saisissez `VueDessin` en guise de nom et validez la création de votre classe. Saisissez ensuite le code du Listing 8.15 dans le fichier `VueDessin.h`.

Listing 8.15 : *VueDessin.h*

```
#import <Foundation/Foundation.h>

@interface VueDessin : UIView {
}
@end
```

Nous importons ici la bibliothèque Foundation afin de dessiner à l'écran de l'iPhone. Complétez la définition de votre classe en modifiant le fichier `VueDessin.m` avec le code du Listing 8.16.

Listing 8.16 : *VueDessin.m*

```
#import "VueDessin.h"

@implementation VueDessin

- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame]) {
        // Initialization code
    }
    return self;
}

- (void)drawRect:(CGRect)rect {
    CGContextRef contexte = UIGraphicsGetCurrentContext();
    CGContextSetLineWidth(contexte, 4.0);
    CGContextSetStrokeColorWithColor(contexte,
        [UIColor blueColor].CGColor);
    CGContextMoveToPoint(contexte, 300, 300);
    CGContextAddLineToPoint(contexte, 50, 50);
    CGContextStrokePath(contexte);
}

- (void)dealloc {
    [super dealloc];
}

@end
```


Ouvrez à nouveau le fichier nib et associez l'outlet `vueDessin` à l'objet `UIImageView`. En parallèle, sélectionnez cet objet puis ouvrez l'inspecteur d'identité (Cmd+4) et saisissez `VueDessin` dans le champ *Class* (voir Figure 8.30). Enregistrez tous les fichiers et compilez le projet : sur l'écran de l'iPhone, une ligne oblique bleue de quatre pixels d'épaisseur apparaît sur la vue (voir Figure 8.31).

Figure 8.30

*N'oubliez pas de modifier la classe de votre objet `UIImageView` sous *Interface Builder*.*

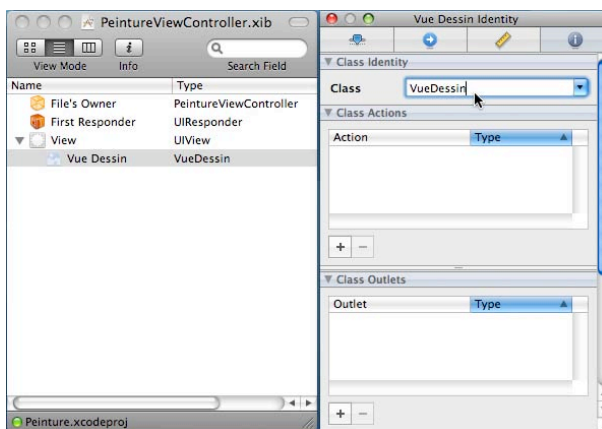


Figure 8.31

Notre programme dessine un trait entre deux points.



Le dessin est défini dans la méthode `drawRect`. Nous commençons par définir le contexte graphique :

```
CGContextRef contexte = UIGraphicsGetCurrentContext();
```

On définit ensuite l'allure de notre "pinceau", en choisissant une épaisseur de quatre pixels et un trait de couleur bleue :

```
CGContextSetLineWidth(contexte, 4.0);
CGContextSetStrokeColorWithColor(contexte, [UIColor blueColor].CGColor);
```

On atteint tout d'abord le point situé aux coordonnées (300, 300) et on trace une ligne vers le point (50, 50). Le même système de coordonnées s'applique au dessin : le point d'origine se situe dans le coin supérieur gauche.

```
CGContextMoveToPoint(contexte, 300, 300);
CGContextAddLineToPoint(contexte, 50, 50);
```

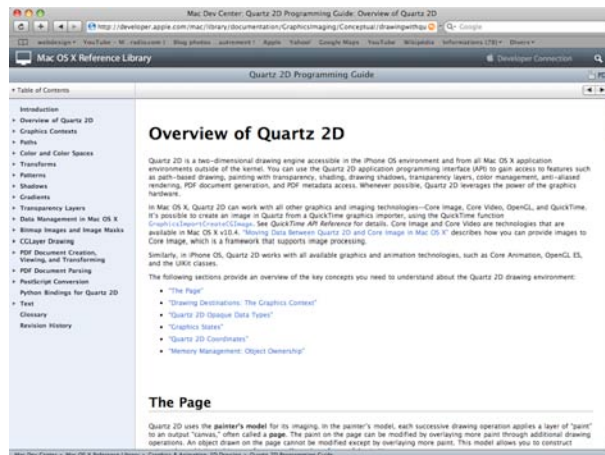
On trace enfin le contexte :

```
CGContextStrokePath(contexte);
```

Reportez-vous au Guide de programmation Quartz, sur l'espace développeur du site d'Apple, pour découvrir plus en détails les autres fonctions de dessin. Elles obéissent toutes au même principe et vous pouvez désormais les ajouter à la méthode `drawRect` afin de passer rapidement en revue cette série d'exemples (voir Figure 8.32).

Figure 8.32

Consultez le Guide de programmation pour découvrir vos autres possibilités de dessin.



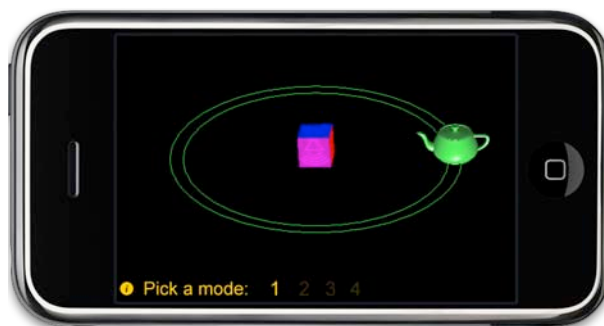
Pour aller plus loin

Vous êtes désormais en mesure d'ajouter de nombreux éléments multimédias à votre application afin de l'enrichir. Le parcours puis le traitement des photographies, et plus généralement des images, sont au centre de vos besoins : vous ajoutez ainsi dynamiquement des éléments visuels à votre interface, qu'ils proviennent d'une source externe, de la bibliothèque personnelle de l'utilisateur ou d'un fichier associé à votre projet. La lecture de données audio ou vidéo suit le même chemin et s'intègre de manière souple à vos projets de développement.

En marge du framework *Core Graphics*, vous pouvez approfondir cet apprentissage en vous intéressant à la bibliothèque *OpenGL ES*. Cette bibliothèque open-source, spécialement adaptée aux périphériques mobiles (*Embedded Systems*), vous permet même d'envisager des effets de dessin 3D. La solution la plus simple pour découvrir les principes généraux de développement avec OpenGL ES consiste à vous reporter sur les exemples de codes sources, diffusés gratuitement sur l'espace développeur du site d'Apple. À l'heure où nous mettons sous presse, vous découvrirez huit projets complets de dessin 3D, dont l'affichage d'un cube que vous manipulez au doigt (voir Figure 8.33). Si tous ces exemples dépassent le cadre de notre ouvrage par leur complexité, ils constituent un excellent moyen de prolonger votre enseignement si vous vous spécialisez dans les effets graphiques.

Figure 8.33

Testez le projet MusicCube, diffusé gratuitement sur le site d'Apple, pour visualiser des effets de dessin 3D.



Puisque le multimédia n'a plus de secrets pour vous, penchons-nous à présent sur l'ultime aspect à maîtriser avant de diffuser vos applications sur l'App Store d'iTunes : la réaction aux gestes de l'utilisateur. Accéléromètre, basculement automatique vers le mode d'affichage Paysage, contrôle d'une vue au doigt : accordez un contrôle maximal à vos utilisateurs.



9

Réagir aux gestes de l'utilisateur

Au sommaire de ce chapitre

- Gérer la rotation de l'affichage
- Contrôler la manipulation de l'écran
- Pour aller plus loin

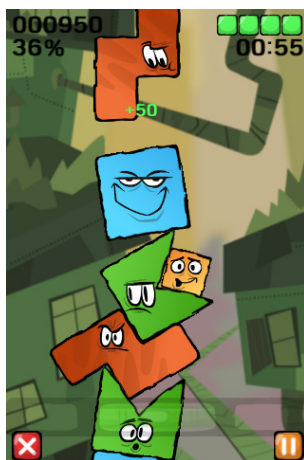
Plébiscités par les utilisateurs et repris par de nombreux constructeurs, l'écran multi-touch et l'accéléromètre de l'iPhone offrent des contrôles avancés et introduisent tout type de manipulations au sein de vos applications. Du modeste utilitaire au jeu vidéo grandiose, tous les projets gagnent à exploiter à fond ces possibilités afin d'accroître leur ergonomie

et de présenter un contrôle intuitif aux utilisateurs. L'accéléromètre constitue à lui seul un élément fondamental de l'iPhone ; il est même à l'origine du concept de nombreuses applications, comme le niveau à bulles ou la plupart des jeux d'adresse par exemple. Il introduit de nouvelles dimensions à vos projets, en vous donnant la possibilité de repérer un iPhone dans l'espace et de relever la vélocité que l'utilisateur lui applique. La simple évocation de ces concepts fait immédiatement jaillir des dizaines d'idées de projets, que vous êtes désormais largement en mesure de développer grâce à l'expérience que vous avez accumulée.

En développant pour un périphérique mobile, vous connaissez précisément le cadre dans lequel s'exécuteront vos applications. L'espace de l'écran est borné, le matériel embarqué est connu et aucun utilisateur ne pourra se prévaloir d'un affichage plus grand ou d'un processeur plus puissant. Vous avez ainsi la possibilité de prévoir des comportements riches, sans vous soucier d'innombrables configurations subalternes ou de résolutions supérieures. C'est un véritable atout : vous pouvez ainsi développer les réactions de votre application face à l'ensemble des gestes et manipulations que les utilisateurs seront susceptibles d'effectuer. Au premier rang de celles-ci, vous devez prévoir le basculement automatique vers la vue Paysage, l'un des modes de visualisation privilégié pour la lecture et la saisie de longs textes. Mais vos possibilités ne s'arrêtent pas là ! Zooms à deux doigts, défilement de vues du bout de l'index, inclinaison de l'iPhone... Envisagez tout type d'interactions en repérant précisément les gestes des utilisateurs. Dans une large mesure, tous nos exemples précédents s'inscrivent d'ores et déjà dans un tel élan. La pression sur un bouton déclenche une action et le parcours de la photothèque s'effectue du bout des doigts, par exemple. Au cours de ce chapitre, nous allons passer en revue l'ensemble de vos possibilités et découvrir comment introduire des réactions plus avancées en fonction des manipulations des utilisateurs. À l'image du très populaire *Topple*, un jeu d'adresse dans lequel vous empilez des briques, vos applications gagneront ainsi une nouvelle dimension (voir Figure 9.1).

Figure 9.1

Le jeu Topple fait un usage constant de l'accéléromètre et offre un plaisir vidéo ludique immédiat.

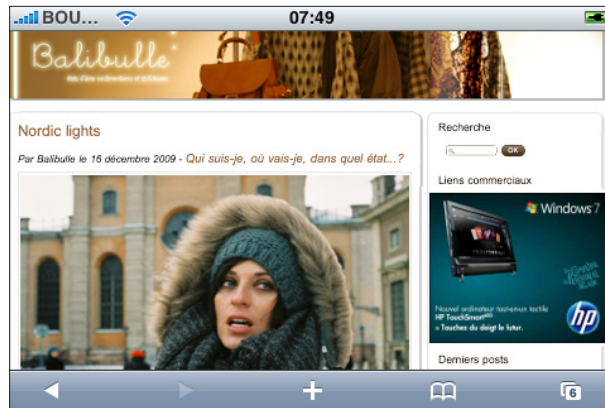


Gérer la rotation de l'affichage

Vous l'avez constaté en naviguant sur le Web avec Safari ou en parcourant vos photos personnelles : l'iPhone bascule automatiquement l'affichage entre le mode portrait (par défaut) et Paysage lorsque vous le manipulez (voir Figure 9.2). Vous pouvez même prévoir jusqu'à quatre vues différentes, en fonction de la rotation du mobile vers la gauche, la droite ou à 180 degrés. Certaines applications restreignent le mode de visualisation à l'une de ces vues ; c'est notamment le cas du lecteur multimédia, qui ne s'exécute qu'en mode paysage, ou de la liste des contacts, que vous ne pouvez parcourir qu'en mode portrait.

Figure 9.2

Le navigateur web prend automatiquement en compte la rotation de l'iPhone.



Le kit de développement de l'iPhone vous aide à agencer vos éléments d'interface en fonction de l'orientation du mobile. Il dispose d'une méthode `shouldAutorotateToInterfaceOrientation` qui relève le mode de visualisation actuel et autorise ou non le basculement vers un autre type d'affichage. Vous l'avez compris : en activant le basculement automatique, vous devez alors prévoir l'agencement de vos objets pour chaque mode considéré. Cette méthode est définie par défaut dans tous les projets que vous créez sous Xcode, mais elle est commentée et n'est donc pas compilée. Sa structure obéit à la nomenclature suivante :

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

À chaque fois que l'utilisateur réoriente son mobile, cette méthode est appelée et on vérifie ainsi que le nouveau mode d'affichage est possible. On associe quatre constantes aux modes d'orientation de l'iPhone :

- `UIInterfaceOrientationPortrait`, mode portrait ;
- `UIInterfaceOrientationLandscapeLeft`, mode paysage vers la gauche ;
- `UIInterfaceOrientationLandscapeRight`, mode paysage vers la droite ;
- `UIInterfaceOrientationPortraitUpsideDown`, mode portrait à 180 degrés ;

En supprimant cette méthode ou en laissant les commentaires, vous empêchez ainsi le basculement vers un autre mode d'orientation. À l'inverse, vous spécifiez les modes supportés à travers la valeur retournée par cette méthode. Ainsi, la déclaration suivante gère le basculement automatique vers les deux modes paysage et le mode portrait par défaut :

