ÉTUDES, DÉVELOPPEMENT & INTÉGRATION

DÉVELOPPER AVEC LES API GOOGLE MAPS

Applications web, iPhone/iPad et Android

Fabien Goblet
Michel Dirix
Loïc Goblet
Jean-Philippe Moreux

DUNOD

DÉVELOPPER AVEC LES API GOOGLE MAPS

Applications web, iPhone/iPad et Android

Fabien Goblet

Ingénieur géomaticien indépendant

Michel Dirix

Analyste programmeur au sein de l'équipe Extranet de la société Cegedim-Activ

Loïc Goblet

Créateur de la société Mooveatis

Jean-Philippe Moreux

Consultant indépendant

DUNOD

Collaboration éditoriale : Jean-Philippe Moreux

Illustration de couverture :
Parc d'armorique © Dominique LUZY - Fotolia.com

ISBN 978-2-10-055682-3

Table des matières

ANNEXES

Avant-propos

Le marché de la géolocalisation est en pleine effervescence, comme le prouve l'annonce de lancement par Facebook, en août 2010, de son service Places, qui permet de partager sa position géographique avec ses amis, mais aussi, et c'est très probable, avec des annonceurs…

La carte a toujours été un outil de pouvoir – que ce soit à une échelle locale pour calculer le montant de l'impôt chez les Égyptiens ou encore à grande échelle pour la domination maritime mondiale lors des explorations maritimes du XV^{e} siècle.

L'évolution de la technique, l'apparition de l'informatique le siècle dernier et l'impressionnante démocratisation de l'Internet ont encore renforcé la place centrale de la carte dans notre société. Google, en mettant à disposition du monde entier et de manière gratuite une cartographie mondiale détaillée, a grandement favorisé l'essor de cette nouvelle technique qu'est la cartographie dynamique (ou *webmapping*). Google Maps en est l'exemple le plus connu et représentatif.

La force de Google a été de donner aux développeurs web, dès juin 2005, la possibilité d'ajouter leurs propres cartes et créations *via* l'utilisation d'une interface de programmation JavaScript. Le succès fut alors immédiat, et des milliers de sites web utilisent ce service pour enrichir d'une touche géographique leur contenu.

Depuis 2005, Internet a beaucoup évolué, et l'apparition des périphériques mobiles de type *smartphones* est un fait majeur de ces cinq dernières années. La capacité de ces nouveaux outils à naviguer sur la toile a favorisé l'essor de nombreuses applications, autrefois destinées aux seuls ordinateurs de bureau – et beaucoup d'entre elles embarquent une carte.

Nous allons découvrir dans cet ouvrage tout d'abord comment utiliser l'API Google Maps pour développer des applications web classiques, puis dans la seconde partie, nous aborderons le développement d'applications spécifiques destinées aux téléphones mobiles iPhone et Android.

À qui s'adresse ce livre ?

Ce livre est destiné principalement aux développeurs de sites web qui souhaiteraient enrichir leurs pages web de cartes dynamiques. Une connaissance des langages HTML et JavaScript est donc souhaitable pour tout internaute désireux d'apprendre ou d'approfondir les différentes API Google Maps.

Cependant, avec l'émergence actuelle des smartphones, nous ne pouvions ignorer le développement d'applications cartographiques sur ces plateformes particulières. C'est pourquoi la seconde partie de l'ouvrage est destinée aux développements sous iOS et Android, pour lesquels une connaissance des SDK spécifiques à ces systèmes d'exploitation et d'un langage de programmation sera nécessaire.

De toutes les manières, une familiarité avec Google Maps en temps qu'utilisateur est souhaitable, et un intérêt pour la représentation de données géographiques sur Internet seront bien sûr un plus...

Suppléments web

Le site www.geotribu.net/dunod présente la plupart des exemples de démonstration de cet ouvrage, mais aussi l'application complète étudiée au chapitre 5. Les exemples sont actifs, ce qui permettra au lecteur de visualiser en regard le code et son résultat.

Remerciements

L'écriture de ce livre n'aurait pas été possible sans l'aide précieuse d'Arnaud Van De Casteele, cofondateur du site GeoTribu et avec qui nous échangeons régulièrement sur le *webmapping*, les nouvelles technologies Internet et bien d'autres choses encore. Merci Arnaud !

Nous remercions également Laurent Jégou pour sa disponibilité et son aide technique de tous les jours.

Enfin, ce livre n'aurait pu être publié sans l'intérêt pour son sujet exprimé par notre éditeur, Jean-Luc Blanc. Qu'il soit remercié ici pour sa confiance et son implication dans la réalisation de ce projet.

PREMIÈRE PARTIE

L'API Google Maps version 3

La conception d'applications cartographiques pour le Web est passionnante. Elle oblige à être au fait des nouvelles technologies et donc de faire une veille permanente. C'est ainsi que les solutions de développement proposées par Google deviennent intéressantes. En effet, depuis mai 2010, la firme de Moutain View propose une nouvelle API de Google Maps qui a la particularité de pouvoir facilement être utilisée sur plusieurs plates-formes : ordinateurs classiques et périphériques mobiles (*smartphones*).

Trois approches de développement sont possibles lors du développement d'une application cartographique pour le Web :

- la première est une approche dite « *full web* » : on n'utilise que le navigateur et la connexion Internet de la plate-forme pour développer l'application : c'est celle que nous découvrirons dans cette première partie de l'ouvrage ;
- la seconde est une approche dite « native » : on utilise alors les fonctionnalités des plates-formes cibles pour développer les applications – dans le cas de l'iPhone ou d'Android, cela passe par leurs SDK respectifs. Cette approche sera abordée dans la seconde partie de l'ouvrage ;
- la troisième approche est à mi-chemin entre les deux premières : basée sur les *webviews*, elle délègue le développement de la cartographie dans une page Internet qui sera intégrée telle qu'elle dans l'application.

Dans cette première partie, nous verrons le développement d'applications cartographiques *via* la première approche, en n'utilisant que l'API Google Maps. À travers une description détaillée des classes d'objets et des services, nous arriverons à la conception d'une application complète.

Par ailleurs, l'annexe A fera découvrir l'API OpenLayers, une concurrente *open source* de l'API Google Maps. Nous essaierons ainsi de pointer les différences majeures entre ces deux technologies.

1

Les concepts

Objectifs

La cartographie dynamique sur Internet est une technique nouvelle de représentation de données géographiques. Elle s'appuie sur les technologies web pour permettre à l'internaute de naviguer dans des cartes et d'aller chercher lui-même l'information dont il a besoin.

L'API Google Maps a permis de démocratiser cette nouvelle technologie, que l'on nomme également « *Web mapping* ».

1.1 CARTOGRAPHIE DYNAMIQUE SUR LE WEB

La cartographie dynamique sur le Web est une forme de cartographie récente dans l'histoire de la géographie. Il s'agit d'une cartographie où l'utilisateur est acteur de sa découverte d'informations : il zoome, il change de fond de carte, il ajoute ou modifie des informations.

D'une manière simplifiée, la cartographie dynamique regroupe l'ensemble des technologies permettant d'afficher une carte sur le Web.

Ces technologies reposent principalement sur les trois composantes que sont le client, le serveur et les données. La cartographie dynamique permet donc, en fonction d'une requête d'un client envoyée au serveur cartographique, de retourner les données désirées sous la forme d'une carte (voir figure 1.1).

Figure 1.1 — Organisation d'une application de cartographie numérique

Une application reposant sur l'API Google Maps est un exemple représentatif de ces nouvelles technologies de cartographie dynamique. L'internaute, *via* son client web – son navigateur – demande aux serveurs Google les cartes qu'il désire visualiser à travers l'interface de navigation. Google lui restitue cette information sous la forme d'images, appelées « tuiles », assemblées dans l'interface de navigation. À chaque nouvelle demande – un déplacement par exemple – l'application web repose une requête aux serveurs, qui envoient l'information correspondante ; cet échange se fait de manière transparente, les interactions entre le client et le serveur devant être les moins intrusives possible.

1.2 L'INTERACTION CLIENT/SERVEUR

1.2.1 AJAX

AJAX (*Asynchronous JavaScript and XML*) est la terminologie qui a été employée pour désigner les applications riches, dynamiques et interactives qui ont pour support Internet et qui fonctionnent donc à l'aide de navigateurs web.

L'utilisation de la classe `XmlHttpRequest` – introduite par Microsoft en 2000 – a rendu possible l'émergence d'un nouveau type d'applications pour le Web. Celles-ci sont fluides et rapides et ne nécessitent pas le rechargement complet de la page web sur laquelle l'internaute navigue lorsqu'il sollicite des ressources provenant du serveur.

Cette interaction entre client et serveur *via* cette classe `XmlHttpRequest` est à la base du Web 2.0. L'augmentation constante des débits de connexion à Internet et l'amélioration continue des navigateurs font des applications Web de véritables logiciels à part entière.

1.2.2 Google Maps, le meilleur exemple d'application AJAX

Apparu en 2004, Google Maps est un service gratuit de cartographie en ligne. La possibilité de naviguer à travers le monde entier de manière fluide avec une simple connexion Internet a fait de ce projet un véritable succès.

En 2010, le Web compte près de 350 000 sites proposant une composante de cartographie utilisant la technologie Google Maps. Le succès de cette API a été et reste toujours la mise à disposition d'une interface de programmation gratuite à destination des développeurs, leur permettant d'intégrer de la cartographie dynamique dans leurs propres sites web.

1.3 ENVIRONNEMENT DE DÉVELOPPEMENT

Le développement de sites web ne nécessite que peu d'outils spécifiques et est par conséquent peu onéreux sur un plan matériel ; un investissement léger permet de commencer immédiatement.

Un simple éditeur de texte sera donc amplement suffisant pour commencer à programmer des cartes dynamiques *via* l'API Google Maps : Gedit sous Unix ou Notepad++ sous Windows par exemple. Cependant, un éditeur avec coloration syntaxique est vivement conseillé.

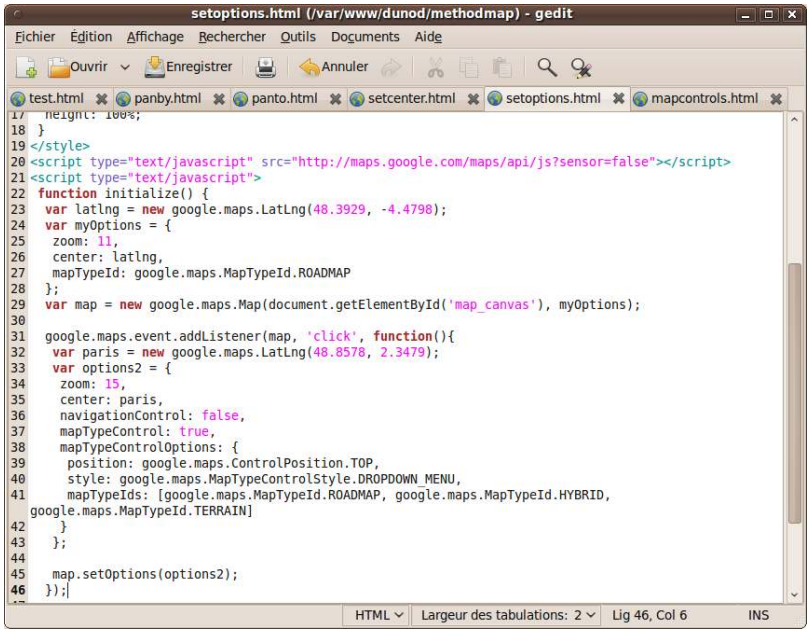

Figure 1.2 — Éditeur de texte classique avec coloration syntaxique

Un navigateur web récent devra également être ouvert en permanence afin de jongler entre le code saisi sur l'éditeur de texte et le résultat sur le Web.

Plusieurs versions de navigateurs de plusieurs sociétés seront toutefois nécessaires. En effet, les différents navigateurs du marché n'interprètent pas de la même façon un même code JavaScript. C'est malheureusement un handicap pour le développement d'applications web riches et dynamiques, mais parions que les principaux acteurs du marché des navigateurs suivent très prochainement les spécifications du W3C.

Note : HTML 5 permettra peut-être de réduire ces problèmes de compatibilité entre les différents navigateurs du marché. Google a fait ce pari avec Chrome ; espérons que tous les navigateurs respectent enfin les mêmes normes...

1.4 SERVEUR HTTP ET SCRIPTING CÔTÉ SERVEUR

Pour développer des cartes avec l'API Google Maps, il est nécessaire de posséder un serveur HTTP en service. En effet, les cartes Google Maps sont imbriquées dans des pages web et pour les déployer, un serveur de type Apache est obligatoire.

Dans le chapitre 3 consacré à l'affichage de données sur une carte Google Maps, nous verrons comment utiliser des ressources stockées en base de données. Alors, nous aurons besoin d'un langage de script côté serveur permettant la connexion à la base de données et d'un système de gestion de base de données.

Pour cet ouvrage, le choix s'est porté sur PHP comme langage de script côté serveur et MySQL comme système de gestion de bases de données.

Nous avons donc besoin d'une architecture du style WAMP ou LAMP :

- **W** pour Windows et L pour Linux,
- **A** pour Apache,
- **M** pour MySQL,
- **P** pour PHP.

Sur Windows, WampServer peut tout à fait convenir pour la phase de développement. Lors de la mise en production des pages web, le recours à un administrateur web est toutefois conseillé pour optimiser les deux serveurs Apache et MySQL.

Note : tout autre serveur de base de données (PostgreSQL par exemple) ou serveur HTTP (lighthtpd par exemple) ou langage de script (Python par exemple) peuvent être utilisés pour produire des cartes grâce à l'API Google Maps.

1.5 LANGAGES DE PROGRAMMATION

Le développement avec l'API Google Maps nécessite des connaissances en HTML et en JavaScript.

1.5.1 HTML

HTML est un format de données permettant la représentation des pages web. Les différentes balises permettent de structurer l'information, d'ajouter des images et des contenus multimédias. On utilise souvent HTML en conjonction avec des langages de programmation afin de fluidifier la navigation et de donner aux pages des qualités de réactivité et d'interaction.

1.5.2 JavaScript

Le principal langage de programmation utilisé en conjonction avec HTML est JavaScript.

L'API Google Maps est développée en JavaScript. La tâche pour les développeurs familiers avec ce langage de script en sera bien évidemment facilitée.

2

Débuter avec l'API Google Maps

Objectifs

Le but de ce chapitre est de se familiariser avec l'API Google Maps ; il exposera les outils de développement à mettre en place avant de commencer un projet.

Nous reviendrons sur les langages de programmation qui seront utilisés. Suite à cette introduction, nous commencerons avec une première carte simple, puis enchaînerons sur l'élément principal de l'API : l'objet Map.

La notion d'événement sera expliquée et nous verrons l'utilisation de ces événements pour interagir avec la carte.

2.1 RETOUR HISTORIQUE SUR L'API

2.1.1 Cinq ans déjà !

L'API Google Maps vient de fêter ces 5 ans. Désormais, dans sa troisième version, plus de 350 000 sites web l'utilisent pour afficher des cartes dynamiques.

Google Maps est d'abord connu pour son célèbre site Internet : http://maps.google.fr. Bien que n'étant pas le premier à proposer un tel service et il est devenu le plus populaire d'entre eux.

L'API a été proposée aux développeurs en juin 2005. Depuis lors, il s'agit de l'interface de programmation pour Internet la plus utilisée à travers le monde. En

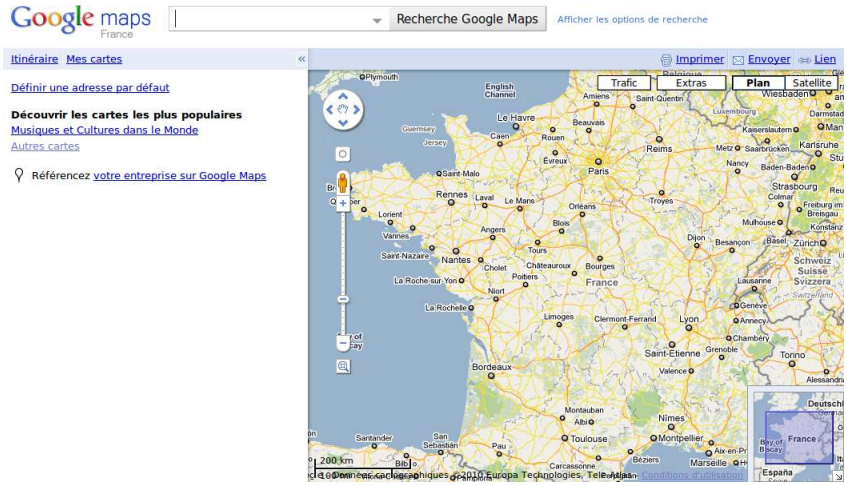

Figure 2.1 — Page d'accueil du site Google Maps

effet, la vitesse d'affichage, les nombreuses possibilités de l'API, la finesse de la cartographie et la gratuité commerciale (dans certaines mesures) ont poussé nombre de développeurs à s'intéresser à ce mode de représentation des données via une cartographie en ligne dynamique.

Définition : une API est une interface de programmation. Dans le cas de Google Maps, il s'agit d'un ensemble de fonctions et classes JavaScript qui permettent de manipuler une carte dynamiquement au sein d'un site web.

Cette manière de visualiser des informations laisse souvent libre court à l'internaute afin que celui-ci navigue au grès de ses envies à travers la carte. Le fait de pouvoir ajouter des objets ou de l'information permet aux développeurs web de guider les visiteurs sur la carte et sur le site.

La première version destinée aux seuls sites web s'est vue agrémentée au fil des années de plates-formes supplémentaires : API pour Flash, API statique (*cf.* annexe B), API pour les smartphones (*cf.* seconde partie). De plus, au fur et à mesure de l'exploitation de cette API, Google a proposé de nouveaux services : géolocalisation, calcul d'itinéraires, calcul d'altitude, etc.

De nombreux sites web proposent une carte provenant de Google Maps dans leurs pages :

- *de type professionnel* : SeLoger.com, par exemple, utilise l'API Google Maps comme support d'informations immobilières (figure 2.2).
- *à but promotionnel* : l'entreprise Jan Staellert propose un concours pour choisir le bâtiment le plus laid de Belgique via le site http://www.lebatimentlepluslaid.be (figure 2.3). L'entreprise se propose alors de détruire le bâtiment choisi – évidemment, il s'agit d'une entreprise de démolition...

Figure 2.2 — Mashup Google Maps SeLoger.com

Figure 2.3 — Mashup Google Maps : Jan Staellert

- *à but institutionnel* : le site http://www.marinetraffic.com/ais (figure 2.4) permet de voir en temps réel l'ensemble des bateaux à haut-tonnage sur une carte mondiale.

Figure 2.4 — Mashup Google Maps : MarineTraffic.com

- *à but de loisir* : le site http://demo.geogarage.com/cassini (figure 2.5) propose une superposition des cartes anciennes de Cassini sur un fond Google Maps.

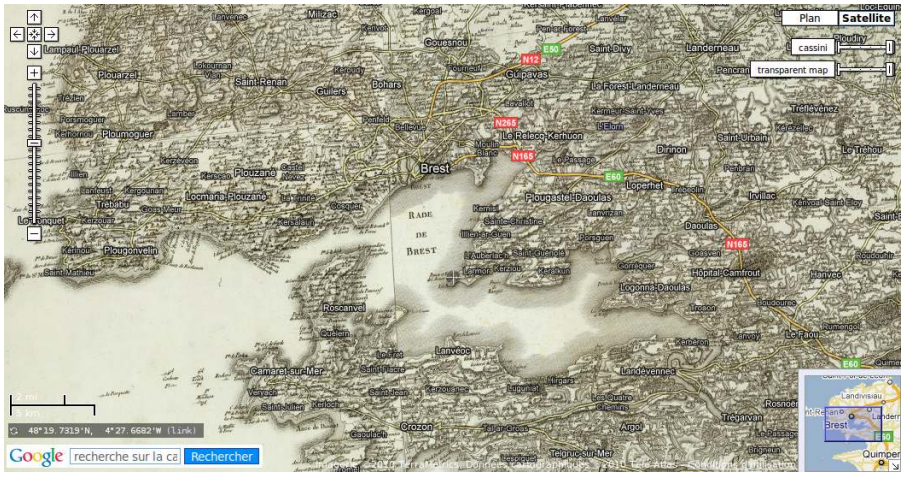

Figure 2.5 — Mashup Google Maps : cartes de Cassini

Ainsi, la version 3 de l'API, officialisée en mai 2010, propose dorénavant toute une palette de services à destination des utilisateurs tout en facilitant la tâche des développeurs.

L'API Google Maps est gratuite si les points suivants sont respectés :

- la cartographie résultante doit être accessible de manière gratuite (un identifiant et un mot de passe peuvent être demandés pour accéder à la carte mais l'accès au site web doit être libre et gratuit) ;
- ne pas utiliser l'API Google Maps dans un environnement Intranet hormis dans le cadre d'un développement destiné pour la suite au grand public ;
- dans des cas d'utilisation pour des usages internes et/ou privés, Google propose le service Google Maps API Premier.

Google a depuis 2005 fait évoluer son concept de cartographie dynamique en l'adaptant à de nombreux autres sujets :

- la Lune (et Google Moon, voir figure 2.6) : une interface similaire à Google Maps mais avec des images satellites de la Lune. Les images proviennent de la NASA ;
- les étoiles (et Google Sky, voir figure 2.7) : une interface similaire avec des images de l'espace prises par le satellite Hubble ;
- et beaucoup d'autres encore : Google Latitude, Google Aerial, Google My Maps, Google StreetView.

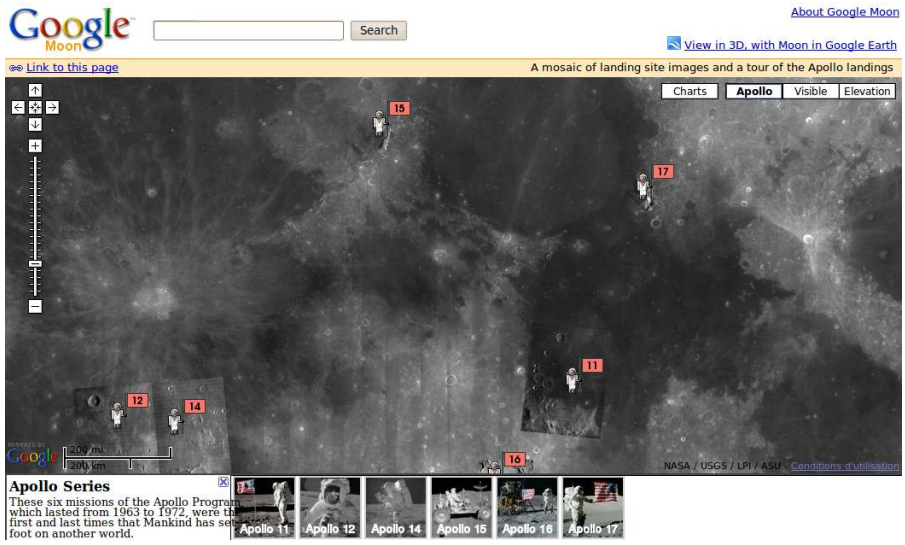

Figure 2.6 — Google Moon

2.1.2 De nombreuses fonctionnalités

Les API cartographiques dynamiques permettent de générer des cartes qui seront intégrées dans des pages web à l'aide de code JavaScript. Elles proposent donc toute une panoplie de services et d'outils à destination de l'internaute afin de l'aider dans la navigation au sein d'une carte :

Figure 2.7 — Google Sky

- contrôles de zoom,
- contrôles de navigation,
- contrôles de choix du type de carte (carte normale, carte relief, carte satellite ou hybride),
- contrôle d'échelle,
- marqueurs et infobulles associées,
- polylignes et polygones,
- événements pour tout type d'actions sur les éléments de la carte,
- service de géocodage à l'adresse,
- StreetView,
- itinéraires,
- superposition de couches.

2.1.3 API version 3

La troisième version est actuellement conseillée par Google pour tout nouveau développement et mise en production. La version 2 est passée en mode « *deprecated* », soit déconseillé pour la production de nouveaux *mashups*.

> **Définition** : un mashup permet d'ajouter sur une page web du contenu provenant d'un autre site. En l'occurrence dans notre cas, il s'agit de Google Maps.

Cette nouvelle API, à la différence de la version 2, repose sur le concept MVC (modèle-vue-contrôleur) qui a permis d'alléger considérablement la taille du code JavaScript et par conséquent d'améliorer la fluidité de la navigation.

Elle est également spécialement développée pour pouvoir être affichée sur de petits écrans, tels que ceux qui équipent les smartphones (iPhone et Android en tête). Ainsi, plus besoin de développement spécifique pour chacune des plates-formes dans le cas d'application ne nécessitant pas l'usage des capteurs spécifiques aux smartphones : GPS, accéléromètres, etc. (dans ce dernier cas, il faudra développer avec les API spécifiques à ces plates-formes : voir la deuxième partie du livre).

De nombreuses simplifications ont été également apportées à cette nouvelle mouture afin de rendre le code plus lisible. Nous verrons des exemples de code dans les chapitres suivants.

Note : avec cette version, il n'est plus nécessaire de s'enregistrer auprès de Google pour obtenir une clé d'utilisation de l'API.

2.2 PREMIÈRE CARTE

2.2.1 Initialisation

Le premier exemple commence toujours par un « Hello World! ». Ne dérogeons pas à cette règle et codons notre première carte.

Définissons tout d'abord le corps de la page HTML pour revenir ensuite à l'en-tête :

```
<body onload="initialize()">
 <div id="map_canvas" style="width: 100%; height: 100%;"></div>
</body>
```

- le bloc `map_canvas` contiendra la carte Google Maps. Il est donc défini dans le corps de la page HTML ;
- la fonction `initialize` sera appelée dès l'ouverture de la page dans le navigateur Internet de l'utilisateur. Elle aura la charge de commander l'affichage de la carte dans le bloc `map_canvas`.

Revenons maintenant à la section `HEAD` de la page HTML ; celle-ci doit contenir les feuilles de styles ainsi que les fonctions JavaScript nécessaires au fonctionnement de la page web. Nous trouverons donc l'appel à l'API Google Maps et la fonction `initialize`.

- la feuille de styles CSS définissant une carte en pleine page sans bordure :

```
<style type="text/css">
 html {
  height: 100%;
 }
 body {
```

```
    height: 100%;
    margin: 0px;
    padding: 0px;
   }
   #map_canvas {
    height: 100%;
   }
  </style>
```

- l'appel à l'API Google Maps :

```
<script type="text/javascript"
  src="http://maps.google.com/maps/api/js?sensor=false" ></script>
```

Le paramètre `sensor` doit nécessairement être indiqué : il indique à l'API si l'application utilisera la position géographique de l'utilisateur ou non. Ce paramètre est notamment important pour les applications mobiles.

- enfin, la fonction `initialize` définissant la carte. Une des nouveautés de l'API v3 est le fait qu'il faille initialiser les paramètres de position et de zoom du centre, ainsi que les paramètres de type de carte avant de construire l'objet principal. Dans un souci de performance, ces options seront des objets « non construits » et donc « littéraux » :

```
<script langage="text/javascript">
 function initialize() {
  var latlng = new google.maps.LatLng(48.3929, -4.4798);
  var myOptions = {
   zoom: 11,
   center: latlng,
   mapTypeId: google.maps.MapTypeId.ROADMAP
  };
  var map = new google.maps.Map(document.getElementById(
                                "map_canvas"), myOptions);
 }
</script>
```

Nous voyons que l'espace de nommage des objets est `google.maps.*`.

Le premier objet `latlng` de la classe `LatLng` est un point géographique défini par ses coordonnées géographiques latitude (entre –90° et +90°) et longitude (entre –180° et +180°).

Note : 48,3929 degrés de latitude et –4,4798 degrés de longitude est un point basé sur Brest.

Le second objet `map` de la classe `Map` désigne la carte Google définie par le bloc `map_canvas` et par des options :

- `zoom` : le niveau de zoom par défaut (chiffre entre 1 et 20) 7 ;
- `center` : le point définissant le centre de la carte ;
- `mapTypeId` : le fond de carte initial.

Ces trois propriétés sont requises pour déclarer un objet de la classe `Map`.

Le code final de cette première carte est donc le suivant :

```
<!DOCTYPE html>
<html>
<head>
<title>Développez avec les API Google Maps</title>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<meta http-equiv="content-type" content="text/html; charset=UTF-8" />
<style type="text/css">
 html {
  height: 100%;
 }
 body {
  height: 100%;
  margin: 0px;
  padding: 0px;
 }
 #map_canvas {
  height: 100%;
 }
</style>
<script type="text/javascript"
src="http://maps.google.com/maps/api/js?sensor=false" ></script>
<script type="text/javascript">
 function initialize() {
  var latlng = new google.maps.LatLng(48.3929, -4.4798);
  var myOptions = {
   zoom: 11,
   center: latlng,
   mapTypeId: google.maps.MapTypeId.ROADMAP
  };
  var map = new google.maps.Map(
        document.getElementById("map_canvas"), myOptions);
 }
</script>
</head>
<body onload="initialize()">
 <div id="map_canvas" style="width: 100%; height: 100%;"></div>
</body>
</html>
```

Dans cet exemple, nous déclarons l'application comme du HTML 5 en utilisant la balise `<!DOCTYPE html>` et nous indiquons que la carte pourra être déclarée en pleine page et ne pourra pas être redimensionnée par l'internaute. Cette spécification est nécessaire pour le navigateur Safari de l'iPhone.

Note : la taille de la carte est définie en CSS par la déclaration du bloc DIV `map_canvas` contenant la carte. Il n'y a pas de restrictions de taille pour une carte élaborée avec l'API Google Maps.

Figure 2.8 — Première carte Google Maps

2.2.2 Optimisation pour smartphone

La majorité des connexions Internet se fera dans les prochaines années *via* des smartphones. L'API Google Maps v3 est optimisée pour un affichage sur ce type de plate-forme : taille et résolution de l'écran moins importante et débit inférieur notamment en 3G.

Pour afficher par défaut une carte en plein écran pour smartphone (voir la figure 2.9) et en définissant la taille en pixels pour tous les autres navigateurs, introduisons une nouvelle fonction qui à l'ouverture de la page détectera le type de plate-forme et appliquera le style approprié.