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    return (interfaceOrientation == UIInterfaceOrientationPortrait
        || interfaceOrientation == UIInterfaceOrientationLandscapeLeft
        || interfaceOrientation == UIInterfaceOrientationLandscapeRight);
}
```

Modifiez la déclaration de cette méthode comme précédemment sur l'un de vos projets puis compilez-le. Dans le Simulateur d'iPhone, déroulez le menu Matériel et cliquez sur Rotation à gauche ou Rotation à droite. Les contrôles se réarrangent dans un léger effet d'animation. Mais le basculement n'est pas toujours gracieux : dans notre exemple *ChantOiseau* du Chapitre 8, les libellés et la photo de l'oiseau se chevauchent, tandis que le bouton déclenchant l'extrait audio n'apparaît plus à l'écran (voir Figure 9.3). Vous devez indiquer précisément la position de tous vos éléments d'interface afin d'éviter ce type de problème. En fonction de la complexité de votre application, vous choisirez de modifier la vue à travers Interface Builder ou la propriété `frame` de vos variables d'instances, en Objective-C. Dans ce dernier cas, vous indiquez précisément les coordonnées de tous vos objets, comme nous l'avons vu au chapitre précédent.

Figure 9.3

L'orientation en mode paysage fait disparaître certains contrôles, en l'absence de propriétés spécifiques.

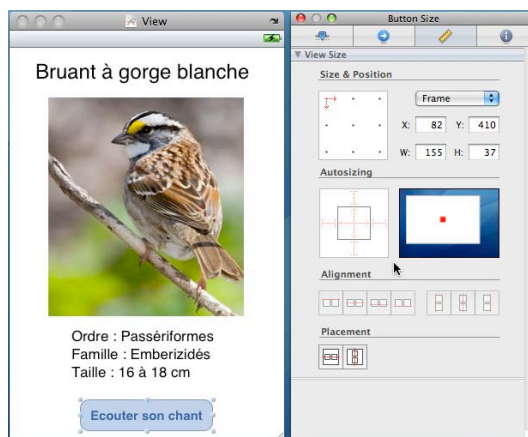


L'inspecteur de dimensions d'Interface Builder vous permet de prévoir ce type de comportement et de retoucher visuellement l'agencement de vos contrôles. Dans le cas d'interfaces

simples qui s'articulent autour d'une série de boutons et de libellés, c'est une technique rapide et efficace qui conduit à de bons résultats. Ouvrez le fichier nib associé à votre projet et faites apparaître cet inspecteur en pressant les touches Cmd+3. Commencez par sélectionner l'un des objets figurant sur votre interface, comme le libellé principal par exemple (voir Figure 9.4).

Figure 9.4

L'inspecteur de dimensions de l'élément considéré, sous Interface Builder.



Observez attentivement la section Autosizing. Le schéma sur la gauche vous indique les propriétés de redimensionnement de l'objet. Sur la droite de l'écran, vous découvrez comment se traduisent ces propriétés à l'affichage. Cet outil est directement hérité du développement d'applications Cocoa pour Mac OS X et définit ainsi les principales caractéristiques de chaque élément d'interface lorsque la vue à laquelle ils sont rattachés change d'aspect et de dimensions. Vous devez cliquer sur les flèches de part et d'autre de la boîte, sur la gauche de l'écran, pour modifier l'aspect de vos éléments. Le carré au centre du schéma représente l'objet actuellement sélectionné. Les flèches horizontales et verticales définissent ses dimensions : si elles apparaissent dans un trait continu, vous indiquez que la largeur et la hauteur de votre élément doit s'adapter à l'espace disponible. Cliquez sur les flèches : elles apparaissent en pointillés et vous exigez ainsi que l'élément conserve rigoureusement ses dimensions. En sélectionnant un libellé et en lui associant une flèche horizontale continue et une flèche verticale en pointillés, vous choisissez alors de redimensionner automatiquement sa largeur mais de préserver la hauteur que vous avez définie (voir Figure 9.5).

Les traits à l'extérieur du carré correspondent à l'espace entre l'objet sélectionné et les quatre bords de la vue qui le soutient. Si un trait apparaît en pointillés, vous indiquez que la distance vis-à-vis du bord correspondant doit être librement réadaptée : l'objet "flotte" ainsi dans l'interface et sa position s'adapte à l'espace disponible. À l'inverse, si un trait continu encadre votre objet, vous conservez la distance qui le sépare du bord de l'interface. En autorisant le redimensionnement d'un objet tout en adaptant librement l'espace qui le sépare des

bords de l'interface, vous vous assurez ainsi qu'il apparaît à l'écran. Le concept vous paraît abstrait ? Passez tout de suite à la pratique, en sélectionnant le bouton déclenchant le cri de l'oiseau, dans l'application *ChantOiseau*. Situé tout en bas de l'interface, il disparaissait de l'écran en mode paysage. Cliquez sur le trait en pointillés, en-dessous du rectangle : il se transforme en une ligne continue et vous indiquez ainsi que l'espace inférieur doit être préservé – votre contrôle apparaît tout en bas de l'écran de l'iPhone, lorsqu'on lui imprime une rotation (voir Figure 9.6).

Figure 9.5

L'élément conservera sa hauteur lors d'une rotation, mais adaptera librement sa largeur à l'espace disponible.

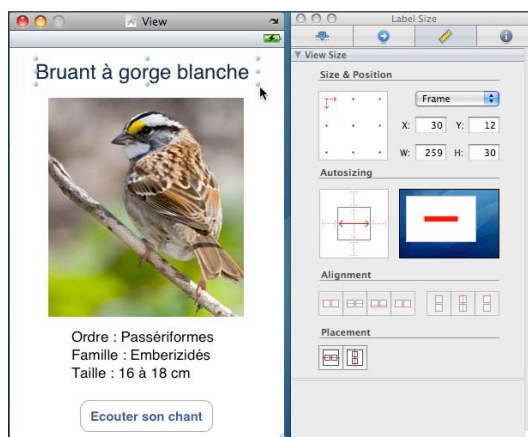
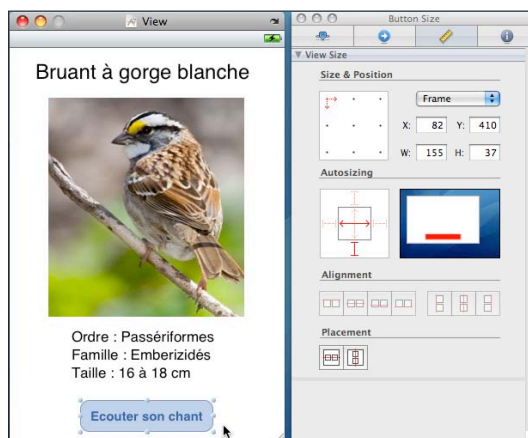


Figure 9.6

Dans ce mode, on redimensionne automatiquement la largeur du bouton mais on maintient l'espace qui le sépare du bas de l'écran.



Cette technique est particulièrement rapide pour les interfaces les plus simples : vous indiquez facilement les éléments à redimensionner et vous vous assurez qu'aucun contrôle ne sort de la vue principale. Elle vous oblige toutefois à un incessant va-et-vient entre le Simulateur d'iPhone et Interface Builder, jusqu'à valider le positionnement et la réorientation de

vos objets. Dans le cadre de notre application *ChantOiseau*, cette technique rencontre vite ses limites : si tous les éléments apparaissent bien à l'écran en mode paysage, ils gagneraient à être positionnés de manière plus précise. En effet, vous ne pouvez pas revoir l'emplacement des éléments du tout au tout avec Interface Builder ! Pour améliorer le résultat, vous devez impérativement vous consacrer à quelques lignes supplémentaires d'Objective-C.

Comme d'habitude, tous vos éléments d'interface doivent correspondre à des outlets pour être modifiés dans le code de votre application. Vous avez naturellement la possibilité de les initialiser directement à travers les méthodes `loadView` ou `viewDidLoad` mais nous privilégions ici la première solution : le fichier `nib` nous permet de définir précisément une interface principale complexe, à travers Interface Builder, que l'on modifie ensuite afin de supporter l'orientation de l'affichage. Comme vous le constaterez à travers cet exercice pratique, le design de la vue principale sous Interface Builder présente un grand intérêt : vous pouvez ainsi relever précisément les coordonnées de tous vos éléments d'interface afin de faciliter la modification de leur propriété `frame` dans votre contrôleur.

Commencez par préparer cette vue initiale, qui apparaîtra en mode portrait. Dans notre exemple, nous prévoyons deux libellés (pour le nom de l'oiseau et ses caractéristiques principales), une vue d'images (pour la photo de l'oiseau) et un bouton (pour lancer le fichier audio). Définissez tous les outlets et l'action `chantOiseau` comme au Listing 9.1 et associez-les à votre contrôleur comme sur la Figure 9.7.

Listing 9.1 : *ChantOiseauViewController.h*

```
#import <UIKit/UIKit.h>
#import <AudioToolbox/AudioServices.h>

@interface ChantOiseauViewController : UIViewController {
    IBOutlet UIButton *bouton;
    IBOutlet UILabel *nomOiseau;
    IBOutlet UILabel *description;
    IBOutlet UIImageView *photoOiseau;
    SystemSoundID bruantGorgeBlanche;
}

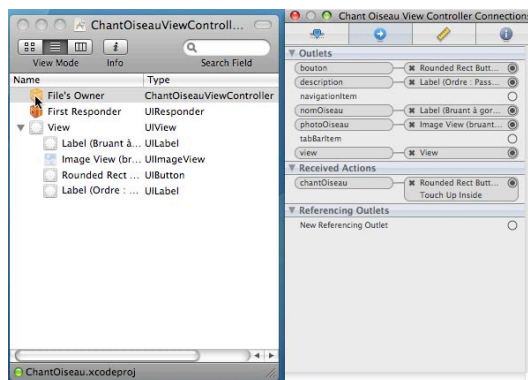
@property (retain, nonatomic) UIButton *bouton;
@property (retain, nonatomic) UILabel *nomOiseau;
@property (retain, nonatomic) UILabel *description;
@property (retain, nonatomic) UIImageView *photoOiseau;

- (IBAction)chantOiseau;

@end
```

Figure 9.7

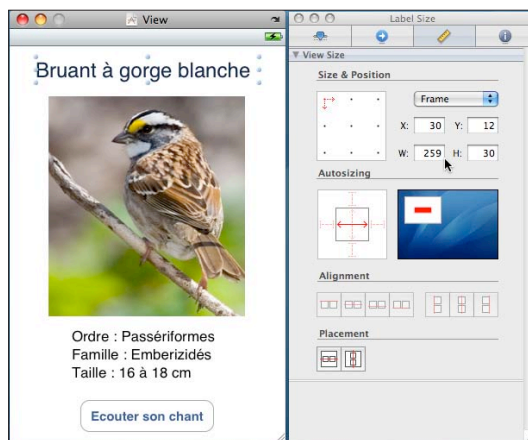
Les liaisons des outlets et actions à notre vue principale.



Comme nous l'avons vu précédemment, nous devons ensuite activer l'orientation de l'iPhone à l'aide de la méthode `shouldAutorotateToInterfaceOrientation`. La méthode `willAnimateSecondHalfOfRotationFromInterfaceOrientation` permet de positionner précisément chaque élément de l'interface en fonction de l'orientation du mobile. Commencez par relever la position actuelle de chacun de vos éléments en mode portrait : ouvrez votre fichier nib sous Interface Builder et lancez l'inspecteur de dimensions. La section "Size & Position" affiche les coordonnées et les dimensions de chaque objet d'interface (voir Figure 9.8).

Figure 9.8

Interface Builder relève à notre place la position et les dimensions de tous les objets d'interface, comme ici le libellé principal.



Le système de coordonnées change fondamentalement lorsque l'on bascule l'écran de l'iPhone dans un autre mode d'affichage. En mode paysage, vous disposez en effet de 480 pixels de largeur et de 320 pixels de hauteur pour positionner vos éléments. Essayez de dessiner sur papier la position attendue de vos objets. Relevez ensuite les valeurs correspon-

dantes pour chacun de vos outlets. Saisissez enfin le code du Listing 9.2 dans l'implémentation de votre contrôleur, `ChantOiseauViewController.m`.

Listing 9.2 : *ChantOiseauViewController.m*

```
#import "ChantOiseauViewController.h"
#import <AudioToolbox/AudioServices.h>

@implementation ChantOiseauViewController
@synthesize bouton;
@synthesize nomOiseau;
@synthesize description;
@synthesize photoOiseau;

// Méthode décrivant la position des éléments d'interface
- (void)willAnimateSecondHalfOfRotationFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
    duration:(NSTimeInterval)duration {
    UIInterfaceOrientation toOrientation = self.interfaceOrientation;
    // Vue Portrait
    if (toOrientation == UIInterfaceOrientationPortrait) {
        nomOiseau.frame = CGRectMake(70, 20, 177, 21);
        photoOiseau.frame = CGRectMake(42, 47, 234, 255);
        description.frame = CGRectMake(72, 323, 175, 72);
        bouton.frame = CGRectMake(82, 400, 155, 37);
    }
    // Vue Paysage
    else
    {
        nomOiseau.frame = CGRectMake(20, 20, 177, 21);
        photoOiseau.frame = CGRectMake(220, 25, 234, 255);
        description.frame = CGRectMake(20, 71, 175, 72);
        bouton.frame = CGRectMake(20, 200, 155, 37);
    }
}

// Action déclenchée pour lire le fichier audio (cf. chapitre 8)
- (void) chantOiseau {
    id fichierAudio = [[NSBundle mainBundle] pathForResource:@"bruant-
        gorge_blanche" ofType:@"wav"];
    CFURLRef cheminAcces = (CFURLRef)[[NSURL alloc] initWithFileURLWithPath:
        fichierAudio];
    AudioServicesCreateSystemSoundID (cheminAcces, &bruantGorgeBlanche);
    AudioServicesPlaySystemSound (bruantGorgeBlanche);
}

// On autorise l'orientation de l'écran
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)
    interfaceOrientation {
    // Return YES for supported orientations
}
```

```

        return (interfaceOrientation == UIInterfaceOrientationPortrait
            || interfaceOrientation ==
                UIInterfaceOrientationLandscapeLeft ||
                interfaceOrientation == UIInterfaceOrientationLandscapeRight);
    }

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
        // Releases the view if it doesn't have a superview
        // Release anything that's not essential, such as cached data
    }

    - (void)dealloc {
        if(bruantGorgeBlanche) AudioServicesDisposeSystemSoundID
            (bruantGorgeBlanche);

        [bouton release];
        [nomOiseau release];
        [description release];
        [photoOiseau release];
        [super dealloc];
    }

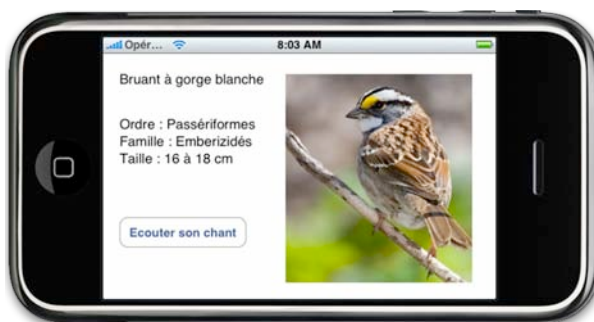
@end

```

Enregistrez puis compilez votre projet. Dans le Simulateur d'iPhone, pressez le pavé fléché en maintenant la touche Ctrl enfoncée : vous basculez automatiquement dans les autres modes d'affichage et vous constatez que le positionnement de vos objets est entièrement réadapté. Nous avons en effet défini les positions précises de chaque élément et un léger effet d'animation accompagne la rotation de l'affichage (voir Figure 9.9).

Figure 9.9

En mode paysage, les éléments d'interface changent totalement d'emplacement.



Contrôler la manipulation de l'écran

Le kit de développement de l'iPhone gère à merveille toutes les interactions de l'utilisateur sur son écran. Du simple effleurement de sa surface aux doubles ou triples pressions consécutives, en passant par les gestes de glissement ou d'écartement de deux doigts, vous disposez d'une palette très riche de comportements à relever. À travers les objets `UITouch` créés par le capteur de mouvements, vous relevez également les différentes propriétés des gestes de l'utilisateur, notamment les coordonnées du point sélectionné et le nombre de pressions d'affilée qu'il a exercées. Quatre méthodes permettent de gérer les temps forts d'un geste :

- `touchesBegan`, une première pression sur l'écran a été détectée.
- `touchesMoved`, la pression sur l'écran est maintenue et le doigt bouge.
- `touchesCancelled`, on a annulé la pression en sortant de la zone de l'écran.
- `touchesEnded`, on a finalement relevé le doigt de l'écran.

Ces quatre méthodes couvrent l'ensemble de vos besoins. Elles sont d'ailleurs intégrées à certaines classes du kit de développement : la règlette (`UISlider`) repère les gestes de l'utilisateur pour mettre à jour son aspect, par exemple. Vous devez toutefois prêter une grande attention aux éléments d'interface impliqués dans un geste ; comme nous le verrons par la suite, il est nécessaire de maîtriser le système de coordonnées de l'iPhone et surtout de définir correctement les zones touchées par l'utilisateur afin de déplacer des objets.

Commençons par un premier exemple simple, que nous compléterons par la suite. Sous Xcode, créez un nouveau projet de type View-Based Application et baptisez-le *Toucher*. Saisissez le code du Listing 9.3 dans le fichier `ToucherViewController.h`.

Listing 9.3 : *ToucherViewController.h*

```
#import <UIKit/UIKit.h>

@interface ToucherViewController : UIViewController {
    IBOutlet UILabel *messageLabel;
    IBOutlet UILabel *coordonneesLabel;
}

@property(retain, nonatomic) UILabel *messageLabel;
@property(retain, nonatomic) UILabel *coordonneesLabel;

@end
```

Nous définissons ici deux outlets correspondant à de simples libellés. Ouvrez le fichier nib associé à votre projet, déposez deux libellés sur votre interface et supprimez leur texte initial. Vous disposez ainsi de deux champs vierges, que nous mettrons à jour avec les coor-

données du point sélectionné par l'utilisateur et par le type de geste réalisé. Connectez les outlets à File's Owner comme vous en avez l'habitude, en sélectionnant respectivement `messageLabel` et `coordonneesLabel`. Saisissez à présent le code du Listing 9.4 dans l'implémentation de votre contrôleur, `ToucherViewController.m`.

Listing 9.4 : *ToucherViewController.m*

```
#import "ToucherViewController.h"

@implementation ToucherViewController
@synthesize messageLabel;
@synthesize coordonneesLabel;

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    messageLabel.text = @"Début du toucher";
    UITouch *touch = [[event allTouches] anyObject];
    // Coordonnées du point touché
    CGPoint location = [touch locationInView:self.view];
    NSString *coordMessage = [[NSString alloc]
initWithFormat:@"Coordonnées : %.0f, %.0f", location.x, location.y];
    coordonneesLabel.text = coordMessage;
    [coordMessage release];
    NSUInteger nbrPressions = [[touches anyObject] tapCount];

    switch (nbrPressions) {
        case 1:
            self.view.backgroundColor = [UIColor grayColor];
            break;
        case 2:
            self.view.backgroundColor = [UIColor blackColor];
            break;
        case 3:
            self.view.backgroundColor = [UIColor purpleColor];
            break;
        case 4:
            self.view.backgroundColor = [UIColor whiteColor];
            break;
        default:
            break;
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    messageLabel.text = @"Annulation du toucher";
    self.view.backgroundColor = [UIColor redColor];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    messageLabel.text = @"Arrêt du toucher";
}
```

```
        self.view.backgroundColor = [UIColor blueColor];
    }

    - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
        messageLabel.text = @"Déplacement détecté";
        // Coordonnées du point touché
        UITouch *touch = [[event allTouches] anyObject];
        CGPoint location = [touch locationInView:self.view];
        NSString *coordMessage = [[NSString alloc]
initWithFormat:@"Coordonnées : %.0f, %.0f", location.x, location.y];
        coordonneesLabel.text = coordMessage;
        [coordMessage release];
        self.view.backgroundColor = [UIColor greenColor];
    }

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
    }

    - (void)dealloc {
        [messageLabel release];
        [coordonneesLabel release];
        [super dealloc];
    }