Il s'agit simplement d'une fonction `detectBrowser` qui modifie le style par défaut de quelques éléments de navigation :

```
function detectBrowser() {
 var useragent = navigator.userAgent;
 var mapdiv = document.getElementById("map_canvas");
 if (useragent.indexOf('iPhone') != -1 || useragent.indexOf('Android') != -1)
{
  mapdiv.style.width = '100%';
  myOptions = {
   navigationControlOptions : {
    style : google.maps.NavigationControlStyle.ANDROID
   },
   mapTypeControlOptions : {
    style : google.maps.MapTypeControlStyle.DROPDOWN_MENU
   }
  };
  map.setOptions(myOptions);
 } else {
  mapdiv.style.width = '100%';
 }
}
```

Figure 2.9 — Première carte avec le navigateur de l'iPhone

2.3 L'ÉLÉMENT PRINCIPAL : L'OBJET MAP

Comme nous l'avons vu, la classe JavaScript `Map` représente une carte.

2.3.1 Les contrôles

Contrôle de navigation

Le contrôle de navigation est l'ensemble composé des boutons de zoom Plus et Moins et des flèches de déplacement. Nous avons déjà vu dans l'exemple précédent qu'il était possible de modifier ce contrôle en fonction du type de plate-forme de l'internaute.

Les contrôles de navigation sont définis par l'objet `NavigationControlOptions` qui possède deux propriétés : `style` et `position`.

Les quatre contrôles de navigation différents proposés par l'API sont définis par la propriété `style` de type :

- `ANDROID` : seulement deux boutons pour les zooms Plus et Moins. Les flèches de déplacement ont été supprimées ;
- `DEFAULT` : les contrôles par défaut de l'API. Celle-ci choisit automatiquement l'interface de contrôle la plus appropriée en fonction de la taille de la fenêtre par exemple. Dans le premier exemple, en l'absence de définition du contrôle de navigation, la valeur par défaut avait été choisie ;
- `SMALL` : affiche seulement les boutons de zoom Plus et Moins ;

- `ZOOM_PAN` : affiche le contrôle de navigation le plus complet avec flèche de direction Nord, Sud, Est et Ouest, et boutons de zoom Plus et Moins. Ce contrôle complet ressemble fortement à l'outil de navigation dans le logiciel Google Earth.

La position sur la carte du contrôle de navigation est définie par la propriété `position` du type `ControlPosition` et prend les valeurs suivantes :

- `BOTTOM` : l'élément est placé en bas au milieu de la carte ;
- `BOTTOM_LEFT` : l'élément est placé en bas à gauche de la carte juste à droite du logo Google ;
- `BOTTOM_RIGHT` : l'élément est placé en bas à droite de la carte juste à gauche des copyrights ;
- `LEFT` : l'élément est placé à gauche, juste au-dessous des éléments positionnés en haut à gauche ;
- `RIGHT` : l'élément est placé à droite, juste au-dessous des éléments positionnés en haut à droite ;
- `TOP` : l'élément est placé au milieu en haut de la carte ;
- `TOP_LEFT` : l'élément est placé en haut à gauche de la carte ;
- `TOP_RIGHT` : l'élément est placé en haut à droite de la carte.

Figure 2.10 — Différents contrôles de navigation

Les contrôles de navigation font partie des options de l'objet `Map` principal. Ils se déclarent donc dans ces dernières.

Voici un exemple déclarant un contrôle de navigation réduit positionné en haut à droite :

```
var myOptions = {
 zoom : 11,
 center : latlng,
```

```
  mapTypeId : google.maps.MapTypeId.ROADMAP,
  navigationControl: true,
  navigationControlOptions : {
   style : navigationControlOptionsStyle.SMALL,
   position : TOP_RIGHT
  }
 };
```

L'état initial du contrôle de navigation est contrôlé par le paramètre booléen `navigationControl` de l'objet `Map`.

Supplément Web : vous pouvez voir les différents types de contrôle de navigation ainsi que des positions différentes des contrôles sur la page http://www.geotribu.net/dunod/gmaps/control/navigationcontrols.html.

Notez l'utilisation de la méthode `bindTo` de la classe `MVC` que nous réutiliserons souvent dans les exemples de cet ouvrage.

Contrôle du type de carte

Google met à disposition des internautes quatre fonds cartographiques différents, qui sont souvent définis par les termes Terrain, Satellite, Mixte et Relief.

L'outil de sélection du type de carte se fait par l'objet `MapTypeControlOptions` qui possède trois propriétés : `mapTypeIds`, `position` et `style`.

De la même manière que les contrôles de navigation, le contrôle du type de carte se définit dans les options de l'objet `Map`.

Les différents types de carte sont :

- `ROADMAP` : la carte routière par défaut de Google ;
- `SATELLITE` : la carte satellite obtenue par satellite ou par ortho-photographie aérienne ;
- `HYBRID` : superposition de la carte satellite avec des éléments de la carte routière classique ;
- `TERRAIN` : la carte topographique représentant le relief et quelques informations importantes.

L'exemple suivant montre comment déclarer un choix de deux cartes à l'utilisateur, la carte routière classique et l'imagerie satellite :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 mapTypeControl: true,
 mapTypeControlOptions: {
  mapTypeIds: [google.maps.MapTypeId.ROADMAP,
               google.maps.MapTypeId.SATELLITE]
 }
};
```

Le positionnement de l'outil est défini de la même manière que pour le contrôle de navigation : il peut prendre huit valeurs différentes.

Voici l'exemple d'un outil de choix de carte positionné en bas à droite de la carte (juste à gauche des copyrights) :

```
var myOptions = {
 zoom: 1,
 center: latlng,
 mapTypId: google.maps.MapTypeId.ROADMAP,
 mapTypeControl: true,
 mapTypeControlOptions: {
  position: google.maps.ControlPosition.BOTTOM_RIGHT
 }
};
```

La troisième propriété de cet outil de contrôle du type de carte est le style. En effet, l'API propose deux types de style complété comme pour le contrôle de navigation d'un style par défaut qui choisit automatiquement quel style appliquer en fonction de la plate-forme de l'internaute et de ses propriétés :

- `DEFAULT` : style par défaut qui choisit automatiquement un des deux styles suivants en fonction de la plate-forme de l'internaute et de ses propriétés ;
- `DROPDOWN_MENU` : menu de sélection qui s'ouvre lorsque l'on clique dessus ;
- `HORIZONTAL_BAR` : menu de sélection qui dispose des boutons horizontalement.

Voici un exemple de l'outil de choix de type de carte positionné en haut au milieu de la carte, proposant la sélection de trois types de carte et ayant le style `DROPDOWN_MENU` :

```
var myOptions : {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 navigationControl: false,
 mapTypeControl: true,
 mapTypeControlOptions: {
  position: google.maps.ControlPosition.TOP,
  style: google.maps.MapTypeControlStyle.DROPDOWN_MENU,
  mapTypeIds: [google.maps.MapTypeId.ROADMAP,
               google.maps.MapTypeId.HYBRID,
               google.maps.MapTypeId.TERRAIN]
 }
};
```

L'état initial du contrôle du choix de type de carte est défini par le paramètre booléen `mapTypeControl` de l'objet `Map`.

Supplément Web : vous trouverez plusieurs exemples d'utilisation du contrôle de type de carte à l'adresse http://www.geotribu.net/dunod/gmaps/control/mapcontrols.html.

Note : l'imagerie satellite n'est pas composée exclusivement d'images satellites. Il arrive parfois à Google d'utiliser des orthophotographies aériennes. L'hétérogénéité de ces données explique en partie le mosaïquage important de ce fond de carte.

Contrôle d'échelle

L'API Google Maps propose la possibilité d'ajouter une échelle à la carte. De la même manière que les contrôles précédents, on peut définir une position et un style (pour le moment unique) par le biais des propriétés de l'objet nommé `scaleControlOptions`.

Voici un exemple pour une échelle affichée en haut à droite de la carte, juste au-dessous du contrôle du type de carte :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeControl: true,
 mapTypeControlOptions: {
  position: google.maps.ControlPosition.TOP_RIGHT,
  style: google.maps.MapTypeControlStyle.DEFAULT,
  mapTypeIds: [google.maps.MapTypeId.ROADMAP,
               google.maps.MapTypeId.SATELLITE]
 },
 navigationControl: true,
 navigationControlOptions: {
  position: TOP_RIGHT,
  style: google.maps.NavigationControlStyle.ZOOM_PAN
 }
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 scaleControl: true,
 scaleControlOptions: {
  position: google.maps.ControlPosition.RIGHT,
  style: google.maps.ScaleControlStyle.DEFAULT
 }
};
```

L'état initial de l'outil de contrôle d'échelle est défini par le paramètre booléen `scaleControl` de l'objet `Map`.

Supplément Web : trois exemples d'implantation de l'échelle dans une carte sont disponibles à l'adresse http://www.geotribu.net/dunod/gmaps/ control/scalecontrols.html.

2.3.2 Propriétés de l'objet *Map*

À l'initialisation de la carte, il est possible de définir un certain nombre de propriétés de la carte. Celles-ci vont de la couleur du fond d'écran de la carte à la possibilité de zoomer à l'aide de la souris.

Supplément Web : tous les exemples suivants sont disponibles à cette adresse http://www.geotribu.net/dunod/gmaps/property.

- `backgroundColor` : permet de définir la couleur d'arrière plan en attendant que les tuiles composant la carte se chargent et s'affichent. Cette propriété prend comme valeur une chaîne de caractères représentant le code couleur HTML hexadécimal ou sa transcription en anglais.

L'exemple ci-dessous affichera un fond rouge en attendant le chargement des tuiles :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 backgroundColor: '#FF0000'
};
```

- `center` : paramètre obligatoire, il désigne le centre de la carte lors de l'initialisation. Il doit être un objet de type `LatLng`.

L'exemple ci-dessous déclare un centre de carte initial vers Brest, soit une latitude de 48,3929 degrés et une longitude de –4,4798 degrés :

```
var latlng = new google.maps.LatLng(48.3929, -4,4798);
var mOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

- `disableDefaultUI` : paramètre de type booléen, il permet de masquer tous les contrôles. Si les contrôles sont déclarés actifs dans les options de la carte, alors ces derniers prendront le pas sur ce paramètre.

L'exemple suivant masque tous les contrôles :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 disableDefaultUI: true
};
```

- `disableDoubleClickZoom` : paramètre de type booléen, il définit la possibilité de zoomer en double-cliquant avec la souris. Par défaut, cette fonctionnalité est activée.

L'exemple suivant désactive le zoom en double-cliquant avec la souris :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 disableDoubleClickZoom: true
};
```

Note : un double-clic avec le bouton gauche de la souris effectue un zoom avant, un double-clic avec le bouton droit de la souris effectue un zoom arrière.

- `draggable` : de type booléen, ce paramètre définit la possibilité ou non de naviguer à l'aide la souris.

L'exemple ci-dessous empêche l'internaute de se déplacer dans la carte autrement que par les flèches de navigation :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 draggable: false
};
```

- `draggableCursor` : de type chaîne de caractères, définit le pointeur de souris par défaut. Il peut prendre comme valeur toutes les valeurs possibles des curseurs en CSS ou une URL d'un curseur.

L'exemple suivant présente un curseur en forme de croix pour tous les objets qui peuvent être déplacés – la carte en fait partie :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 draggableCursor: 'crosshair'
};
```

- `draggingCursor` : identique au paramètre précédent, mais pour les objets en cours de déplacement.

L'exemple suivant présente une combinaison des deux paramètres de changement de curseur :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 draggableCursor: 'crosshair',
 draggingCursor: 'move'
};
```

- `keyboardShortcuts` : paramètre de type booléen. Par défaut, il active la possibilité de se mouvoir dans la carte grâce aux flèches de navigation.

L'exemple suivant désactive la possibilité de naviguer avec les flèches de direction :

```
var myOptions = {
 zoom: 11,
 center: latlng,
```

```
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 keyboardShorcuts: false
};
```

- `mapTypeId` : paramètre obligatoire, définit le type de carte par défaut. Doit être défini par un identifiant de la classe `MapTypeId`.

L'exemple suivant précise une carte satellite par défaut :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.SATELLITE
};
```

- `noClear` : de type booléen, précise la possibilité d'effacer le contenu du bloc HTML qui contient la carte.
- `scrollwheel` : de type booléen, active ou non la possibilité de zoomer grâce à la molette de la souris. Ce paramètre est activé par défaut.

L'exemple suivant désactive ce paramètre :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 scrollwheel: false
};
```

- `streetView` et `streetViewControl` : sont abordés par la suite à la section 4.4, qui leur est consacrée.
- `zoom` : paramètre obligatoire de type entier. Ce paramètre définit le niveau de zoom initial de la carte. Celui-ci est compris entre 1 et 21 en fonction de la précision voulue.

L'exemple suivant définit une carte avec un zoom initial à 7 :

```
var mOptions = {
 zoom: 7,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP
};
```

2.3.3 Événements sur l'objet *Map*

De même que pour les autres objets de l'API Google Maps, la carte – objet `Map` – interagit avec des événements. La carte réagit donc en fonction des événements qui pourront se produire.

Supplément Web : tous les exemples suivants sont disponibles à l'adresse http://www.geotribu.net/dunod/gmaps/eventmap.

- `click` : événement clic sur la carte. Le développeur d'une application Google Maps peut décider d'intercepter un clic sur la carte pour afficher des informations.

Dans l'exemple suivant, un clic sur la carte provoque l'ouverture d'une simple alerte JavaScript. Nous reprenons l'exemple simple du chapitre précédent :

```
function initialize() {
 var latlng = new google.maps.LatLng(48.3929, -4.4798);
 var myOptions = {
  zoom: 11,
  center: latlng,
  mapTypeId: google.maps.MapTypeId.ROADMAP
 };
 var map = new google.maps.Map(document.getElementById("map_canvas"),
                               mOptions);

 google.maps.event.addListener(map, 'click', function(){
  alert("Vous avez cliqué sur la carte.");
 });
}
```

Nous avons utilisé la méthode `addListener` de la classe `MapEventListener`. Le premier argument est l'objet sur lequel l'événement doit être appliqué, le deuxième argument est le type d'événement, le troisième argument est la fonction qui sera exécutée lorsque le programme aura intercepté cet événement.

Ici, lors d'un clic sur tout endroit de la carte, une alerte JavaScript s'ouvre.

Figure 2.11 — Événement `click`

Note : nous verrons dans la section consacrée aux méthodes de l'objet `Map`, comment lier des événements avec des actions sur la carte elle-même.

- `dblclick` : événement double-clic sur la carte. Similaire à l'événement `click`.

```
google.maps.event.addListener(map, 'dblclick', function(){
 alert("Vous avez double-cliqué sur la carte.");
});
```

Note : les événements `click` et `dblclick` ne fonctionnent qu'avec le clic gauche de la souris.

- `dragend` : événement qui intervient lors de la fin d'un déplacement de la carte effectué à l'aide de la souris.

```
google.maps.event.addListener(map, 'dragend', function(){
 alert("Événement de fin de déplacement avec la souris.");
});
```

- `dragstart` **:** identique à l'événement précédent mais déclenché lors du début d'un déplacement de la carte avec la souris.
- `drag` : identique à l'événement précédent mais déclenché chaque fois que la carte est déplacée avec la souris.

Conseil : il vaut mieux éviter de positionner une alerte JavaScript, ce qui mettrait en pause l'application avec cet événement !

- `bounds_changed` : événement qui se déclenche lorsque les coordonnées des coins de la carte sont modifiées.

```
google.maps.event.addListener(map, 'bounds_changed', function(){
 alert('Les coordonnées des coins de la carte ont changé.');
});
```

- `maptypeid_changed` : événement déclenché lorsque le type de carte est modifié.

```
google.maps.event.addListener(map, 'maptypeid_changed', function(){
 alert('Le type de carte a changé.');
});
```

- `mouseover` : événement déclenché lorsque la souris entre dans le bloc HTML contenant la carte.

```
google.maps.event.addListener(map, 'mouseover', function(){
 alert('La souris entre sur la carte.');
});
```

- `mouseout` : événement déclenché lorsque la souris sort du bloc HTML contenant la carte.

```
google.maps.event.addListener(map, 'mouseout', function(){
 alert('La souris sort de la carte.');
});
```

- `rightclick` : événement déclenché lors d'un clic droit sur la carte.

```
google.maps.event.addListener(map, 'rightclick', function(){
 alert('Clic droit');
});
```

- `zoom_changed` : événement déclenché lorsque le zoom de la carte a été modifié.

```
google.maps.event.addListener(map, 'zoom_changed', function(){
 alert('Modification du niveau de zoom de la carte.');
});
```

- `center_changed` : événement déclenché lorsque le centre de la carte a été modifié.

```
google.maps.event.addListener(map, 'center_changed', function(){
 alert('Modification du centre de la carte.');
});
```

Remarque : les deux derniers événements `zoom_changed` et `center_changed` risquent d'être appelés très régulièrement. En effet, chaque fois que l'internaute naviguera dans la carte, l'événement sera déclenché. Il convient donc de les utiliser avec prudence et parcimonie.

Nous avons utilisé dans les exemples précédents la méthode `addListener` de la classe `MapEventListener`. Cependant, cette classe en possède plusieurs dont notamment la méthode `addListenerOnce` qui permet de ne déclencher un événement qu'une seule fois ou encore la méthode `clearInstanceListeners` qui supprime les « écouteurs » sur des événements :

```
google.maps.event.addListener(map, 'click', function(){
 alert("Nous avons instancié l'événement clic.\nMais ce clic supprime tous les
'écouteurs' d'événements sur l'objet map grâce à la méthode
clearInstanceListeners déclarée juste après cette alerte.");
 google.maps.event.clearInstanceListeners(map);
});
```

La méthode `clearListeners`, quant à elle, supprime un événement sur un objet donné :

```
google.maps.event.addListener(map, 'click', function(){
 alert('Un simple événement clic.');
});
```

```
google.maps.event.addListener(map, 'rightclick', function(){
 alert("Un clic droit supprime l'événement click de l'objet map.");
 google.maps.event.clearListeners(map, 'click');
});
```

2.3.4 Méthodes de l'objet *Map*

Tout comme les autres objets de l'API Google Maps, l'objet principal `Map` comporte une quinzaine de méthodes. Celles-ci sont divisées principalement entre des *getter* et des *setter* : qui permettent soit de récupérer le contenu d'une variable, soit de modifier cette variable (on les appelle les accesseurs d'attributs).

Supplément Web : tous les exemples de ce chapitre sont disponibles à l'adresse suivante http://www.geotribu.net/dunod/gmaps/methodmap.

Les exemples suivants utilisent tous les événements de l'objet `Map` vus au chapitre précédent pour pouvoir interagir avec la carte. Il s'agit de clic gauche, de double-clic ou de clic gauche sur la carte.

- `getCenter` : récupère le centre de la carte courante sous la forme d'un objet `LatLng`.

```
var latlng = new google.maps.LatLng(48.3929, -4.4798);
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP
};
var map = new google.maps.Map(document.getElementById('map_canvas'),
myOptions);
google.maps.event.addListener(map, 'center_changed', function(){
 var currentcenter = map.getCenter();
 var latcenter = currentcenter.lat();
 var lngcenter = currentcenter.lng();
 alert('Le centre de la carte est le point de coordonnées :
       '+latcenter+' ,
       '+lngcenter);
});
```

Note : pour récupérer les valeurs des latitude et longitude d'un objet `LatLng`, il faut utiliser les méthodes `lat` et `lng`.

- `setCenter` : méthode qui modifie le centre de la carte. Prend en argument un objet de type `LatLng`.

```
var paris = new google.maps.LatLng(48.8578, 2.3479);
google.maps.event.addListener(map, 'click', function(){
 map.setCenter(paris);
 alert('Direction Paris');
});
```

```
google.maps.event.addListener(map, 'rightclick', function(){
 map.setCenter(latlng);
 alert('Retour à Brest');
});
```

- `getZoom` : retourne le niveau de zoom de la carte.

```
google.maps.event.addListener(map, 'click', function(){
 var mapzoom = map.getZoom();
 alert('Le niveau de zoom est le suivant : '+mapzoom);
});
```

- `setZoom` : modifie le niveau de zoom de la carte. Prend en argument un entier compris entre 1 et 21.

```
google.maps.event.addListener(map, 'click', function(){
 map.setZoom(18);
});
google.maps.event.addListener(map, 'rightclick', function(){
 map.setZoom(2);
});
```

- `getMapTypeId` : retourne le type de carte courant.

```
google.maps.event.addListener(map, 'click', function(){
 alert('Le type de carte courant est : '+map.getMapTypeId());
});
```

- `setMapTypeId` : modifie le type de carte. Prend en argument un objet de type `MapTypeId`.

```
google.maps.event.addListener(map, 'click', function(){
 map.setMapTypeId(google.maps.MapTypeId.SATELLITE);
});
google.maps.event.addListener(map, 'rightclick', function(){
 map.setMapTypeId(google.maps.MapTypeId.TERRAIN);
});
```

- `getBounds` : retourne un rectangle (objet de type `LatLngBounds`) correspondant aux limites visibles de la carte.

```
google.maps.event.addListener(map, 'click', function(){
 var bounds = map.getBounds();
 var northeast = bounds.getNorthEast();
 var southwest = bounds.getSouthWest();
 alert('Les coordonnées du coin haut droit (nord-est) de la carte sont :
'+northeast.lat()+' , '+northeast.lng()+'\nLes coordonnées du coin bas gauche
(sud-ouest) sont : '+southwest.lat()+' , '+southwest.lng());
});
```

Note : Il est également possible d'afficher les coordonnées des coins de la carte en utilisant la méthode `toString` de l'objet `Bounds`. Toutes les

méthodes de cet objet sont documentées dans la référence de l'API : http://code.google.com/intl/fr/apis/maps/documentation/javascript/reference.html.

- `fitBounds` : positionne la carte pour qu'elle s'adapte aux limites données en argument.

```
google.maps.addListener(map, 'click', function(){
  var northeast = new google.maps.LatLng(43, 10);
  var southwest = new google.maps.LatLng(41, 8);
  var bounds = new google.maps.LatLngBounds(southwest, northeast);
  map.fitBounds(bounds);
});
```

- `panBy` : déplace le centre de la carte du nombre de pixels passé en arguments. Le premier argument est le déplacement en X. Le second argument est le déplacement en Y.

```
google.maps.event.addListener(map, 'click', function(){
  map.panBy(100, 100);
});
google.maps.event.addListener(map, 'rightclick', function(){
  map.panBy(-50, -50);
});
```

Note : dans l'exemple précédent, un clic gauche suivi de deux clics droits déplace la carte pour ensuite revenir à sa position initiale.

- `panTo` : déplace le centre de la carte vers le point de coordonnées passé en argument. L'argument est de type `LatLng`.

```
google.maps.event.addListener(map, 'click', function(){
  var berlin = new google.maps.LatLng(52.5287, 13.3588);
  map.panTo(berlin);
});
google.maps.event.addListener(map, 'rightclick', function(){
  map.panTo(latlng);
});
```

- `setOptions` : modifie les options initiales de l'objet `Map`. Prend en argument un objet de type `MapOptions`.

```
google.maps.event.addListener(map, 'click', function(){
 var paris = new google.maps.LatLng(48.8578, 2.3479);
 var options2 = {
  zoom: 15,
  center: paris,
  navigationControl: false,
  mapTypeControl: true,
  mapTypeControlOptions: {
   position: google.maps.ControlPosition.TOP,
   style: google.maps.MapTypeControlStyle.DROPDOWN_MENU,
```

```
    mapTypeIds: [google.maps.MapTypeId.ROADMAP,
                 google.maps.MapTypeId.HYBRID,
                 google.maps.MapTypeId.TERRAIN]
  }
};
```

- `getStreetView` et `setStreetView` : nous verrons ces méthodes à la section 4.4 consacrée à ce service.

2.4 DOCUMENTATION GOOGLE MAPS

L'API Google Maps comporte de nombreuses fonctionnalités que nous essaierons d'introduire dans cet ouvrage. Pour retrouver les multiples classes et objets ainsi que leurs propriétés d'utilisation, il est nécessaire de se référer en permanence au document qui liste toutes les fonctionnalités de l'API : « Google Maps JavaScript API V3 Reference ». Il se trouve à cette adresse : http://code.google.com/intl/fr/apis/maps/documentation/javascript/reference.html.

En résumé

Outre un bref historique sur l'API Google Maps et un rappel sur les langages de programmation qu'il est nécessaire de maîtriser avant de se lancer dans la production de cartes dynamiques sur le Web, ce deuxième chapitre nous a montré comment coder notre première carte afin qu'elle soit compatible notamment avec toutes les plates-formes supportées par l'API Google Maps, soient les smartphones et les navigateurs de postes classiques.

Ce chapitre a été l'occasion de voir en détail l'objet `Map` qui est le socle de toute application Google Maps.

Une carte est donc définie par ses options ou propriétés, par des méthodes propres permettant de manipuler la carte (obtenir des informations sur celle-ci ou se mouvoir géographiquement), par des types de contrôle sur la navigation, le choix du type de carte ou le choix d'une échelle, et enfin sur des événements qui peuvent être interceptés afin de rendre l'application interactive.

3

Afficher des données

Objectifs

Nous verrons dans ce chapitre comment afficher des données sur une carte de l'API Google Maps.

Les marqueurs, polylignes, polygones et fichiers KML ont en commun des coordonnées géographiques. Ils sont donc superposables à une carte Google Maps.

Tout comme la carte principale, ces objets ont des propriétés, des méthodes et des événements propres. Nous verrons donc comment utiliser ces objets pour enrichir notre cartographie dynamique.

3.1 MARQUEURS ET INFOBULLES

Tout le monde a déjà vu un marqueur sur une carte web. La punaise Google s'est imposée dans l'inconscient des internautes comme la solution pour pointer une information sur une carte.

En association avec une infobulle, cette punaise permet l'affichage d'information et pour le moment, rien n'a été inventé de plus efficace. Dans la majorité des applications Google Maps, les marqueurs sont la raison d'être de la carte ; ils servent à indiquer la position géographique de toute sorte d'informations :

- du restaurant au magasin de vêtements,
- d'une mairie à un stade football,
- d'un parc naturel à une île déserte,

- des points formant la trace GPS d'une randonnée,
- et plus généralement, toute information pouvant être positionnée grâce à ses coordonnées.

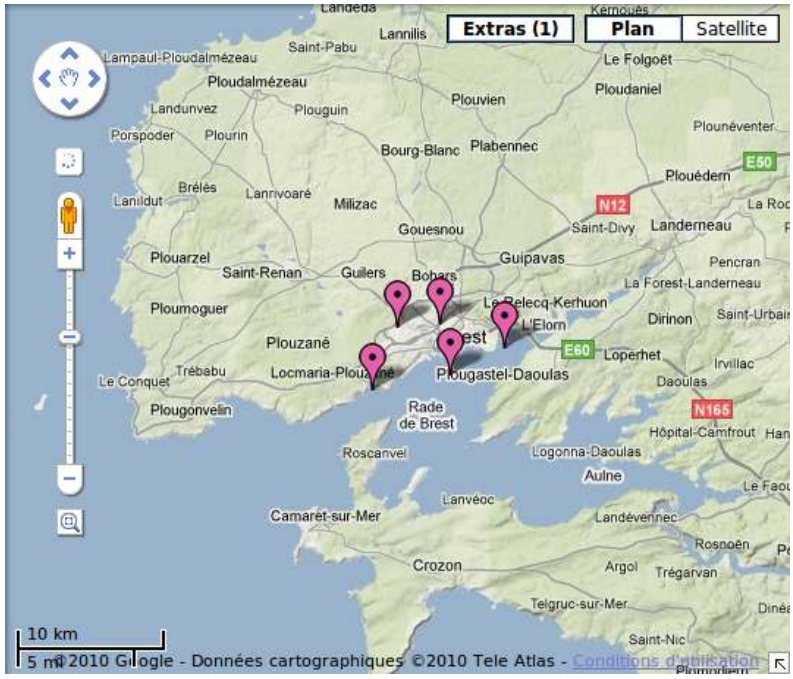

Figure 3.1 — Exemple de marqueurs autour de Brest

3.1.1 Initialisation

Dans l'API Google Maps, la classe `Marker` permet la construction et l'affichage des marqueurs :

```
function initialize(){
 var latlng = new google.maps.LatLng(48.3929, -4.4798);
 var myOptions = {
  zoom: 11,
  center: latlng,
  mapTypeId: google.maps.MapTypeId.ROADMAP
 };
 var map = new google.maps.Map(document.getElementById(
                                'map_canvas'), myOptions);
 var brest = new google.maps.LatLng(48.3929, -4.4798);
 var marker = new google.maps.Marker({
  position: brest,
  map: map
 });
}
```

La figure 3.2 montre la carte produite par ce code.

Figure 3.2 — Exemple de marqueur simple

Seule une option pour le marqueur est nécessaire, `position`. De type `LatLng`, elle indique la position géographique du marqueur.

Pour indiquer sur quelle carte afficher le marqueur, il faut spécifier dans les options du marqueur l'identifiant `map` de type `Map`. Il indique la carte sur laquelle sera superposé le marqueur.

> **Supplément Web** : tous les exemples suivants se trouvent à l'adresse http://www.geotribu.net/dunod/gmaps/displaydata/marker.

3.1.2 Paramètres du marqueur

Les options ou paramètres du marqueur peuvent être passés lors de l'appel au constructeur. L'unique paramètre obligatoire est la position, vue précédemment.

> **Note** : lorsque le paramètre `map` est indiqué, l'API Google Maps se charge d'afficher le marqueur sans qu'on en fasse la demande (à la différence de l'API Google Maps version 2).

- `clickable` : de type booléen. Permet de définir si le marqueur est cliquable ou non :

```
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 clickable: true
});
```

- `cursor` : de type chaîne de caractères. Permet de spécifier le type de pointeur de souris lorsque celle-ci survole le marqueur :

```
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 cursor: 'crosshair'
});
```

- `draggable` : de type booléen. Indique si le marqueur est déplaçable :

```
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 draggable: true
});
```

- `flat` : de type booléen. Indique si l'ombre du marqueur doit être affichée ou non (voir figure 3.3) :

```
var map = new google.maps.Map({
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP
});

var map2 = new google.maps.Map({
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP
});

var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 flat: false
});

var marker2 = new google.maps.Marker({
 position: brest,
 map: map2,
 flat: true
});
```

- `icon` : de type `MarkerImage`. Nous y revenons dans la section suivante qui lui est consacrée ;
- `shadow` : de type `MarkerImage` (voir aussi la section suivante) ;

Figure 3.3 — Propriété `flat`

- `title` : de type chaîne de caractères. Affiche un titre lors du survol du marqueur par la souris. Il s'agit du même comportement que la propriété `title` d'une image en HTML (voir figure 3.4) ;

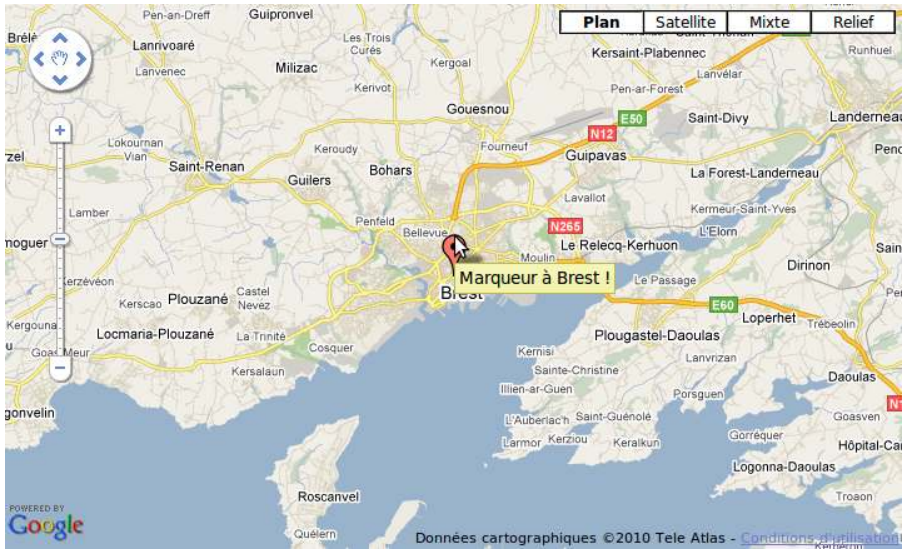

Figure 3.4 — Propriété `title`

- `visible` : de type booléen. Indique si le marqueur est caché ou non :

```
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 visible: true
});
```

- `zIndex` : de type entier. Par défaut, si deux marqueurs se superposent, le marqueur ayant la latitude la plus faible s'affiche au-dessus du second. Si `zIndex` est spécifié,

alors le marqueur ayant le `zIndex` le plus élévé s'affichera au-dessus des autres (voir figure 3.5) :

```
var brest = new google.maps.LatLng(48.3928, -4.4798);
var brest2 = new google.maps.LatLng(48.3927, -4.4798);
var marker = new google.maps.Marker({
 position: brest;
 map: map
});

var marker2 = new google.maps.Marker({
 position: brest2;
 map: map
});
```

Figure 3.5 — Propriété `zIndex`

3.1.3 Icônes du marqueur

Le marqueur Google Maps peut être personnalisé grâce à une classe spéciale `MarkerImage` de l'API. L'icône est alors définie par cinq paramètres :

- `url` : de type chaîne de caractères. Correspond à l'URL de l'image représentant l'icône. Celle-ci doit être accessible par Internet ;
- `size` : de type `Size`. Taille de l'icône en pixel : largeur × hauteur ;
- `origin` : de type `Point`. Correspond à l'origine en pixels de l'image ;
- `anchor` : de type `Point`. Correspond au point d'ancrage. Défini en pixels ;
- `scaledSize` : de type `Size`. Correspond à l'échelle de l'icône :

```
var icontest = new google.maps.MarkerImage('/images/icon.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
```

```
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 icon: icontest
});
```

L'icône utilisée à la figure 3.6 fait 32 pixels de largeur sur 32 pixels de hauteur. Le point d'origine est le point de coordonnées (en pixel) 0 en abscisse et 0 en ordonnée (soit en haut à gauche).

Le point d'ancrage pointera sur le point géographique : il est ici de coordonnées (en pixel) 16 en abscisse (moitié de la largeur) et 32 en ordonnée (totalité de la hauteur).

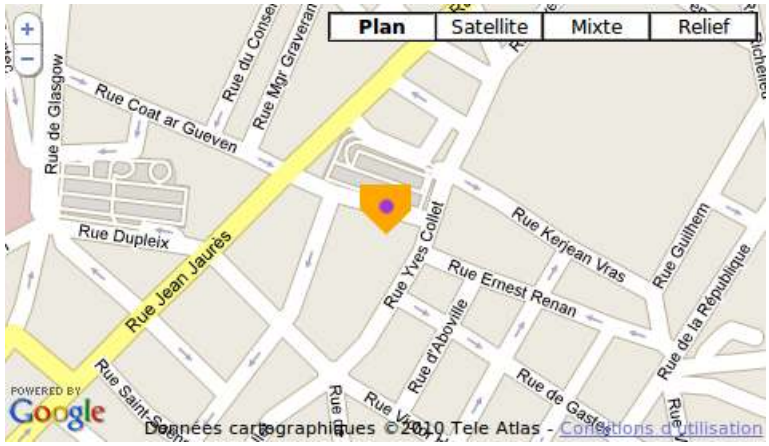

Figure 3.6 — Personnalisation du marqueur

Astuce : avec ces paramètres, nous pouvons utiliser des images comportant plusieurs icônes et sélectionner celle que nous voulons.

Maintenant que l'icône est définie, il est possible de définir une ombre. Nous avons vu dans la partie précédente que celle-ci se définissait avec un objet de type `MarkerImage` également.

Il est possible de construire l'ombre avec un logiciel de dessin (Gimp, Photoshop) mais il existe des logiciels en ligne qui permettent de les générer en quelques secondes (voir par exemple http://www.cycloloco.com/shadowmaker) :

```
var icontest = new google.maps.MarkerImage('./images/icon1.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));

var shadowicontest = new google.maps.MarkerImage('./images/
  shadow-icon1.png', new google.maps.Size(49, 32),
  new google.maps.Point(0, 0), new google.maps.Point(16, 32));

var marker = new google.maps.Marker({
```

```
 position: brest,
 map: map,
 icon: icontest,
 shadow: shadowicontest
});
```

Figure 3.7 — Ombre d'un marqueur personnalisé

Note : l'icône générée par le logiciel en ligne nous a fourni une image légèrement plus large que celle d'origine.

3.1.4 Événements sur les marqueurs

Comme la plupart des objets de l'API Google Maps, les marqueurs peuvent réagir à des événements.

Supplément Web : tous les exemples suivants se trouvent à l'adresse http://www.geotribu.net/dunod/gmaps/displaydata/marker/event.

- `click` : événement déclenché lors d'un clic sur un marqueur. Cet événement est principalement utilisé pour ouvrir une infobulle correspondant à l'objet.

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 clickable: true,
 title: 'Cliquez-moi'
});

google.maps.event.addListener(marker, 'click', function(){
 alert('Evénement click');
});
```

- `clickable_changed` : événement déclenché lorsque le marqueur passe d'un état cliquable à un état non cliquable (ou inversement) :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 clickable: true,
 title: 'Test de l\'événement clickable_changed'
});

google.maps.event.addListener(map, 'click', function(){
 if (marker.getClickable()){
  marker.setClickable(false);
 } else {
  marker.setClickable(true);
 }
});

google.maps.event.addListener(marker, 'clickable_changed',
  function(){
   alert('Etat clic du marqueur : '+marker.getClickable());
  });
```

Le code ci-dessus utilise des méthodes sur l'objet marqueur que nous verrons en détail dans la section suivante.

Il exécute les actions suivantes : lors d'un clic sur la carte, l'état clic du marqueur est modifié, et chaque fois que cet état est modifié, une alerte intervient pour nous donner la valeur de cet état.

- `cursor_changed` : événement déclenché lorsque le pointeur affiché lors du survol de la souris est modifié :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 clickable: true,
 cursor: 'crosshair'
});

google.maps.event.addListenerOnce(marker, 'click', function(){
 marker.setCursor('wait');
});

google.maps.event.addListener(marker, 'cursor_changed', function(){
 alert('Le curseur de la souris a changé.');
});
```

Nous avons utilisé ici la méthode `setCursor` que nous verrons en détail dans la section suivante.

- `dblclick` : événement déclenché lors d'un double-clic sur le marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 clickable: true,
 title: 'Test événement doucleclic'
});

google.maps.event.addListener(marker, 'dblclick', function(){
 alert('Evénement doucle-clic sur le marqueur.');
});
```

- `drag` : événement déclenché lorsque le marqueur est déplacé.

Note : il vaut mieux éviter une alerte JavaScript sur cet événement au risque de planter l'application.

- `dragend` : événement déclenché lorsque le marqueur a fini d'être déplacé :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 draggable: true,
 title: 'Déplacez-moi'
});

google.maps.event.addListener(marker, 'dragend', function(){
 alert('Vous avez déplacé le marqueur.');
});
```

- `dragstart` : événement déclenché au début du déplacement du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 draggable: true,
 title: 'Test événement dragstart'
});

google.maps.event.addListener(marker, 'dragstart', function(){
 alert('Début de déplacement du marqueur.');
});
```

- `draggable_changed` : événement déclenché lorsque l'état de déplacement du marqueur change :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
```

```
 map: map,
 draggable: true,
 title: 'Test événement daggable_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getDraggable()){
  marker.setDraggable(false);
 } else {
  marker.setDraggable(true);
 }
});

google.maps.event.AddListener(marker, 'draggable_changed',
 function(){
 if (marker.getDraggable()){
  alert('Le marqueur est déplaçable.');
 } else {
  alert('Le marqueur n\'est plus déplaçable.');
 }
});
```

Nous avons utilisé dans le code ci-dessus les méthodes `getDraggable` et `setDraggable` que nous verrons à la section suivante. Celles-ci renseignent ou modifient l'état de déplacement du marqueur.

Cet exemple permet de modifier l'état de déplacement du marqueur lorsqu'on clique sur celui-ci.

- `flat_changed` : événement déclenché lorsque l'état de l'ombre est modifié :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 posiion: brest,
 map: map,
 flat: true
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getFlat()){
  marker.setFlat(false);
 } else {
  marker.setFlat(true);
 }
});

google.maps.event.addListener(marker, 'flat_changed', function(){
 if (marker.getFlat()){
  alert('Affichage de l\'ombre.');
 } else {
  alert('L\'ombre n\'est plus affichée.');
 }
});
```

Le code ci-dessus permet de modifier l'état de l'affichage de l'ombre du marqueur lors d'un clic sur celui-ci. Il utilise les méthodes `setFlat` et `getFlat` qui seront détaillées à la prochaine section.

- `icon_changed` : événement déclenché lorsque l'icône du marqueur est modifiée :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var icontest = new google.maps.MarkerImage('./images/icon1.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
var icontest2 = new google.maps.MarkerImage('./images/icon2.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
var shadowicontest = new google.maps.MarkerImage('./images/
  shadow-icon1.png', new google.maps.Size(49, 32),
  new google.maps.Point(0, 0), new google.maps.Point(16, 32));
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 icon: icontest,
 shadow: shadowicontest,
 title: 'Test événement icon_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getIcon() == icontest){
  marker.setIcon(icontest2);
 } else {
  marker.setIcon(icontest);
 }
});

google.maps.event.addListener(marker, 'icon_changed', function(){
 alert('Modification de l\'icône.');
});
```

Le code ci-dessus modifie l'icône lors d'un clic sur celle-ci. Les méthodes `getIcon` et `setIcon` utilisées seront détaillées à la section suivante.

- `mouseover` : événement déclenché lorsque la souris survole le marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 title: 'Test événement mouseover'
});

google.maps.event.addListener(marker, 'mouseover', function(){
 alert('Passage au-dessus du marqueur.');
});
```

- `mouseout`: événement déclenché lorsque la souris quitte la zone au-dessus du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 title: 'Test événement mouseout'
});
google.maps.event.addListener(marker, 'mouseout', function(){
 alert('Le pointeur de la souris quitte l\'icône.');
});
```

- `position_changed` : événement déclenché lorsque la position du marqueur est modifiée :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var paris = new google.maps.LatLng(48.8578, 2.3479);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 title: 'Test événement position_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getPosition() == paris){
  marker.setPosition(brest);
 } else {
  marker.setPosition(paris);
 }
});

google.maps.event.addListener(marker, 'position_changed', function(){
 alert('Modification de la position du marqueur.');
});
```

- `rightclick` : événement déclenché lors d'un clic droit sur le marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 title: 'Test événement rightclick'
});

google.maps.event.addListener(marker, 'rightclick', function(){
 alert('Clic droit.');
});
```

- `shadow_changed` : événement déclenché lorsque l'image de l'ombre du marqueur est modifiée :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var icontest = new google.maps.MarkerImage('./images/icon1.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
var shadowicontest = new google.maps.MarkerImage('./images/
  shadow-icon1.png', new google.maps.Size(49, 32),
```

```
  new google.maps.Point(0, 0), new google.maps.Point(16, 32));
var shadowicontest2 = new google.maps.MarkerImage('./images/
  shadow-icon2.png', new google.maps.Size(49, 32),
  new google.maps.Point(0, 0), new google.maps.Point(16, 32));
var marker = new google.maps.Marker({
  position: brest,
  map: map,
  icon: icontest,
  shadow: shadowicontest,
  title: 'Test événement shadow_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getShadow() == shadowicontest){
  marker.setShadow(shadowicontest2);
 } else {
  marker.setShadow(shadowicontest);
 }
});

google.maps.event.addListener(marker, 'shadow_changed', function(){
 alert('Modification de l\'ombre de l\'icône.');
});
```

Les méthodes `getShadow` et `setShadow` seront détaillées dans la partie suivante.

- `title_changed` : événement déclenché lorsque le titre est modifié :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 title: 'Titre n° 1'
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getTitle() == 'Titre n° 1'){
  marker.setTitle('Titre n° 2');
 } else {
  marker.setTitle('Titre n° 1');
 }
});

google.maps.event.addListener(marker, 'title_changed', function(){
 alert('Le titre du marqueur devient : '+marker.getTitle());
});
```

Les méthodes `setTitle` et `getTitle` seront détaillées à la section suivante.

- `visible_changed` : événement déclenché lorsque la propriété visible est modifiée :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
```

```
 visible: true,
 title: 'Evénement visible_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 marker.setVisible(false);
});

google.maps.event.addListener(marker, 'visible_changed', function(){
 alert('Le marqueur disparaît.');
});
```

La méthode `setVisible` sera détaillée à la section suivante.

- `zindex_changed` : événement déclenché lorsque la valeur du `zIndex` du marqueur est modifiée :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var brest2 = new google.maps.LatLng(48.3927, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 zIndex: 2,
 title: 'Evénement zindex_changed'
});

var marker2 = new google.maps.Marker({
 position: brest2,
 map: map,
 zIndex: 1,
 title: 'Evénement zindex_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 marker2.setZIndex(3);
});

google.maps.event.addListener(marker2, 'zindex_changed', function(){
 alert('Le premier marqueur passe sous le second.');
});
```

La méthode `setZIndex` sera détaillée à la section suivante.

3.1.5 Méthodes sur les marqueurs

Nous avons déjà vu à la section précédente, consacrée aux événements des marqueurs, quelques méthodes de l'objet `Marker`. Ces fonctions permettent de modifier le comportement du marqueur, le déplacer, le positionner sur une autre carte, modifier l'icône, etc.

Supplément Web : tous les exemples suivants se trouvent à cette adresse : http://www.geotribu.net/dunod/gmaps/displaydata/marker/method.

- `getClickable` : renvoie un booléen qui indique si le marqueur est cliquable ou non. Cet état est défini soit lors de l'initialisation (*cf.* la section sur les paramètres des marqueurs) soit avec la méthode `setClickable` :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 clickable: true
});

if (marker.getClickable()){
 alert('Le marqueur est cliquable.');
} else {
 alert('Il n\'est pas cliquable.');
}
```

- `setClickable` : modifie l'état `clickable` du marqueur. Prend en argument un booléen :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 clickable: true
});

marker.setClickable(false);
if (marker.getClickable()){
 alert('Le marqueur est cliquable.');
} else {
 alert('Le marqueur n\'est pas cliquable.');
}
```

- `getCursor` : renvoie la chaîne de caractères correspondant au pointeur de souris :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 cursor: 'crosshair'
});

google.maps.event.addListener(marker, 'click', function(){
 alert('Le pointeur de la souris est le suivant :
       '+marker.getCursor());
});
```

- `setCursor` : modifie le type de pointeur de la souris lors d'un survol du marqueur. Prend en argument une chaîne de valeurs correspondant aux valeurs CSS usuelles de type de curseur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

google.maps.event.addListener(marker, 'click', function(){
 marker.setCursor('crosshair');
});
```

- `getFlat` : renvoie un booléen correspondant à l'état d'affichage de l'ombre du marqueur. Si vrai alors l'ombre n'est pas affichée :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 flat: true
});

google.maps.event.addListener(marker, 'click', function(){
 alert('L\'ombre n\'est pas affichée.');
});
```

- `setCursor` : modifie l'état d'affichage de l'ombre du marqueur. Prend en argument un booléen :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 flat: true
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getFlat()){
  marker.setFlat(false);
 } else {
  marker.setFlat(true);
 }
});
```

- `getIcon` : renvoie l'objet de type `MarkerImage` correspondant à l'icône du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var icontest = new google.maps.MarkerImage('../images/icon1.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
var icontest2 = new google.maps.MarkerImage('../images/icon2.png', new
google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
var shadowicontest = new google.maps.MarkerImage('../images/
```

```
  shadow-icon1.png', new google.maps.Size(49, 32),
  new google.maps.Point(0, 0), new google.maps.Point(16, 32));

var marker = new google.maps.Marker({
 position: brest,
 map: map,
 icon: icontest,
 shadow: shadowicontest
});

var marker2 = new google.maps.Marker({
 position: paris,
 map: map
});

google.maps.event.addListener(marker, 'click', function(){
 marker2.setIcon(marker.getIcon());
});
```

- `setIcon` : modifie l'icône du marqueur. Prend un argument un objet de type `MarkerImage` :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);

var icontest = new google.maps.MarkerImage('../images/icon1.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
var icontest2 = new google.maps.MarkerImage('../images/icon2.png',
  new google.maps.Size(32, 32), new google.maps.Point(0, 0),
  new google.maps.Point(16, 32));
var shadowicontest = new google.maps.MarkerImage('../images/
  shadow-icon1.png', new google.maps.Size(49, 32),
  new google.maps.Point(0, 0), new google.maps.Point(16, 32));

var marker = new google.maps.Marker({
 position: brest,
 map: map,
 icon: icontest,
 shadow: shadowicontest,
 title: 'Cliquez sur le marqueur pour modifier son apparence'
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getIcon() == icontest){
  marker.setIcon(icontest2);
 } else {
  marker.setIcon(icontest);
 }
});
```

- `getMap` : renvoie la carte – un objet de type `Map` – sur laquelle a été initialisé le marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
```

```
 map: map
});

google.maps.event.addListener(marker, 'click', function(){
 alert('Le type de la carte est : '+marker.getMap().getMapTypeId());
});
```

- `setMap` : modifie la carte à laquelle est affecté le marqueur :

```
function initialize() {
 var latlng = new google.maps.LatLng(48.3929, -4.4798);
 var myOptions = {
  zoom: 11,
  center: latlng,
  mapTypeId: google.maps.MapTypeId.ROADMAP,
  draggable: true
 };

 var map = new google.maps.Map(document.getElementById('map_canvas'),
 myOptions);
 var map2 = new google.maps.Map(document.getElementById(
   'map_canvas2'), myOptions);

 var brest = new google.maps.LatLng(48.3929, -4.4798);
 var icontest = new google.maps.MarkerImage('../images/icon1.png',
   new google.maps.Size(32, 32), new google.maps.Point(0, 0),
   new google.maps.Point(16, 32));
 var shadowicontest = new google.maps.MarkerImage('../images/
  shadow-icon1.png', new google.maps.Size(49, 32),
  new google.maps.Point(0, 0), new google.maps.Point(16, 32));

 var marker = new google.maps.Marker({
  position: brest,
  map: map,
  icon: icontest,
  shadow: shadowicontest
 });

 google.maps.event.addListener(marker, 'mouseover', function(){
  if (marker.getMap() == map){
   marker.setMap(map2);
  } else {
   marker.setMap(map);
  }
 });

 map2.bindTo('center', map, 'center');
 map2.bindTo('zoom', map, 'zoom');
}
```

Avec l'exemple ci-dessus, nous voyons comment utiliser un marqueur sur deux cartes différentes.

- `getPosition` : retourne la position – un objet `LatLng` – du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

google.maps.event.addListener(marker, 'click', function(){
 alert('Le marqueur a pour coordonnées :
       '+marker.getPosition().lat()+' , '+marker.getPosition().lng());
});
```

- `setPosition` : modifie la position du marqueur. Prend en argument un objet de type `LatLng` :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var paris = new google.maps.LatLng(48.8578, 2.3479);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getPosition() == brest){
  marker.setPosition(paris);
 } else {
  marker.setPosition(brest);
 }
});
```

- `getShadow` : renvoie l'objet de type `MarkerImage` correspondant à l'ombre du marqueur. Cette méthode a le même comportement que la méthode `getIcon` ;
- `setShadow` : modifie l'ombre du marqueur. Prend un argument un objet de type `MarkerImage`. Cette méthode a le même comportement que la méthode `setIcon` ;
- `getTitle` : renvoie le titre du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 title: 'Titre qui apparaît lorsque la souris s\'attarde un peu
        sur le marqueur'
 });

google.maps.event.addListener(marker, 'click', function(){
 alert('Voici le titre du marqueur :\n'+marker.getTitle());
});
```

- `setTitle` : modifie le titre du marqueur. Prend en argument une chaîne de caractères :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 title: 'Le titre du marqueur à l\'initialisation'
});

google.maps.event.addListenerOnce(marker, 'click', function(){
 marker.setTitle('Le nouveau titre');
});
```

- `getVisible` : renvoie le booléen correspondant à la propriété `visible` du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 visible: true
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker.getVisible()){
  alert('Le marqueur est visible, sinon nous n\'aurions pu cliquer
        dessus  !');
 }
});
```

- `setVisible` : modifie l'état de la propriété `visible` du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var paris = new google.maps.LatLng(48.8578, 2.3479);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 visible: true,
 title: 'Cliquez-moi pour afficher/masquer le second marqueur'
});

var marker2 = new google.maps.Marker({
 position: paris,
 map: map
});

google.maps.event.addListener(marker, 'click', function(){
 if (marker2.getVisible()){
  marker2.setVisible(false);
 } else {
  marker2.setVisible(true);
 }
});
```

- `getZIndex` : renvoie l'entier correspondant à la propriété `zIndex` du marqueur :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var brest2 = new google.maps.LatLng(48.3927, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 zIndex: 2,
 title: 'Evénement zindex_changed'
});

var marker2 = new google.maps.Marker({
 position: brest2,
 map: map,
 zIndex: 1,
 title: 'Evénement zindex_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 alert('Valeur du zIndex : '+marker.getZIndex());
});

google.maps.event.addListener(marker2, 'click', function(){
 alert('Valeur du zIndex : '+marker2.getZIndex());
});
```

- `setZIndex` : modifie la valeur du `zIndex` du marqueur. Prend en argument un entier :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var brest2 = new google.maps.LatLng(48.3927, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map,
 zIndex: 2
});

var marker2 = new google.maps.Marker({
 position: brest2,
 map: map,
 zIndex: 1,
 title: 'Evénement zindex_changed'
});

google.maps.event.addListener(marker, 'click', function(){
 var temp = marker2.getZIndex();
 marker2.setZIndex(marker.getZIndex());
 marker.setZIndex(temp);
});
google.maps.event.addListener(marker2, 'click', function(){
 var temp = marker2.getZIndex();
 marker2.setZIndex(marker.getZIndex());
 marker.setZIndex(temp);
});
```

3.1.6 Infobulles

Une des fonctionnalités les plus connues de l'API Google Maps est bien la possibilité d'ouvrir une infobulle lorsque l'on clique sur un marqueur (voir figure 3.8). Celle-ci contient souvent des informations relatives au marqueur cliqué.

La méthode pour déclarer une infobulle passe par la classe `InfoWindow` et par la déclaration d'un événement : le plus souvent le clic sur un marqueur pour afficher l'infobulle.

Figure 3.8 — Ouverture d'une infobulle

Les infobulles, comme les autres objets de l'API Google Maps, possèdent des propriétés, réagissent à des événements et ont des méthodes qui permettent de les manipuler.

La déclaration d'une infobulle passe donc par le constructeur de la classe `InfoWindow` :

```
var infobulle = new google.maps.InfoWindow({
 ...
});
```

Une infobulle possède cinq paramètres principaux que l'on peut initialiser lors de la construction de l'objet ou modifier par la suite avec les méthodes *ad hoc*.

- `content` : chaîne de caractères qui représente le contenu de l'infobulle. Il s'agit du texte que l'on veut afficher dans l'infobulle. Cette chaîne de caractères peut être du HTML ou du texte simple. En effet, l'objet `InfoWindow` a la particularité d'interpréter le HTML. Il est donc possible de donner à l'infobulle la présentation souhaitée :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Contenu de l\'infobulle, peut être du HTML ou du texte
  simple.'
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});
```

- `disableAutoPan` : booléen. Lors de l'ouverture de l'infobulle, si cette dernière est trop proche d'une bordure de carte, alors la carte se déplace automatiquement afin que l'affichage soit complet (voir figure 3.9). Ce paramètre permet de contrôler cette fonctionnalité. Par défaut, initialisé à `false` :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Contenu de l\'infobulle, peut être du HTML ou du texte simple.',
 disableAutoPan: true
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});
```

- `maxWidth` : entier. Correspond à la largeur maximale que peut prendre l'infobulle (voir figure 3.10) :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Contenu de l\'infobulle, peut être du HTML ou du texte simple.',
 maxWidth: 35
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});
```

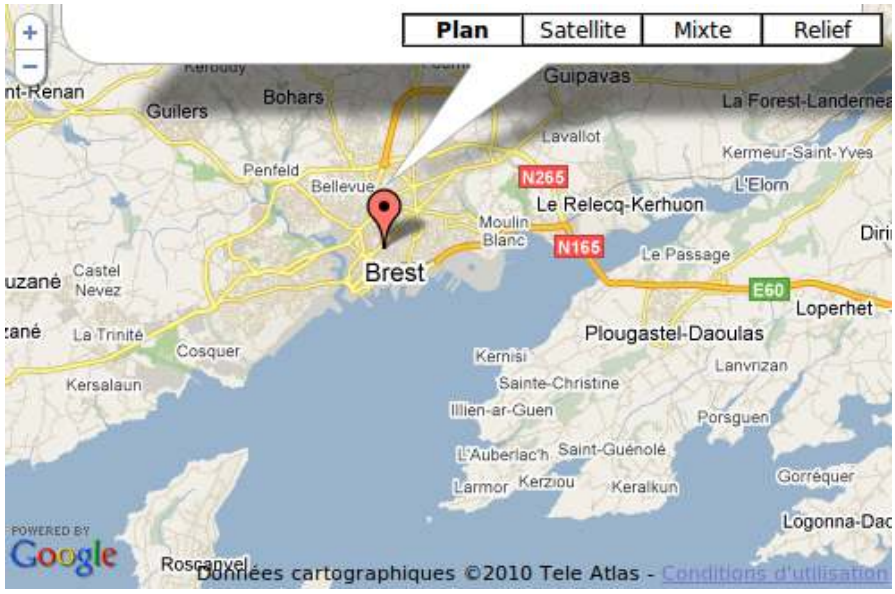

Figure 3.9 — Propriété `disableAutoPan`

Figure 3.10 — Propriété `maxWidth`

- `pixelOffset` : de type `Size`. Décalage de l'affichage de l'infobulle par rapport à l'ouverture par défaut (voir figure 3.11) :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Contenu de l\'infobulle, peut être du HTML ou du texte
   simple.',
 pixelOffset: new google.maps.Size(50, 0)
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});
```

Figure 3.11 — Propriété `pixelOffset`

- `zIndex` : de la même manière que pour la superposition des marqueurs, il est possible de définir un ordre de couche pour les infobulles (voir figure 3.12). Par défaut, l'infobulle dont le marqueur a la latitude la plus faible s'affiche au-dessus. Si la propriété `zIndex` est définie, alors la superposition se fait dans l'ordre décroissant des `zIndex` :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var brest2 = new google.maps.LatLng(48.3927, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var marker2 = new google.maps.Marker({
 position: brest2,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Infobulle 1'
});
var infobulle2 = new google.maps.InfoWindow({
 content: 'Infobulle 2'
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});

google.maps.event.addListener(marker2, 'click', function(){
 infobulle2.open(map, marker2);
});
```

Figure 3.12 — Propriété `zIndex`

Bien que moins nombreux que les événements pour les marqueurs, ceux pour les infobulles existent néanmoins. Ils permettent de capturer notamment la fermeture d'une infobulle, la modification du contenu ou le changement de superposition.

- `closeclick` : événement déclenché lors de la fermeture d'une fenêtre lors d'un clic sur la croix en haut à droite (voir figure 3.13) :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Infobulle 1'
});

google.maps.event.addListener(infobulle, 'closeclick', function(){
 alert('Capture de l\'événement closeclick.');
});
```

Figure 3.13 — Propriété `closeclick`

- `content_changed` : événement déclenché lors de la modification du contenu de l'infobulle.

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Infobulle'
```

```
});

google.maps.event.addListener(map, 'click', function(){
 infobulle.setContent('Autre contenu pour l\'infobulle.');
});

google.maps.event.addListener(infobulle, 'content_changed',
  function(){
   alert('Le contenu de l\'infobulle a été modifié.');
});
```

La méthode `setContent` sera détaillée par la suite.