@end
```

Enregistrez puis compilez votre projet. Le Simulateur d'iPhone remplace le doigt de l'utilisateur par la souris : en exerçant des pressions successives ou en maintenant le clic enfoncé, les différents messages apparaissent sur votre libellé ("Déplacement détecté", "Arrêt du toucher", etc.). En parallèle, l'arrière-plan de la vue principale est modifié en fonction du geste réalisé. Le début d'un geste et la pression maintenue, détectés par les méthodes `touchesBegan` et `touchesMoved`, créent un objet de la classe `UITouch` dont on relève les coordonnées. Elles figurent alors dans le second libellé. Vous disposez ainsi d'un mini-outil permettant de définir correctement les coordonnées de vos éléments d'interface, dans le cas d'un changement de mode d'affichage par exemple ! Enfin, nous comptons le nombre de pressions successives au sein de la méthode `touchesBegan` et nous modifions une nouvelle fois l'arrière-plan en conséquence. Une simple pression change le fond en gris, deux pressions le remplissent en noir, trois pressions font apparaître un fond violet et quatre pressions affichent un cadre blanc (voir Figure 9.10). Le nombre de pressions relevées est illimité et vous pouvez ainsi envisager des réactions très différentes au sein de votre application, en distinguant le simple "clic" du double-clic.

Un outil de dessin à main levée

Le code précédent suffit d'ores et déjà à mettre en place un outil de dessin sommaire. Comme vous pouvez le constater, on relève les coordonnées du point touché par l'utilisateur. Il vous suffit d'afficher un point à cet emplacement et de poursuivre le trait à travers la méthode `touchesMoved` pour laisser l'utilisateur dessiner du bout des doigts. Ajoutez des boutons permettant de changer de couleur ou d'épaisseur de trait et vous obtenez une application qui n'a rien à envier à l'honorable Photoshop !

Figure 9.10

L'arrière-plan et les libellés se transforment en fonction des gestes de l'utilisateur.



Déplacer un élément du bout des doigts

Complétons ce premier exemple en ajoutant une image PNG transparente, que l'utilisateur va déplacer sur l'écran du bout des doigts. Cet ajout illustre notamment la technique de détection des points communs entre plusieurs zones : l'utilisateur doit en effet toucher l'espace correspondant à l'image pour la déplacer. Les gestes exécutés en-dehors de cette zone ne doivent pas être considérés.

Cliquez sur le dossier `/Resources` de votre projet en maintenant la touche `Ctrl` enfoncée. Choisissez `Add > Existing Files` et sélectionnez une image en la copiant dans le répertoire de votre projet. Ouvrez ensuite le fichier nib et ajoutez une instance de la classe `UIImageView`. Ouvrez l'inspecteur d'attributs (`Cmd+1`) et sélectionnez votre image dans le premier menu déroulant. Adaptez éventuellement les dimensions de l'objet `UIImageView` à celles de l'image que vous avez importée. Modifiez à présent le code de `ToucherViewController.h` à l'aide du Listing 9.5.

Listing 9.5 : ToucherViewController.h

```
#import <UIKit/UIKit.h>

@interface ToucherViewController : UIViewController {
    IBOutlet UILabel *messageLabel;
    IBOutlet UILabel *coordonneesLabel;
    IBOutlet UIImageView *logoPearson;
}

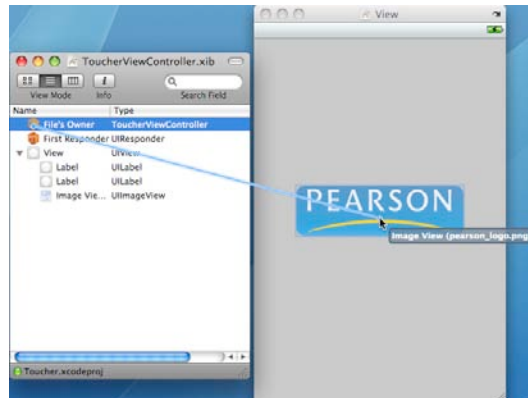
@property(retain, nonatomic) UILabel *messageLabel;
@property(retain, nonatomic) UILabel *coordonneesLabel;
@property(retain, nonatomic) UIImageView *logoPearson;

@end
```

Nous créons ici l'outlet correspondant à notre objet UIImageView. Sous Interface Builder, pressez la touche Ctrl du clavier tout en cliquant sur File's Owner et déplacez la flèche vers l'instance de la classe UIImageView. Choisissez alors `logoPearson` dans le menu contextuel qui apparaît : notre image correspond au logo de Pearson (voir Figure 9.11).

Figure 9.11

On associe les outlets à notre vue principale.



Enregistrez votre interface puis attaquez-vous au fichier `ToucherViewController.m`. Ajoutez les lignes figurant en gras au Listing 9.6.

Listing 9.6 : ToucherViewController.m

```
#import "ToucherViewController.h"

@implementation ToucherViewController
@synthesize messageLabel;
@synthesize coordonneesLabel;
@synthesize logoPearson;

```

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    messageLabel.text = @"Début du toucher";
    UITouch *touch = [[event allTouches] anyObject];
    // Coordonnées du point touché
    CGPoint location = [touch locationInView:self.view];
    NSString *coordMessage = [[NSString alloc]
initWithFormat:@"Coordonnées : %.0f, %.0f", location.x, location.y];
    coordonneesLabel.text = coordMessage;
    [coordMessage release];
    NSUInteger nbrPressions = [[touches anyObject] tapCount];

    switch (nbrPressions) {
        case 1:
            self.view.backgroundColor = [UIColor grayColor];
            break;
        case 2:
            self.view.backgroundColor = [UIColor blackColor];
            break;
        case 3:
            self.view.backgroundColor = [UIColor purpleColor];
            break;
        case 4:
            self.view.backgroundColor = [UIColor whiteColor];
            break;
        default:
            break;
    }
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
    messageLabel.text = @"Annulation du toucher";
    self.view.backgroundColor = [UIColor redColor];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    messageLabel.text = @"Arrêt du toucher";
    self.view.backgroundColor = [UIColor blueColor];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    messageLabel.text = @"Déplacement détecté";
    // Coordonnées du point touché
    UITouch *touch = [[event allTouches] anyObject];
    CGPoint location = [touch locationInView:self.view];
    NSString *coordMessage = [[NSString alloc]
initWithFormat:@"Coordonnées : %.0f, %.0f", location.x, location.y];
    coordonneesLabel.text = coordMessage;
    [coordMessage release];
    if(CGRectContainsPoint([logoPearson frame], location)) {
        self.logoPearson.center = location;
    }
}
```

```
        self.view.backgroundColor = [UIColor greenColor];
    }

    - (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
    }

    - (void)dealloc {
        [messageLabel release];
        [coordonneesLabel release];
        [logoPearson release];
        [super dealloc];
    }

@end
```

Enregistrez votre projet puis compilez-le. Notre image apparaît à l'écran. En imprimant des gestes sur la surface de l'écran, nos deux libellés sont toujours mis à jour : on détecte ainsi les coordonnées de chaque point touché. La méthode `frame` de notre objet `UIImageView` renvoie les coordonnées du cadre de cet élément d'interface.

À l'aide de la méthode `CGRectContainsPoint`, on vérifie ainsi que le point touché appartient à cette zone : si l'utilisateur a bien exercé une pression à cet endroit, tout en déplaçant son doigt, on recentre l'image. Celle-ci "suit" donc le doigt de l'utilisateur, tant qu'il pointe précisément au-dessus de sa zone d'affichage (voir Figure 9.12). Essayez de déplacer le doigt au-dessus d'un espace vierge de l'écran : l'image ne bouge pas.

Figure 9.12

*On déplace l'image
du bout des doigts.*



Gérer la navigation à partir de gestes

L'exemple précédent illustre la puissance de la classe `UITouch` et plus généralement des fonctions de détection du toucher de l'iPhone. Vous pouvez facilement implémenter les méthodes `touchesBegan`, `touchesMoved` et `touchesEnded` au sein de vos contrôleurs ou de vos classes personnalisées afin de contrôler la navigation de votre application ou d'offrir des fonctions supplémentaires à l'utilisateur. De nombreux éléments prédéfinis de l'iPhone s'articulent autour de tels comportements intuitifs ; c'est notamment le cas du parcours de la photothèque, que nous allons reproduire à travers une nouvelle application, *Diaporama*.

Comme vous allez le constater, l'enjeu est plus complexe que notre exemple précédent. S'il est facile de déplacer une instance unique de la classe `UIImageView` en repérant ses coordonnées, il est en revanche plus délicat d'animer une série de vues par simples glissements du doigt. Reportez-vous à la photothèque de votre iPhone : de part et d'autre de l'image centrale, vous apercevez la première moitié des images de gauche et droite. Celles-ci apparaissent progressivement en fonction du déplacement du doigt dans un sens ou l'autre de l'écran. Dans notre application, nous devons ainsi charger non seulement l'image centrale mais également l'image précédente et suivante, en préparant les vues correspondantes et en redéfinissant dynamiquement leur cadre en fonction des gestes de l'utilisateur.

Sous Xcode, créez un nouveau projet de type View-Based Application. Saisissez *Diaporama* en guise de nom, puis validez. Notre exemple s'articule autour d'une série d'images que l'on ajoute à notre projet (`01.jpg`, `02.jpg`, `03.jpg`, etc.). Déroulez le menu `File` et cliquez sur `New File`. Dans la fenêtre qui apparaît, sélectionnez `UIView subclass`, à la section `Cocoa Touch Classes` et cliquez sur le bouton `Next`. Saisissez `DiaporamaView` en guise de nom et validez. Il s'agit d'une instance personnalisée de la classe `UIView`, que l'on va exploiter pour créer les vues d'images et gérer les effets de transition. Commencez par saisir le Listing 9.7 dans le fichier `DiaporamaView.h`.

Listing 9.7 : *DiaporamaView.h*

```
#import <UIKit/UIKit.h>

@interface DiaporamaView : UIView {
    NSArray * images;

    UIImageView * vueImageGauche;
    UIImageView * vueImageActuelle;
    UIImageView * vueImageDroite;

    NSUInteger imageActuelle;

    BOOL transition;
    CGFloat debutTransition;
}
```

```

    }

    - (id)chargementImages:(NSArray *)inImages;

@end

```

Nous déclarons ici un tableau contenant les références aux images, trois instances de la classe `UIImageView` (pour la vue centrale et les deux vues de part et d'autre) et un entier correspondant à l'index de l'image actuellement chargée. La valeur booléenne indique si une transition doit être effectuée : elle prendra la forme d'une animation entre deux vues. Saisissez à présent le code du Listing 9.8 dans le fichier `DiaporamaView.m`.

Listing 9.8 : *DiaporamaView.m*

```

#import "DiaporamaView.h"

@implementation DiaporamaView

// Chargement initial des vues d'images
- (id)chargementImages:(NSArray *)inImages {
    if (self = [super initWithFrame:CGRectZero]) {
        images = [inImages retain];

        NSUInteger nbrImages = [inImages count];
        if (nbrImages > 0) {
            vueImageActuelle = [self creerVueImage:0];
            [self addSubview:vueImageActuelle];

            // Il y a plus de deux images : on prépare la vue suivante
            if (nbrImages > 1) {
                vueImageDroite = [self creerVueImage:1];
                [self addSubview:vueImageDroite];
            }
        }

        self.opaque = YES;
        self.backgroundColor = [UIColor blackColor];
        self.autoresizingMask = UIViewAutoresizingFlexibleWidth |
                                UIViewAutoresizingFlexibleHeight;
    }

    return self;
}

// Création des vues d'images
- (UIImageView *)creerVueImage:(NSUInteger)inImageIndex {
    if (inImageIndex >= [images count]) {
        return nil;
    }
}

```

```
// Création d'une vue
// On la charge avec l'image courante du tableau
UIImageView * vueTraitee = [[UIImageView alloc] initWithImage:
    [images objectAtIndex:indexInImageIndex]];
vueTraitee.opaque = YES;
vueTraitee.userInteractionEnabled = NO;
vueTraitee.backgroundColor = [UIColor blackColor];
vueTraitee.contentMode = UIViewContentModeScaleAspectFit;
vueTraitee.autoresizingMask = UIViewAutoresizingFlexibleWidth |
    UIViewAutoresizingFlexibleHeight;

return vueTraitee;
}

// Début du toucher : on repère le point pressé au départ
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    if ([touches count] != 1)
        return;

    debutTransition = [[touches anyObject] locationInView:self].x;
    transition = YES;

    vueImageGauche.hidden = NO;
    vueImageActuelle.hidden = NO;
    vueImageDroite.hidden = NO;
}

// Déplacement détecté
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    // On cesse le traitement si on détecte plusieurs pressions
    //successives
    if (! transition || [touches count] != 1)
        return;

    // On calcule la distance du déplacement
    CGFloat distanceDeplacement = [[touches anyObject]
        locationInView:self].x - debutTransition;

    // On repère les dimensions de la vue actuelle
    CGSize tailleVue = self.frame.size;

    // On modifie dynamiquement le cadre des vues afin de faire
    // apparaître des "morceaux" des images de gauche et de droite
    // au fur et à mesure du déplacement
    vueImageGauche.frame = CGRectMake(distanceDeplacement -
        tailleVue.width, 0.0f, tailleVue.width, tailleVue.height);
    vueImageActuelle.frame = CGRectMake(distanceDeplacement, 0.0f,
        tailleVue.width, tailleVue.height);
    vueImageDroite.frame = CGRectMake(distanceDeplacement +
        tailleVue.width, 0.0f, tailleVue.width, tailleVue.height);
}
```

```
// Fin du toucher : fin de l'animation et affichage complet de l'image
// centrale
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    if (! transition)
        return;

    // On repère les dimensions de la vue actuelle
    CGSize tailleVue = self.frame.size;

    // Doit-on afficher une image à gauche et à droite ?
    // S'il s'agit de la dernière image, on ne prépare pas
    // la vue de droite.
    NSUInteger nbrImages = [images count];

    // Si l'utilisateur a déplacé le doigt sur plus de 100 pixels
    // vers la droite, on remplace la vue centrale par l'image de droite.
    CGFloat distanceDeplacement = [[touches anyObject]
                                   locationInView:self].x - debutTransition;
    if (imageActuelle > 0 && distanceDeplacement > 100.0f) {
        [vueImageDroite removeFromSuperview];
        [vueImageDroite release];

        vueImageDroite = vueImageActuelle;
        vueImageActuelle = vueImageGauche;

        // La vue centrale précédente devient la vue de gauche
        imageActuelle--;
        if (imageActuelle > 0) {
            vueImageGauche = [self creerVueImage:imageActuelle - 1];
            vueImageGauche.hidden = YES;

            [self addSubview:vueImageGauche];
        }
        else {
            vueImageGauche = nil;
        }
    }

    // Si l'utilisateur a déplacé le doigt sur plus de 100 pixels
    // vers la gauche, on remplace la vue centrale par l'image de gauche.
    else if (imageActuelle < nbrImages - 1 && distanceDeplacement <
            -100.0f) {
        [vueImageGauche removeFromSuperview];
        [vueImageGauche release];

        vueImageGauche = vueImageActuelle;
        vueImageActuelle = vueImageDroite;

        // La vue centrale précédente devient la vue de droite
        imageActuelle++;
        if (imageActuelle < nbrImages - 1) {
```



```
        vueImageDroite= [self creerVueImage:imageActuelle + 1];
        vueImageDroite.hidden = YES;

        [self addSubview:vueImageDroite];
    }
    else {
        vueImageDroite = nil;
    }
}

// On effectue la suite de l'animation
[UIView beginAnimations:@"transition" context:NULL];
[UIView setAnimationCurve:UIViewAnimationCurveEaseOut];
[UIView setAnimationDuration:0.3f];

// On redéfinit le cadre des vues, à l'issue de l'animation
vueImageGauche.frame = CGRectMake(-tailleVue.width, 0.0f, tailleVue.
                                width, tailleVue.height);
vueImageActuelle.frame = CGRectMake(0.0f, 0.0f, tailleVue.width,
                                tailleVue.height);
vueImageDroite.frame = CGRectMake(tailleVue.width, 0.0f, tailleVue.
                                width, tailleVue.height);

[UIView commitAnimations];

transition = NO;
}

- (void)dealloc {
    [images release];
    [super dealloc];
}

@end
```

Le code est relativement explicite. La première méthode, `chargementImages`, est appelée dès l'initialisation de notre contrôleur (nous le verrons au Listing 9.10 qui suit). Elle crée les vues de l'image au centre de l'écran et de celle qui suit immédiatement. La création d'une vue d'image passe par la méthode `creerVueImage` : on initialise un objet `UIImageView` à partir de l'image relevée dans le tableau global contenant les références aux fichiers que vous avez associés à votre projet. Ce sont les fonctions de gestion du toucher qui sont responsables de l'animation des vues. Dans la méthode `touchesBegan`, nous relevons la position du point touché par l'utilisateur. C'est la distance horizontale qu'il imprime à son mouvement qui nous intéresse : la coordonnée `x` est stockée dans la variable `debutTransition`. La méthode `touchesMoved` est appelée lorsque l'utilisateur maintient son doigt sur l'écran et qu'il se met à le déplacer. On calcule la distance parcourue et on modifie dynamiquement le cadre des trois vues (l'image centrale et les deux images de part et d'autre) en fonction de

ce déplacement. La fonction `touchesEnded` intervient lorsque l'utilisateur relève son doigt de l'écran. On calcule alors la distance totale parcourue par le doigt : si elle est supérieure à 100 pixels, on considère que l'utilisateur souhaite passer à la vue précédente ou suivante. On affiche un effet d'animation pour remplacer la vue centrale par l'image de gauche ou de droite, en fonction du sens du déplacement.