- `zindex_changed` : événement déclenché lors de la modification de la propriété `zIndex` de l'infobulle.

Il est possible de manipuler les infobulles grâce à leurs méthodes. Depuis le début de cette section, nous en avons notamment utilisé la méthode `open`.

- `close` : méthode qui permet de fermer une infobulle :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Infobulle'
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});

google.maps.event.addListener(map, 'click', function(){
 infobulle.close();
});
```

- `getContent` : cette méthode permet de récupérer le contenu de l'infobulle. Elle retourne une chaîne de caractères :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Ceci est le contenu de l\'infobulle'
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});
```

```
google.maps.event.addListener(map, 'click', function(){
 alert('Contenu de l\'infobulle :\n"'+infobulle.getContent()+'"');
});
```

- `getZIndex` : retourne le paramètre `zIndex` de l'infobulle ;
- `open` : méthode principale de l'objet `InfoWindow`. Cette méthode prend en argument deux paramètres : l'identifiant de la carte et un point d'ancrage (souvent le marqueur auquel est associé l'infobulle) :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Ceci est le contenu de l\'infobulle'
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});
```

- `setContent` : permet de modifier le contenu de l'infobulle. Cette méthode prend en argument une chaîne de caractères qui peut être un contenu HTML :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Ceci est le contenu de l\'infobulle'
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});

google.maps.event.addListener(map, 'click', function(){
 infobulle.setContent('Ceci est un autre contenu pour
   l\'infobulle.');
});
```

- `setOptions` : méthode qui permet de mettre à jour les paramètres de l'infobulle. Prend en paramètre une liste d'options de la classe `InfoWindowOptions` :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});
```

```
var infobulle = new google.maps.InfoWindow({
 content: 'Ceci est le contenu de l\'infobulle'
});

google.maps.event.addListener(marker, 'click', function(){
 infobulle.open(map, marker);
});

google.maps.event.addListener(map, 'click', function(){
 infobulle.setOptions({
  maxWidth: 35,
  content: 'On peut changer le contenu avec cette méthode également.'
 });
});
```

- `setZIndex` : permet de modifier la valeur de la superposition de l'infobulle. Cette méthode prend un entier en argument :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var brest2 = new google.maps.LatLng(48.3927, -4.4798);
var marker = new google.maps.Marker({
 position: brest,
 map: map
});

var marker2 = new google.maps.Marker({
 position: brest2,
 map: map
});

var infobulle = new google.maps.InfoWindow({
 content: 'Infobulle 1',
 zIndex: 1
});

var infobulle2 = new google.maps.InfoWindow({
 content: 'Infobulle 2',
 zIndex: 2
});

infobulle.open(map, marker);
infobulle2.open(map, marker2);

google.maps.event.addListener(map, 'click', function(){
 var temp = infobulle2.getZIndex();
 infobulle2.setZIndex(infobulle.getZIndex());
 infobulle.setZIndex(temp);
});
```

3.2 POLYLIGNES

Dans les systèmes d'information géographique, les entités sont décomposées en trois types : les points, les lignes et les polygones. Nous avons vu dans la première partie de ce chapitre comment créer des points (des marqueurs).

L'API Google Maps permet également d'afficher des lignes nommées « polylignes » grâce à la classe `Polyline` (voir figure 3.14).

Une polyligne peut être considérée comme une succession de points géographiques reliés entre eux. On utilise souvent les polylignes pour représenter un trajet – un itinéraire routier par exemple.

Comme les points, pour initialiser une polyligne, il faut utiliser le constructeur en passant des paramètres :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var polyligne = new google.maps.Polyline({
 path: [brest, paris],
 map: map
});
```

Figure 3.14 — Une polyligne simple

Supplément Web : tous les exemples sur les polylignes se trouvent à cette adresse : http://www.geotribu.net/dunod/gmaps/displaydata/polyline.

3.2.1 Propriétés

Lors de l'initialisation de l'objet, il est possible de choisir entre huit propriétés :

- `clickable` : de type booléen. Cette propriété permet de définir si l'utilisateur peut cliquer sur la polyligne. Dans l'affirmative, au survol de la ligne par la souris, un nouveau pointeur apparaîtra. Par défaut, une polyligne est cliquable :

```
var brest = new google.maps.LatLng(48.3929, -4.4798);

var paris = new google.maps.LatLng(48.8578, 2.3479);
var polyligne = new google.maps.Polyline({
 path: [brest, paris],
 map: map,
 clickable: true
});
```

- `geodesic` : de type booléen. Cette propriété indique si l'on veut que le tracé suive la rotondité de la Terre et non une ligne droite en 2D. Pour afficher le chemin le plus court pour une ligne composée de deux points, il faut spécifier cette propriété (voir figure 3.15) :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var polyligne = new google.maps.Polyline({
 path: [nyc, paris],
 map: map,
 geodesic: true
});
```

Figure 3.15 — Propriété `geodesic`

- `map` : paramètre d'affectation de la polyligne à une carte. Ce paramètre n'est pas requis à l'initialisation, il est possible d'affecter une ligne à un marqueur via la méthode `setMap`.
- `path` : il s'agit du chemin composant la polyligne (voir figure 3.16). Cette séquence ordonnée est un tableau d'objets de type `LatLng`. Dans tous les cas, le tableau d'objets sera transformé en un tableau `MVCArray` défini par l'API. Ainsi l'objet chemin pourra être interrogé – méthode de calcul du nombre d'éléments, etc.

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var chemin = new google.maps.MVCArray();
chemin.push(nyc);
chemin.push(paris);

var polyligne = new google.maps.Polyline({
 path: chemin,
 map: map,
 geodesic: true
});

var cayenne = new google.maps.LatLng(4.9379, -52.3273);

google.maps.event.addListenerOnce(map, 'click', function(){
 chemin.push(cayenne);
});
```

Figure 3.16 — Propriété `path`

La classe `MVCArray` permet de définir le chemin qui compose la polyligne, mais la puissance de cette classe vient du fait de la possibilité de modifier directement le

chemin qui compose une ligne en agissant seulement sur ce tableau. Ainsi il devient très simple, avec les méthodes dont dispose cette classe, d'ajouter un nœud, de modifier un nœud à un index donné ou de supprimer un nœud. Cette classe permet d'envisager la construction d'un outil de numérisation en ligne par exemple.

- `strokeColor` : couleur de la polyligne. Ce paramètre est défini par une chaîne de caractères correspondant au code HTML de la couleur de la ligne, par exemple # BFF980 pour un vert pâle :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var polyligne = new google.maps.Polyline({
 path: [nyc, paris],
 map: map,
 geodesic: true,
 strokeColor: '#BFF980'
});
```

- `strokeOpacity` : transparence de la ligne. Ce paramètre est défini par un nombre entre 0.0 et 1.0 : 0.0 pour transparent, 1.0 pour opaque :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var polyligne = new google.maps.Polyline({
 path: [nyc, paris],
 map: map,
 geodesic: true,
 strokeOpacity: 0.2
});
```

- `strokeWeight` : la largeur en pixels de la polyligne (voir figure 3.17) :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var polyligne = new google.maps.Polyline({
 path: [nyc, paris],
 map: map,
 geodesic: true,
 strokeWeight: 8
});
```

- `zIndex` : identique à la propriété `zIndex` pour les marqueurs. Définit l'ordre de superposition lorsqu'il y a plusieurs polylignes.

3.2.2 Événements

La classe `Polyline` possède tous les événements usuels déjà vus notamment pour les points. Une application Google Maps peut donc intercepter des clics de souris ou des passages de souris au-dessus d'une ligne.

Figure 3.17 — Propriété `strokeWeight`

- `click` : événement déclenché lors d'un clic sur la polyligne :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var chemin = new google.maps.MVCArray();
chemin.push(nyc);
chemin.push(paris);
var polyligne = new google.maps.Polyline({
 path: chemin,
 map: map,
 geodesic: true,
 strokeOpacity: 0.5,
 strokeColor: '#BFF980',
 strokeWeight: 8
});

google.maps.event.addListener(polyligne, 'click', function(){
 alert('La polyligne peut réagir aux clics.');
});
```

- `dblclick` : événement déclenché lors d'un double-clic sur la polyligne :

```
google.maps.event.addListener(polyligne, 'dblclick', function(){
 alert('La polyligne peut réagir aux double-clics.');
});
```

- `rightclick` : événement déclenché lors d'un clic droit sur la polyligne :

```
google.maps.event.addListener(polyligne, 'rightclick', function(){
 alert('La polyligne peut réagir aux double-clics.');
});
```

- `mouseover` : événement déclenché lorsque le curseur de la souris passe au-dessus de la ligne :

```
google.maps.event.addListener(polyligne, 'mouseover', function(){
 alert('Evénement lors du survol de la ligne.');
});
```

- `mouseout` : événement déclenché lorsque le curseur sort de la ligne :

```
google.maps.event.addListener(polyligne, 'mouseout', function(){
 alert('La souris quitte la ligne.');
});
```

3.2.3 Méthodes

La classe `Polyline` ne contient que peu de méthodes. En effet, la classe `MVCArray` qui définit le tableau de points composant la ligne en possède une bonne partie. Les seules méthodes dont on dispose pour cet objet sont des accesseurs d'attributs :

- `getMap` : méthode qui renvoie la carte courante sur laquelle est définie la polyligne ;
- `setMap` : méthode qui permet de modifier ou de préciser la carte sur laquelle est affectée la polyligne ;
- `getPath` : méthode qui renvoie le chemin. Le retour est de type `MVCArray` ;
- `setPath` : méthode qui permet de modifier ou de préciser la liste de points qui compose la polyligne ;
- `setOptions` : méthode permettant de repréciser les paramètres de la polyligne :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var chemin = new google.maps.MVCArray();
chemin.push(nyc);
chemin.push(paris);

var polyligne = new google.maps.Polyline({
 path: chemin,
 map: map
});

google.maps.event.addListener(map, 'click', function(){
 polyligne.setOptions({
  geodesic: true,
  strokeOpacity: 0.5,
```

```
    strokeColor: '#BFF980',
    strokeWeight: 8
   });
});
```

3.2.4 Classe MVCArray

Nous avons vu dans la section précédente que le chemin d'une polyligne est défini par un objet de type `MVCArray`. L'avantage de cet objet vient du fait que nous pouvons le modifier en utilisant des méthodes qui lui sont propres et plus nombreuses que les simples accesseurs d'attributs.

Ainsi il est possible de modifier immédiatement une polyligne en insérant par exemple un nouveau nœud dans le tableau ou en supprimant un nœud à un index donné. Nous verrons ici quelques méthodes aperçues dans le point précédent à propos de la construction des polylignes :

- `getAt` : permet de récupérer l'élément du tableau à l'index précisé. Prend en argument un entier ;
- `setAt` : précise l'élément à modifier à l'index précisé. Prend en argument l'index et l'élément :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);

var chemin = new google.maps.MVCArray();
chemin.push(nyc);
chemin.push(paris);

var polyligne = new google.maps.Polyline({
 path: chemin,
 map: map
});

var moscou = new google.maps.LatLng(55.754043, 37.61971);
google.maps.event.addListener(map, 'click', function(){
 chemin.setAt(0,moscou);
});
```

- `insertAt` : insère un élément à l'index précisé. Prend en argument l'index et l'élément ;
- `removeAt` : supprime l'élément de l'index précisé en paramètre ;
- `pop` : supprime le dernier élément du tableau et retourne cet élément ;
- `push` : insère l'élément passé en paramètre à la fin du tableau et retourne la nouvelle longueur de ce tableau.

3.3 POLYGONES

L'API Google Maps permet d'afficher des points (classe `Marker`) et des lignes (classe `Polyline`). Il reste donc à présenter les polygones, définis avec la classe `Polygon`.

Un polygone est normalement une ligne fermée. Dans ce cas, la seule différence avec une polyligne est la possibilité de définir une couleur et une opacité de remplissage du polygone (voir figure 3.18).

Cependant, un polygone peut comporter plusieurs polygones : c'est pourquoi il est possible de définir plusieurs chemins composant un même polygone.

La construction de cet objet se déroule de la même façon que pour une ligne ou un marqueur :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);
var captown = new google.maps.LatLng(-33.922, 18.413);

var chemin = new google.maps.MVCArray();
chemin.push(nyc);
chemin.push(paris);
chemin.push(captown);
chemin.push(nyc);

var polygone = new google.maps.Polygon({
 map: map,
 paths: chemin
});
```

Figure 3.18 — Polygone simple

Normalement, un polygone doit comporter un point de départ et un point d'arrivée identique. Dans l'exemple ci-dessus, le chemin possède quatre points pour définir un triangle. Il est cependant possible de ne définir que les trois points définissant le polygone, l'API se chargeant de terminer la construction du polygone. On peut donc omettre le retour au point d'origine.

3.3.1 Propriétés des polygones

Les propriétés sont les suivantes :

- `clickable` : de type booléen, possibilité de cliquer sur l'objet ;
- `geodesic` : de type booléen, trace les parties entre chaque point du polygone par le plus court chemin (voir figure 3.19) :

```
var map = new google.maps.Map(document.getElementById('map_canvas'),
  myOptions);

var map2 = new google.maps.Map(document.getElementById(
  'map_canvas2'), myOptions);

var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);
var captown = new google.maps.LatLng(-33.922, 18.413);

var chemin = new google.maps.MVCArray();
chemin.push(nyc);
chemin.push(paris);
chemin.push(captown);

var polygone = new google.maps.Polygon({
 map: map,
 paths: chemin
});

var polygone = new google.maps.Polygon({
 map: map2,
 paths: chemin,
 geodesic: true
});

map2.bindTo('center', map, 'center');
map2.bindTo('zoom', map, 'zoom');
```

- `strokeColor` : couleur du bord du polygone. Ce paramètre est défini par une chaîne de caractères correspondant au code HTML de la couleur de la ligne ;
- `strokeOpacity` : transparence du bord du polygone. Ce paramètre est défini par un nombre entre 0.0 et 1.0 : 0.0 pour transparent, 1.0 pour opaque ;
- `strokeWeight` : largeur en pixels du bord du polygone ;
- `zIndex` : ordre de superposition des polygones ;
- `fillColor` : couleur de l'intérieur du polygone. Ce paramètre est défini par une chaîne de caractères correspondant au code HTML couleur ;

Figure 3.19 — Propriété `geodesic`

- `fillOpacity` : opacité de l'intérieur du polygone. Ce paramètre est défini par un nombre entre 0.0 et 1.0 : 0.0 pour transparent, 1.0 pour opaque.

```
var map = new google.maps.Map(document.getElementById('map_canvas'),
   myOptions);

var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);
var captown = new google.maps.LatLng(-33.922, 18.413);

var chemin = new google.maps.MVCArray();
chemin.push(nyc);
chemin.push(paris);
chemin.push(captown);

var polygone = new google.maps.Polygon({
 map: map,
 paths: chemin,
 fillOpacity: 0.5,
 fillColor: '#FF0000'
});
```

Un objet de la classe `Polygon` peut être composé de plusieurs polygones (voir figure 3.20). Dans ce cas, il suffit d'instancier un tableau de tableaux :

```
var nyc = new google.maps.LatLng(40.7734, -73.9779);
var paris = new google.maps.LatLng(48.8578, 2.3479);
var cayenne = new google.maps.LatLng(4.9379, -52.3273);
var captown = new google.maps.LatLng(-33.922, 18.413);
```

```
var chemin = new google.maps.MVCArray();
chemin.push(new google.maps.LatLng(49.3, 66.1));
chemin.push(new google.maps.LatLng(44.8, 103.3));
chemin.push(new google.maps.LatLng(25.8, 79.8));

var chemin2 = new google.maps.MVCArray();
chemin2.push(nyc);
chemin2.push(paris);
chemin2.push(captown);
chemin2.push(cayenne);

var patio = new google.maps.MVCArray();
patio.push(chemin);
patio.push(chemin2);

var polyligne = new google.maps.Polygon({
 paths: patio,
 map: map,
 geodesic: true
});
```

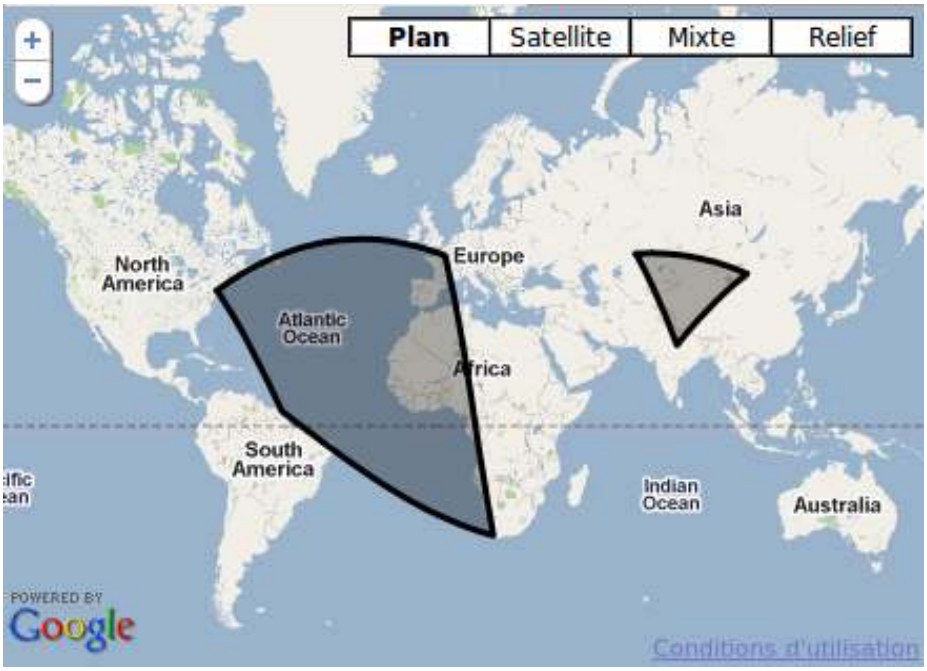

Figure 3.20 — Plusieurs géométries dans un polygone

3.3.2 Méthodes et événements

Les méthodes et les événements de l'objet `Polygon` sont strictement identiques à celles des polylignes à l'exception de la possibilité de gérer plusieurs chemins dans les polygones.

- `click` : événement déclenché lors d'un clic sur le polygone ;
- `dblclick` : événement déclenché lors d'un double-clic sur le polygone ;
- `rightclick` : événement déclenché lors d'un clic droit sur le polygone ;
- `mouseout` : événement déclenché lors de la sortie du pointeur du polygone ;
- `mouseover` : événement déclenché lors de l'entrée du pointeur dans le polygone ;
- `getMap` : retourne la carte sur laquelle est positionné le polygone ;
- `getPath` : retourne le premier chemin composant le polygone ;
- `getPaths` : retourne un tableau de l'ensemble des chemins composant le polygone ;
- `setMap` : indique la carte sur laquelle le polygone sera affiché ;
- `setOptions` : méthode permettant de repréciser les paramètres du polygone ;
- `setPath` : méthode pour préciser le premier chemin composant le polygone ;
- `setPaths` : méthode pour préciser l'ensemble des chemins composant le polygone.

3.4 KML

Dans le monde de la cartographie numérique, il existe de nombreux formats d'échange de données. En effet, les applications – Internet ou non – peuvent utiliser des fichiers externes pour présenter et afficher des données sur une carte.

En vrac nous pouvons citer des formats comme GML (*Geographic Markup Language*), GeoRSS ou encore KML (*Keyhole Markup Language*) pour les plus connus. Ces trois exemples sont des formats dérivés du XML.

Il s'agit donc de fichiers d'échange qui sont semblables car tous vectoriels et compréhensibles par le commun des mortels à l'aide d'un éditeur de texte.

3.4.1 Introduction au format

Le format KML (du nom de la société à l'origine du logiciel Google Earth, Keyhole, rachetée par Google en 2004) est un dérivé de XML et permet de gérer des données géographiques, notamment les points, les lignes et les polygones.

Initialement utilisé par Google Earth, ce format d'échange s'est vite démocratisé jusqu'à devenir un standard dans la communauté géomatique. KML est maintenant utilisé par tous les logiciels de système d'information géographique pour afficher et diffuser des données géolocalisées.

Très vite, Google a donc rendu possible l'affichage de données KML à l'intérieur de son API Google Maps. En effet, il n'est pas toujours facile d'aborder les notions d'objets `Marker`, `Polyline` et `Polygon` alors qu'un simple fichier de description de données peut faire la même chose et que l'API permet de les afficher simplement.

Voici un exemple d'un simple fichier KML contenant un point :

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Placemark>
 <name>Test</name>
 <Point>
  <coordinates>-4.508852043550391,48.38157384398676,0</coordinates>
 </Point>
</Placemark>
</kml>
```

3.4.2 Construction

L'affichage du contenu d'un fichier KML dans une cartographie de l'API (voir figure 3.21) passe par la classe `KmlLayer`, qui permet d'ajouter à la carte des données provenant non seulement des fichiers KML mais aussi des fichiers GeoRSS et des fichiers KMZ (il s'agit d'un fichier KML zippé).

```
function initialize() {
 var latlng = new google.maps.LatLng(48.3929, -4.4798);
 var myOptions = {
  zoom: 10,
  center: latlng,
  mapTypeId: google.maps.MapTypeId.ROADMAP
 };

 var map = new google.maps.Map(document.getElementById('map_canvas'),
   myOptions);

 var kml = new google.maps.KmlLayer('url_du_fichier_kml.kml', {
  map: map,
  preserveViewport: true
 });
}
```

3.4.3 Propriétés

Un objet `KmlLayer` prend en argument l'URL du fichier KML et une liste d'options.

L'URL doit être accessible depuis Internet : cela signifie qu'il n'est pas possible de tester l'affichage de fichier KML sur un environnement en local, il est obligatoire de mettre une copie du fichier KML quelque part sur la toile.

La liste des options que peut prendre un objet de cette classe est la suivante :

- `map` : la carte à laquelle est affecté le KML ;
- `preserveViewport` : de type booléen. Par défaut, l'API modifie l'emprise de la carte pour optimiser au mieux l'affichage des objets contenus dans le fichier KML. Cette option permet de contrôler cette fonctionnalité :

Figure 3.21 — Affichage d'un fichier KML

```
var kml = new google.maps.KmlLayer('url_du_fichier_kml.kml', {
 map: map,
 preserveViewport: true
});
```

- `suppressInfoWindows` : de type booléen. Par défaut, un clic sur un objet du fichier KML affiché sur la carte ouvre une infobulle. Ce paramètre permet d'autoriser ou non cette possibilité :

```
var kml = new google.maps.KmlLayer('url_du_fichier_kml.kml', {
 map: map,
 suppressInfoWindows: true
});
```

3.4.4 Méthodes et événements

Les objets `KmlLayer` peuvent réagir à deux événements et possèdent cinq méthodes accesseurs d'attributs :

- `click` : événement déclenché lorsque la couche KML est cliquée :

```
var kml = new google.maps.KmlLayer('url_du_fichier_kml.kml', {
 map: map,
 preserveViewport: true
});

google.maps.event.addListener(kml, 'click', function(){
```

```
 alert('Normalement, un clic sur la couche KML ouvre une infobulle.
       Il est possible de modifier ce comportement.');
});
```

- `getMap` : méthode qui renvoie la carte sur laquelle est affiché le KML ;
- `setMap` : indique la carte sur laquelle la couche KML sera affichée ;
- `getUrl` : renvoie la chaîne de caractères correspondant à l'URL précisée dans les paramètres ;
- `getMetadata` : renvoie un objet de type `KmlLayerMetadata`. Le retour correspond aux métadonnées contenues dans le fichier KML. Il sera alors possible d'en extraire l'auteur, la date ou d'autres informations.

3.5 AJOUTER DES POINTS PROVENANT D'UNE BASE DE DONNÉES

Souvent lorsque l'on développe des applications avec l'API Google Maps, survient la nécessité d'afficher un grand nombre de points d'intérêt. Et souvent ces derniers sont stockés en base de données : ils sont géolocalisés, ils possèdent une description et un titre.

Comment est-il possible de lier la carte avec ces données en base ?

Pour que la carte reste fluide et que l'internaute n'est pas à recharger la page, il faut tirer parti d'AJAX et donc de la classe `XMLHttpRequest`.

Le processus est le suivant :

1. Faire une requête JavaScript qui « demande » au serveur les objets contenus en base de données.
2. Éditer un script côté serveur qui formate les données dans un fichier XML par exemple et qui retourne ce fichier au client.
3. Parser le XML en JavaScript et afficher les points sur la carte.

3.5.1 Préparation de la base de données

Dans le cadre de cet ouvrage, nous utiliserons le système de gestion de base de données MySQL et le langage de script PHP. Il s'agit d'un couple très utilisé sur le Web (cf. section 1.4).

Imaginons une table contenant des points d'intérêts divers et variés. Ces points sont désignés par un titre, une description et des coordonnées géographiques : latitude et longitude.

```
CREATE TABLE 'poi' (
'id' SERIAL NOT NULL ,
'titre' VARCHAR( 100 ) NOT NULL ,
'latitude' DOUBLE NOT NULL ,
```

```
'longitude' DOUBLE NOT NULL ,
'description' TEXT NOT NULL
);
```

Ajoutons quelques données dans cette table :

```
INSERT INTO 'poi' (
'id' ,
'titre' ,
'latitude' ,
'longitude' ,
'description'
)
VALUES (
NULL , 'Mont Saint-Michel', '48.63620', '-1.51137', 'Deuxième site touristique
de France'
);

INSERT INTO 'poi' (
'id' ,
'titre' ,
'latitude' ,
'longitude' ,
'description'
)
VALUES (
NULL , 'Tour Eiffel', '48.85820', '2.29457', 'Elle est restée pendant près de
40 ans la plus haute construction humaine.'
);

INSERT INTO 'poi' (
'id' ,
'titre' ,
'latitude' ,
'longitude' ,
'description'
)
VALUES (
NULL , 'Place Stanislas', '48.693448', '6.18340', 'La plus belle place du
monde'
);
INSERT INTO 'poi' (
'id' ,
'titre' ,
'latitude' ,
'longitude' ,
'description'
)
VALUES (
NULL , 'Le Capitole', '43.60449', '1.44349', 'L''autre plus belle place du
monde'
);
INSERT INTO 'poi' (
'id' ,
'titre' ,
'latitude' ,
'longitude' ,
```

```
'description'
)
VALUES (
NULL , 'Chateau des ducs de Bretagne', '47.216225', '-1.54948', 'Début de
construction au XIIIe siècle'
);
```

3.5.2 Script PHP

Préparons dès à présent le script PHP qui édite le fichier XML. Ce script doit se connecter à la base de données, l'interroger et construire le fichier XML qui sera envoyé à l'API Google Maps :

```
<?php

  $user = "****";
  $password = "****";
  $host = "localhost";
  $bdd = "dunod";

  mysql_connect($host,$user,$password);
  mysql_select_db($bdd) or die("erreur de connexion à la base
   de données");

  $sql = "SELECT * FROM poi";
  $res = mysql_query($sql) or die(mysql_error());

  $dom = new DomDocument('1.0', 'utf-8');
  $node = $dom->createElement("markers");
  $parnode = $dom->appendChild($node);

  while ($result = mysql_fetch_array($res)){
    $node = $dom->createElement("marker");
    $newnode = $parnode->appendChild($node);
    $newnode->setAttribute("titre", $result['titre']);
    $newnode->setAttribute("lat", $result['latitude']);
    $newnode->setAttribute("lng", $result['longitude']);
    $newnode->setAttribute("description",
                           utf8_encode($result['description']));
  }

  $xmlfile = $dom->saveXML();
  echo $xmlfile;
?>
```

Ce script utilise les fonctions DOM XML de PHP pour construire le fichier XML, notamment les fonctions `createElement`, `appendChild` ou `setAttribute`.

3.5.3 Côté client

L'API Google Maps ne propose plus la classe `GDownloadUrl` dans sa version 3. Celle-ci permettait de charger directement en AJAX une ressource externe avec l'API.

Dans la nouvelle version de l'API, il faut passer par un code JavaScript qui permet d'ouvrir une requête AJAX :

```
function createXmlHttpRequest() {
 try {
  if (typeof ActiveXObject != 'undefined') {
   return new ActiveXObject('Microsoft.XMLHTTP');
  } else if (window["XMLHttpRequest"]) {
   return new XMLHttpRequest();
  }
 } catch (e) {
  changeStatus(e);
 }
 return null;
};
```

Il faut définir la fonction `downloadUrl` qui téléchargera la ressource externe :

```
function downloadUrl(url, callback) {
 var status = -1;
 var request = createXmlHttpRequest();
 if (!request) {
  return false;
 }

 request.onreadystatechange = function() {
  if (request.readyState == 4) {
   try {
    status = request.status;
   } catch (e) {
   }
   if (status == 200) {
    callback(request.responseText, request.status);
    request.onreadystatechange = function() {};
   }
  }
 }

 request.open('GET', url, true);
 try {
  request.send(null);
 } catch (e) {
  changeStatus(e);
 }
};
```

Et définir un parser XML :

```
function xmlParse(str) {
 if (typeof ActiveXObject != 'undefined' && typeof GetObject != 'undefined') {
  var doc = new ActiveXObject('Microsoft.XMLDOM');
  doc.loadXML(str);
  return doc;
 }

 if (typeof DOMParser != 'undefined') {
  return (new DOMParser()).parseFromString(str, 'text/xml');
```

```
 }

 return createElement('div', null);
}
```

Définissons maintenant une fonction de création de marqueurs prenant en arguments les latitude et longitude du point ainsi que le titre et la description :

```
function createMarker(lat, lng, titre, description){
 ...
}
```

Et construisons le marqueur et son infobulle comme vu précédemment dans le chapitre :

```
function createMarker(lat, lng, titre, description){
 var latlng = new google.maps.LatLng(lat, lng);
 var marker = new google.maps.Marker({
  position: latlng,
  map: map,
  title: titre
 });

 var infobulle = new google.maps.InfoWindow({
  content: description
 });
 google.maps.event.addListener(marker, 'click', function(){
  infobulle.open(map, marker);
 });
}
```

Il reste donc la fonction principale de construction de la carte. Le début de cette fonction nous est familier :

```
function initialize() {
 var latlng = new google.maps.LatLng(46.7, 2.5);
 var myOptions = {
  zoom: 6,
  center: latlng,
  mapTypeId: google.maps.MapTypeId.ROADMAP
 };

 map = new google.maps.Map(document.getElementById('map_canvas'),
                           myOptions);
...
}
```

Puis insérons l'appel AJAX vers le script PHP et construisons les marqueurs :

```
downloadUrl("getPoi.php", function(data) {
 var xml = xmlParse(data);
 var markers = xml.documentElement.getElementsByTagName("marker");
 for (var i = 0; i < markers.length; i++) {
```

```
  createMarker(parseFloat(markers[i].getAttribute("lat")),
parseFloat(markers[i].getAttribute("lng")), markers[i].getAttribute('titre'),
markers[i].getAttribute('description'));
 }
});
```

Nous avons donc maintenant une chaîne de diffusion de données complète. En effet, nous administrons une base de données qui contient des points d'intérêts ; nous avons un script serveur qui permet d'extraire des données de la base et de les envoyer au client JavaScript ; et enfin, une application Google Maps qui récupère les données envoyées et les interprète pour les afficher sur une carte (cf. figure 3.22).

Figure 3.22 — Connexion avec une base de données

Supplément Web : l'exemple ci-dessus est accessible à l'adresse suivante : http://www.geotribu.net/dunod/gmaps/displaydata/sgbd.

En résumé

Ce long chapitre a essayé de décrire les différents objets qu'il est possible d'afficher sur une carte Google Maps. En effet, le principal intérêt d'une cartographie dynamique est de diffuser des données géolocalisées.

Nous avons vu que l'API pouvait afficher les principales primitives géométriques, c'est-à-dire des points, des lignes et des polygones ainsi que des fichiers de description de données KML ou d'afficher des points stockés en base de données.

L'affichage de données stockées en base permet d'entrevoir des applications riches offrant à l'internaute d'interagir avec les points d'intérêt : les événements sur les objets permettent d'imaginer toutes sortes d'applications web.

4

Les services

Objectifs

Les services sont la deuxième principale fonctionnalité de l'API Google Maps après l'affichage de primitives (points, lignes, polygones).

Ceux-ci utilisent toute l'intelligence des serveurs de Google pour proposer des outils de géocodage, d'itinéraires, de calcul d'altitude ainsi qu'un outil d'exploration de photos 3D, StreetView.

Nous verrons dans ce chapitre comment utiliser ces différents services et comment les piloter à partir de l'API Google Maps.

4.1 GÉOCODAGE

Le géocodage est la possibilité d'obtenir les coordonnées géographiques d'une adresse donnée. Nous avons vu que le positionnement de marqueur sur une carte passait obligatoirement par un objet de type `LatLng` qui renseignait les latitude et longitude.

Mais comment faire pour afficher une adresse dont nous ne connaissons pas ces coordonnées géographiques ? C'est là qu'intervient le géocodage. Vous l'avez sûrement déjà utilisé des dizaines de fois sans avoir donné un nom à cette fonctionnalité : lorsque sur la page d'accueil de Google Maps vous saisissez une adresse et cliquez sur « Recherche », Google va chercher dans sa base de données les coordonnées géographiques correspondant à cette requête.

L'API Google Maps fournit exactement le même service : nous envoyons une adresse et Google nous renvoie les coordonnées géographiques si ces dernières lui sont connues.

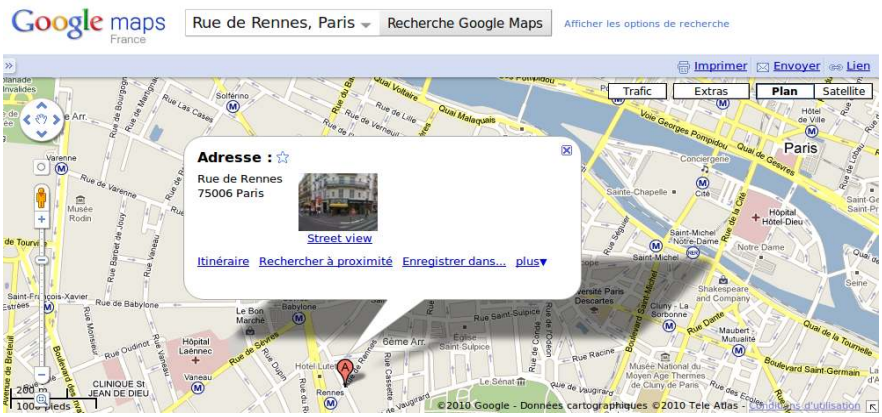

Figure 4.1 — Géocodage à l'adresse sur le site Google Maps

4.1.1 Construction

Une requête de géocodage passe par un objet de la classe `Geocoder`. Celle-ci ne possède qu'une seule méthode, *geocode*, qui permet d'envoyer la requête :

- `geocode` : prend en argument la question – un objet de type `GeocoderRequest` –, une fonction de retour qui nous permettra de faire l'appel AJAX vers le service Google et un code de retour.

Un objet de type `GeocoderRequest` prend quant à lui les arguments suivants :

- `address` : obligatoire si l'argument `latLng` (voir ci-après) n'est pas renseigné. Il s'agit de l'adresse à géocoder. C'est une chaîne de caractères ;
- `latLng` : obligatoire si `address` ci-dessus est non renseigné. Il s'agit d'un point dont on veut trouver l'adresse. On parle alors de « géocodage inverse ». C'est un objet de type `LatLng` ;
- `language` : la langue dans laquelle on souhaite obtenir le résultat.

Le code de retour peut prendre les valeurs suivantes :

- `ERROR` : indique un problème inattendu ;
- `INVALID_REQUEST` : indique que la question posée est erronée ;
- `OK` : la réponse est valide et donc exploitable ;
- `OVER_QUERY_LIMIT` : indique que l'application a atteint son quota de questions qui est actuellement de 2 500 requêtes par jour ;
- `REQUEST_DENIED` : indique que la requête a été rejetée ;
- `UNKNOW_ERROR` : indique que la requête a échoué. Cependant il est possible qu'elle fonctionne lors d'un essai ultérieur. Indique donc un problème serveur ;
- `ZERO_RESULTS` : indique que la question ne renvoie aucun résultat.

4.1.2 Spécifications de la réponse

Lorsque l'on envoie une question de géocodage via l'API, le service répond par un objet de type `GeocoderResult`.

On peut extraire de cette réponse plusieurs informations :

- les coordonnées géographiques : propriété `geometry.location` de type `LatLng` ;
- la précision du retour : propriété `geometry.location_type` de type chaîne de caractères ;
- les coordonnées des coins de la carte recommandées pour visualiser le résultat : propriété `geometry.viewport` de type `LatLngBounds`.

4.1.3 Exemple de géocodage

L'exemple suivant montre comment utiliser très simplement le service de géocodage.

> **Supplément Web** : cet exemple se trouve à l'adresse http://www.geotribu.net/dunod/gmaps/service/geocode.

Tout d'abord, créons en plus du bloc HTML contenant la carte, un champ `INPUT` de type `TEXT` et un bouton de validation :

```
<html>
..
<body onload='initialize()'>
 <div id='map_canvas' style='width: 640px; height: 480px'></div>
 <div>
  <input id='champ_adresse' type='text' value='Rue de Rennes, Paris'>
  <input type='submit' value='Géocode' onclick='searchAddress()'>
 </div>
</body>
</html>
```

Notre fonction `initialize` est similaire à celle de tous les exemples précédents :

```
var map;
var geocoder = new google.maps.Geocoder();
function initialize(){
  var latlng = new google.maps.LatLng(46.7, 2.5);
  var myOptions = {
   zoom: 10,
   center: latlng,
   mapTypeId: google.maps.MapTypeId.ROADMAP
  };

  map = new google.maps.Map(document.getElementById('map_canvas'),
    myOptions);
}
```

Et enfin nous ajoutons la fonction `searchAddress` qui est appelée dès que le bouton Géocode est pressé :

```
function searchAddress(){
 var adresse = document.getElementById("champ_adresse").value;
 geocoder.geocode({
  address: adresse
 }, function(results, status){
  if (status == google.maps.GeocoderStatus.OK){
   map.setCenter(results[0].geometry.location);
   var marker = new google.maps.Marker({
    position: results[0].geometry.location,
    map: map
   });
  } else {
   alert('Le géocodage a échoué.');
  }
 });
}
```

Figure 4.2 — Géocodage simple en utilisant l'API

4.2 ITINÉRAIRE

Le deuxième service de l'API Google Maps est le calcul d'itinéraire. Très abouti, il est en concurrence avec des outils similaires chez Mappy, ViaMichelin ou encore Bing Maps.

L'API dans sa version 3 continue de mettre à disposition des développeurs cette fonctionnalité très populaire. En effet, qui n'a jamais calculé d'itinéraire sur le site de Google Maps ?

La méthode de fonctionnement de ce service est très similaire à celle du géocodage. En effet, il faut poser une question bien formatée au service de Google qui répond par un objet directement exploitable par l'API.

4.2.1 Construction du service

Le service d'itinéraire de l'API passe par les classes `DirectionsService` et `DirectionsRendered`. La première étant le constructeur du service et la seconde la classe qui exploite le résultat de la requête.

Une demande d'itinéraire doit comporter les paramètres suivants :

- `origin` : point de départ de l'itinéraire. Peut être un objet `LatLng` ou une adresse que le service se chargera de géocoder ;
- `destination` : point d'arrivée de l'itinéraire. Peut être un objet `LatLng` ou une adresse que le service se chargera de géocoder ;
- `unitSystem` : le type de système de mesure. Deux possibilités : métrique ou système anglophone ;
- `travelMode` : le type de calcul d'itinéraire désiré. Il est possible de calculer des itinéraires en voiture ou à pied ;
- `avoidHighways` : autorise le service à calculer un itinéraire empruntant les autoroutes ;
- `avoidTolls` : autorise le service à calculer un itinéraire empruntant des routes à péage ;
- `waypoints` : tableau de points intermédiaires. Peuvent être de type `LatLng` ou une chaîne de caractères que le service se chargera de géocoder ;
- `optimizeWaypoints` : autorise le service à modifier le tableau de points intermédiaires si le coût de l'itinéraire est moins élevé.

```
var paris = new google.maps.LatLng(48.86, 2.341);
var toulouse = new google.maps.LatLng(43.6, 1.44);

var directionsService = new google.maps.DirectionsService();
directionsService.route({
 origin: paris,
 destination: toulouse,
 unitSystem: google.maps.DirectionsUnitSystem.METRIC,
 avoidTolls: true,
 avoidHighways: true,
 travelMode: google.maps.DirectionsTravelMode.DRIVING
}, function(result, status){
 ...
)};
```

Comme le service de géocodage, la construction d'une requête passe par une fonction de retour qui exploitera par AJAX la réponse du service. Le résultat de la requête est un tableau de routes envoyé au format JSON. Il est possible d'en tirer des informations telles que la distance entre deux intersections, le fait de tourner à gauche ou à droite, etc.

Nous verrons ici simplement comment afficher la totalité de l'itinéraire sur la carte et la totalité des informations textuelles dans un bloc, grâce à la classe `DirectionsRendered` qui possède les méthodes suivantes :

- `setDirections` : prend en argument le résultat d'une requête au service d'itinéraire et affiche sur la carte l'itinéraire en question ainsi que l'ensemble des informations textuelles dans le bloc HTML passé en paramètre avec la méthode `setPanel` ;
- `setMap` : indique la carte sur laquelle sera affiché l'itinéraire ;
- `setPanel` : indique le bloc HTML où les informations de l'itinéraire seront affichées.

4.2.2 Exemples d'application

L'exemple présenté ci-après propose de calculer un itinéraire et d'afficher les informations du parcours. Cet exemple est identique à celui sur le géocodage à la différence de l'utilisation des classes appropriées pour le calcul d'itinéraire.

La partie HTML est similaire à l'exemple dédié au géocodage :

```
<body onload="initialize()">
 <div>
  <input id='depart' type='text' value='Rue de Rennes, Paris'>
  <input id='arrivee' type='text' value='Rue de Metz, Toulouse'>
  <input type='submit' value='Calculer' onclick='showItineraire()'>
 </div>
 <div id="map_canvas" style="width: 500px; height: 400px;"></div>
 <div id="displayinfo_canvas" style="width: 250px;
    height: 400px;"></div>
</body>
</html>
```

Dans le code JavaScript, nous déclarons la fonction `showItineraire` et modifions quelque peu la fonction `initialize` :

```
 var map;
 var directionsDisplay;
 var directionsService = new google.maps.DirectionsService();

 function initialize() {
  var latlng = new google.maps.LatLng(46.7, 2.5);
  var myOptions = {
   zoom: 10,
   center: latlng,
   mapTypeId: google.maps.MapTypeId.ROADMAP
  };
```

```
  map = new google.maps.Map(document.getElementById('map_canvas'),
     myOptions);
  directionsDisplay = new google.maps.DirectionsRenderer();
  directionsDisplay.setMap(map);
  directionsDisplay.setPanel(document.getElementById(
     'displayinfo_canvas'));
}

function showItineraire(){
 directionsService.route({
  origin: document.getElementById('depart').value,
  destination: document.getElementById('arrivee').value,
  unitSystem: google.maps.DirectionsUnitSystem.METRIC,
  travelMode: google.maps.DirectionsTravelMode.DRIVING
 }, function(result, status){
  if (status == google.maps.DirectionsStatus.OK){
   directionsDisplay.setDirections(result);
  } else {
   alert('Le calcul d\'itinéraire a échoué.');
  }
 });
}
```

Pour utiliser le service d'itinéraire de Google Maps, il suffit de construire un objet de la classe `DirectionsService` auquel on passe une demande d'itinéraire entre une ville de départ et une ville d'arrivée.

La déclaration d'un objet de la classe `DirectionsRendered` nous permet d'afficher le résultat sur la carte (voir figure 4.3) ainsi que l'ensemble des indications textuelles avec la méthode `setDirections`.

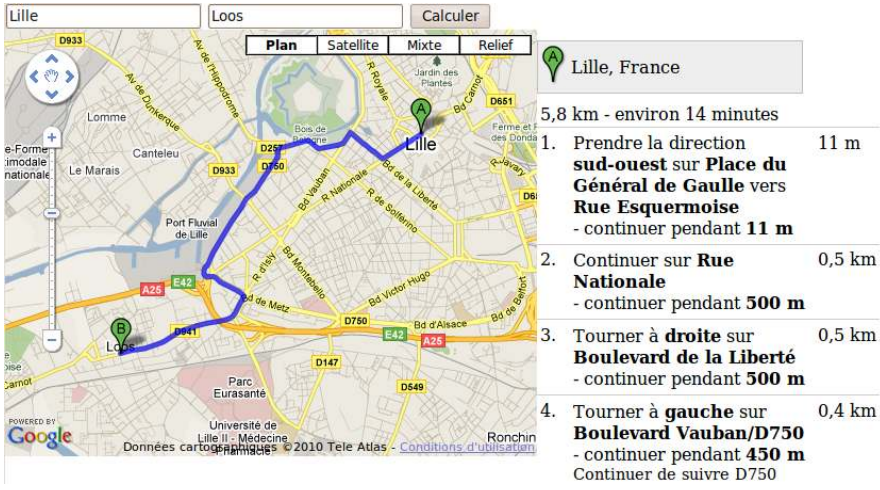

Figure 4.3 — Calcul d'itinéraire avec l'API

Supplément Web : cet exemple se trouve à l'adresse suivante : http://www.geotribu.net/dunod/gmaps/service/route.

4.3 ALTITUDE

Il est parfois intéressant de connaître l'altitude d'un point à la surface de la Terre. L'API Google Maps propose ce nouveau service depuis mars 2010.

L'obtention de l'altitude de tous les points de la surface de la Terre peut être nécessaire pour les applications où cette information est importante : les randonnées par exemple.

La seule restriction de l'API sur ce service est une limitation de 2 500 requêtes par jour et que l'information tirée de ce service soit exploitée sur une carte Google Maps. Ici pas de problème, nous allons utiliser la classe `ElevationService` de l'API.

Dans ce simple exemple – qui donne l'altitude d'un point cliqué sur la carte –, nous allons utiliser la classe `ElevationService` ainsi que la classe `InfoWindow` pour présenter les résultats sur la carte.

Tout d'abord, commençons par la fonction `initialize` et les variables :

```
var map;
var elevation;
var infobulle = new google.maps.InfoWindow();

function initialize() {
 var latlng = new google.maps.LatLng(46.7, 2.5);
 var myOptions = {
  zoom: 5,
  center: latlng,
  mapTypeId: google.maps.MapTypeId.ROADMAP
 };

 map = new google.maps.Map(document.getElementById('map_canvas'),
    myOptions);
 elevation = new google.maps.ElevationService();
 google.maps.event.addListener(map, 'click', getElevation);
}
```

La seule nouveauté provient de l'objet de la classe `ElevationService`.

Nous instancions un événement qui se déclenche lors d'un clic sur la carte et nous appelons la fonction `getElevation` qui pose la question de l'altitude au service et affiche une infobulle avec l'altitude indiquée :

```
function getElevation(event){
 var tab = [];
 tab.push(event.latLng);
 elevation.getElevationForLocations({locations: tab},
     function(results, status){
  if (status == google.maps.ElevationStatus.OK){
   if (results[0]){
    infobulle.setContent('Altitude : '+results[0].elevation+' m');
    infobulle.setPosition(event.latLng);
    infobulle.open(map);
   }
```

```
    } else {
     alert('Le service n\'a pas trouvé l\'altitude de ce point.');
    }
   });
}
```

Expliquons le fonctionnement de la fonction `getElevation` :

- Elle possède le paramètre qui correspond à l'événement clic de la souris qui possède la propriété `latLng`.
- Nous ajoutons donc ce point dans un tableau avec la fonction `push`.
- Nous appelons la méthode `getElevationForLocations` qui est l'une des deux méthodes de l'objet `ElevationService`, qui permet de poser une question au service. Elle prend en paramètre un tableau de positions de type `LatLng` et une fonction de retour qui exploitera ainsi le résultat.
- Le résultat est de type `ElevationResult` qui possède deux attributs : `elevation` et `position`. Nous utilisons ici `elevation`.
- Si le statut de la requête est positif, alors nous affichons à l'endroit du clic une infobulle dans laquelle nous écrivons l'altitude (voir figure 4.4).

Figure 4.4 — Affichage de l'altitude dans une infobulle

4.4 STREETVIEW

Google StreetView est un service qui propose des photographies panoramiques à 360° d'un grand nombre de routes de part le monde.

En septembre 2010, la couverture de StreetView en France était celle visualisée à la figure 4.5.

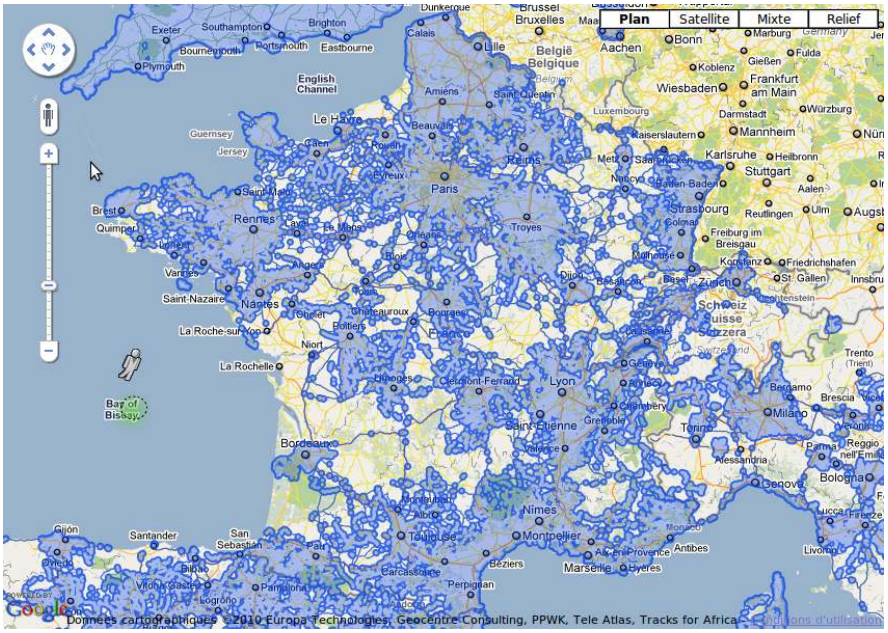

Figure 4.5 — Couverture StreetView en 2010

Campagne commencée en juillet 2008 à l'occasion du Tour de France, les célèbres Google Cars parcourent depuis le pays en quête d'une couverture toujours plus complète.

4.4.1 Implantation dans Google Maps

Ajouter le service StreetView dans une carte Google Maps est extrêmement facile avec l'API ; une simple ligne suffit :

```
var myOptions = {
 zoom: 11,
 center: latlng,
 mapTypeId: google.maps.MapTypeId.ROADMAP,
 streetViewControl: true
};
```

Ce contrôle se caractérise par l'ajout d'un petit bonhomme orange au-dessus du contrôle de navigation : Pegman (voir figure 4.6).

Figure 4.6 — Utilisation (1) du service StreetView avec l'API

Il est alors possible de cliquer sur ce bonhomme et de le déplacer sur la carte, sur une route couverte par le service – surbrillance en bleu. La carte s'efface alors pour laisser place à la navigation StreetView dans des photographies panoramiques à 360° (voir figure 4.7).

Figure 4.7 — Utilisation (2) du service Streetview avec l'API

4.4.2 Utilisation du panorama en dehors de la cartographie

L'utilisation précédente du service StreetView ne permet cependant pas d'avoir le panorama en correspondance avec la carte.

Pour cela, il est nécessaire de déclarer un objet `StreetViewPanorama` qui permet d'externaliser le panorama en-dehors de la carte.

Les options du panorama sont les suivantes :

- `addressControl` : de type booléen. Contrôle l'affichage de l'adresse dans le panorama (voir figure 4.8) :

```
function initialize() {
 var saintleu = new google.maps.LatLng(49.89655, 2.30408);
 var panoramaOptions = {
  position: saintleu,
  pov: {
   heading: 165,
   pitch: 0,
   zoom: 1
  }
 };

 var panoramaOptions2 = {
  addressControl: false,
  position: saintleu,
  pov: {
   heading: 165,
   pitch: 0,
   zoom: 1
  }
 };

 var myPano = new google.maps.StreetViewPanorama(
    document.getElementById("pano_canvas"), panoramaOptions);

 myPano.setVisible(true);

 var myPano2 = new google.maps.StreetViewPanorama(
    document.getElementById("pano_canvas2"), panoramaOptions2);
 myPano2.setVisible(true);
}
```

- `enableCloseButton` : de type booléen. Affiche ou non la croix de fermeture de la fenêtre.
- `linksControl` : de type booléen. Contrôle l'affichage des liens de contrôle. Si non, il n'est plus possible de se déplacer vers les panoramas suivant et précédent (voir figure 4.9) :

```
 var panoramaOptions2 = {
  addressControl: false,
  linksControl: false,
  position: saintleu,
```

Figure 4.8 — Propriété `addressControl`

Figure 4.9 — Propriété `linksControl`

```
  pov: {
   heading: 165,
   pitch: 0,
   zoom: 1
  }
 };
```

- `navigationControls` : de type booléen. État du contrôle de navigation ;
- `position` : de type `LatLng`. Il s'agit de la position géographique du panorama ;
- `pov` : de type `StreetViewPov`. Il s'agit du contrôle de l'inclinaison, du zoom et la direction de l'angle de la caméra ;
- `visible` : de type booléen. Contrôle la visibilité du panorama.

Un objet de type `StreetViewPanorama` peut réagir à tous les changements d'état des propriétés précédentes : `links_changed`, `pano_changed`, `position_changed`, `pov_changed`, `visible_changed` ; ainsi que les événements `closeclick` et `resize`.

Cet objet panorama possède donc des méthodes qui peuvent piloter toutes ces propriétés, il s'agit des accesseurs d'attributs :

- `getPano` : retourne l'identifiant du panorama courant ;
- `setPano` : édite l'identifiant du panorama ;
- `getPosition` : retourne la position `LatLng` du panorama courant ;
- `setPosition` : modifie la position du panorama. Reçoit un objet de type `LatLng` en paramètre ;
- `getPov` : retourne le point de vue de la caméra sous la forme d'une chaîne de caractère (inclinaison, angle et zoom de la caméra) ;
- `setPov` : modifie le point de vue de la caméra. Prend un objet de type `StreetViewPov` en paramètre ;
- `getVisible` : retourne la visibilité du panorama ;
- `setVisible` : modifie l'état de visibilité du panorama avec un paramètre vrai ou faux.

En résumé

Ce chapitre a introduit les services que propose l'API Google Maps. Nous avons vu que ces derniers étaient simples à mettre en place dans une application Google Maps à condition de suivre scrupuleusement les objets et leur définition.

Le géocodage et le calcul d'itinéraire peuvent être utilisés dans des applications de suivi de flotte et d'optimisation de parcours – dans le cas d'une utilisation privée, il faut alors passer par l'API Google Maps Premier.

Le calcul de l'altitude est destiné plus particulièrement aux applications telles que la randonnée ou le cyclotourisme.

Le service StreetView permet, quant à lui, une nouvelle forme d'exploration du territoire en immersion dans un panorama en trois dimensions.

5

Exemple d'application

Objectifs

Ce chapitre va nous permettre d'utiliser les différents services vus dans les parties précédentes en développant une application complète. Elle aura pour objectif d'afficher en temps réel l'état des bornes de vélo de la ville de Rennes.

Pour mener à bien ce projet, nous allons avoir besoin des données relatives à l'état des bornes de vélo dans la ville de Rennes. Celles-ci sont utilisables et téléchargeables via un service Web mis en place par la commune de Rennes. Cette possibilité s'inscrit dans un mouvement mondial d'ouverture des données : l'*open data* qui est un peu aux données ce qu'est l'*open source* aux logiciels.

Rennes ouvre la voie en France avec la mise à disposition des données en temps réel de ses bornes de vélo – nombre de vélos disponibles et nombre d'emplacements libres – *via* une API. Cette dernière fonctionne très simplement : il suffit d'ouvrir un compte sur le site http://data.keolis-rennes.com et de poser une simple requête.

5.1 PLATES-FORMES CIBLES

Une application Web récente se doit de fonctionner sur les ordinateurs de bureau mais aussi maintenant sur les smartphones : il est prévu qu'en 2015 le volume des données téléchargées via les smartphones dépasse celui des ordinateurs de bureau.

Il est donc indispensable de penser notre application sur les deux types de plate-forme. Pour cela, reprenons la fonction `detectBrowser` détaillée à la section 2.2.2 en la modifiant quelque peu pour détecter lorsqu'un smartphone accède à la page et ainsi

le renvoyer vers une seconde page HTML appropriée ; en effet l'application sur les deux plates-formes n'aura pas les mêmes fonctionnalités :

```
function detectBrowser() {
 var useragent = navigator.userAgent;
 if (useragent.indexOf('iPhone') != -1 ||
      useragent.indexOf('Android') != -1) {
  window.location.replace("mobile.html");
 }
}
```

Supplément Web : l'application étudiée est disponible à l'adresse http://www.geotribu.net/dunod/gmaps/application.

5.2 APPLICATION WEB

5.2.1 Préparation HTML

Le début du développement passe par la construction des blocs `DIV` en construisant le corps de la page HTML et en préparant le code JavaScript qui accueillera les fonctions Google Maps :

```
<body onload="initialize(); detectBrowser();">
 <div id="map_canvas" style="width: 50%; height: 100%"></div>
 <div id="pano_canvas" style="width: 50%; height: 100%"></div>
</body>
<html>
```

Notre application web pour ordinateur comportera deux parties égales : la première présentant la cartographie de Rennes, la seconde les panoramas StreetView.

Dans l'élément `HEAD` de la page, déclarons le titre de l'application, les feuilles de styles, l'appel à l'API Google Maps et nos fonctions JavaScript `initialize` et `detectBrowser` :

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<meta http-equiv="content-type" content="text/html; charset=UTF-8" />
<title>Développez avec les API Google Maps - Application</title>
<style type="text/css">
 html {
  height: 100%;
 }
 body {
  height: 100%;
  margin: 0px;
  padding: 0px;
 }
 #map_canvas {
```

```
   height: 100%;
   float: left;
  }
  #pano_canvas {
   height: 100%;
   float: left;
  }
</style>

<script type="text/javascript" src="http://maps.google.com/maps/api
    /js?sensor=false"></script>
<script type="text/javascript">
 var map;
 var panorama;
 var infobulle;

 function initialize() {
  ..
 }

 function detectBrowser() {
  var useragent = navigator.userAgent;
  if (useragent.indexOf('iPhone') != -1 ||
        useragent.indexOf('Android') != -1) {
   window.location.replace("mobile.html");
  }
 }

</script>
</head>
```

L'application comportera donc deux parties égales sur la largeur de la page.

Enfin dans cette première partie, éditons notre fonction `initialize` pour afficher une carte centrée sur Rennes ainsi qu'un panorama :

```
 function initialize() {
  var rennes = new google.maps.LatLng(48.111, -1.6778);
  var myOptions = {
   zoom: 14,
   center: rennes,
   mapTypeId: google.maps.MapTypeId.ROADMAP,
   streetViewControl: true
  };

  map = new google.maps.Map(document.getElementById('map_canvas'),
       myOptions);

  var panoramaOptions = {
   position: rennes,
   pov: {
    heading: 34,
    pitch: 10,
    zoom: 1
   }
  };
```

```
    panorama = new  google.maps.StreetViewPanorama(
          document.getElementById("pano_canvas"), panoramaOptions);
    map.setStreetView(panorama);

  }
```

5.2.2 L'API Keolis

Le fonctionnement de l'API Keolis est détaillé à cette adresse : http://data.keolis-rennes.com/ .