Il ne nous reste plus qu'à implémenter notre contrôleur de vue. Modifiez l'interface et l'implémentation du contrôleur `DiaporamaViewController` par le code des Listings 9.9 et 9.10.

Listing 9.9 : *DiaporamaViewController.h*

```
#import <UIKit/UIKit.h>
#import "DiaporamaView.h"

@interface DiaporamaViewController : UIViewController {
}

@end
```

Listing 9.10 : *DiaporamaViewController.m*

```
#import "DiaporamaViewController.h"
#import "DiaporamaView.h"

@implementation DiaporamaViewController

// Chargement initial du tableau d'images
// Création de l'instance DiaporamaView
- (void)loadView {
    NSArray * images = [NSArray arrayWithObjects:[UIImage
        imageNamed:@"01.jpg"], [UIImage imageNamed:@"02.jpg"],
        [UIImage imageNamed:@"03.jpg"], [UIImage imageNamed:@"04.jpg"],
        [UIImage imageNamed:@"05.jpg"], nil];
    self.view = [[[DiaporamaView alloc] chargeImages:images]
        autorelease];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}

- (void)dealloc {
    [super dealloc];
}

@end
```

Notre contrôleur de vue charge le tableau `images` à partir des fichiers que vous avez importés dans votre projet. En dernier lieu, modifiez la méthode `applicationDidFinishLaunching` du délégué de l'application, comme au Listing 9.11.

Listing 9.11 : *DiaporamaAppDelegate.m*

```
#import "DiaporamaAppDelegate.h"
#import "DiaporamaViewController.h"

@implementation DiaporamaAppDelegate

@synthesize window;
@synthesize viewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application {

    viewController = [[DiaporamaViewController alloc] init];
    viewController.view.frame = [window frame];

    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}

- (void)dealloc {
    [viewController release];
    [window release];
    [super dealloc];
}

@end
```

Tout est prêt : enregistrez votre projet et compilez-le. Dans la fenêtre du Simulateur d'iPhone, vos images apparaissent de manière horizontale, à la suite les unes des autres. Un simple déplacement du doigt sur plus de 100 pixels de longueur conduit à un effet d'animation, qui vient charger la vue précédente ou suivante (voir Figure 9.13).

Figure 9.13

Notre application affiche un diaporama dynamique, à travers lequel vous déplacez des images du bout des doigts.



Interagir avec l'accéléromètre de l'iPhone

Figure emblématique du succès de l'iPhone, l'accéléromètre qu'il intègre autorise des manipulations avancées qui jouent un grand rôle dans l'ergonomie de votre application. Grâce aux trois capteurs qui le composent, vous êtes en mesure de relever la force inertielle et l'accélération imprimées sur les trois axes perpendiculaires du mobile : vous repérez ainsi le téléphone dans l'espace et vous détectez les inclinaisons vers la gauche ou la droite (axe X), le haut ou le bas (axe Y) et l'avant ou l'arrière (axe Z).

L'accéléromètre relève en particulier la force de gravité exercée sur le mobile. En position statique, celle-ci est de 1g : seule la pesanteur terrestre s'applique à l'iPhone. Si vous provoquez un geste soudain, cette valeur augmente radicalement jusqu'à se situer aux alentours de 2g ou 2,5g. En créant une simple boucle conditionnelle qui vérifie cette valeur, vous détectez à coup sûr les secousses intentionnelles et vous prévoyez des comportements spécifiques au sein de votre application.

À l'instar de l'accès au capteur photo/vidéo de l'iPhone, vous interrogez l'accéléromètre en créant une classe qui se conforme à un protocole spécifique, `UIAccelerometerDelegate`. Vous l'initialisez puis vous relevez ses valeurs à l'aide de la méthode `sharedAccelerometer` :

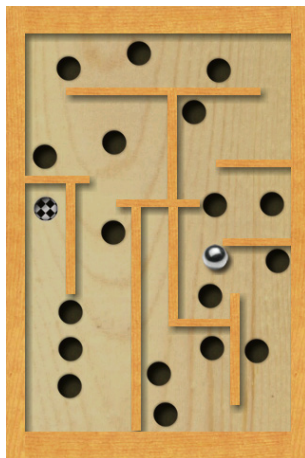
```
UIAccelerometer *accelerometre = [UIAccelerometer sharedAccelerometer];
```

Comme pour les interactions avec les gestes des utilisateurs, vous disposez ensuite d'une méthode qui relève le type d'accélération exercée, `didAccelerate`. Celle-ci renvoie l'accélération exprimée sur les trois axes vus précédemment.

Vous savez ainsi si l'utilisateur incline son mobile vers la gauche, la droite, le haut, le bas, l'avant ou l'arrière. En parallèle, la force exercée sur le mobile traduit l'accélération : on peut ainsi repérer si l'utilisateur bouge *rapidement* ou non son téléphone dans l'une des directions. Cette valeur joue sur la viscosité des effets d'animation que vous imprimez à vos objets ; en déplaçant à l'écran une instance de la classe `UIImageView` en fonction de l'inclinaison du mobile, vous pouvez ainsi accélérer ou décélérer le mouvement en fonction de cette valeur (voir Figure 9.14).

Figure 9.14

Le jeu Labyrinth utilise l'accéléromètre pour animer une balle à l'écran : la force exercée se traduit par des mouvements précis, qui vous permettent d'accélérer ou de ralentir la balle et de contourner les obstacles.



Tout ceci vous paraît encore abstrait ? Passons sans plus tarder à un exemple concret ! Nous allons déplacer une simple image sur l'ensemble de l'écran de l'iPhone, en fonction de l'inclinaison du mobile. Créez un nouveau projet sous Xcode de type View-Based Application et baptisez-le "Accelerometre". Importez une image PNG à votre projet. Notre modèle d'exemple est une image de 128 × 128 pixels (les dimensions sont importantes afin de prévoir la butée contre l'un des bords de l'iPhone), qui provient du site d'icônes gratuites www.iconspedia.com. Saisissez ensuite le code du Listing 9.12 dans le fichier `AccelerometreViewController.h`.

Listing 9.12 : `AccelerometreViewController.h`

```
#import <UIKit/UIKit.h>

@interface AccelerometreViewController : UIViewController
    <UIAccelerometerDelegate> {
    UIImageView *vueImage;
    UIImageView *personnage;
    float accelX, accelY, velociteX, velociteY;
}

@end
```

Rien de bien particulier : nous déclarons nos deux vues d'images ainsi qu'une série de nombres flottants, qui correspondent à l'accélération de l'iPhone sur ses axes X et Y. L'essentiel du traitement réside dans l'implémentation du contrôleur, que vous saisissez à partir du Listing 9.13.

Listing 9.13 : *AccelerometreViewController.m*

```
#import "AccelerometreViewController.h"

@implementation AccelerometreViewController

#define SIGNE(x) ((x < 0.0f) ? -1.0f : 1.0f)

- (void) déplacement {
// On modifie dynamiquement la position de l'objet
// UIImageView contenant l'image.
    CGRect zonePersonnage = [personnage frame];
    zonePersonnage.origin.x += velociteX;
    zonePersonnage.origin.y += velociteY;

// Si le cadre bute contre l'un des bords de l'iPhone,
// on cesse le déplacement.
    if (zonePersonnage.origin.x > 192.0f) zonePersonnage.origin.x
                                         = 192.0f;
    if (zonePersonnage.origin.x < 0.0f) zonePersonnage.origin.x = 0.0f;
    if (zonePersonnage.origin.y > 332.0f) zonePersonnage.origin.y
                                         = 332.0f;
    if (zonePersonnage.origin.y < 0.0f) zonePersonnage.origin.y = 0.0f;

    [personnage setFrame:zonePersonnage];
}

// Une accélération est détectée : on la traite
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration {

// On relève l'accélération sur les axes X et Y
    float xAccel = -[acceleration x];
    float yAccel = [acceleration y];

// On vérifie la direction du déplacement
    float directAccelX = SIGNE(velociteX) * 1.0f;
    float nouvDirectX = SIGNE(xAccel);
    float directAccelY = SIGNE(velociteY) * 1.0f;
    float nouvDirectY = SIGNE(yAccel);

// Le téléphone s'incline dans la direction
// suivie actuellement par l'image : on l'accélère.
    if (directAccelX == nouvDirectX)
```

```
        accelX = (abs(accelX) + 0.9f) * SIGNE(accelX);
        if (directAccelY == nouvDirectY)
            accelY = (abs(accelY) + 0.9f) * SIGNE(accelY);

// On modifie alors la valeur de l'accélération
// appliqué à l'image.
// Le Timer modifie la position de l'image toutes les 0,03 sec.
        velociteX = -accelX * xAccel;
        velociteY = -accelY * yAccel;
    }

// Chargement initial de la vue
- (void)loadView {
    vueImage = [[UIImageView alloc] initWithFrame:[UIScreen mainScreen]
                                                    applicationFrame]];
    self.view = vueImage;
    [vueImage release];

// Création de la vue contenant l'image
    personnage = [[UIImageView alloc]
                  initWithFrame:CGRectMake(25, 153, 128, 128)];
    [personnage setImage:[UIImage imageNamed:@"personnage.png"]];
    [vueImage addSubview:personnage];
    [personnage release];

// On définit la position d'origine de l'image.
    [personnage setCenter:CGPointMake(150, 150)];

    accelX = 2.0f;
    accelY = 2.0f;
    velociteX = 0.0f;
    velociteY = 0.0f;

// On active l'écoute de l'accéléromètre
    [[UIAccelerometer sharedAccelerometer] setDelegate:self];

// On définit un Timer qui va exécuter toutes les 0,03 sec
// le déplacement de l'image.
    [NSTimer scheduledTimerWithTimeInterval: 0.03f
                                target:self
                                selector:@selector(deplacement)
                                userInfo:nil
                                repeats:YES];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as cached data
}
```

```
- (void)dealloc {  
    [vueImage release];  
    [super dealloc];  
}  
  
@end
```

Enregistrez le programme. Cette fois, il est impossible de le tester directement dans le Simulateur d'iPhone. Vous devez générer un certificat de développeur et associer le profil de provisionnement à votre propre iPhone, avant de le compiler vers votre mobile à l'aide de Xcode. Reportez-vous au Chapitre 10 pour découvrir la procédure en détails. Dans notre exemple, nous avons créé une instance `UIImageView` contenant notre image de départ. Après avoir activé l'écoute de l'accéléromètre, un objet `NSTimer` appelle la méthode `déplacement` toutes les 0,03 seconde. Celle-ci modifie le point d'origine du conteneur `UIImageView` en fonction de l'inclinaison détectée par l'accéléromètre. Si l'image atteint l'un des bords (soit 320 pixels en largeur et 460 pixels en hauteur moins les dimensions de notre image de 128 × 128 pixels), sa position se maintient.

La méthode `didAccelerate` est chargée de relever les valeurs de l'accéléromètre. Les méthodes `x` et `y` renvoient l'accélération sur les axes X et Y. On vérifie tout d'abord le sens de l'accélération, en fonction du signe de ces deux valeurs. Si l'inclinaison correspond bien au sens de déplacement actuel de l'image, on modifie les valeurs de son accélération. À la prochaine exécution de la méthode `déplacement`, l'image se repositionne en conséquence. Ainsi, lorsque vous inclinez le téléphone vers le haut, la variable `yAccel` contient une valeur positive. On l'applique sous forme de coefficient à la variable `velocityY`. Attention, comme il s'agit de déplacer l'image vers le haut, nous devons ainsi inverser sa valeur – `velocityY` contient une valeur négative, ce qui se traduit à l'écran par un déplacement vers le haut de l'objet `UIImageView` (voir Figure 9.15).

Figure 9.15

Le déplacement de l'image correspond à l'inclinaison de l'iPhone.



Pour aller plus loin

En gérant l'accéléromètre ou le capteur de l'écran de l'iPhone, vous offrez une toute nouvelle dimension à vos applications. Vous améliorez considérablement leur prise en main, en réinventant totalement des contrôles plus intuitifs. Inspirez-vous des exemples de ce chapitre pour définir une ergonomie d'un ordre nouveau : vous pouvez ainsi envisager tout type d'interactions et sortir largement des sentiers battus. Sans exploiter coûte que coûte ces fonctions au sein de tous vos projets, sachez qu'elles font souvent la différence entre deux projets opérant dans le même domaine. Lecteur multimédia à déclencher en une secousse, application de dessin à manipuler du bout des doigts ou jeu vidéo ambitieux : vous êtes désormais en mesure de vous attaquer aux développements les plus importants.

N'hésitez pas non plus à vous reporter aux multiples exemples diffusés sur le centre de développement d'Apple. En triant les projets en fonction de leur framework, vous remarquerez une dizaine d'exemples s'articulant autour de la classe `UITouch`. Dans le même ordre d'idée, la gestion de l'accéléromètre implique des connaissances sur la physique des éléments qui dépassent le cadre de notre ouvrage. Là encore, le centre de développement de l'iPhone vous offre un aperçu de vos possibilités. Consultez en particulier les sections User Experience et Graphics & Animation, à l'adresse <http://developer.apple.com/iphone/library>.



10

Publication et marketing

Au sommaire de ce chapitre

- Adhérer au programme de développeur iPhone
- Déployer une application iPhone
- Soumettre votre application à Apple
- Populariser votre application
- Pour aller plus loin

Vous avez enfin trouvé l'idée d'application du siècle et vous avez achevé sa programmation, en exploitant toutes vos connaissances du développement pour iPhone. Après l'avoir testée sous toutes les coutures dans le simulateur, vous bouillonnez d'impatience de la soumettre au monde entier, convaincu de son intérêt pour un large public. Félicitations ! La route

conduisant au succès n'est plus qu'à quelques clics de souris... si vous disposez toutefois d'un compte développeur afin d'introduire votre projet sur l'App Store d'iTunes.

Moyennant 79 euros, vous intégrez l'iPhone Developer Program et vous profitez d'un accès complet à l'ensemble des outils de distribution et de test d'Apple. Outre la possibilité de diffuser vos propres créations à travers iTunes Connect, ce contrat de licence vous permet aussi de compiler directement vos projets Xcode sur votre iPhone. Vous les testerez en situation réelle et vous vous assurerez de leur ergonomie sur un véritable mobile.

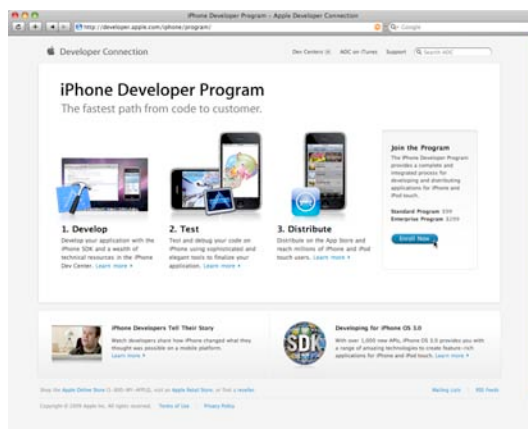
Mais votre tâche ne consiste pas uniquement à vous acquitter du montant de cette licence. Vous devez soigner tout particulièrement l'ultime étape de soumission de votre application afin qu'Apple la valide dans de brefs délais. Description détaillée, captures d'écran d'exemple, création d'un site web associé à votre projet, promotion à travers un large réseau... Le succès d'une application ne tient parfois qu'à une campagne de marketing savamment organisée. Il ne s'agit pas d'impressionner coûte que coûte vos utilisateurs et de leur promettre des avantages que votre application ne saurait tenir, mais bien de révéler sa quintessence et son intérêt fondamental.

Adhérer au programme de développeur iPhone

Vous êtes fin prêt : il est temps de souscrire une licence auprès d'Apple afin de diffuser vos applications. Rendez-vous à l'adresse <http://developer.apple.com/iphone/program> et cliquez sur le bouton Enroll Now, à droite de l'écran (voir Figure 10.1).

Figure 10.1

Achetez une licence standard à 79 euros pour distribuer vos applications.

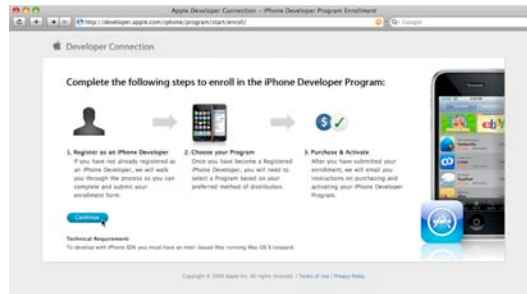


Apple propose deux licences, *standard* et *enterprise*, qui correspondent à des utilisations différentes. Le second cas n'est à réserver qu'aux entreprises comptant plus de 500 salariés, qui souhaitent inscrire leurs propres applications propriétaires dans leur cycle de produc-

tion. Ils profiteront ainsi d'options plus étendues liées à la diffusion de licences et de certificats. Choisissez le programme standard, proposé à 79 euros.

Figure 10.2

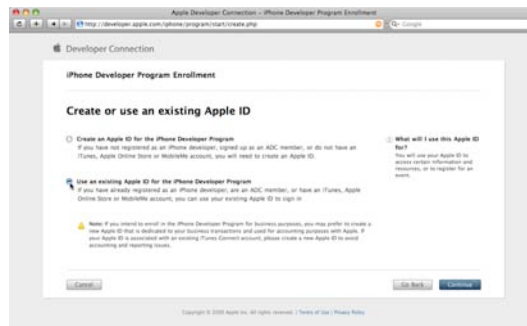
Vous devez successivement vous authentifier, choisir votre programme, puis procéder au paiement.