Cette API – après un enregistrement et l'obtention d'une clé – permet, à l'aide d'une simple requête HTTP, de récupérer la liste des stations de vélo avec leur position géographique, leur nom, le nombre de vélos disponibles et le nombre d'emplacements libres :

- la requête http://data.keolis-rennes.com/json/?version=1.0&key=votre_clé_ici &cmd=getstation nous fournira un fichier au format JSON de la liste des stations de vélos de la ville avec toutes les informations sur celles-ci :

```
{"opendata":{
  "request":"http:\/\/data.keolis-
 rennes.com\/json\/?version=1.0&key=votre_clé_ici&cmd=getstation",
  "answer":{
   "status":{
    "@attributes":{
     "code":"0",
     "message":"OK"
    }
   },
   "data":{
    "station":[{
      "id":"75",
      "number":"75",
      "name":"ZAC SAINT SULPICE",
      "state":"1",
      "latitude":"48.1321",
      "longitude":"-1.63528",
      "slotsavailable":"21",
      "bikesavailable":"9",
      "pos":"0",
      "district":"Maurepas - Patton",
      "lastupdate":"2010-08-28T15:13:06+02:00"
     },{
      ...
      ...
     }]
    }
   }
  }
}
```

Nous obtenons ainsi toutes les informations en temps réel relatives à chaque station.

- la requête http://data.keolis-rennes.com/json/?version=1.0&key= votre_clé_ici &cmd=getstation&param[request]=number&param[value]=numero_station nous fournira toutes les informations sur une station en particulier.

Pour protéger notre clé d'une utilisation abusive, encapsulons notre requête dans un fichier PHP (getStation.php) et interrogeons ce fichier PHP via une requête AJAX (cf. sections 3.5.2 et 3.5.3) :

```
<?php header('Content-Type: text/html;charst=utf-8;');

 $content = ""; // si $content == 1 alors erreur dans la requête

 if (!isset($_GET['numberStation'])){
  $url = "http://data.keolis-rennes.com/json/?version=1.0
       &key=votre_clé_ici&cmd=getstation";
  $fp = @fopen($url,"r");
  if($fp){
   while(!feof($fp)){
    $content .= fgets($fp,1024);
   }
  } else{
   $content = 1;
  }
 } else{
  $num = $_GET['numberStation'];
  $url = "http://data.keolis-rennes.com/json/?version=1.0
      &key=votre_clé_ici&cmd=getstation&param[request]=number
      &param[value]=".$num;

  $fp = @fopen($url,"r");
  if($fp){
   while(!feof($fp)){
    $content .= fgets($fp,1024);
   }
  } else{
   $content = 1;
  }
 }

 echo $content;

?>
```

5.2.3 Affichage des stations sur la carte

Pour récupérer les données de la requête HTTP précédente, nous avons besoin des fonctions permettant d'ouvrir une connexion AJAX – fonctions `createXmlHttpRequest` et `downloadUrl`.

Affichons dès à présent les stations de vélos sur la carte en ajoutant ce code :

```
  var url = "getStation.php";
  downloadUrl(url, function(data) {
   // var jsonObject = JSON.parse(data);
```

```
    // JSON.parse ne fonctionne pas sur mobile Safari
    var jsonObject = eval('(' + data + ')');
    var stations = jsonObject.opendata.answer.data.station;
    for (var i = 0 ; i < stations.length; i++){
     var station = stations[i];
     var latlng = new google.maps.LatLng(parseFloat(station.latitude),
                                         parseFloat(station.longitude));
     var nbSlot = station.slotsavailable;
     var nbBike = station.bikesavailable;

     var marker = new google.maps.Marker({
      position: latlng,
      title: station.name,
      map: map
     });
    }
   });
```

Le retour de la requête est au format JSON qui a l'avantage d'être un objet JavaScript ce qui permet d'évaluer rapidement l'expression pour la transformer en un objet JavaScript.

Nous faisons cependant ici appel à la méthode `eval` (potentiellement dangereuse car il y a exécution de code JavaScript) en lieu et place de la fonction `JSON.parse` parce que cette dernière n'est pas supportée par le navigateur Safari de l'iPhone.

Nous obtenons donc une carte affichant des marqueurs correspondant aux stations de vélo ainsi qu'un panorama (voir figure 5.1).

Figure 5.1 — Application (1)

Améliorons maintenant les marqueurs afin que d'un coup d'œil, l'internaute puisse avoir l'information de la disponibilité d'un vélo ou d'un emplacement pour en déposer un. Nous utiliserons pour cela un code couleur :

- vert : supérieur à 5,
- orange : entre 1 et 5,
- rouge : aucun.

Et coupons nos marqueurs en deux : à gauche la disponibilité des vélos, à droite le nombre d'emplacements libres. Déclarons tous ces marqueurs :

```
var icongo = new google.maps.MarkerImage(
    'resources/icons/mm_20_go.png',
    new google.maps.Size(12,20),
    new google.maps.Point(0,0),
    new google.maps.Point(6,20));
```

Il nous reste maintenant à définir une fonction `createMarker` prenant en argument le nombre de vélos disponibles, le nombre d'emplacements libres, les coordonnées géographiques et le nom de la station. Ainsi nous pourrons choisir le marqueur :

```
function createMarker(latlng, name, nbSlot, nbBike){
 var marker = new google.maps.Marker({
  position: latlng,
  title: name,
  map: map
 });

 if (nbBike == 0){
  if (nbSlot == 0){
   marker.setIcon(iconrr);
  }
  if (nbSlot > 0 && nbSlot <= 4){
   marker.setIcon(iconro);
  }
  if (nbSlot > 4){
   marker.setIcon(iconrg);
  }
 }

 if (nbBike > 0 && nbBike <= 4){
  if (nbSlot == 0){
   marker.setIcon(iconor);
  }
  if (nbSlot > 0 && nbSlot <= 4){
   marker.setIcon(iconoo);
  }
  if (nbSlot > 4){
   marker.setIcon(iconog);
  }
 }

 if (nbBike > 4){
  if (nbSlot == 0){
   marker.setIcon(icongr);
  }
  if (nbSlot > 0 && nbSlot <= 4){
   marker.setIcon(icongo);
  }
  if (nbSlot > 4){
   marker.setIcon(icongg);
  }
 }
}
```

Et enfin ajoutons une infobulle qui s'ouvre lors d'un clic sur le marqueur :

```
infobulle = new google.maps.InfoWindow();
function createMarker(latlng, name, nbSlot, nbBike){

...
 google.maps.event.addListener(marker, "click", function() {

  infobulle.setContent('<b>Station : '+name+'</b><br> - nombre
  de vélos disponibles : '+nbBike+ '<br> - nombre d\'emplacements
  libres : '+nbSlot);

  infobulle.open(map, marker);

 });
}
```

Appelons cette fonction de création de marqueurs à l'initialisation de la carte dans la boucle parcourant toutes les stations de vélos :

```
createMarker(latlng,station.name,nbSlot,nbBike);
```

5.2.4 Affichage des stations sur la carte

La partie cartographie étant finie, il nous reste à travailler sur le panorama StreetView : le but étant d'afficher le panorama autour des stations de vélos lorsque l'utilisateur clique sur une station.

Lors de l'initialisation, le panorama doit donc être invisible :

```
 var panoramaOptions = {
  position: rennes,
  pov: {
   heading: 34,
   pitch: 10,
   zoom: 1
  },
  visible: false
 };
```

Lors d'un clic sur un marqueur, ajoutons l'édition de la position géographique de la caméra et rendons visible le panorama :

```
 google.maps.event.addListener(marker, "click", function() {
  infobulle.setContent('<b>Station : '+name+'</b><br> - nombre
  de vélos disponibles : '+nbBike+ '<br> - nombre d\'emplacements
  libres : '+nbSlot);
  infobulle.open(map, marker);

  panorama.setPosition(latlng);
  panorama.setVisible(true);
 });
```

Masquons le panorama lorsque l'utilisateur ferme une infobulle :

```
google.maps.event.addListener(infobulle, 'closeclick', function(){
 panorama.setVisible(false);
});
```

Et enfin supprimons la possibilité d'aller dans les photos suivantes et précédentes du panorama et masquons l'affichage de la rue dans le panorama :

```
var panoramaOptions = {
 pov: {
  heading: 34,
  pitch: 10,
  zoom: 1
 },
 visible: false,
 linksControl: false,
 addressControl: false
};
google.maps.event.addListener(infobulle, 'closeclick', function(){
 panorama.setVisible(false);
});
```

La figure 5.2 montre la nouvelle version de notre application.

Figure 5.2 — Application (2)

5.2.5 Optimisation pour smartphones

La version de l'application pour les ordinateurs de bureau étant finalisée, il reste à développer celle pour les smartphones (iPhone et Android). Toutes les fonctionnalités développées pour les ordinateurs sont utilisables également pour les smartphones ; ainsi peu de modifications seront nécessaires.

En effet, la seule différence résidera dans les contrôles de navigation spéciaux pour les smartphones et dans une utilisation différente du service StreetView : en effet, les petits écrans ne nous permettent pas de « découper » l'application en deux.

Sur smartphone, un clic sur une station ouvrira une infobulle et nous laisserons le loisir à l'utilisateur d'utiliser StreetView en déplaçant le Pegman sur la carte.

La feuille de styles CSS ainsi que le corps de la page HTML ont été un peu modifiés :

```
<style type="text/css">
 html {
  height: 100%;
 }
 body {
  height: 100%;
  margin: 0px;
  padding: 0px;
 }
 #map_canvas {
  height: 100%;
 }
</style>
...
<body onload="initialize();">
 <div id="map_canvas" style="width: 100%; height: 100%"></div>
</body>
</html>
```

La fonction `detectBrowser` a été supprimée, le panorama simplifié et les événements relatifs au diaporama supprimés :

```
 function initialize() {
  var rennes = new google.maps.LatLng(48.111, -1.6778);
  var myOptions = {
   zoom: 14,
   center: rennes,
   mapTypeId: google.maps.MapTypeId.ROADMAP,
   streetViewControl: true,
   navigationControlOptions: {
    style: google.maps.NavigationControlStyle.ANDROID
   },
   mapTypeControlOptions: {
    style: google.maps.MapTypeControlStyle.DROPDOWN_MENU
   }
  };
map = new google.maps.Map(document.getElementById('map_canvas'),
     myOptions);

  var url = "getStation.php";
  downloadUrl(url, function(data) {
   // var jsonObject = JSON.parse(data);
   // JSON.parse ne fonctionne pas sur mobile Safari
   var jsonObject = eval('(' + data + ')');
   var stations = jsonObject.opendata.answer.data.station;
   for (var i = 0 ; i < stations.length; i++){
```

```
    var station = stations[i];
    var latlng = new google.maps.LatLng(parseFloat(station.latitude),
         parseFloat(station.longitude));
    var nbSlot = station.slotsavailable;
    var nbBike = station.bikesavailable;

    createMarker(latlng,station.name,nbSlot,nbBike);
   }
  });

 }
...
  google.maps.event.addListener(marker, "click", function() {
   infobulle.setContent('<b>Station : '+name+'</b><br> - nombre
   de vélos disponibles : '+nbBike+ '<br> - nombre d\'emplacements
   libres : '+nbSlot);
   infobulle.open(map, marker);
  });
```

La figure 5.3 montre la version pour périphérique mobile de notre application.

Figure 5.3 — Application (3)

En résumé

Cet exemple de cartographie nous a permis de reprendre les notions vues dans les chapitres précédents et de les mettre en application pour des plates-formes différentes – PC et smartphones.

Une application Google Maps est assez simple à coder une fois les concepts intégrés et avec un peu d'habitude à la programmation JavaScript, le développement d'une application est assez rapide.

DEUXIÈME PARTIE

Utiliser les API sur les périphériques mobiles

Cette partie traite de l'intégration de l'API Google Maps dans les périphériques mobiles, et plus particulièrement dans les SDK du système d'exploitation de l'iPhone et de l'iPad, qui se nomme iOS, et celui du système d'exploitation produit par Google, qui s'appelle Android.

Avant de présenter les différentes façons d'utiliser ces composants intégrés dans ces deux systèmes, nous allons présenter le contexte du développement d'applications mobiles, ainsi que leur distribution.

Nous allons ensuite parcourir le processus de développement pour iOS et pour Android.

Nous présenterons les différents environnements de développement pour ces deux SDK et nous verrons comment adhérer au programme de déploiement des applications.

Nous verrons ensuite comment intégrer une carte dans une application, comment la manipuler et comment ajouter des couches d'informations. Enfin, nous parlerons de l'interaction avec le capteur GPS de ces périphériques mobiles.

6

Introduction

Objectif

Ce chapitre a pour but de faire découvrir le concept d'application mobile ainsi que d'essayer de répondre aux questions qui se posent généralement à leur propos. En quoi ces applications diffèrent-elles d'une application issue d'un développement web ? Comment les distribuer ?

Il présente également Android et iOS, représentatifs de ces nouveaux systèmes d'exploitation pour périphériques mobiles, et dans quel contexte ils ont été lancés.

6.1 PRÉSENTATION DU CONTEXTE

Les plus récents *smartphones* proposent aux développeurs des kits de développement d'application très riches. En effet, ces téléphones sont de véritables petits ordinateurs, embarquant beaucoup d'électronique. Il est donc possible d'élaborer des applications complexes.

Bien entendu, ces téléphones possèdent tous un navigateur Internet, et comme ils interprètent bien le langage JavaScript, ils sont pour la plupart capables d'afficher des cartes Google Maps développées pour le Web.

Cependant, il est intéressant de pouvoir accéder à des cartes lors de l'utilisation d'applications spécifiques. Par exemple, pour les applications de localisation du genre « où se trouve le plus proche restaurant? », il est utile de pouvoir passer de la liste des restaurants à une carte les affichant, juste en cliquant sur un onglet, et connaître sa position actuelle sur la carte grâce au GPS embarqué. Nous pouvons aussi coupler l'exploration d'information sur une carte à la possibilité d'appeler un numéro de téléphone.

Au-delà de ces nouvelles perspectives, on peut se demander si les coûts de développement ne vont pas exploser, s'il faut déployer son application sur de multiples cibles (iPhone, Android, BlackBerry, Nokia, etc.). Devant le succès de certains smartphones, les différents acteurs du marché veulent prendre leur part, car ne pas être présent sur ces marchés d'avenir peut être préjudiciable...

Bref, il n'existe pas de recettes miracles ! Mais pour mieux se faire une idée du développement de telles applications, les chapitres 7 et 8 présentent l'intégration des API Google Maps au sein des SDK de l'iOS et d'Android.

Déploiement des applications

Ici, pour distribuer ses applications, il ne s'agit pas d'héberger sur un serveur son travail, comme on peut le faire en mode Web. Il faut passer par un marché virtuel officiel pour chacun des smartphones.

Pour l'iPhone et l'iPad, il y a l'AppStore et pour Android, AndroidMarket. Par ce système, les acteurs garantissent un bon fonctionnement des applications téléchargées, et profitent au passage d'un prélèvement sur les achats d'applications payantes. Cela réduit d'autant la marge possible du développeur.

Cette garantie de bon fonctionnement est plus élevée chez Apple, puisqu'il existe un vrai processus de validation, à la fois électronique et humain, afin de garantir que l'expérience utilisateur du client sera conforme aux exigences ergonomiques d'Apple. Cependant, le cahier des charges fait plus de 200 pages et le temps de validation peut être long, même si l'on note une franche amélioration par rapport au début.

Google est plus souple sur ce point. Il faut évidemment soumettre des applications qui fonctionnent, mais le développeur est assez libre sur l'ergonomie. Il existe aussi des règles qui interdisent un contenu illicite. Mais les applications seront plutôt éliminées après coup, et la licence du développeur peut être suspendue.

6.2 LES SYSTÈMES ANDROID ET IOS

6.2.1 iOS

iOS est le nom du système d'exploitation développé par Apple pour l'iPod, l'iPhone et l'iPad.

Origines

Le nom iPhone OS est apparu en mars 2008 lors du lancement du SDK pour iPhone. Ce système d'exploitation a pris le nom d'iOS plus récemment, en juin 2010.

Dès septembre 2009, Apple annonça que la barre des 50 millions d'iPhone et d'iPod Touch avait été franchie. Ce système d'exploitation a donc connu un succès fulgurant. Dans le modèle économique d'Apple, l'apparition du AppStore s'est révélée décisive. Ce magasin virtuel, qui permet aux développeurs de proposer leurs applications au public, a vraiment augmenté l'intérêt d'iOS.

On comptait en mars 2010 plus de 200 000 applications différentes pour un nombre de téléchargements s'élevant à 4 milliards.

Apple reçoit 30 % des revenus liés à la vente d'application sur l'AppStore, le développeur gardant les 70 % restants.

App Store

Conditions de déploiement

Pour pouvoir déployer une application sur l'AppStore, il faut ouvrir un compte sur le portail développeur de l'iPhone, et payer une licence de développement de 79 €. Cette licence permet d'avoir accès à l'atelier de développement et de déployer les applications sur quelques téléphones de test. On peut ensuite compiler grâce au certificat de développeur en mode Release, ce qui permet d'envoyer les applications achevées sur le serveur de publication de l'AppStore.

Conditions d'acceptation

Apple reste le seul décideur de la publication ou non de l'application sur l'AppStore. L'application doit notamment suivre à la lettre les recommandations en termes d'ergonomie. L'application sera testée en conditions réelles d'utlisation, pour voir si l'expérience utilisateur est conforme aux spécifications d'Apple. Suivant la nature de l'application, ce processus de validation peut sembler long, parfois plus d'une semaine.

6.2.2 Android

Afin de pouvoir se démarquer de ses concurrents et imposer son système d'exploitation, Google se devait d'être innovant. Pour cela, le géant américain a notamment choisi de proposer son système sous une licence de type logiciel libre.

Nous allons voir dans cette section quelle est l'origine d'Android et comment il a su se positionner par rapport à une concurrence féroce.

Origines

Android est à l'origine une startup spécialisée dans le développement mobile, rachetée par Google en août 2005. Android est maintenant un système d'exploitation *open source* basé sur un noyau Linux ; il cible *smartphones*, assistants personnels (PDA), terminaux mobiles et même désormais téléviseurs.

Afin de faciliter le lancement du système d'exploitation, l'Open Handset Alliance a été créé le 5 novembre 2007 à l'initiative de Google, dont le but est de développer des normes ouvertes pour les téléphones portables dont les principaux acteurs sont HTC, LG, Motorola, Samsung, Archos, Sony Ericsson, Samsung, Toshiba, T-Mobile, China Mobile.

Android fut d'abord lancé aux États-Unis le 22 octobre 2008 par HTC sous le nom de « HTC Dream » sur le réseau T-Mobile. Son arrivée en France a eu lieu le 12 mars 2009 par Orange. Google a depuis lancé son propre téléphone le

5 janvier 2010 ; il s'agit d'un téléphone HTC mais la marque commerciale Nexus One a bien été déposée par Google.

En lançant son propre système d'exploitation, Google vient se frotter à une concurrence déjà rude, composée principalement d'Apple avec iOS, de Microsoft avec Windows Mobile, de Nokia avec Symbian OS, et bien sûr de RIM avec Blackberry.

Au 2e trimestre 2008, avant le lancement d'Android, les principales parts de marché étaient les suivantes :

- Nokia : 47,5 %
- Windows Mobile : 20 %
- Blackberry : 17,7 %
- Apple : 12,9 %

Début 2010, une fois Android lancé et l'iPhone en pleine expansion, les parts de marchés avaient bien évolué :

- Apple : environ 30 %
- Windows Mobile : environ 20 %
- Android : environ 20 %
- Blackberry : environ 20 %
- Nokia : environ 10 %

Android a bien su s'implanter face à une rude concurrence, et cette tendance semble se confirmer depuis.

Android Market

Comme on l'a vu précédemment, les applications écrites pour les smartphones, qu'ils soient basés sur Android ou iOS, se déploient sur des « marchés » : Android Market pour les applications Android et l'Apple Store pour les applications iPhone.

Le téléchargement d'applications pour Android *via* l'Android Market se fait directement par le téléphone.

L'Android Market a été ouvert fin 2008. En juillet 2010, il comptait plus de 83 000 applications, dont environ 61 % gratuites. Le succès de ce market est énorme ; en effet, environ 1 000 applications sont ajoutées chaque jour et plus d'un milliard de téléchargements ont déjà été réalisés.

Conditions de déploiement

Pour pouvoir déployer ses applications sur Android, il faut créer un compte développeur et acquérir une licence coûtant 25 $. La page http://market.android.com/publish/signup guide le développeur qui souhaiterait déployer ces applications Android.

Les applications peuvent être gratuites ou payantes. Dans le cas d'une application payante, le développeur reçoit 70 % du prix de l'application et Google 30 %.

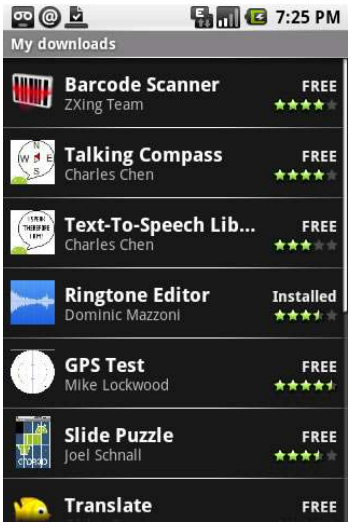

Figure 6.1 — Android Market

Elles doivent également être signées ; cette signature comporte le nom du développeur et elle permet de faire le lien entre les applications et les développeurs.

Conditions d'acceptation

Contrairement à Apple, Google n'a pas instauré de conditions quant à la diffusion ou la vente d'applications. On trouve donc sur l'Android Market tous les types d'applications, des jeux vidéo pour enfants aux applications pour adultes. C'est peut-être aussi grâce à cela que le nombre d'applications disponibles explose.

En résumé

Après ce bref descriptif des systèmes d'exploitation Android et iOS, ainsi que de leurs contextes technique et commercial, les chapitres suivants entrent dans le vif du sujet, en abordant le développement d'applications Google Maps sur ces systèmes.

7

iOS pour iPhone et iPad

Objectifs

Le but de ce chapitre est de découvrir comment utiliser les API Google Maps au sein même du SDK de l'iOS.

Nous allons d'abord présenter l'environnement de développement d'application Xcode, puis voir comment ajouter une carte avec les composants disponibles dans le SDK.

Nous allons ensuite découvrir les multiples possibilités de manipulation de ces cartes et enfin voir comment il est possible d'interagir avec le récepteur GPS.

7.1 INSTALLATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

7.1.1 Prérequis au développement d'application pour iOS

Pour développer une application sur iOS, il est nécessaire de posséder un Mac avec processeur Intel. Il est bien entendu préférable de posséder un iPhone ou un iPad pour tester les applications développées.

Afin de pouvoir télécharger le SDK, il faut s'enregistrer auprès d'Apple. Un enregistrement gratuit existe à cette adresse : http://developer.apple.com/programs/register ; il vous permettra de développer des applications, mais vous ne pourrez pas les tester ailleurs que sur le simulateur iPhone inclus dans l'atelier de développement. Et vous ne pourrez pas les distribuer sur l'AppStore.

Pour pouvoir installer vos applications sur votre iPhone et sur ceux de vos clients avec l'AppStore, vous devez adhérer au programme Apple Developer Program à cette adresse : http://developer.apple.com/programs/start/standard. Cette adhésion est payante (79 € pour une licence simple).

7.1.2 Installation du SDK

Grâce à votre adhésion au programme Apple, vous pouvez vous connecter sur le site de ressources dédiées aux développements d'application iPhone : http://developer.apple.com/iphone (voir figure 7.1). Vous y trouverez la documentation complète des bibliothèques disponibles dans le SDK et de multiples exemples de code. Il existe aussi une section Downloads, où se trouvent le dernier SDK et le logiciel Xcode qui permet de développer les applications iPhone.

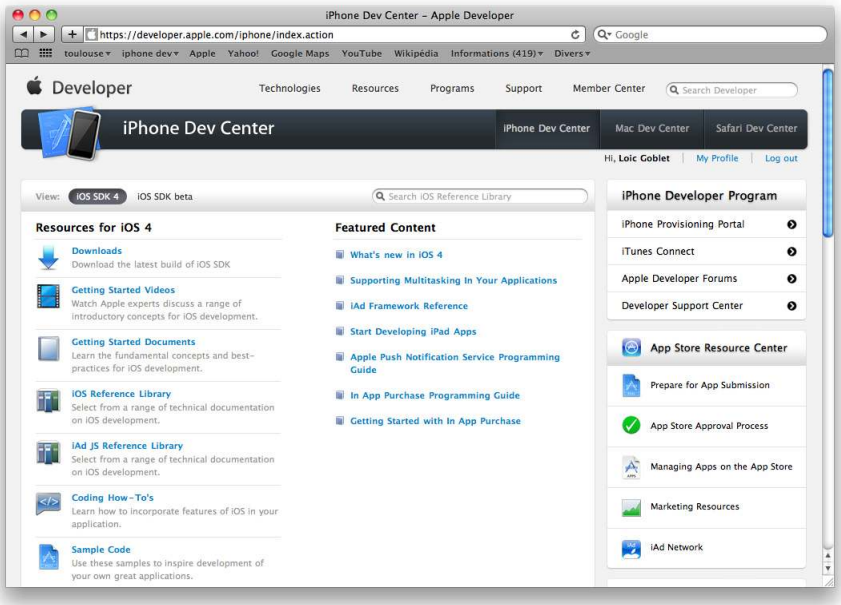

Figure 7.1 — L'iPhone Dev Center

La dernière version à ce jour de ces deux composants (Xcode 3.2.3 et iOS_SDK 4.0.2) représente un total de 2,68 Go.

Une fois téléchargé, il suffit de double cliquer sur l'archive .dmg, et un assistant d'installation démarre.

Le logiciel Xcode est alors installé dans le répertoire Developer du disque dur que vous avez sélectionné pendant l'installation. On peut le lancer en double-cliquant sur l'application Xcode située dans le répertoire Developer/Applications.

7.1.3 Une première application

Nous allons créer un premier projet d'application iPhone, dans lequel nous ajouterons ensuite une carte.

Afin de bien comprendre la programmation d'applications iPhone, nous n'allons pas utiliser les composants graphiques pour la conception de l'interface ; nous allons les coder. Le langage de programmation utilisé est Objective-C, qui est un langage orienté objet dérivé du C.

Pour créer un nouveau projet, démarrez Xcode, puis cliquez sur File>New Project... Un assistant de création de projet s'ouvre (voir figure 7.2).

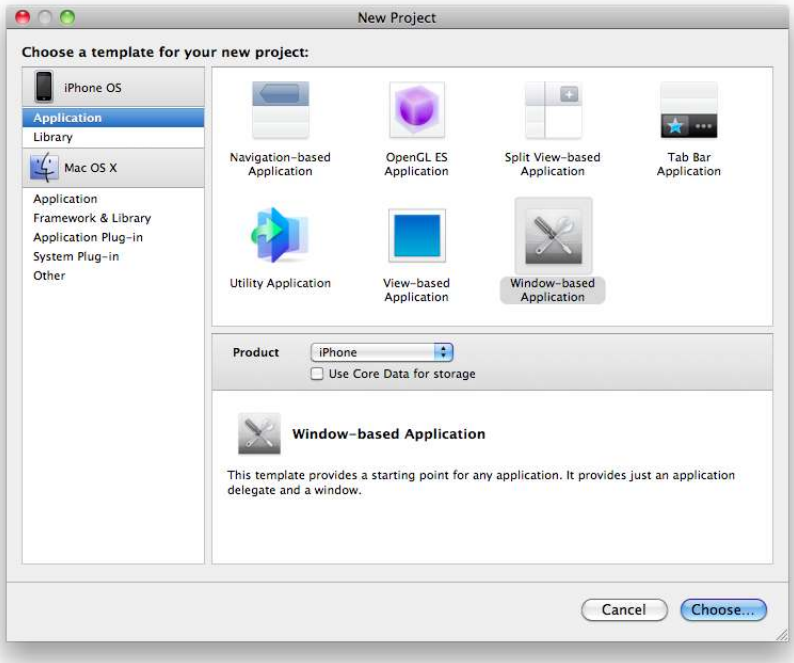

Figure 7.2 — L'assistant de création de projet Xcode

Sélectionnez le modèle « Window-based Application » dans la rubrique iPhone OS/Application. La case Product permet de sélectionner le périphérique cible, soit l'iPhone, soit l'iPad. Après avoir nommé et rangé votre nouveau projet, vous devez voir l'atelier de développement dans son ensemble (voir figure 7.3).

Dans le répertoire Classes, nous pourrons ajouter des classes : les fichiers d'interface avec l'extension .h et leur implémentation avec l'extension .m .

Nous avons ici choisi un développement pour iPhone.

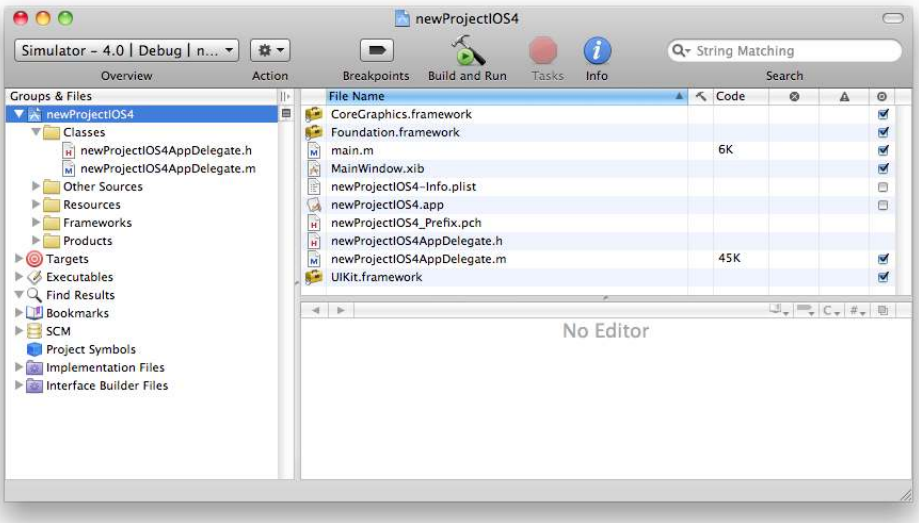

Figure 7.3 — Xcode, l'atelier de développement

Nous pouvons déjà compiler le projet et le lancer dans le simulateur d'iPhone. Il suffit pour cela de cliquer sur Build and Run. Un iPhone apparaît alors, avec une page blanche à l'écran (voir figure 7.4).