Sur la page suivante, cliquez sur le bouton Continuer (voir Figure 10.2). Vous devez alors vous authentifier. Vous pouvez librement utiliser votre compte Apple ID standard, mais sachez que votre nom complet figurera à côté de tous vos projets, sur l'App Store d'iTunes. Réfléchissez donc à deux fois : il peut être intéressant de créer un nouvel identifiant de revendeur, tout en conservant votre compte d'origine pour un usage plus personnel. Sélectionnez votre option, puis cliquez sur le bouton Continuer (voir Figure 10.3).

Figure 10.3

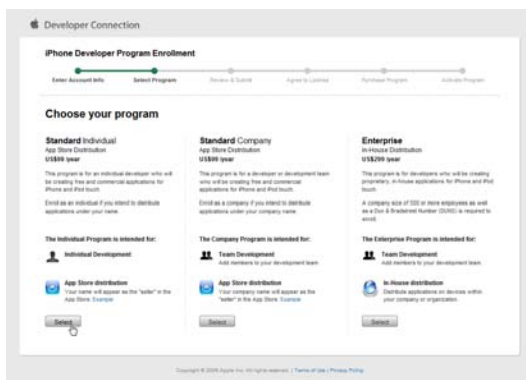
Créez un nouvel identifiant Apple ID ou utilisez votre compte actuel.



Vous devez enfin valider votre choix. Vous remarquerez qu'il existe deux types de licences, *Standard Individual* et *Standard Company*, proposées à 79 euros. La première concerne les particuliers : leur nom apparaît directement sur leur compte de développeur, tandis que pour la seconde le nom d'une société pourra apparaître. À ce stade, faites attention : dans le second cas, Apple pourra exiger des documents officiels au moment de vous rétribuer. Vous devez également opérer dans le cadre de la législation française et définir précisément la structure de votre société. Les entreprises unipersonnelles ou le statut d'auto-entrepreneur conviennent à ce type de compte, par exemple. Si vous hésitez ou si vous souhaitez diffuser rapidement des applications développées en solo, choisissez plutôt la première option (voir Figure 10.4).

Figure 10.4

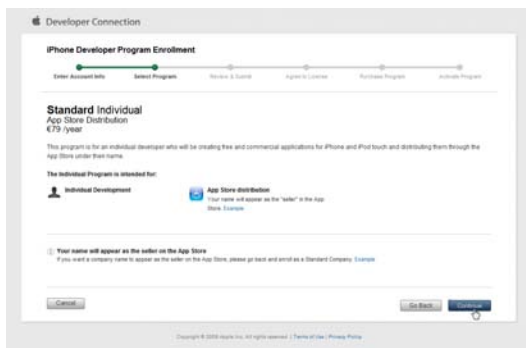
Choisissez d'adhérer au programme standard.



Le site vous résume les droits liés à votre licence et son montant. Cliquez à nouveau sur le bouton Continuer pour procéder au paiement et finaliser l'opération (voir Figure 10.5).

Figure 10.5

Vous avez adhéré au programme standard : il ne reste plus qu'à passer à la caisse !



Il y a quelques mois encore, l'achat d'une licence de développeur iPhone passait irrémédiablement par les services d'Apple USA. Après avoir créé votre compte, vous deviez indiquer vos motivations principales et même répondre à un coup de fil en provenance de Cupertino afin d'obtenir le précieux sésame. La procédure était lourde et nécessitait des semaines de délais. Vous obtenez aujourd'hui votre licence en moins de vingt-quatre heures, à travers l'Apple Store local. Cliquez sur le bouton Add to Cart pour basculer vers l'Apple Store français (voir Figure 10.6).

Figure 10.6

Si le prix de la licence est fixé à 99 dollars, vous vous acquittez en réalité de 79 euros.



Louons Apple pour avoir simplifié la procédure ! Sur l'Apple Store, vous découvrez que le montant de la licence est de 79 euros et que votre compte en banque ne sera ainsi pas débité en dollars, ce qui aurait engendré des frais de conversion de devises supplémentaires (voir Figure 10.7). Procédez au paiement et surveillez vos e-mails : vous allez recevoir un message automatique contenant un code d'activation.

Figure 10.7

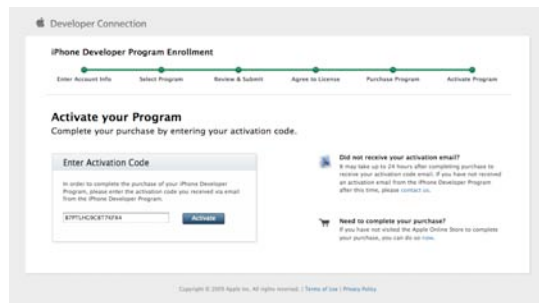
Passez commande sur l'Apple Store français, comme si vous achetiez n'importe quel accessoire Apple.



Une fois l'e-mail de l'iPhone Developer Program dans votre boîte de réception, suivez le lien qu'il contient. Vous activez ainsi votre compte (voir Figure 10.8). Prenez une bonne dernière inspiration et cliquez sur le bouton Activer : vous faites désormais partie de la grande famille des développeurs pour iPhone !

Figure 10.8

Le succès n'est plus qu'à un clic de souris.



Déployer une application iPhone

Associé au précieux sésame que vous avez acheté, votre compte de développeur s'enrichit désormais d'un module supplémentaire. Rendez-vous à l'adresse <http://developer.apple.com/iphone> et authentifiez-vous. Remarquez le bloc "iPhone Developer Program", sur la droite de l'écran : il contient un lien direct vers le portail de l'iPhone Developer Program et vers iTunes Connect. Cliquez tout d'abord sur "iPhone Developer Program Portal". La diffusion d'une application sur l'iPhone de votre choix tient en une série d'étapes que vous devrez impérativement suivre dans l'ordre :

1. La création d'un certificat de développeur, à travers l'application Trousseau d'accès de Mac OS X.
2. La signature de ce certificat par Apple.
3. L'ajout d'iPhones autorisés à installer l'application, *via* leur identifiant unique UDID.
4. La création d'un identifiant d'application, associé à chacun de vos projets.
5. La création d'un profil de provisionnement, qui lie votre certificat à un identifiant d'application et à une série d'iPhone autorisés.
6. L'installation du profil sur l'iPhone du développeur et la liaison avec Xcode.

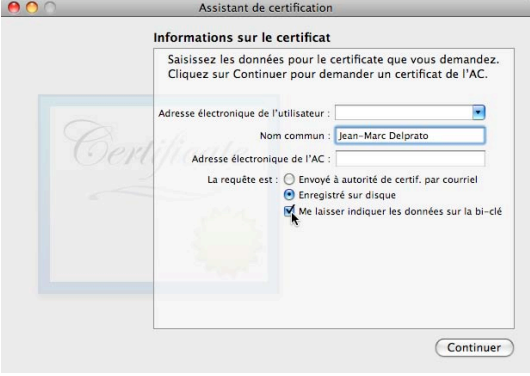
Créer un certificat de développeur

Le certificat électronique permet à Xcode de signer votre application et de gérer vos droits d'accès. Chiffré, il garantit ainsi l'authenticité d'une application et vous autorise à la tester en-dehors de l'environnement de développement. Ouvrez une fenêtre du Finder, parcourez le dossier Applications/Utilitaires et lancez le Trousseau d'accès. Déroulez le menu Trousseau d'accès et sélectionnez Assistant de certification > Demander un certificat à une autorité de certificat. Remplissez le formulaire initial en indiquant votre adresse e-mail et votre nom puis cochez les cases Enregistré sur disque et Me laisser indiquer les données sur la bi-clé (voir Figure 10.9).

Cliquez sur le bouton Continuer et validez l'enregistrement du fichier `.certSigning`. À l'écran suivant, vérifiez que la clé générée est bien basée sur l'algorithme RSA en 2 048 bits et cliquez à nouveau sur Continuer (voir Figure 10.10).

Figure 10.9

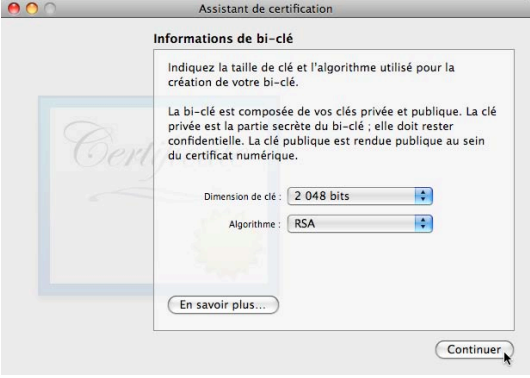
Saisissez les informations liées à votre demande de certificat.



The screenshot shows a window titled 'Assistant de certification' with a tab labeled 'Informations sur le certificat'. The text inside says: 'Saisissez les données pour le certificat que vous demandez. Cliquez sur Continuer pour demander un certificat de l'AC.' There are three input fields: 'Adresse électronique de l'utilisateur :' (with a dropdown arrow), 'Nom commun : Jean-Marc Delprato', and 'Adresse électronique de l'AC :'. Below these, it says 'La requête est :'. There are three radio buttons: 'Envoyé à autorité de certif. par courriel' (unselected), 'Enregistré sur disque' (selected), and 'Me laisser indiquer les données sur la bi-clé' (checked with a mouse cursor). A 'Continuer' button is at the bottom right.

Figure 10.10

Vérifiez le niveau de chiffrement de votre clé de certificat.

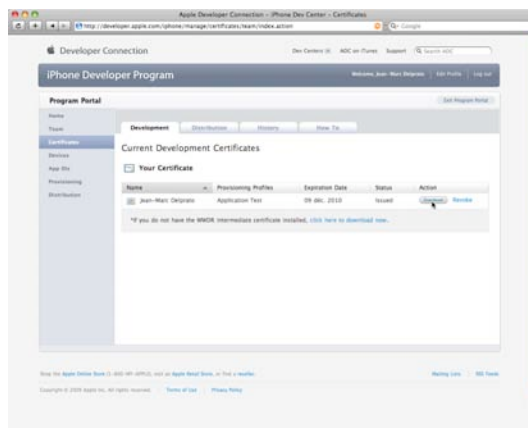


The screenshot shows the same window with a tab labeled 'Informations de bi-clé'. The text says: 'Indiquez la taille de clé et l'algorithme utilisé pour la création de votre bi-clé.' Below this, it explains: 'La bi-clé est composée de vos clés privée et publique. La clé privée est la partie secrète du bi-clé ; elle doit rester confidentielle. La clé publique est rendue publique au sein du certificat numérique.' There are two dropdown menus: 'Dimension de clé : 2 048 bits' and 'Algorithme : RSA'. A mouse cursor is pointing at the 'Continuer' button at the bottom right. There is also an 'En savoir plus...' button.

Vous devez à présent faire signer votre certificat par Apple, à travers la section Certificates du portail de développement de l'iPhone. Cliquez sur le bouton Add Certificate, tout en haut à droite. Envoyez ensuite votre demande de certificat en cliquant sur le bouton Parcourir puis en sélectionnant le certificat généré précédemment. Il apparaît à la section Current Development Certificates. Il n'est pas encore approuvé : cliquez sur le bouton Approve et patientez quelques minutes, le temps que le certificat se prépare. Vous pouvez ensuite le télécharger sur votre poste de développeur en cliquant sur le bouton Download (voir Figure 10.11).

Figure 10.11

Le certificat est approuvé : vous pouvez le télécharger sur votre poste de développeur.



Associer des iPhones enregistrés

Cliquez sur la rubrique Devices, sur le volet gauche de l'écran. Votre compte vous autorise à ajouter 100 appareils par an. Vous pouvez à tout moment supprimer ou ajouter des périphériques mobiles, mais sachez que les appareils que vous retirez de la liste comptent malgré tout parmi les cent périphériques autorisés. Connectez votre propre iPhone ou iPod Touch à votre ordinateur, puis cliquez sur le bouton Add Devices. Commencez par saisir le nom de votre périphérique dans le champ Device Name. Ouvrez iTunes, sélectionnez votre mobile dans la section Appareils puis cliquez sur l'intitulé Numéro de série. Celui-ci se remplace par un identifiant unique de 40 caractères. Cliquez sur ce numéro et pressez les touches Cmd+C pour copier cette chaîne de caractères (voir Figure 10.12). Vous obtenez la même information à travers la fenêtre Organizer de Xcode, située dans le menu Window.

Figure 10.12

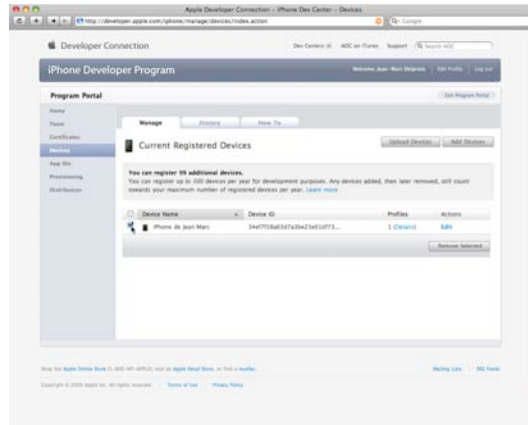
On relève l'identifiant unique associé à l'iPhone que l'on souhaite ajouter.



De retour sur le portail de développement, collez la chaîne dans le champ Device ID. Validez en cliquant sur le bouton Submit. Votre mobile figure désormais parmi les appareils enregistrés (voir Figure 10.13).

Figure 10.13

Ajoutez votre propre iPhone à la liste des appareils autorisés.

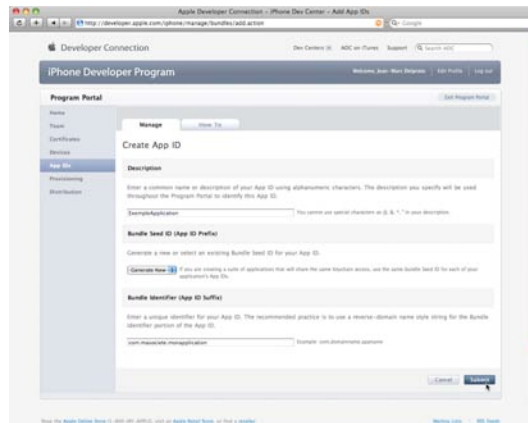


Créer un identifiant d'application

Cliquez sur la rubrique App IDs, puis sur le bouton New App ID. L'identifiant de l'application se compose d'une chaîne de dix caractères générée par Apple (*Bundle Seed ID*) et du nom de votre application, en notation de nom de domaine inversé (*Bundle Identifier*). Il prendra alors la forme d'une longue chaîne unique, comme "A1B2C3D4E5.com.masociete.monapplication" par exemple. En procédant ainsi, vous créez un identifiant unique que vous associez à une application précise. Si vous avez l'intention de tester de nombreux projets sur votre propre iPhone, vous pouvez également générer un identifiant d'application multiple, en remplaçant le Bundle Identifier par un astérisque.

Figure 10.14

Créez un identifiant d'application, que vous associez à chacun de vos projets au moment de les distribuer.



Dans la page qui apparaît, commencez par saisir un nom dans le champ Description (voir Figure 10.14). Remplissez ensuite le champ Bundle Identifier, en indiquant un identifiant complet (com.masociete.monapplication, par exemple) si vous envisagez de déployer une application précise, ou un identifiant plus large (com.masociete.*) afin d'utiliser la même clé sur une série de projets. Validez en cliquant sur le bouton Submit.

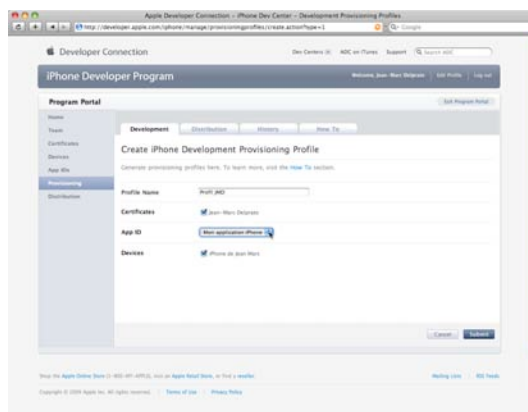
Créer un profil de provisionnement

Ultime étape, la création du profil de provisionnement lie l'un de vos périphériques enregistrés à l'identifiant d'une application. Vous pouvez ainsi utiliser votre propre appareil pour tester votre application, au-delà du Simulateur d'iPhone. Cliquez sur la rubrique Provisioning, sur le volet gauche de l'écran, puis sur le bouton New Profile (voir Figure 10.15).

Dans la mesure où vous avez rempli toutes les opérations précédentes, la tâche ne devrait pas être difficile : saisissez le nom de votre profil, puis cochez la case correspondant à votre certificat et votre appareil. Déroulez le menu App ID et sélectionnez l'identifiant d'application que vous venez de composer. Cliquez sur le bouton Submit pour valider l'opération. Vous devez ensuite télécharger le profil pour l'installer sur votre iPhone. Sur le large tableau résumant tous vos profils de provisionnement, cliquez sur le bouton Download. Vous téléchargez ainsi un fichier .mobileprovision.