Figure 7.4 — Le simulateur d'iPhone démarre sur une page vide

L'atelier de développement d'applications iPhone est maintenant installé, et nous allons pouvoir entrer dans le vif du sujet.

7.2 AJOUTER UNE CARTE AVEC LES COMPOSANTS INTÉGRÉS DANS LE SDK

7.2.1 Ajout du framework MapKit

Pour utiliser les composants Google Maps du SDK iOS, il faut ajouter à votre projet (que vous aurez nommé « MapView ») le framework MapKit, qui contient les bibliothèques cartographiques.

Pour ajouter ce framework, il suffit de se placer sur le dossier Frameworks dans l'arborescence du projet, puis à l'aide d'un clic droit, de sélectionner Add>Existing Framework, puis dans cette longue liste, de choisir MapKit.

Vous devriez voir apparaître le framework comme dans la figure 7.5.

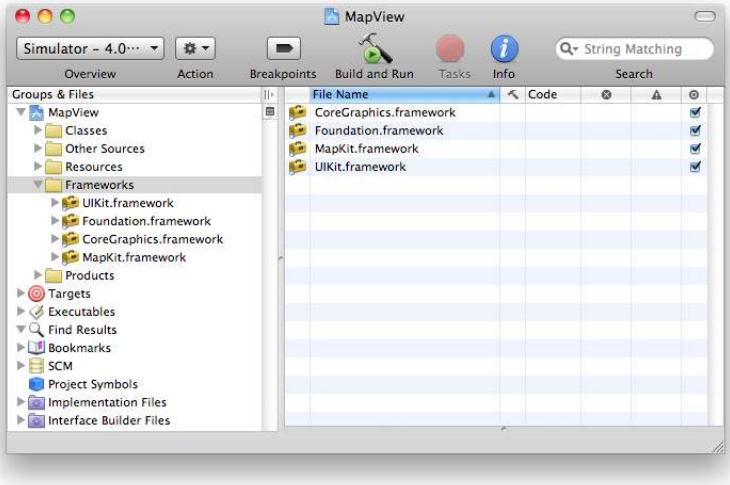

Figure 7.5 — L'ajout du framework MapKit

7.2.2 Ajout d'un contrôleur de vue

Nous allons ajouter une vue au projet. Cette vue contiendra la carte. Il faut ici cliquer sur File>New File et choisir le template « UIViewController subclass ».

Le projet contient une classe dont le nom se termine par `AppDelegate` qui régit le lancement de l'application. La classe du contrôleur de vue a été appelée `MapViewController`. Il faut donc ici déclarer un objet de type `MapViewController` dans l'interface de la classe `AppDelegate`.

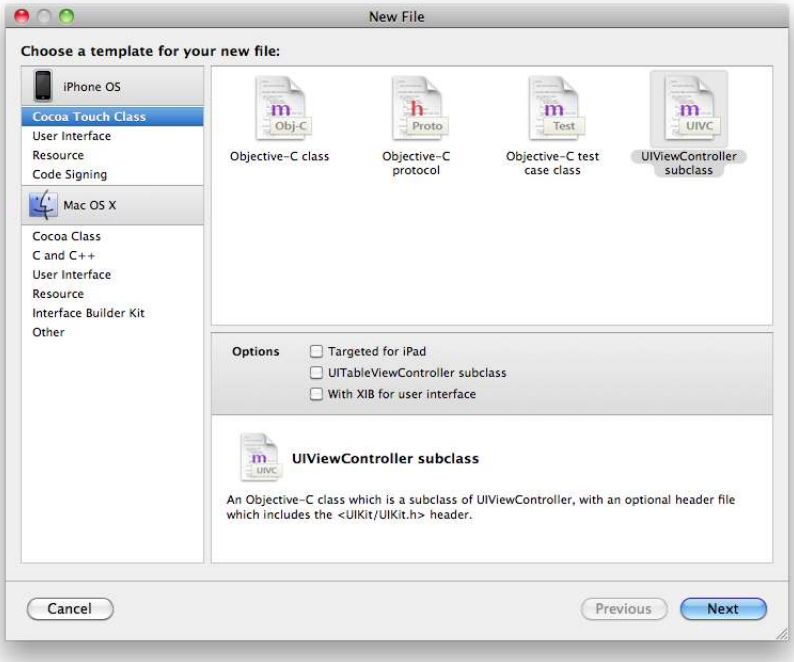

Figure 7.6 — L'assistant de création de fichier de code

```
@class MapViewController;

@interface MapViewAppDelegate : NSObject <UIApplicationDelegate>
{
  UIWindow *window;
  MapViewController *myMapViewController;
}

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

@end
```

Il faut ensuite placer au premier plan ce contrôleur de vue dans la fenêtre de l'iPhone. La méthode `applicationDidFinishLaunching` est appelée à la fin du lancement du projet. C'est dans cette méthode qu'il convient d'appeler la nouvelle vue que l'on vient de créer.

Voici le code du fichier MapViewAppDelegate :

```
#import "MapViewAppDelegate.h"
#import "MapViewController.h"

@implementation MapViewAppDelegate

@synthesize window ;
```

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
  myMapViewController = [[MapViewController alloc]
initWithNibName:nil bundle:nil];
  [window addSubview:myMapViewController.view];
  [window makeKeyAndVisible];
}

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

@end
```

On voit donc comment mettre en avant notre vue : il faut l'initialiser puis l'ajouter à la vue principale.

Attention : On remarque ici un aspect majeur de la programmation Objective-C, la gestion de la mémoire occupée par les objets : il est important de bien libérer les objets en fin de vie. Ici, on libère (méthode `release`) `myMapViewController`.

7.2.3 Mise en place d'une carte

Nous souhaitons maintenant afficher une carte dans notre vue de type `MapViewController`. Dans l'interface de cette classe, il faut inclure mapKit/MapKit.h, car c'est dans ce fichier que se trouve la classe `MKMapView`, qui représente la carte.

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

@interface MapViewController : UIViewController
{
  MKMapView *mapView;
}
```

Nous allons ensuite initialiser cette `MKMapView` dans l'implémentation de `MapViewController`. Cela est fait dans la méthode `viewDidLoad` :

```
-(void)viewDidLoad{
  mapView=[[MKMapView alloc] inithWithFrame:self.view.bounds];
  [self.view insertSubView:mapView atIndex:0];
}
```

Nous voyons que nous avons initialisé la carte avec le paramètre `Frame` qui permet de définir la zone d'affichage de la carte. Ici, nous avons choisi la taille de la fenêtre principale pour avoir une carte en plein écran. Il convient ensuite d'ajouter cette vue à la vue principale (et de la libérer à la fin).

```
- (void)viewDidUnload {
   mapView = nil;
}

- (void)dealloc {
   [mapView release];
   [super dealloc];
}
```

La figure 7.7 montre le résultat lorsque l'on compile et lance l'application dans le simulateur. Nous voyons une carte simple, en plein page, centrée sur les États-Unis. Il s'agit bien d'une carte Google Maps comme l'atteste le logo de cette entreprise, en bas à gauche.

Figure 7.7 — Une carte s'affiche

En allant dans l'onglet Project et en choisissant Edit Project Settings, on peut changer la version du SDK. En sélectionnant iPhone Device 3.2, on cible plus particulièrement l'iPad.

La figure 7.8 montre le résultat du lancement du simulateur, qui démarre alors automatiquement en mode iPad.

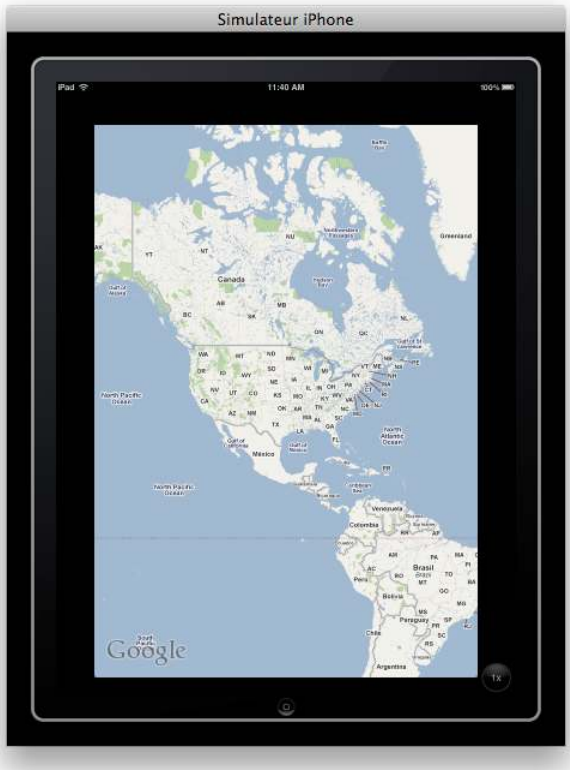

Figure 7.8 — La version iPad

7.3 MANIPULATION DES CARTES

Cette section présente les différentes possibilités offertes par le framework MapKit.

Le choix du type de carte, la possibilité de zoomer, le positionnement de la carte et la manipulation de marqueurs sont disponibles dans iOS 4 ainsi que dans iOS 3.2. Cela signifie que ces API fonctionneront de la même manière sur iPhone et sur iPad.

Cependant, la manipulation de couches sur les cartes `MKOverlay` n'est disponible que pour l'iOS 4. Elle ne fonctionne donc pas sur l'iPad.

7.3.1 Choisir le type de carte

Nous avons le choix entre différentes couches de représentation de la carte : Standard, Satellite ou Hybride (voir section 2.3.12).

La classe `MKMapView` possède une propriété `mapType` qui peut prendre trois valeurs différentes de type `MKMapType`. Il s'agit de `MKMapTypeStandard`, `MKMapTypeSatellite` et `MKMapTypeHybrid`. On peut accéder à cette propriété grâce à la notation point.

```
mapView.mapType = MKMapTypeSatellite ;
```

Par défaut il s'agit du type Standard comme nous l'avons vu précédemment. La figure 7.9 donne un aperçu des deux autres possibilités.

Figure 7.9 — Des cartes de types satellite et hybride

7.3.2 Zoomer

La propriété `zoomEnabled` détermine si l'on peut zoomer dans la carte grâce à un double-clic ou avec deux doigts qui s'écartent, et dézoomer avec deux doigts qui se rapprochent. Par défaut, la valeur de cette propriété est `YES`. Nous pouvons donc interdire le zoom, et figer une carte de cette manière :

```
mapView.zoomEnabled = NO ;
```

Il convient alors de bien centrer la carte avec le bon niveau de zoom. Nous allons voir comment réaliser cela.

7.3.3 Centrer la carte et fixer un niveau de zoom

Pour positionner la carte à un endroit précis, et avec un niveau de zoom désiré, nous allons utiliser quelques nouvelles structures présentes dans la bibliothèque MapKit.

Tout d'abord, présentons `MKCoordinateRegion`. Cette structure représente la portion de carte à afficher. Elle est composée d'un objet `center` de type `CLLocationCoordinate2D` et d'un objet `span` de type `MKCoordinateSpan`.

`CLLocationCoordinate2D` est une structure qui contient les coordonnées géographiques d'un point (latitude et longitude).

`MKCoordinateSpan` est une structure qui définit la surface représentée par la carte que nous souhaitons afficher. Cette structure est composée de deux objets de type `CLLocationDegrees`, `latitudeDelta` et `longitudeDelta`. Pour `latitudeDelta`, il s'agit de la distance du nord au sud, mesurée en degrés que l'on souhaite voir s'afficher. Un degré pour ce paramètre correspond à 111 kilomètres. Pour `longitudeDelta`, il s'agit cette fois de la distance en degré d'est en ouest. Contrairement aux distances en latitude, ici 1 degré représente 111 kilomètres à l'équateur, mais représente 0 kilomètre aux pôles.

Voici comment on peut déclarer une carte centrée sur Toulouse et qui représente la ville entière.

```
MKCoordinateSpan span;
span.latitudeDelta=0.1;
span.longitudeDelta=0.1;

CLLocationCoordinate2D location;
location.latitude = 43.604363;
location.longitude = 1.442951;

MKCoordinateRegion region ;
region.span=span;
region.center=location;
```

Figure 7.10 — Une carte centrée sur Toulouse

Il faut ensuite attacher cette région à la carte que nous affichons. Cela est effectué grâce à la méthode `setRegion` qui s'applique à une `MKMapView`.

```
[mapView setRegion:region animated:TRUE];
```

Le paramètre `animated` permet de spécifier si l'on souhaite une transition animée entre une précédente localisation de la carte et la nouvelle position.

Nous pouvons ensuite ajuster l'aspect d'une région pour être sûr qu'il sera adapté à la taille spécifiée par les bordures de la `MKMapView`. On utilise pour cela la méthode `regionThatFits:(MKCoordinateRegion)region`. Cette méthode renvoie une nouvelle région toujours centrée au même endroit mais avec une surface différente.

7.3.4 Insérer des marqueurs

Comme avec l'API Google Maps en JavaScript (voir section 3.1), il est possible avec le SDK iPhone de positionner toutes sortes de marqueurs sur une carte.

Un marqueur est un objet qui doit respecter le protocole `MKAnnotation`. Ce protocole oblige l'implémentation d'une propriété nommée `coordinate` qui représente les coordonnées géographiques d'un point (latitude et longitude avec `CLLocationCoordinate2D`).

Nous pouvons aussi définir un titre et un sous-titre pour ce marqueur.

Voici un exemple de code qui respecte ce protocole. En premier lieu, le fichier d'interface :

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>


@interface MyAnnotation : NSObject <MKAnnotation>{
    CLLocationCoordinate2D      coordinate;
    NSString                    *aTitle;
    NSString                    *aSubTitle;
}

@property (nonatomic, retain) NSString *aTitle;
@property (nonatomic, retain) NSString *aSubTitle;

- (id)initWithCoordinate:(CLLocationCoordinate2D)coord;

@end
```

La notation `<MKAnnotation>` permet de définir le fait que l'objet `MyAnnotation` doit se conformer au protocole `MKAnnotation`.

Nous avons ajouté une méthode d'initialisation à partir des coordonnées géographiques.

Passons maintenant au fichier d'implémentation :

```
#import "MyAnnotation.h"

@implementation MyAnnotation

@synthesize coordinate;
@synthesize aTitle;
@synthesize aSubTitle;

- (id)initWithCoordinate:(CLLocationCoordinate2D)c {
   coordinate = c;
   self.aSubTitle = @"un sous-titre";
   self.aTitle = @"un titre";
   return self;
}

- (NSString *)subtitle {
   return self.aSubTitle;
}

- (NSString *)title {
   return self.aTitle;
}

- (void)dealloc {
   [aTitle release];
   [aSubTitle release];
   [super dealloc];
}

@end
```

Les méthodes `title` et `subtitle` renvoient les chaînes de caractères contenant le titre et le sous-titre du marqueur.

Nous pouvons dès maintenant ajouter ce marqueur à la carte. Il suffit, dans le contrôleur de vue de la carte, de créer un objet de type `MyAnnotation` et de l'ajouter à la carte.

```
MyAnnotation *annot = [[MyAnnotation alloc] initWithCoordinate:location];

annot.aTitle=@"un marqueur";
annot.aSubTitle=@"le centre de la carte";

[mapView addAnnotation:annot];
```

Le simulateur permet d'observer le résultat (voir figure 7.11).

Cette façon de procéder est assez simple, mais n'offre pas beaucoup de souplesse quant à la personnalisation de l'apparence du marqueur.

Pour changer quelques options de notre marqueur, il va falloir intervenir dans la méthode `viewForAnnotation` de la classe `MKMapView`. Pour cela, nous allons utiliser la notion de « *delegate* ».

Figure 7.11 — Un marqueur au centre de la carte

Notre contrôleur de vue va pouvoir se conformer à `MKMapViewDelegate`, et ainsi nous pourrons agir au sein de l'implantation de notre contrôleur de vue sur la méthode `viewForAnnotation`.

Cela se déclare d'abord dans l'interface MapViewController.h :

```
@interface MapViewController : UIViewController <MKMapViewDelegate>
```

Ensuite, toujours dans la méthode `viewDidLoad` où l'on agit sur `MkmapView`, il faut déclarer que le `delegate` de notre carte est bien la classe du contrôleur de vue :

```
mapView.delegate = self;
```

Nous allons maintenant pouvoir agir sur la méthode `viewForAnnotation`. Cette méthode renvoie un objet de type `MKAnnotationView`. Nous allons utiliser la classe `MKPinAnnotationView` qui hérite de `MKAnnotationView`, et qui procure la vue d'une icône en forme d'épingle comme celle présente dans l'application Google Maps.

Il existe deux attributs spécifiques à cette classe, `pinColor` et `animatesDrop`. Le premier permet de choisir parmi trois couleurs l'aspect de la tête d'épingle représentant le marqueur : rouge par défaut, `MKPinAnnotationColorRed`, vert avec `MKPinAnnotationColorGreen` et violet avec `MKPinAnnotationColorPurple`.

La seconde propriété `animatesDrop` est un booléen qui indique si le marqueur doit être animé lors de son affichage. L'animation représente la chute de l'épingle vers son point d'ancrage.

D'autres propriétés sont disponibles dans la classe `MKAnnotationView`. Il y en a une qui permet d'indiquer si l'on souhaite voir les informations de notre `MKAnnotation` dans une infobulle lorsque l'on clique sur le marqueur. Elle se nomme `canShowCallout`.

Il est obligatoire d'utiliser un identifiant pour des raisons de gestion de mémoire.

```
- (MKAnnotationView *) mapView:(MKMapView *)mapView viewForAnnotation:(
    id <MKAnnotation>) annotation{
      MKPinAnnotationView *annView=[[MKPinAnnotationView alloc]
      initWithAnnotation:annotation reuseIdentifier:@"currentloc"];
  [annView setAnimatesDrop:TRUE];
  [annView setPinColor:MKPinAnnotationColorGreen];
  [annView setCanShowCallout:YES];
  return annView;
}
```

Ici, la seule différence avec l'exemple précédent est la couleur verte de la tête d'épingle et l'animation à l'affichage.

Il existe d'autres propriétés intéressantes dans `MKAnnotationView` :

- on peut ajouter des accessoires à droite ou à gauche de l'infobulle avec `leftCalloutAccessoryView` et `rightCalloutAccessoryView`, typiquement, une petite flèche pour aller vers une page contenant plus d'informations ;
- l'infobulle peut être placée où souhaité grâce à `calloutOffset` ;
- la propriété `draggable` permet de spécifier si l'on peut déplacer le marqueur ;
- la propriété `enabled` permet de rendre le marqueur insensible au clic ;
- on peut changer l'aspect du marqueur grâce à la propriété `image` et déplacer le marqueur de quelques pixels avec la propriété `centerOffset`.

Voici un exemple où nous changeons l'aspect du marqueur. Il est nécessaire d'ajouter une image dans les ressources du projet, ici le fichier blue.png. Il faut aussi utiliser un objet `MKAnnotationView` et non plus un `MKPinAnnotationView`. Nous allons aussi décaler le marqueur de 10 pixels vers la droite et 30 pixels vers le haut pour qu'il ne recouvre pas l'intitulé « Toulouse ».

```
- (MKAnnotationView *) mapView:(MKMapView *)mapView viewForAnnotation:(
    id <MKAnnotation>) annotation{
  MKAnnotationView *annView=[[MKAnnotationView alloc]
      initWithAnnotation:annotation reuseIdentifier:@"currentloc"];

  [annView setCanShowCallout:YES];
  [annView setImage:[UIImage imageNamed:@"blue.png"]];
  CGPoint decalage = CGPointMake(10.0,-30.0);
  [annview setCenterOffset:decalage];

  return annView;
}
```

La figure 7.12 montre le résultat en image.

Figure 7.12 — Un marqueur personnalisé

7.3.5 Ajouter des polylignes sur une carte

Depuis la dernière version du SDK, l'iOS 4,0, il est possible d'ajouter des couches supplémentaires au-dessus de la carte. Ces couches doivent respecter le protocole `MKOverlay`. Ces couches peuvent donc représenter des cercles, des rectangles ou bien des lignes à plusieurs segments.

Comme nous l'avons dit précédemment, cette fonctionnalité n'est pas encore implémentée pour l'iPad, qui utilise actuellement iOS 3.2.

Nous allons donc examiner comment créer une ligne à plusieurs segments, comment l'insérer sur la carte et comment agir sur la vue de cette couche quand la carte se demande s'il faut l'afficher.

Pour les besoins de l'exemple, nous quittons Toulouse pour New York. Nous reprenons donc la méthode `viewDidLoad` de notre contrôleur de vue `MapViewController`.

```
- (void)viewDidLoad {
  [super viewDidLoad];
  mapView =[[MKMapView alloc] initWithFrame:self.view.bounds];
  mapView.mapType=MKMapTypeStandard;
  mapView.delegate = self ;
  mapView.zoomEnabled = YES ;
  MKCoordinateSpan span;
  span.latitudeDelta=0.05;
  span.longitudeDelta=0.05;
```

```
    CLLocationCoordinate2D location;
    location.latitude = 40.78;
    location.longitude = -73.97;
    MKCoordinateRegion region ;
    region.span=span;
    region.center=location;
    [mapView setRegion:region animated:TRUE];
    CLLocationCoordinate2D mapCoords[4];
    mapCoords[0].latitude = 40.767995;
    mapCoords[0].longitude = -73.981876;
    mapCoords[1].latitude = 40.764265;
    mapCoords[1].longitude = -73.973014;
    mapCoords[2].latitude = 40.796858;
    mapCoords[2].longitude = -73.949260;
    mapCoords[3].latitude = 40.800554;
    mapCoords[3].longitude = -73.958208;
    MKPolyline *polyLine = [MKPolyline polylineWithCoordinates:mapCoords
        count:4];
    [mapView insertOverlay:polyLine atIndex:0];
    [self.view insertSubview:mapView atIndex:0];
}
```

Et maintenant, voyons comment configurer l'affichage de cette ligne à multiples segments. Il s'agit d'écrire dans la méthode `viewForOverlay` qui nous vient du `MKMapViewDelegate`.

```
- (MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(
    id <MKOverlay>)overlay {
      MKPolylineView *polylineView = [[[MKPolylineView alloc]
         initWithOverlay:overlay] autorelease];
  polylineView.strokeColor = [UIColor blueColor];
  polylineView.lineWidth = 8.0;
  return polylineView;
}
```

L'objet `MKPolylineView` hérite de `MKOverlayPathView` qui lui-même hérite de `MKOverlayView`. Nous pouvons l'insérer ici. Les attributs `strokeColor` et `lineWidth` représentent la couleur de la ligne et son épaisseur.

La figure 7.13 montre le résultat dans le simulateur, une délimitation de Central Park.

7.3.6 Ajouter des cercles sur une carte

Voyons maintenant comment ajouter un cercle sur notre carte, et remplissons-le d'une teinte transparente.

Il faut utiliser la classe `MKCircle` et la classe `MKCircleView`.

Dans l'exemple de code ci-dessus, il suffit de supprimer la définition de `mapCoords` et à la place de `MKPolyline`, utilisez ceci :

Figure 7.13 — Un segment à lignes multiples sur une carte

```
MKCircle *myCircle = [MKCircle circleWithCenterCoordinate:location
                      radius:1000];
[mapView addOverlay:myCircle];
```

Nous avons utilisé le centre de la carte comme centre du cercle et nous avons choisi un rayon de 1 000 mètres.

Ensuite il suffit de remplacer le code de la méthode `viewForOverlay` comme suit :

```
- (MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(
          id <MKOverlay>)overlay {
       MKCircleView* circleView = [[[MKCircleView alloc]
       initWithOverlay:overlay] autorelease];
  circleView.strokeColor = [UIColor blueColor];
      circleView.lineWidth = 5.0;
      circleView.fillColor = [UIColor colorWithRed:0.0 green:0.0 blue:1.0
                              alpha: 0.4];
      return circleView;
}
```

Nous avons ici utilisé la propriété `fillColor` qui permet de remplir la forme que nous souhaitons dessiner (voir figure 7.14). Le paramètre `alpha` représente un niveau de transparence.

Figure 7.14 — Un cercle sur une carte

7.3.7 Ajouter des polygones sur une carte

Nous allons maintenant dessiner un polygone qui est une forme fermée constituée de plusieurs points. Nous allons pour cela réutiliser la série de points `mapCoords` de la section 7.3.4.

Cet exemple s'apparente grandement aux deux exemples précédents. Voici pour le polygone :

```
MKPolygon *polygon = [MKPolygon polygonWithCoordinates:mapCoords count:4];
[mapView insertOverlay:polygon atIndex:0];
```

Et voici pour sa vue :

```
- (MKOverlayView *)mapView:(MKMapView *)mapView viewForOverlay:(
    id <MKOverlay>)overlay {
      MKPolygonView *polygonView = [[[MKPolygonView alloc]
      initWithOverlay:overlay] autorelease];
  polygonView.strokeColor = [UIColor blueColor];
  polygonView.fillColor = [UIColor colorWithRed:0.0 green:0.0 blue:1.0
    alpha: 0.4];
  polygonView.lineWidth = 8.0;
  return polygonView;
}
```

Et voici le résultat sur le simulateur (figure 7.15).

Figure 7.15 — Un polygone sur une carte

7.4 INTERACTION AVEC LE RÉCEPTEUR GPS

7.4.1 Récupération des coordonnées de l'utilisateur

L'iPhone possède un GPS intégré et Apple a bien entendu mis à disposition des développeurs une bibliothèque complète pour mettre en œuvre ce composant.

En utilisant la bibliothèque CoreLocation/CoreLocation.h présente dans le framework `CoreLocation`, on a accès au *delegate* `CLLocationManagerDelegate`. On peut donc ainsi démarrer un `CLLocationManager` et donner des filtres pour les notifications de changement de position. La mise à jour de la position appelle la méthode `didUpdateToLocation: newLocation fromLocation:oldLocation`. Il est donc assez simple de récupérer la nouvelle position.

Avec les cartes, c'est encore plus simple. En effet, dans la classe `MKMapView`, il existe une propriété appelée `showsUserLocation`. Il s'agit d'un booléen, qui est par défaut initialisé à `NO`. Mais lorsque nous le changeons à la valeur `YES`, apparaît sur la carte un point bleu clignotant, qui correspond à la position du téléphone (voir la figure 7.16).