Figure 10.15

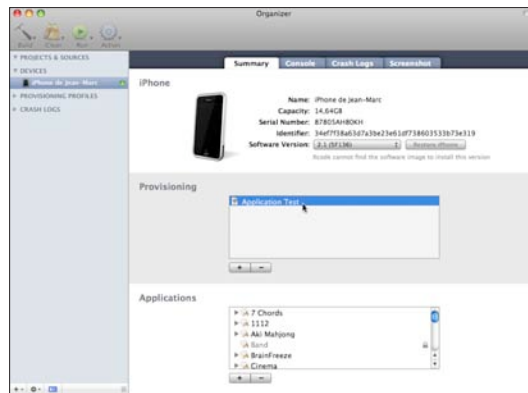
La création du profil de provisionnement lie un certificat, un identifiant d'application et un appareil enregistré.



Pour l'ajouter à votre iPhone, rien de plus simple : ouvrez Xcode, déroulez le menu Window et cliquez sur Organizer. Glissez-déposez le profil téléchargé précédemment vers la section Provisioning de votre mobile (voir Figure 10.16). Tout est prêt !

Figure 10.16

Installez enfin le profil de provisionnement sur votre propre iPhone, à travers Xcode.

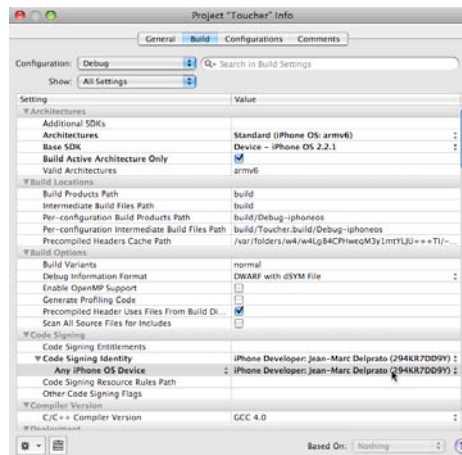


Tester une application sur votre iPhone

Vous avez installé le profil de provisionnement sur votre iPhone. Ouvrez le projet Xcode correspondant à l'application que vous souhaitez installer. Effectuez ensuite un double-clic sur son intitulé, au sommet du volet gauche de l'environnement de développement. Cliquez sur l'onglet Build et vérifiez tout d'abord que le menu Configuration indique bien Debug avant de vous reporter à la section Code Signing (voir Figure 10.17).

Figure 10.17

On prépare la diffusion d'une application sur un iPhone.



Aux lignes Code Signing Identity et Any iPhone OS Device, déroulez le menu et sélectionnez la clé liée à votre propre iPhone. Fermez la fenêtre et sélectionnez ensuite le fichier Info.plist associé à votre projet. Dans le champ Bundle identifier, saisissez l'identifiant que vous avez défini précédemment lors de la création d'un App ID. Enregistrez votre mo-

dification, déroulez le menu tout en haut à gauche de Xcode et sélectionnez Device – iPhone OS x (où x correspond à la version d’iPhone OS installée sur votre mobile). Cliquez enfin sur le bouton Build and Go, en ayant naturellement connecté votre iPhone à votre Mac en USB. Xcode compile votre projet et l’envoie vers votre iPhone et non le Simulateur. Vous pouvez ainsi librement tester vos applications, en renouvelant l’opération pour chacun de vos projets.

En cas d’erreur

Un message d’erreur apparaît à l’écran lors de la compilation et vous empêche d’installer l’application sur votre iPhone (erreur 0XE800003A) ? Pas de panique : déroulez le menu Build de Xcode et cliquez sur Clean All Targets. Quittez Xcode, relancez votre projet. Vérifiez les propriétés de votre application, en particulier à la section Code Signing Identity, puis compilez à nouveau votre projet.

Soumettre votre application à Apple

L’exercice précédent vous a permis de tester votre application en situation réelle. N’hésitez pas à ajouter d’autres mobiles à la liste des appareils enregistrés, notamment des iPod Touch ou des iPhone s’appuyant sur une autre version du système, en sollicitant l’aide de vos amis. Vous procédez ainsi à une batterie de tests supplémentaires et vous vous assurez que votre application ne posera pas de problèmes majeurs face à la multitude de configurations possibles. Il est maintenant l’heure de soumettre votre application à Apple, afin de la diffuser sur iTunes.

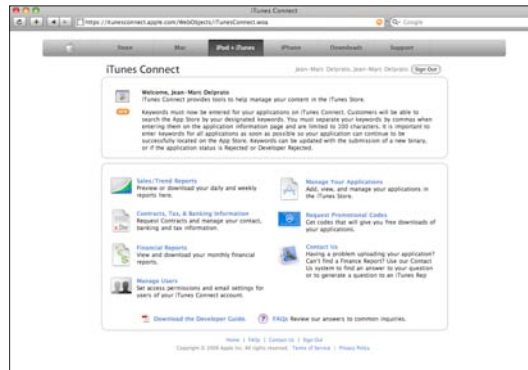
De retour sur le portail de développement pour l’iPhone, à l’adresse <http://developer.apple.com/iphone>, cliquez sur le lien iTunes Connect à droite de l’écran. Cet espace vous permet de soumettre de nouveaux projets, de gérer vos informations bancaires, de visualiser les rapports de ventes ou d’obtenir des codes promotionnels visant à télécharger gratuitement vos propres applications (voir Figure 10.18).

Cliquez sur le lien Manage Your Applications, puis sur le bouton Add New Application. Tout commence par le choix d’une langue principale : il s’agit de la langue dans laquelle vous saisissez les informations liées à votre application. Ce choix est important, dans la mesure où il définit la langue principale associée au projet que vous soumettez. Si votre application se destine exclusivement à un public francophone, sélectionnez le français. Vous aurez ainsi plus de chances de figurer en bonne place dans l’iTunes Store francophone. Vous pourrez toutefois ajouter des langues supplémentaires par la suite et même publier des captures d’écran dans plusieurs langues : nous y reviendrons. En parallèle, saisissez votre nom ou celui de votre société. Là encore, vous ne pourrez pas modifier ce choix ultérieurement.

Il s'agit du nom qui figurera à côté de votre application, sous iTunes. Rien ne vous oblige à saisir le nom d'une entité réellement déposée auprès de la Chambre de commerce et d'industrie ou du statut des auto-entrepreneurs, mais prenez garde : si vous utilisez un nom très fantaisiste ou pompeux (en prétendant par exemple que votre application émane d'Apple), elle sera immédiatement rejetée. Dans la mesure du possible, essayez d'utiliser votre propre nom en diffusant votre application à titre individuel.

Figure 10.18

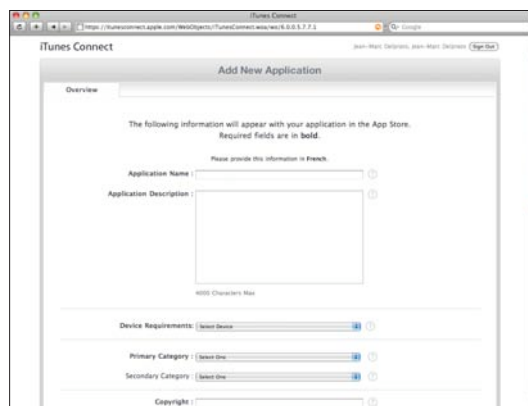
Les différents outils de la section iTunes Connect.



Apple vous demande ensuite si votre application exploite des technologies de chiffrement. Dans la mesure où elle sera diffusée au monde entier, vous devez faire attention aux législations locales portant sur le chiffrement de données. Il y a toutefois de grandes chances pour que votre application n'exploite pas ce type de procédé : poursuivez la soumission de votre projet. Vous devez enfin remplir un large formulaire pour finaliser la soumission de votre application auprès d'Apple (voir Figure 10.19). Procédez avec minutie : vous ne serez plus en mesure de modifier la plupart de ces informations par la suite et Apple se montre intransigeant vis-à-vis des conditions d'utilisation. La qualité générale des applications de l'App Store est à ce prix !

Figure 10.19

Le formulaire de soumission d'applications d'iTunes Connect.



Nom de l'application. Ce nom figurera dans l'App Store d'iTunes et non sur l'iPhone de vos clients (ce dernier est défini au champ Bundle display name du fichier Info.plist associé à votre projet, sous Xcode). Vous êtes limité à 255 caractères, mais évitez de choisir un nom si long. N'oubliez pas que les utilisateurs peuvent également consulter le catalogue d'applications depuis leur propre iPhone – l'écran y est plus limité et vous devez rester synthétique. Par ailleurs, vous n'avez pas le droit d'utiliser des marques déposées, y compris le mot "iPhone". Apple voit également d'un très mauvais œil l'utilisation du préfixe "i", comme *iVideo* ou *iReader* par exemple. Si vous choisissez un nom différent pour votre application et sa fiche descriptive diffusée sur iTunes, optez pour une base commune. Par exemple, une célèbre application liée aux métros parisiens s'intitule "*Paris-ci*" sur un iPhone et "*Paris-ci la sortie du métro*" sur iTunes. Cet intitulé plus complet permet à cette application d'être référencée sur le terme "métro" lors d'une recherche sur iTunes.

Description de l'application. Ce long champ de texte contient jusqu'à 4 000 caractères. Vous y définissez le rôle et l'intérêt de votre application, en mettant en relief ses principales fonctionnalités. Vous n'avez pas la possibilité d'utiliser de balises HTML dans ce champ ; la plupart des développeurs divisent cette description en plusieurs sections (Fonctionnalités, Caractéristiques, Nouveautés, etc.) en soulignant les titres à l'aide de codes ASCII, en ajoutant une série d'étoiles par exemple. Soignez tout particulièrement ce texte initial : mis à part son intitulé et son icône, c'est le premier contact qu'ont vos futurs clients avec votre application.

Configuration requise. Vous définissez dans ce menu les appareils compatibles avec votre application : iPhone uniquement, iPhone et iPod Touch de seconde génération ou toutes les versions de l'iPhone et de l'iPod Touch. Si votre application exploite le capteur de l'iPhone ou son GPS, sélectionnez la première catégorie.

Catégories de l'application. iTunes trie les applications selon vingt catégories différentes (actualités, divertissement, utilitaires, météo, jeux, enseignement...). Sélectionnez la catégorie principale de votre application : elle figurera dans cette section, sous iTunes. La catégorie secondaire affine le genre de votre application. Un lecteur de flux RSS pointant vers le blog d'un photographe appartiendra probablement à la catégorie principale "News" et à la catégorie secondaire "Photography", par exemple.

Informations de copyright. Saisissez ensuite vos informations de copyright, en notation anglaise ("2010, Ma société" par exemple). Définissez également le numéro de version ("1.0"). Le champ SKU Number correspond à la gestion des stocks : son acronyme signifie *Stock-Keeping Unit*. Il s'agit d'un numéro unique qui vous permettra de suivre les ventes de vos applications. Ainsi, votre premier projet pourra hériter du numéro de produit 1, votre deuxième application du numéro 2 et ainsi de suite. Ces numéros figurent également dans

les différents documents de facturation et vous aident à conserver une trace des applications que vous diffusez sur l'App Store.

Mots-clés. Fondamentale, cette section vous permet d'associer des mots-clés à votre application. Ils seront pris en compte au moment de la recherche. Vous ne disposez que de cent caractères, en séparant chaque terme d'un espace. Soyez précis et essayez de définir au mieux les termes liés à votre application.

Sites web du support. Afin de valider la soumission de votre application, vous devez impérativement indiquer l'adresse d'un site web associé à votre projet. Cette page doit comprendre un rapide guide d'utilisation, autour d'une série de captures d'écrans, et mentionner une information de contact afin de permettre aux utilisateurs de vous solliciter. En réalité, ces pages ne sont que rarement consultées par les utilisateurs, qui préfèrent déposer directement leurs questions sous forme de commentaires sur iTunes. Si vous êtes pressé, n'hésitez pas à utiliser un service de blogging ou d'hébergement gratuit afin d'aboutir rapidement un résultat. Le service Google Sites (<http://sites.google.com>) constitue une solution simple et conviviale pour créer ce type de page. En quelques clics de souris, vous déployez un site complet autour d'un modèle et vous enrichissez vos pages de captures d'écrans et d'une description précise de votre application. À l'inverse, vous pouvez prendre le temps de créer un site complet aux couleurs de votre application, en veillant à son bon référencement auprès des moteurs de recherche. Une telle solution constitue alors un second point d'entrée vers votre application, au-delà d'iTunes : des utilisateurs potentiels la découvriront au gré d'une recherche sur Google, avant de se rabattre vers iTunes pour la télécharger.

Compte de démonstration. Votre application nécessite-t-elle un compte utilisateur pour accéder à un service précis ? Si c'est le cas, vous devez initialiser un compte de test et en indiquer l'identifiant à Apple. La maison-mère ne prendra effectivement pas le temps de créer manuellement un compte sur votre service et rejettera votre application si vous ne lui en fournissez pas un.

Contrat de licence. Si votre application nécessite un contrat de licence particulier, vous pouvez le fournir en dernier lieu. Ce type de contrat est nécessaire lorsque votre application effectue des traitements particuliers (stockage ou publication d'informations, par exemple) ou lorsqu'elle s'inscrit dans un réseau communautaire qui exige le respect de certaines règles. Dans la plupart des cas, le contrat de licence standard suffit. Cliquez sur le bouton Continuer pour valider votre saisie.

Niveau de contrôle parental. Vous devez ensuite indiquer le niveau de contrôle parental de votre application, en répondant à une série de questions (voir Figure 10.20). Votre application contient-elle de la violence ? Des références à l'alcool ou au tabac ? Des illustrations pouvant effrayer les utilisateurs ? Répondez sereinement à chaque point, en cochant la case "Jamais", "Modérément" ou "Fréquemment". Sur la droite de l'écran, un pictogramme vous indique le niveau de contrôle parental correspondant (4+, 9+, 12+ et 17+). Apple ne plaisante pas du tout avec cet aspect et vous avez tout intérêt à vous montrer très prudent. Si votre application interroge un service à distance, comme un lecteur de flux RSS par exemple, veillez à choisir un niveau de contrôle parental relativement élevé (12+). Dans tous les cas, sachez que les utilisateurs d'iPhone de moins de douze ans sont relativement rares et que vous ne vous priverez donc pas d'une cible potentielle !

Figure 10.20

Sélectionnez le niveau de contrôle parental de votre application.

The screenshot shows the 'Add New Application' page in iTunes Connect. The 'App Rating' section is active, displaying a table of content descriptions and their frequency levels. A large '9+' rating icon is shown on the right.

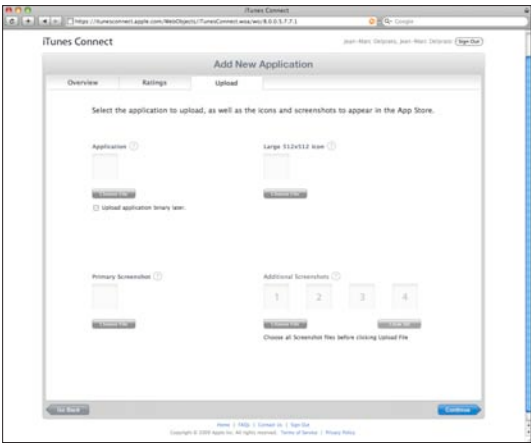
Apple Content Descriptions	None	Infrequent/Mild	Frequent/Intense
Cartoon or Fantasy Violence	☐	☒	☐
Realistic Violence	☒	☐	☐
Sexual Content or Nudity	☒	☐	☐
Profanity or Crude Humor	☒	☐	☐
Alcohol, Tobacco, or Drug Use or References	☒	☐	☐
Mature/Suggestive Themes	☒	☐	☐
Simulated Gambling	☒	☐	☐
Horror/Fear Themes	☒	☐	☐
Prolonged Graphic or Sadistic Realistic Violence	☒	☐	☐
Graphic Sexual Content and Nudity	☒	☐	☐

9+ App Rating

Éléments de votre application. Ouvrez à nouveau votre projet sous Xcode et sélectionnez Release dans le menu déroulant en haut à gauche. Compilez votre application : vous obtenez un fichier .app dans le dossier /build/release de votre projet. Vous devez impérativement ajouter une icône de 57 × 57 pixels au fichier Info.plist, comme nous l'avons vu au cours des premiers chapitres. Vous devez ajouter ce fichier .app à la section Application de l'étape suivante d'iTunes Connect (voir Figure 10.21). Choisissez également une large icône de 512 × 512 pixels en 72 dpi, de préférence au format TIFF ou PNG. Celle-ci sera utilisée sur la page associée à votre application, sous iTunes. Enfin, ajoutez cinq captures d'écran en choisissant l'une d'entre elles en guise de capture principale. Ces images présentent votre application en action : sélectionnez-les attentivement, elles constituent l'un des premiers critères de choix. Pour réaliser une capture d'écran sur votre propre iPhone, pressez simultanément la touche Home et le bouton Power, en haut à droite. Vous avez également la possibilité d'utiliser Xcode : déroulez le menu Window > Organizer, puis cliquez sur l'onglet Screenshot après avoir relié votre iPhone en USB.

Figure 10.21

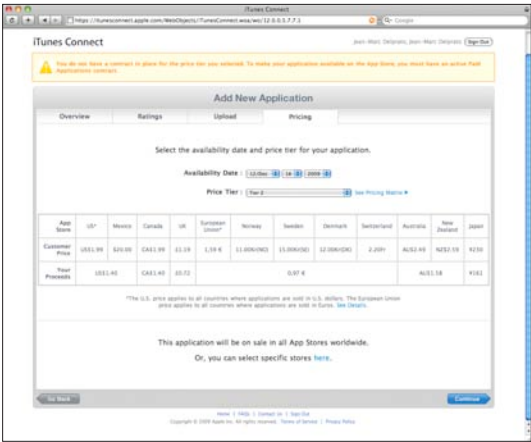
Associez des éléments graphiques à votre application.



Prix et date de disponibilité. À l’écran suivant, vous êtes invité à définir le prix de votre application (voir Figure 10.22). Vous ne pouvez pas saisir directement le montant de votre choix : sélectionnez l’une des catégories. Reportez-vous au tableau ci-dessous pour connaître la correspondance entre les principales tranches tarifaires et le prix réellement pratiqué en France, à l’heure où nous mettons cet ouvrage sous presse (voir Tableau 10.1).

Figure 10.22

Indiquez le niveau de prix et la date de disponibilité de votre application.



Comme vous le constatez, la grille tarifaire évolue largement. Nous ne vous recommandons pas de choisir un prix trop élevé : tâchez de vous maintenir dans les cinq premières catégories de tarifs, à moins de proposer une application réellement innovante. Envisagez également de diffuser gratuitement votre application, afin de vous constituer une vitrine initiale sur iTunes. Vous bâtirez des versions plus approfondies de vos applications par la suite, après avoir brillé une première fois dans les rayons de l’App Store ! Sélectionnez également

une date de disponibilité. Par défaut, il s'agit de la date du jour : vous indiquez ainsi à Apple que l'application peut être commercialisée dès qu'elle sera validée. Si votre application se rattache à un événement spécifique (rencontre sportive, foire ou salon, fêtes de fin d'année, etc.) indiquez une date précise.

Tableau 10.1 : Les principaux tarifs et rétributions pratiqués sur iTunes

<i>Tranche tarifaire</i>	<i>Prix pratiqué sur iTunes</i>	<i>Reversement net au développeur</i>
Gratuit	0 €	0 €
Tranche 1	0,79 €	0,48 €
Tranche 2	1,59 €	0,97 €
Tranche 3	2,39 €	1,45 €
Tranche 4	2,99 €	1,82 €
Tranche 5	3,99 €	2,43 €
Tranche 6	4,99 €	3,04 €
Tranche 7	5,49 €	3,34 €
Tranche 8	5,99 €	3,65 €
Tranche 9	6,99 €	4,25 €
Tranche 10	7,99 €	4,86 €
Tranche 25	19,99 €	12,17 €
Tranche 50	39,99 €	24,34 €
Tranche 85	799 €	485,95 €

En dernier lieu, iTunes Connect vous propose de soumettre un descriptif détaillé de votre application dans des langues complémentaires. Si vous aviez choisi le français à l'étape précédente, vous avez ainsi la possibilité de renseigner une nouvelle fiche en anglais, allemand, espagnol ou italien par exemple, comme nous le verrons un peu plus loin. Vous ajoutez au passage des captures d'écran localisées. Apple résume ensuite toutes les informations que vous avez saisies jusqu'à présent. Cliquez sur le bouton Submit Application pour lancer votre requête. Patientez : le processus de validation prend deux à trois semaines environ, en fonction de la demande. Apple traite près de 8 500 nouvelles applications par semaine et indique que 95 % d'entre elles sont analysées selon ce délai. Le tableau dressant la liste de toutes les applications que vous soumettez indique la progression de cette opération ("waiting for review", "in review", "ready for sale", etc.).

L'équipe de validation d'Apple comprend à l'heure actuelle plus de quarante membres, dont deux seront délégués à votre application. Les critères de validation sont relativement flous, mais la plupart d'entre eux relèvent du bon sens. Votre application plante-t-elle ? Génère-t-elle des messages d'erreur ? Que se passe-t-il lorsqu'on l'utilise sans capter le moindre réseau ? Contient-elle tous les éléments nécessaires, en particulier une icône ? Affiche-t-elle du contenu offensant ? Utilise-t-elle des éléments dont vous ne détenez pas les droits d'auteurs ? Avez-vous sous-estimé le niveau de contrôle parental requis ? N'hésitez pas à télécharger le guide de développement complet, sur la page d'accueil d'iTunes Connect. Il vous renseigne sur les meilleures pratiques et sur les éléments à ne pas oublier. Si Apple rejette votre application, l'équipe vous explique précisément les points que vous n'avez pas respectés. À l'heure actuelle, 20 % des applications en moyenne échouent lors de leur première soumission.

Validez votre contrat auprès d'Apple

Sur la page d'accueil d'iTunes Connect, cliquez sur le lien "Contracts, Tax & Banking Information". Par défaut, Apple a contracté un premier document auprès de vous : il vous permet de diffuser des applications gratuites dans le monde entier. Si vous envisagez de diffuser des applications payantes, vous devrez expressément demander un contrat élargi et indiquer vos coordonnées bancaires internationales. Cliquez sur le premier lien et validez les conditions générales. Vous obtenez alors un PDF dans votre boîte aux lettres. De retour sur iTunes Connect, saisissez vos informations de contact à la rubrique "Contact Info" puis indiquez vos informations bancaires, notamment le numéro IBAN qui figure sur l'un de vos RIB. Dans la mesure où Apple est une société américaine, vous devrez valider un formulaire de "cessation du statut de revendeur à un non-résident des États-Unis" (sauf si vous avez la nationalité américaine, bien entendu, ce que vous indiquez en début de formulaire). Quelques heures plus tard, votre contrat est finalement validé par Apple et vous avez la possibilité de revendre des applications aux quatre coins du monde.

Populariser votre application

S'il existait une recette magique conduisant invariablement au succès d'une application, tous les développeurs seraient millionnaires...! La réussite de votre projet tient parfois à une série de détails qu'il est difficile de prévoir et de maîtriser. Mettez toutefois toutes les chances de votre côté, en envisageant une campagne promotionnelle et en soignant les derniers aspects de votre application.

L'intérêt de votre application. En premier lieu, veillez à proposer une application qui remplit un rôle spécifique et qui enrichit réellement le quotidien de ses utilisateurs. Programmer la dix-millième calculatrice pour iPhone, sans offrir de réels atouts par rapport à

L'application intégrée par défaut, est peine perdue : elle sombrera dans les profondeurs du classement, même si Apple la valide rapidement. Si cette notion est largement subjective, vous avez intérêt à soumettre votre application au verdict de votre cercle familial dans un premier temps. Des idées d'améliorations jailliront spontanément, si vous êtes ouvert à la discussion et si vous acceptez de repousser la diffusion de votre application en replongeant dans son code source. Dans le même ordre d'idée, réfléchissez au cadre d'exécution de votre application. N'oubliez jamais que vous développez pour un périphérique nomade, qui accompagne l'utilisateur lors de ses déplacements. Ce dernier n'a ni le temps ni la concentration de se pencher sur des traitements trop complexes ou des manipulations avancées. Par exemple, une application de recettes de cuisine qui exigerait des manipulations incessantes de l'écran ne remplirait pas totalement l'objectif qu'elle s'est fixée : l'utilisateur n'apprécierait pas forcément de toucher son mobile avec les doigts dans la farine. Si votre application s'exerce dans un cadre spécifique, mettez-vous en situation et vérifiez que son usage est réellement possible.

L'analyse de l'existant. Intimement liée à cette première étape de validation, l'analyse de vos applications concurrentes en dit long sur la portée de votre projet. Vous avez créé un utilitaire de calcul de trajets dans le métro ? Passez en revue l'ensemble des solutions concurrentes, en particulier celles s'inscrivant dans la même grille tarifaire que la vôtre. Si leurs fonctions dépassent largement ce que vous proposez, vous aurez du mal à vous démarquer... à moins d'offrir une ergonomie et une interface réellement nouvelles. À l'inverse, si votre application renvoie aussitôt un air de fraîcheur et complète ces approches concurrentes, vous avez toutes les chances de vous imposer en quelques semaines.

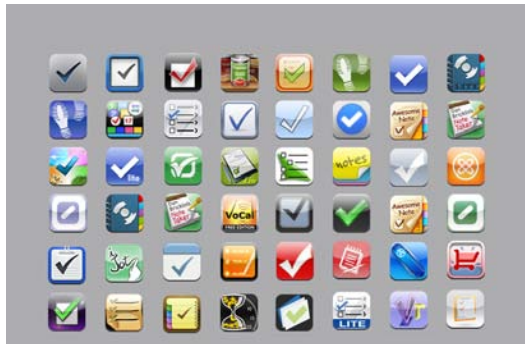
Optimisation du code source. C'est un aspect qu'Apple considère au moment de valider votre application : consomme-t-elle des ressources démesurées ? Validez consciencieusement votre application avant de la soumettre sur iTunes. Vérifiez que vous avez bien supprimé les références à vos outlets lors de l'arrêt de vos traitements et que la gestion de la mémoire est optimisée. Pour vous aider dans cette tâche, utilisez les instruments de Xcode, en particulier Memory Leaks et Activity Monitor. Lorsque vous compilez votre projet, évitez également d'alourdir le fichier résultant par des éléments que vous n'utilisez pas (images importées, frameworks non utilisés, etc.). Supprimez les commentaires de vos codes sources, en particulier les appels à la classe NSLog. Les utilisateurs n'auront pas accès à la console de débogage et ce type d'appels amoindrit la réactivité de votre application.

Charte et éléments graphiques. C'est l'un des premiers aspects lié à votre application, sur iTunes : soignez tout particulièrement son icône, son logo et les captures d'écran. Là encore, il n'existe pas réellement de règles établies qui garantissent le succès d'une charte graphique. À titre de comparaison, reportez-vous à la Figure 10.23 : elle réunit des exemples d'icônes d'applications de "listes de tâches". Ces icônes, alliées au nom de chaque projet, apparaissent sous la forme d'un tableau lorsqu'on effectue une recherche sur iTunes. Cer-

taines d'entre elles paraissent plus explicites et mieux dessinées – deux critères de choix essentiels. Si vous ne disposez pas d'un talent de graphiste particulier, sollicitez l'aide d'un spécialiste, à travers un forum de discussion, ou veillez à rester le plus sobre possible. Évitez dans la mesure du possible les photos réduites en 57×57 pixels. Elles donnent un côté très amateur au projet et passent mal sur l'écran de l'iPhone.

Figure 10.23

Les applications de listes de tâches abondent : saurez-vous distinguer les meilleures d'entre elles à partir de leur icône ?



Le site web de référence. Lors de la soumission de votre application à Apple, vous êtes invité à créer un site web associé à votre projet. S'il est possible d'utiliser un modèle prêt à l'emploi, qui ne contiendra que les informations essentielles (description du projet, fonctionnalités de l'application, support technique...), vous pouvez aussi envisager de créer un site haut en couleurs. Celui-ci accompagnera les utilisateurs et pourra même contenir un forum (voir Figure 10.24). Là encore, si vous envisagez de créer un vaste site web, profitez-en pour soigner son référencement. De nombreux utilisateurs pourraient ainsi tomber sur vos pages à travers une recherche sur Google : convainquez-les de tester votre application à travers iTunes.

Figure 10.24

Le studio français Agharta diffuse des jeux pour iPhone et a mis en place un site web très riche.



Sites partenaires et réseaux communautaires. Il existe de nombreux blogs ou webzines liés à l'iPhone et aux applications diffusées sur iTunes. Il est délicat de solliciter leurs rédacteurs en leur soumettant votre propre création : n'envoyez jamais directement un e-mail les invitant à tester votre projet, ce serait largement contre-productif. Participez aux forums de discussion, déposez des commentaires constructifs et intégrez-vous progressivement à leur communauté de lecteurs. Petit à petit, vous gagnerez en crédibilité et vous pourrez ainsi plus sereinement vous soumettre à leur verdict. Utilisez également les codes promotionnels que vous pouvez créer sur iTunes Connect pour offrir un accès gratuit à votre application. Parmi les sites de référence, reportez-vous à iPhoneTests.com, ApplicationIphone.com, Test-Appstore.com et Appstore-test.fr (voir Figure 10.25). Les réseaux sociaux comme Facebook et Twitter jouent également un grand rôle dans la diffusion initiale de votre application. Publiez un Tweet ou modifiez votre statut, en indiquant que vous venez de créer une première application. Vos amis et contacts directs iront peut-être l'essayer : le nombre de téléchargements effectués dans les premiers jours de diffusion de votre application jouent largement sur son classement. Faites ensuite marcher le bouche-à-oreille.

Figure 10.25

De nombreux sites testent régulièrement les nouvelles applications de l'App Store. Soumettez-leur vos propres créations.



Versions localisées et mises à jour. Afin de toucher un public plus large, vous pouvez prévoir des versions localisées de vos applications. Sous Xcode, votre projet peut en effet abriter des traductions de tous vos éléments d'interface et chaînes de caractères : le résultat apparaît automatiquement dans la langue de l'utilisateur, telle qu'il l'a choisie dans les réglages de son mobile. Reportez-vous à la documentation du SDK de l'iPhone pour de plus amples détails à ce sujet. La localisation d'une application ne met pas en jeu des techniques complexes, mais elle exige de la minutie et de l'organisation. Heureusement, Xcode supporte pleinement cet aspect et vous aide en préparant automatiquement des sous-dossiers contenant vos éléments de traduction. Vous devez tout d'abord connaître l'identifiant ISO des langues supplémentaires que vous souhaitez proposer. Rendez-vous à l'adresse

http://www.iso.org/iso/country_codes.htm pour en découvrir la liste complète. L'Allemand est reconnu par le code "de_DE" et le Français par "fr_FR" par exemple.

En Objective-C, on récupère le paramètre local de l'utilisateur à l'aide de la méthode `currentLocale` :

```
NSLocale *localisation = [NSLocale currentLocale];
```

Vous définissez ensuite tous les termes apparaissant à l'écran à travers des instances de la classe `NSLocalizedString`. Cliquez sur l'icône de votre projet, sous Xcode, puis déroulez le menu File et cliquez sur Get Info. Cliquez ensuite sur le bouton Make File Localizable. Parmi les ressources associées à votre projet, vous découvrez alors un sous-dossier /English. Ouvrez à nouveau le panneau des informations associées à votre projet et cliquez à présent sur le bouton Add Localization. Saisissez le code ISO correspondant à la langue que vous souhaitez ajouter, "fr_FR" par exemple. Vous créez ainsi automatiquement un nouveau fichier prêt à l'emploi, dans lequel vous définirez les versions localisées de vos chaînes de caractères. La structure de votre projet est prête : reportez-vous au guide de développement Cocoa Touch intégré à Xcode pour découvrir la nomenclature exacte de la classe `NSLocalizedString`.

Sachez par ailleurs que ce marché s'est considérablement ouvert ces derniers mois et qu'il est désormais possible de proposer ses services de traduction. La société iPhoneLocalizer (www.iphonelocalizer.com) s'en fait une spécialité et propose de traduire une application dans près de quatre-vingt langues. Dans le même ordre d'idée, n'oubliez pas de proposer des mises à jour régulières de votre application (voir Figure 10.26).

Figure 10.26

La publication de mises à jour régulières renforce l'image de votre application auprès de ses utilisateurs.

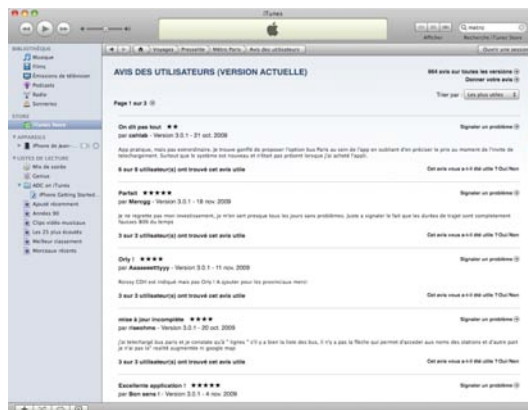


Vous devrez à nouveau les soumettre à Apple et vous inscrire dans le cycle de validation – choisissez un numéro de version différent afin d'éviter de saisir toutes les informations initiales. Vos utilisateurs sont prévenus de la mise à jour par un petit "badge" associé à l'icône de l'application App Store.

Tirez des leçons des commentaires. On ne le sait que trop bien : les commentaires sur les forums ne font jamais dans la demi-mesure et peuvent vite tourner à l'insulte à l'orthographe approximative. Ne vous découragez pas si un premier utilisateur vocifère sur la page des commentaires liés à votre application. Attardez-vous plutôt sur les commentaires constructifs ; de nombreux internautes prennent le temps d'exprimer leur avis complet et de vous suggérer des pistes d'amélioration. Écoutez-les attentivement, notamment pour faire remonter la note moyenne de votre application (voir Figure 10.27).

Figure 10.27

Consultez fréquemment les commentaires, sans toutefois prendre au premier degré les avis les plus acerbes et personnels.



Trouver des idées originales de développement. À l'issue de la lecture de cet ouvrage, vous maîtrisez la quasi-totalité des comportements spécifiques de l'iPhone et de l'iPod Touch. Appareil photo, support de l'accéléromètre, lecture de fichiers externes, interfaces complexes s'articulant autour de listes ou d'onglets... Votre champ d'action s'est très largement ouvert et vous êtes en mesure de vous confronter à des projets de grande ampleur. Premier réflexe salutaire : consultez attentivement la liste des applications les plus populaires de l'App Store d'iTunes et essayez de comprendre au premier coup d'œil la logique de tous ces projets. Un simple jeu consiste à déplacer une balle à l'écran en inclinant son mobile ? Son auteur a utilisé l'accéléromètre pour déterminer les coordonnées à appliquer à la balle. Il faut éviter des obstacles et des trous ? Ceux-ci correspondent à des instances de la classe `UIImageView` dont les coordonnées ont été soigneusement repérées : si la balle pénètre dans leur périmètre, elle disparaît et la partie est perdue. Procédez de même avec l'ensemble des applications les plus populaires : vous verrez à quel point certains projets auréolés de succès ne tiennent qu'en quelques modestes lignes de code. À l'inverse, vous constaterez que cer-

taines catégories d'applications paraissent bouchées et qu'il sera bien difficile de proposer des programmes supplantant les meilleurs projets. C'est notamment le cas des utilitaires de prise de vue, dont certains exemples dépassent largement l'application générique proposée par défaut par Apple. Ne vous découragez pas pour autant : si d'illustres développeurs se sont déjà emparés d'un sujet particulier que vous convoitiez, vous avez toujours votre mot à dire. Réfléchissez en particulier aux spécificités françaises de votre projet : n'y a-t-il pas des services nationaux (transport, météo, dictionnaires, etc.) qui mériteraient d'être intégrés au sujet et qui vous offriraient immédiatement un large public francophone ? Sans pour autant dupliquer l'application d'origine, essayez d'en proposer une nouvelle ergonomie. Si votre interface est plus intuitive et facile à maîtriser, vous partirez d'un bon pied et votre projet pourra être salué sur iTunes. Par ailleurs, n'oubliez jamais que les applications les plus populaires ont généralement bénéficié d'une dizaine de mises à jour depuis leur lancement. N'essayez donc pas du premier coup de proposer une application aussi bluffante – vous l'étofferez progressivement, en vous basant sur les commentaires des utilisateurs.

Pour aller plus loin

Nous arrivons au terme de notre exploration du développement d'applications pour iPhone et iPod Touch. Comme vous l'avez constaté à travers chaque chapitre, c'est un exercice aussi rigoureux que ludique : le kit de développement offre un socle riche et intuitif, qui vous oblige à respecter de nombreuses règles mais qui se solde par des applications originales et innovantes. Complétez votre apprentissage initial en vous reportant au guide de développement inclus dans Xcode, en approfondissant tous les aspects de Cocoa Touch et d'Objective-C. Après avoir soumis votre application à Apple, faites-la vivre en l'enrichissant de nouveaux comportements et en sollicitant l'avis de ses utilisateurs. À ce titre, les rapports de plantage que vous adressez Apple sur votre interface de publication constituent une formidable source d'informations pour résoudre les principaux problèmes et étendre les fonctionnalités de vos projets.

L'App Store d'iTunes contient aujourd'hui plus de 100 000 applications... dont les vôtres ! Vous participez au succès de cette plateforme innovante et originale et vous vous inscrivez parmi les pionniers de ce développement d'un type nouveau, promis à un grand avenir. Sur les bases de votre expérience initiale, n'hésitez pas à vous intéresser au jour le jour à toutes les nouveautés du SDK, en suivant régulièrement ses mises à jour. La demande est aujourd'hui très forte et vous pouvez même proposer vos services à de nombreuses entreprises, qui souhaitent disposer d'une certaine visibilité sur l'App Store. Devenez développeur professionnel, complétez vos connaissances en Objective-C et travaillez en équipe afin d'assurer des projets de plus grande ampleur.

Plus qu'un simple mobile, l'iPhone s'affirme comme une plateforme de développement à part entière qui s'adapte librement à vos projets les plus ambitieux. Participez activement à ce nouvel élan et développez à votre tour toutes les applications dont vous rêvez. Le succès est au bout de vos doigts !

Index

A

Accéléromètre 311
déplacer une image 315

Accesseurs 98

Actions 81

Activity Indicator View 60

Activity Monitor 72

Affichage
alerte à l'utilisateur 177
images 248, 250
modes 290
rotation 289

afficheDate 188

Ajouter une réglette 173

alerteClic 179

allKeys 204, 214

allowsImageEditing 272

animated 124

Appareil photo de
l'iPhone 274

Applications

appareil photo de
l'iPhone 274
catégories 19
formulaire de sou-
mission 330
icône 66
interactions 75
langue 334
multivues 116
populariser 335
soumettre à Apple 328
tarif et rétribution Apple 334
tester lors du déploie-
ment 327
vues 110, 150
Voir aussi Déployer
une application

Audio 275

déclencher un son 275
formats 276

Audio Toolbox 275

Audio Unit 275

AVFoundation 275, 280

B

backgroundColor 257

Bar Button Item 60

beginAnimations 128

Bienvenue_Prefix.pch 50

BienvenueViewController.
xib 50

Blog 231

Boutons prédéfinis 169

Bundle de préférences 232

C

Caractères spéciaux 88

cellForRowAtIndexPath 216

Certificat de développeur 322

CGRect 253

CGRectContainsPoint 303

Champs de saisie 140

Charger des données 212

Classes 50
 fichier d'implémentation 86
 sérialiser 238

Cocoa 34

Cocoa Touch 32

commitAnimations 128, 131

commitEditingStyle 243

composant 205

Conteneurs de données 58

contentMode, constantes 251

Contexte graphique 282

Contrat de développeur 12

Contrôles

bibliothèque 153

localiser 273

roulette 184

segmentés 173, 179

spécifiques 153, 154

Contrôleurs

implémenter 87

logique interne 89, 160

navigation 124

Coordonnées écran 254

Core Graphics 253

créerUIImage 308

D

Date & Time 185

Date Picker 59, 185

Dates 185

formater 188, 189

Déboguer 62

outils 70

debutTransition 308

Déclencher un son 275

Délégué 99

Déplacer un élément au doigt
 300

Déployer une application 322

associer des iPhones

enregistrés 324

certificat de déve-

loppeur 322

étapes 322

formulaire de sou-

mission 330

identifiant d'application 325

langue 334

profil de provision-

nement 326

soumettre à Apple 328

tarif et rétribution 334

tester l'application 327

validation Apple 335

Design

ergonomie 144

tendances 144

vues 143

Dessin 282

Développement

outils 30

vue d'ensemble 29

Dictionary 200

Dictionnaire 201

didAccelerate 315

Did End On Exit 97

didFinishPickingImage 267

didSelectRow 198

didSelectRowAtIndexPath

221, 229

Dimensions Voir Inspecteur de
 dimensions

Données 207

charger 212

écrire un fichier 237

enregistrer 231

interactivité 221

lire 209

listes 214

cellules 215

lire 209

mise en forme 220

polices 217

taille 218

récupérer 194

tableau 194, 198

drawRect 284

E

Écrire un fichier de données
 237

Editing Changed 97

Editing Did Begin 97

Editing Did En 97

@end 98

Enregistrer des données 231

Événements, bibliothèque 96

Exemples

Chant d'oiseaux 277, 293

dessin 282

Diaporama 304

données 210

flux RSS 223

générateur de mot de

pas 155

quiz 80

F

File's Owner 52

First Responder 53

Fixed Space Bar Button
Item 61

Flexible Space Bar Button
Item 61

Flux RSS 208, 223
charger 243
TouchXML, intégrer 224

font 216

Formats audio 276

Foundation 37

frame 253, 290, 303

Frameworks 51

G

Générateur de mot de passe
155

Gestes de l'utilisateur 287
accéléromètre 311
déplacer un élément 300
écran 299
navigation 304

Glyphish 119

H

heightForRowAtIndex
Path 218

I

IBAction 83

Identifiant d'application 325

imagePickerControllerDid-
Cancel 267

Images 119, 248
afficher 248
charger depuis une source
externe 258
mise à jour 259

déplacer avec l'accé-
léromètre 315

dimensions de la vue 253

importer 204

légende 255

personnelles et retouche 267

photothèque de l'uti-
lisateur 264

préserver l'aspect 251

taille 250

UIWebView 261

Image View 59

indexPath 214

Info.plist 51

initWithNibName 124

Inspecteur
de connexions 94
de dimensions 290

Installer le SDK 13

Interactions 75
ergonomie 77
type de 95
Voir aussi Données et
Gestes de l'utilisateur

@interface 98

Interface
définir 165
déposer des objets 92
disposer les contrôles 171
préparer sous Pho-
toshop 170

Interface Builder 8, 52
bibliothèque d'objets 58
fenêtres 52
héritage 56
objets, ajouter 54

Interrupteurs 158

isEqualToString 103

isSourceTypeAvailable 265

K

Key 236

L

Label 59

Leaks 72

libxml2 223

Licence 318
programme standard 320
types 319

Limites de l'iPhone 39

Lire des données 209

Liste de propriétés 199, 210,
232
créer 200
préférences des appli-
cations 233

loadView 250

Locale 185

Logo 170

M

main.m 50

MainWindow.xib 51

Manipulation de l'écran 297

Marketing 317
analyse de l'existant 336
charte et éléments gra-
phiques 336
commentaires 340
idées 340
intérêt d'une appli-
cation 335
optimisation du code
source 336
site web de référence 337

versions localisées et mises à jour 338

Media Player 275

Méthodes
objets et 91

Modèle-Vue-Contrôleur.
Voir MVC

MultiConvertisseur 131

Multimédia 245

Mutateurs 98

MVC
architecture 78
contrôleur 82
vue et contrôles 80

N

Navigation
barre de 113
boutons 121
choisir 111
gestes de l'utilisateur 304
listes 111
listes et tableaux 127
onglets 114, 116, 121
utilitaires 113

Navigation Bar 60

Navigation-Based Application
48

Navigation Controller 124

Navigation Item 60

nodesForXPath 237

nonatomic 85

NSArray 192, 196

NSBundle 204

NSData 258

NSDate 187

NSDateFormatter 188

NSDateFormatterFullStyle
189

NSDateFormatterLongStyle
189

NSDateFormatterMediumStyle
189

NSDateFormatterShortStyle
189

NSSearchPathForDirectoriesInDomains 237

NSString 89

NSTemporaryDirectory 238

NSURL 258

NSUserDefaults 232, 236

NSXMLParser 223

numberOfComponentsInPickerView 198

O

Objective-C 32, 35

Objets
ajouter *via* Interface Builder 54
bibliothèque 91
dimensions 294
méthodes, associer 91
types 58
Voir aussi Inspecteur de dimensions

Opacité 182

OpenAL 275

OpenGL ES Application 48

Optimisation du code source
336

Other Sources 50

Outils de développement 30

Outlets 81

P

Page Control 60

PDF, charger avec UIWebView 261

PhoneGap 17

Photographies *Voir* Images

Photothèque de l'utilisateur
264

pickers 184

Picker View 59

Pointeur 81

PreferenceSpecifiers 233

Products 51

Profil de provisionnement 326

Programme de développeur
iPhone 318
Voir aussi Licence

Progress View 60

Projets
cahier des charges 18
complier 63
conseils 24
créer 46
débuguer 62
icône 105
modèles 48
officiels d'Apple 68
pièges 23
préparer 17
structure 49
tester 62

@property 84, 99, 124

Property List Editor 201

Propriétés
déclarer 84
Objective-C 98

PSGroupSpecifier 233

PSMultiValueSpecifier 235
 PSSliderSpecifier 233
 PSTextFieldSpecifier 233
 PSToggleSwitchSpecifier 233
 Publication 317
Voir aussi Déployer une application et Licence

Q

QuickConnect 17

R

RacineViewController 124
 Réglette 173
 ajouter 173
 valeurs 174
 reloadComponent 205
 Resources 50
 Root.plist 232
 RootViewController 156
 Rotation
 affichage 289
 modes 290
 Roulettes 184
 actualiser 205
 de temps 185
 personnaliser 190
 Round Rect Button 59
 row 205

S

Saisie utilisateur,
 récupérer 140
 Scroll View 59

SDK 7, 45
 alternatives 17
 configuration requise 8
 installer 13
 outils 8
 télécharger 9
 Search Bar 60
 Segmented Controls 59, 180
 setAlpha 182
 setBackgroundColor 182
 setDateFormat 189
 setLocale 189
 shouldAutorotateToInterfaceOrientation 289
 Simulateur d'iPhone 8, 105
 paramètres linguistiques 64
 vérifier un projet 65
 Slider 60, 174
 sliderClic 177
 Soumettre une application à Apple 328
 Sous-vue
 ajouter 179
 opacité 181
 Splash screen 150
 Streaming 281
 Switch 60
 @synthesize 99, 124
 Systèmes de navigation 60
 SystemSoundID 278

T

Tab Bar 61
 Tab Bar Application 48, 117, 132

Tab Bar Item 61
 Table View. 58
 Table View Cell 59
 Tarifs et rétributions Apple 334
 Télécharger des images supplémentaires 119
 TextField 59
 Text View 59
 Time 185
 Timezone 185
 toggleView 156
 Toolbar 60
 Touch Cancel 97
 Touch Down 97
 Touch Down Repeat 97
 Touch Drag Enter 97
 Touch Drag Exit 97
 Touch Drag Inside 97
 Touch Drag Outside 97
 touchesBegan 297, 304
 touchesCancelled 297
 touchesEnded 297, 304
 touchesMoved 297, 304
 Touch Up Inside 97
 Touch Up Outside 97
 TouchXML 223
 parseur XML 226
 Transitions 127
 constantes 131

U

UIAlertView 177
 UIButton 83, 169
 UIColor 219
 UIControlState 90
 UIDatePicker 185
 UIFont 219
 UIImage 258
 UIImagePickerController 265, 272
 UIImageView 170, 300
 UIKit 37
 UILabel 83
 UINavigationController 155
 UINavigationController 116, 123
 UIPickerView 190
 UIPickerViewDataSource 190
 UIPickerViewDelegate 190
 UIScreen 250
 UISegmentedControl 173, 182
 UISlider 297
 UISwitch 158
 UITableViewCell 214, 216
 UITextField 142
 UITouch 297
 UIView beginAnimations 130
 UIViewController 94, 116
 UIWebView 261
 UIWindow 110

Utilitaire

créer 158
 interface 159, 169
 lier outlets et actions 169
Voir aussi Générateur
 de mot de passe

Utility Application 48

Utility Application, modèle 154

V

Validation Apple 335

Value Changed 98

velociteY 315

Vidéo

framework 280
 lire 280
 streaming 281

View 53, 60

View-Based Application 48

viewDidAppear 130

viewDidDisappear 130

viewDidLoad 194, 236

ViewDidLoad 214

viewWillAppear 130

viewWillDisappear 130

VueDessin 283

Vues 107

basculement 169
 design 143
 hiérarchiser 109
 image, dimensions 253
 navigation par onglets 125
 secondaire 173
 transitions 127

W

Webcam 258

Web View 59

willAnimateSecondHalfOfRotationFromInterfaceOrientation 294

window 101

Window 60

Window-Based Application 48, 121

X

Xcode 8, 43, 46
 aide à la frappe 85

Y

yAccel 315

Le Programmeur

Développez des applications originales pour iPhone et iPod Touch

Transformez vos idées en applications !

Apple met à disposition des développeurs un kit de développement complet pour créer des applications tirant partie des fonctions originales de l'iPhone ou de l'iPod Touch – écran multi-touch, accéléromètre, capteur photo/vidéo, accès aux services web... Chacun peut ensuite vendre ses applications sur iTunes. Pourquoi pas vous ?

Grâce à cet ouvrage complet, écrit dans une langue vivante et accessible, découvrez progressivement le langage de programmation Objective-C et les principales fonctionnalités du SDK. Pénétrez dans les coulisses de l'iPhone, décryptez son système, profitez de ses contrôles originaux et modernes et réinventez à votre tour l'usage mobile. Des exemples commentés vous permettront de vous exercer au développement d'applications, mais aussi de réfléchir à la conception de vos propres interfaces, afin qu'elles soient conviviales et originales. Enfin, des conseils en marketing vous aideront à optimiser la diffusion de vos applications.

De la création de vos premiers projets à leur publication sur iTunes, vous aurez toutes les clés en main pour devenir un développeur professionnel et briller sur l'App Store.

- Le SDK de l'iPhone et de l'iPod Touch
- Vue d'ensemble du développement pour iPhone
- À la découverte du SDK de l'iPhone
- Interactions avec vos applications
- Les vues de votre application
- Les contrôles spécifiques
- Lire et écrire des données
- Le multimédia
- Réagir aux gestes de l'utilisateur
- Publication et marketing

À propos de l'auteur

Jean-Marc Delprato est journaliste dans la presse spécialisée informatique et a collaboré à plusieurs revues consacrées à la programmation (*Code(r)*, *Windows News*, *Computer Arts*...). Il a par ailleurs écrit et traduit une quinzaine d'ouvrages spécialisés.

Niveau : Débutant / Intermédiaire

Catégorie : Développement mobile

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers 75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4127-3