```
mapView.showsUserLocation = YES ;
```

Sur le simulateur, la position trouvée correspond au siège de l'entreprise Apple à Cupertino en Californie.

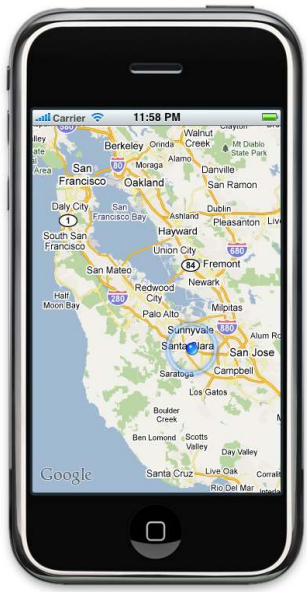

Figure 7.16 — L'utilisateur est localisé

Si l'on clique sur ce point bleu, une infobulle apparaît avec le texte « Current location » sur le simulateur, ou « Lieu actuel » sur votre iPhone, s'il est configuré en français.

Le fait de modifier cette propriété à `YES` enclenche l'utilisation du framework `CoreLocation` (nous n'avons ici pas besoin de le déclarer explicitement dans l'arborescence du projet). Tant que cette valeur reste à `YES` la `MKMapView` va continuer de chercher la position de l'utilisateur, et mettre à jour périodiquement cette information sur la carte.

Cette valeur à `YES` donne aussi la possibilité d'obtenir la latitude et la longitude de l'utilisateur, grâce à la propriété `userLocation`.

Voici comment récupérer cette position actuelle :

```
CLLocationCoordinate2D maPosition;
maPosition.latitude = mapView.userLocation.coordinate.latitude;
maPosition.longitude = mapView.userLocation.coordinate.longitude;
```

7.4.2 Le géocodage inverse

Le framework MapKit et son implémentation avec les API Google Maps donnent la possibilité d'utiliser ce service. De quoi s'agit-il ?

Comme on l'a vu section 4.1, ce service permet de convertir une coordonnée (la paire latitude/longitude) en une information relative à cette position, comme le nom de la rue, la ville ou le pays.

Attention : Google n'autorise l'utilisation de ce service que s'il est relié à une utilisation de carte Google Maps. Un deuxième aspect à tenir compte concerne le nombre limité de requêtes de ce type qu'une application pourra effectuer. Il n'y a pas de chiffres précis à ce propos mais il s'agit d'être vigilant. Il est conseillé de lancer ce service seulement à la demande de l'utilisateur, et pas automatiquement dans votre programme.

Pour utiliser ce service, il faut déclarer `MKReverseGeocoderDelegate` dans votre interface et déclarer une propriété de type `MKReverseGeocoder`.

Dans l'implémentation de votre classe, il faut initialiser cet objet `MKReverseGeocoder` avec une localisation, par exemple `mapView.userLocation` :

```
myReverseGeocoder = [[[MKReverseGeocoder alloc]
               initWithCoordinate: mapView.userLocation.coordinate]
                 autorelease];
myReverseGeocoder.delegate = self;
[myReverseGeocoder start];
```

Pour suivre le protocole de `MKReverseGeocoderDelegate`, il faut maintenant implémenter deux méthodes : `didFindPlacemark` et `didFailWithError`. La première est appelée quand le service a trouvé une réponse à la requête de géocodage inverse, et la seconde lorsqu'il a échoué.

Voici un exemple avec une alerte en cas d'erreur et un message dans la console si la position est trouvée :

```
- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder didFailWithError:
     (NSError *)error{
        NSString *msg = [[NSString alloc]
               initWithString:error.localizedDescription];
    UIAlertView *alert = [[UIAlertView alloc]
                           initWithTitle:@"Echec"
                           message:msg
                           delegate:nil
                           cancelButtonTitle: @"Done"
                           otherButtonTitles:nil];
    [alert show];
    [msg release];
    [alert release];
}

- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder didFindPlacemark:
      (MKPlacemark *)placemark
{
   NSLog(@"adresse actuelle : %@, %@",
      placemark.thoroughfare ,
      placemark.locality);
  }
```

Ici, on récupère donc le nom de la rue et le nom de la ville. Il existe d'autres propriétés liées à cet objet `MKPlaceMark`. Ces propriétés sont les suivantes :

- `administrativeArea` : correspond à la région en France ou à l'État aux États-Unis (comme Californie) ;
- `country` : correspond au nom du pays ;
- `countryCode` : correspond à l'abréviation standard du pays comme FR pour la France ;
- `locality` : correspond au nom de la ville ;
- `postalCode` : correspond au code postal ;
- `subAdministrativeArea` : correspond au département ;
- `subLocality` : correspond au nom du quartier lorsque celui-ci est renseigné ;
- `subThoroughfare` : correspond au numéro dans la rue ;
- `thoroughfare` : correspond au nom de la rue.

En résumé

Ce chapitre nous a permis d'aborder l'intégration de cartes Google Maps dans une application iOS, cela dans toute ses dimensions : contrôle de l'apparence de la carte, ajout de marqueurs et de primitives géométriques, interaction avec le GPS du périphérique mobile. Cette dernière fonctionnalité ouvre notamment un vaste champ d'applications interactives pour iPhone et iPad.

8

Android

Objectifs

Après avoir vu comment mettre en place des cartes Google Maps sur l'iPhone, voyons l'équivalent pour les téléphones sous Android.

Tout au long de ce chapitre, nous allons créer un projet que nous ferons évoluer selon nos besoins. Il a pour but de créer une application affichant la position courante sur une carte Google Maps, se mettant à jour selon les changements de position et envoyant ses données à un serveur ou à un autre téléphone.

8.1 INSTALLATION DE L'ENVIRONNEMENT DE DÉVELOPPEMENT

Cette section montre comment mettre en place l'environnement de développement et quels sont les outils à installer et utiliser afin de développer et tester vos applications.

8.1.1 SDK Android

Le SDK (*Software Development Toolkit* ou kit de développement logiciel) est l'ensemble des outils fourni par Google afin de pouvoir développer des applications Android. Il se présente sous la forme d'un dossier .zip qu'il suffit de décompresser où bon vous semble. Au cours de l'élaboration de notre projet, nous utiliserons la version 2.1.

Une fois le dossier décompressé, on y trouve plusieurs sous-dossiers, parmi lesquels :

- **Docs** : ce dossier contient la documentation HTML d'Android. Il s'agit en fait du contenu du site http://developer.android.com ; une version locale peut toutefois s'avérer utile !
- **Samples** (au sein de docs) : recense divers exemples d'applications ;
- **Tools** : contient les différents utilitaires Android ; ceux-ci ne seront pas détaillés ici, puisqu'avec l'utilisation du plug-in pour Eclipse (*cf.* section suivante), ils sont utilisés de manière transparente pour le développeur ;
- **Add-on** : contient des bibliothèques optionnelles, telle bien sûr celle de Google Maps ;
- **Platforms** : la bibliothèque android.jar y est présente, qui contient aussi bien les classes des packages propres à Android que les classes du JDK ;
- **Usb-drivers** : regroupe les drivers Windows permettant d'effectuer le débogage des applications directement sur le téléphone.

Notez bien le chemin vers le dossier contenant le SDK, celui-ci vous servira par la suite.

8.1.2 Installation d'Eclipse

Pour installer Eclipse, rien de plus simple. Cet environnement de développement intégré (ou IDE, *Integrated Development Environment*) vous aidera à développer vos projets pour Android ; il créera vos projets avec les bons dossiers et fichiers XML nécessaires.

Eclipse est disponible à l'adresse suivante : http://www.eclipse.org/downloads. Disponible pour Linux, Windows et Mac OS, il devrait faire le bonheur de tous...

Note : Une alternative à Eclipse est Netbeans. Pour ces deux IDE, des plug-in existent afin de faciliter la création de projet, le développement et le test des applications. Celui pour Netbeans se nomme « Undroid ».

8.1.3 Plug-in Android pour Eclipse

Installation

La mise en place du plug-in est très simple, elle se fait directement dans Eclipse. Pour cela lancez Eclipse et cliquez dans le menu sur Help puis sur Install New Software. Ensuite, cliquez sur Add pour ajouter le site de téléchargement du plug-in. Dans la fenêtre apparaissant à l'écran, ajoutez l'URL https://dl-ssl.google.com/android/eclipse dans le champ Location et donnez-lui le nom que vous souhaitez. Dans notre exemple, le nom est Android (voir figure 8.1).

Une fois le site ajouté, sélectionnez les outils développeurs, cliquez sur Next et laissez-vous guider. Si le site n'est pas reconnu, remplacez « https » par « http ».

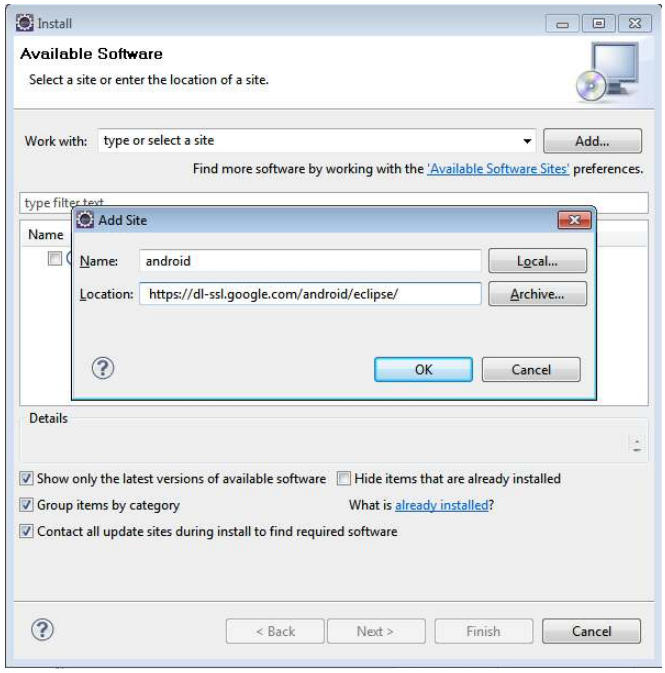

Figure 8.1 — Ajout du site de téléchargement

Ajout du chemin vers le SDK

Pour lier Eclipse au SDK, il faut lui indiquer le chemin vers ce dernier. Pour cela, il suffit d'aller dans le menu Window, puis dans Preferences. Dans le menu gauche, un menu Android est présent ; en le cliquant, vous pourrez indiquer le chemin vers le SDK (que vous avez noté précédemment, comme conseillé section 8.1).

Android Virtual Devices

L'Android Virtual Device (AVD) est un outil permettant de lancer l'émulateur. Cet émulateur est utile pour tester les applications avant de les déployer sur un périphérique. Il faut donc en créer un. Le menu permettant de gérer les AVD se situe dans Window – Android SDK And AVD Manager.

Tout d'abord, la première étape est de récupérer les *platforms* du SDK (qui sont en fait les différentes versions du SDK pour lesquelles vous aller tester vos applications). Pour cela, allez dans Installed Packages et cliquez sur Update All ; suivez ensuite les instructions.

Une fois les *platforms* installées, vous allez pouvoir créer votre AVD. Il suffit de cliquer sur Virtual Devices et sur New. Dans la fenêtre qui apparaît (*cf.* figure 8.2), donnez un nom à votre AVD et sélectionnez la version du SDK. Ici, nous utiliserons « Google API's – API Level 7 », qui correspond au SDK 2.1 (« Google API's » signifiant que nous utiliserons les bibliothèques optionnelles du SDK). Pour notre exemple de test, il faut créer un Android 2.1 – API Level 7.

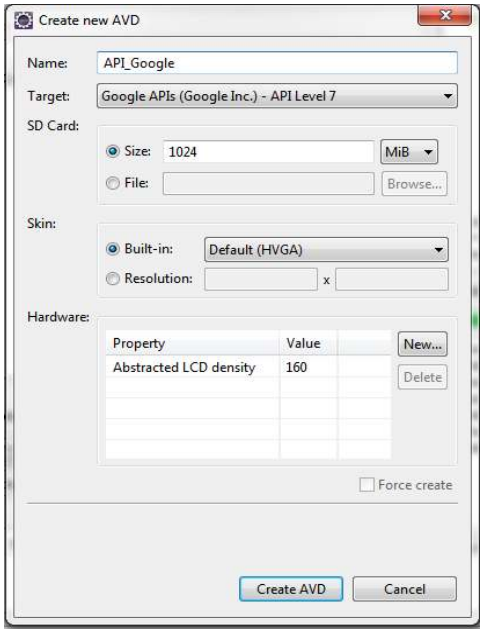

Figure 8.2 — Création d'un nouvel AVD

Vous voici maintenant prêt pour créer votre premier projet sous Android.

8.1.4 Le célèbre Hello World!

Dans cette section, nous allons enfin attaquer les choses sérieuses. Nous allons commencer par créer un projet élémentaire afin d'examiner comment est structuré un projet.

Création de notre premier projet

Pour créer un projet, rien de plus simple : dans le menu File, choisissez New et Android Project (*cf.* figure 8.3). Une fenêtre apparaît alors pour configurer le projet. Saisissez alors le nom du projet et la version du SDK. Ici, comme nous créons notre application pour le SDK 2.1, nous cochons Android 2.1 comme Build Target et saisissons « 7 » comme version minimale du SDK. Le champ CreateActivity sera la classe exécutée par défaut lors du lancement de l'application.

Suite à la création de ce projet, vous pouvez constater qu'un certain nombre de dossiers ont été créés ; voyons maintenant à quoi ceux-ci servent.

Structure du projet

À la racine du projet (*cf.* figure 8.4), se trouvent deux packages : src et gen. src regroupe les sources du projet, comme pour n'importe quel projet Java « classique ». gen, quant à lui, contient le fichier R.java, généré automatiquement par Eclipse et permet d'appeler

les images, textes, etc. contenus dans les fichiers XML. Il ne faut surtout pas toucher à ce fichier puisqu'il est géré et tenu à jour par Eclipse.

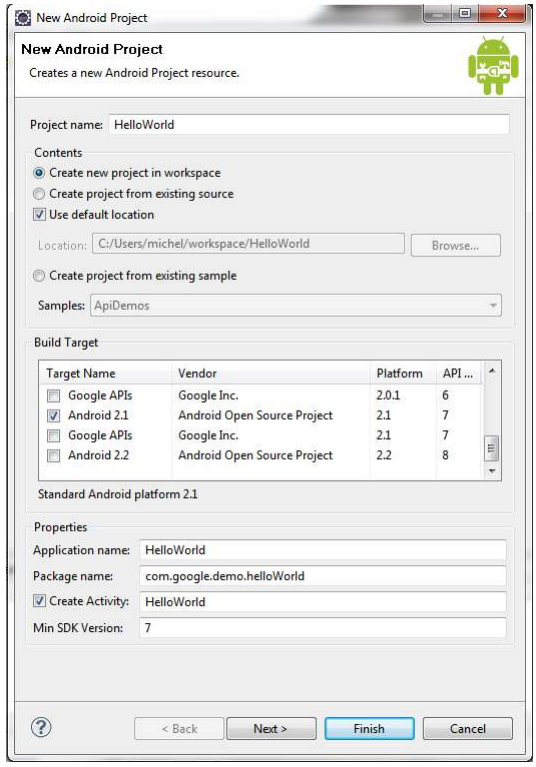

Figure 8.3 — Création du projet Hello World !

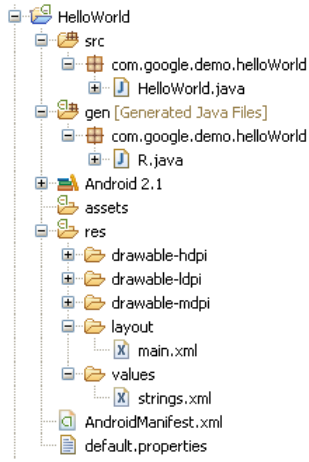

Figure 8.4 — Structure d'un projet

Autre dossier très utile, le dossier res, qui contiendra les fichiers ressources nécessaires à votre projet (images ou fichiers de propriétés). Par défaut, plusieurs dossiers y sont créés. Tout d'abord, trois dossiers « drawable », qui contiendront les images du projet, chaque dossier correspondant à des résolutions d'écran différentes. Le répertoire layout comprend le fichier main.xml, qui permet de créer l'interface graphique de l'application, ce que nous verrons par la suite. Pour finir, dans le dossier values se trouve le fichier strings.xml, qui recense un ensemble de chaînes de caractères utilisées au sein de l'application.

Enfin, à la racine de ce dossier res se situe le fichier AndroidManifest.xml, reprenant le nom de l'application, une icône ainsi que des autorisations (nous verrons l'utilité de celles-ci par la suite, *cf.* section 8.4.1).

Maintenant que notre projet est créé et que nous savons à quoi servent les dossiers, exécutons notre projet !

Exécution du Hello World

Pour exécuter notre application, c'est toujours aussi simple : un clic droit sur l'application puis Run as et Android Application. À cet instant, l'émulateur se lance et après quelques secondes de patience, vous pouvez voir votre première application apparaître (*cf.* figure 8.5).

Pour l'instant, vous n'avez besoin de saisir aucun code, puisque lorsque vous créez votre projet, « Hello World » est le code par défaut.

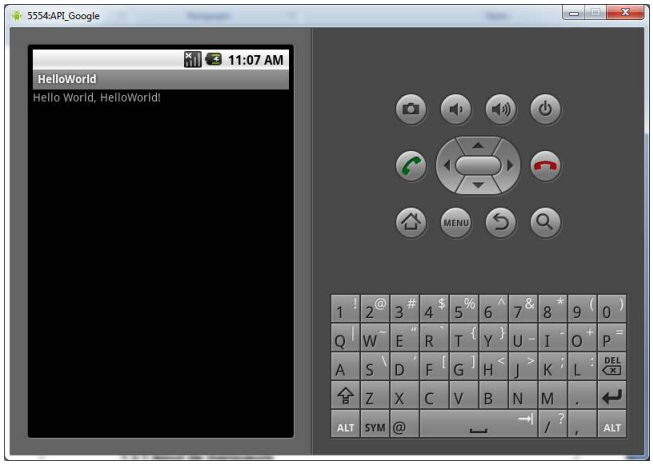

Figure 8.5 — Lancement de l'application sur l'émulateur

Maintenant que vous savez créer un projet et lancer une application, nous allons voir comment y intégrer des cartes Google Maps.

8.2 AJOUTER UNE CARTE AVEC LES COMPOSANTS INTÉGRÉS DANS LE SDK

L'ajout de cartes au sein de votre application ne sera pas beaucoup compliqué que celles listées lors des étapes précédentes, puisque grâce aux composants dédiés intégrés au SDK, peu de code suffit. Suivez bien les étapes suivantes !

8.2.1 *MD5 checksum*

La première étape consiste à obtenir un numéro de certificat. Pour cela, ouvrez une console et tapez la commande suivante :

```
keytool --list --alias androiddebugkey --keystore
<chemin_vers_le_fichier_debug>.keystore --storepass android --keypass android
```

Pour trouver le chemin vers votre fichier debug, dans Eclipse allez dans le menu Windows, puis Prefs, Android et Build.

Une fois cette commande saisie dans la console, vous devriez voir apparaître une clé sous cette forme (*cf.* figure 8.6) :

```
XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX
```

Conseil : La commande `keytool` se situe dans le dossier bin de votre JDK. Si la commande n'est pas connue, déplacez-vous dans le bon dossier.

Figure 8.6 — Résultat de la commande

Notez cette clé, elle vous sera utile pour la seconde étape.

8.2.2 Génération de la clé API

Une fois cette première clé obtenue, nous allons pouvoir générer la clé API nécessaire pour utiliser les API Google Maps.

Pour cela, rendez-vous sur le site http://code.google.com/intl/fr/android/maps-api-signup.html (un compte Google est nécessaire pour réaliser cette opération). Vous n'avez qu'à cocher la case signalant que vous avez bien lu les termes et conditions et saisissez la clé MD5 checksum obtenue lors de l'étape précédente et cliquez sur Generate API Key. Vous voyez maintenant votre clé API et un descriptif pour savoir où placer cette clé (*cf.* figure 8.7).

Tout est maintenant prêt pour afficher votre première carte Google Maps.

Figure 8.7 — Génération de la clé API

8.2.3 Création de la carte

Nous sommes donc prêts à nous lancer dans le vif du sujet avec la création d'une carte. Nous allons voir deux façons de la créer : *via* un fichier XML à l'intérieur duquel il est possible d'initialiser l'interface graphique et *via* du code.

Une première étape à effectuer avant de démarrer est d'ajouter à notre projet la clé que nous venons de générer. Tout d'abord, créez un projet nommé Google Maps, dont le Build Target est « Google API's en version 7 ». Ici, le nom de l'application est GoogleMaps, le nom du package est com.demo.googleMaps, le nom de l'*activity* est GoogleMaps et la version du SDK est la version 7 (*cf.* figure 8.8).

Ensuite, il faut ajouter la clé que nous avons générée dans la section précédente. Ajoutez la ligne suivante dans le fichier strings.xml :

```
<string name="google_key">0kdsEUYLxHcfJM10W3yc2er_CoLySzm_bvgJ9Gw</string>
```

en remplaçant bien sûr la clé par celle qui a été générée. Vous devriez obtenir un fichier ressemblant à celui-ci :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, GoogleMaps!</string>
    <string name="app_name">GoogleMaps</string>
    <string name="google_key">0kdsEUYLxHcfJM10W3yc2er_CoLySzm_bvgJ9Gw</string>
</resources>
```

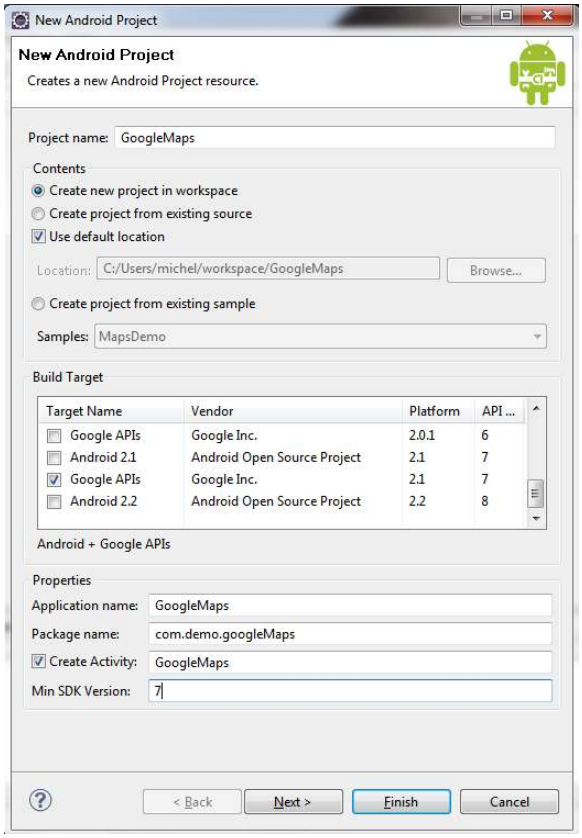

Figure 8.8 — Création du projet Google Maps

Pour finir, il faut indiquer dans le projet que nous utilisons une bibliothèque Google Maps. Pour cela, il faut ouvrir le fichier AndroidManifest.xml et ajouter la ligne suivante :

```
<uses-library android:name="com.google.android.maps"></uses-library>
```

Puis ajouter une autorisation à utiliser Internet avec la ligne :

```
<uses-permission android:name="android.permission.INTERNET" />
```

de manière à obtenir un fichier XML comme suit :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.demo.googleMaps"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label=
                                          "@string/app_name">
      <uses-library android:name="com.google.android.maps"></uses-library>
```

```
        <activity android:name=".GoogleMaps"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="7" />
    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

Via un fichier XML

Comme il a été dit précédemment, il est possible de générer les interfaces graphiques grâce à un fichier XML. Il s'agit du fichier main.xml, présent dans le dossier layout.

Conseil : Le fichier main.xml est un fichier créé par défaut lors de la création du projet, mais il est possible d'utiliser n'importe quel fichier xml que vous aurez créé et ajouté préalablement dans le dossier layout.

À l'intérieur de ce fichier, vous allez définir un `LinearLayout` à l'intérieur duquel sera ajoutée la carte.

Note : La classe `LinearLayout` est proposée dans le SDK d'Android et est nécessaire pour le positionnement des objets de l'interface graphique. Celle-ci indique que les éléments seront placés les uns en dessous des autres. D'autres classes existent comme `RelativeLayout`, `TableLayout`, `GridLayout`, etc.

Le code est le suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    >
<com.google.android.maps.MapView
  android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="@string/google_key"
    android:id="@+id/map"/>
</LinearLayout>
```

Votre classe principale doit maintenant étendre `MapActivity` et surcharger la méthode `isRouteDisplayed` et `onCreate`. Vous devez étendre `MapActivity` puisqu'il s'agit d'un type particulier d'`Activity`, une classe de la bibliothèque Maps. Une `Activity` est un écran permettant d'interagir avec l'utilisateur. Ici, `MapActivity` ajoute plus de contrôles et d'interactions par rapport à `Activity`.

La méthode `isRouteDisplayed`, à surcharger, indique simplement si les directions doivent être affichées sur la carte. `onCreate` doit également être surchargée pour indiquer que faire lors du chargement de l'`Activity`.

La classe principale sera donc :

```
public class GoogleMaps extends MapActivity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        MapView map = (MapView)findViewById(R.id.map);
        MapController mapController = map.getController();
        GeoPoint point = new GeoPoint((int)(48.512899*1E6),
                                      (int)(2.173944*1E6));
        mapController.setCenter(point);
        mapController.setZoom(17);
        map.invalidate();
    }

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }
}
```

La méthode `setContentView()` permet de dire que le fichier main.xml sera la vue courante pour cet écran.

Ensuite, l'appel à la méthode `findViewById(R.id.map)` permet de récupérer l'objet `MapView` défini dans le fichier main.xml dont l'id qui lui a été donné se nomme `map`. La récupération de l'objet `MapController` permet de pouvoir changer la position courante sur la carte et de changer le zoom.

Vous devriez obtenir un résultat similaire à celui présenté ci-dessous.

Via du code

La manière de procéder par code est assez similaire à celle *via* un fichier XML. Ici pas besoin du fichier main.xml, nous allons tout de suite nous occuper du fichier GoogleMaps.java.

Pour cela, nous allons « traduire » le fichier XML en code Java. Dans le fichier XML précédent, nous avons créé la carte à l'intérieur d'un `LinearLayout`. Dans le code, ceci se traduit ainsi :

```
public class GoogleMaps extends MapActivity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        LinearLayout linearLayout = new LinearLayout(this);
        linearLayout.setOrientation(LinearLayout.VERTICAL);
        MapView map = new MapView(this,
              this.getResources().getString(R.string.google_key));
        linearLayout.addView(map);
        MapController mapController = map.getController();
        GeoPoint point = new GeoPoint((int)(48.512899*1E6),
                                      (int)(2.173944*1E6));
        mapController.setCenter(point);
        mapController.setZoom(17);
        map.invalidate();
        setContentView(linearLayout);
    }

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }
}
```

Contrairement à la méthode précédente, la méthode `setContentView` ne récupère pas le fichier XML pour la vue courante mais l'objet `LinearLayout` que nous avons créé et à l'intérieur duquel nous avons placé notre `MapView`. Cette `MapView` est créée à partir de la vue courante et de la clé API.

Vaut-il mieux choisir une méthode plutôt qu'une autre ?

Non, ces deux méthodes sont équivalentes. Néanmoins, le fait de définir l'interface graphique dans le fichier XML permet d'alléger le code et de bien séparer le code métier de la partie affichage, notamment pour des applications de taille importante. Ce qui signifie aussi qu'il sera plus facile de modifier un fichier XML plutôt que du code Java.

Il est aussi possible de coupler les deux méthodes en déclarant les layouts et orientations d'écran dans le fichier XML et de préciser certaines choses dans le code, comme le texte d'un bouton.

8.2.4 Tests sur le téléphone

Voir sa carte s'afficher sur un émulateur est satisfaisant, mais exécuter son application sur un support concret est bien plus réjouissant. Voyons donc comment afficher la carte sur son téléphone.

Pour cela, branchez votre téléphone en mode USB. Vous devrez certainement installer les drivers afin que cette étape fonctionne ou réaliser une petite manipulation. Pour ce faire, la page http://developer.android.com/guide/developing/device.html pourra vous être utile. Ensuite, sur votre téléphone, activez le mode débogage en allant dans les paramètres du téléphone, dans Applications/Développement et cocher Débogage USB.

Pour finir, dans Eclipse, faites un clic-droit sur votre projet et au lieu de faire Run As/Android Application, choisissez Run As/Run Configurations. Dans l'onglet Target et cochez ensuite Manual (*cf.* figure 8.9).

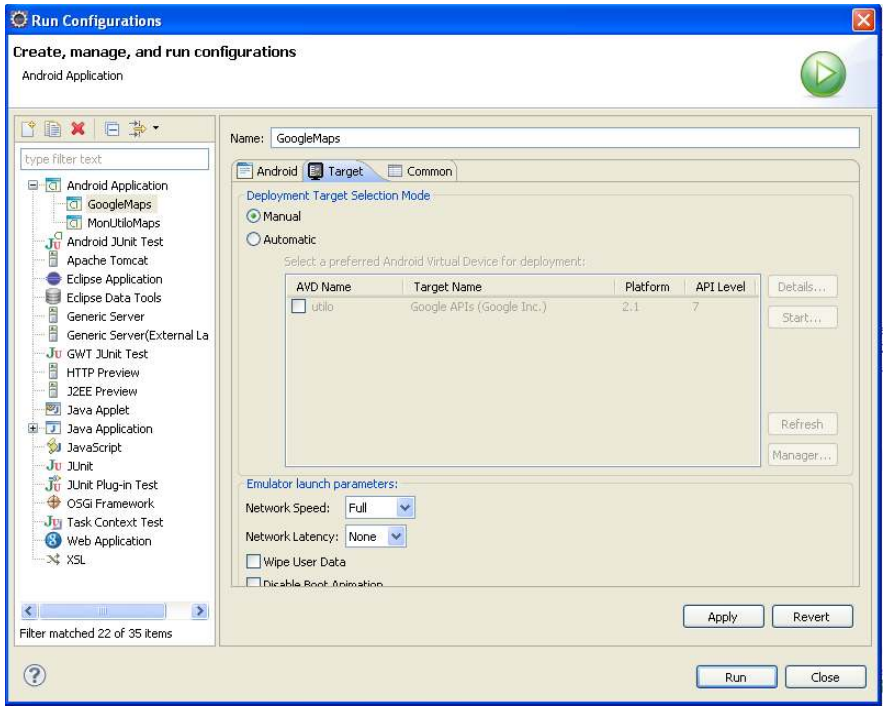

Figure 8.9 — Déploiement manuel sur le téléphone

Lancez maintenant votre application en sélectionnant votre téléphone (*cf.* figure 8.10) et admirez votre carte !

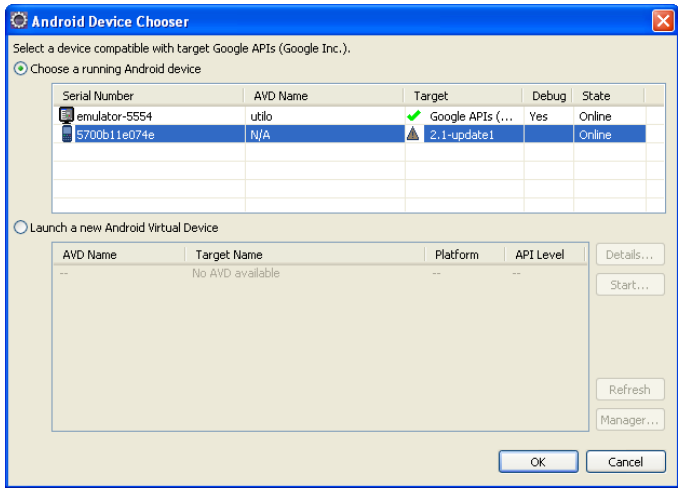

Figure 8.10 — Sélection du téléphone

8.3 MANIPULATION DES CARTES

Après avoir vu comment ajouter des cartes dans notre application, nous allons maintenant les enrichir.

8.3.1 Ajouter des marqueurs

Vous avez déjà très certainement utilisé Google Maps sur le Web et lors de vos recherches, vous avez dû remarquer la présence de marqueurs indiquant des points géoréférencés sur une carte. C'est ce que nous allons faire.

Pour cela, nous allons créer une classe `Marqueur` héritant de `Overlay` – une classe permettant de positionner un marqueur sur la carte – afin d'obtenir un marqueur personnalisé. Le constructeur a besoin du fichier image servant de marqueur, du point où placer celui-ci et du contexte courant. Le code est le suivant :

```
public class Marqueur extends Overlay {

  private Bitmap imageMarqueur;
  private String texteAAfficher;
  private GeoPoint point;
  private Context contexte;

  public Marqueur(Bitmap imageMarqueur, String texteAAfficher, GeoPoint point,
       Context context) {
    this.imageMarqueur = imageMarqueur;
    this.texteAAfficher = texteAAfficher;
    this.point = point;
    this.contexte = context;
  }

  @Override
```

```
    public void draw(Canvas canvas, MapView mapView, boolean shadow) {

      super.draw(canvas, mapView, shadow);
      Paint paint = new Paint();

      Point screenCoords = new Point();
      mapView.getProjection().toPixels(point, screenCoords);

      paint.setStrokeWidth(1);
      paint.setColor(Color.GRAY);
      paint.setAntiAlias(true);
      paint.setStyle(Paint.Style.STROKE);

      canvas.drawBitmap(imageMarqueur, screenCoords.x, screenCoords.y, paint);
    }

    @Override
    public boolean onTap(GeoPoint p, MapView mapView) {
      Point tapPx = new Point();
      Point markPx = new Point();
      mapView.getProjection().toPixels(p, tapPx);
      mapView.getProjection().toPixels(point, markPx);
      if ((tapPx.x > markPx.x - 30) && (tapPx.x < markPx.x + 30)
          && (tapPx.y > markPx.y - 30) && (tapPx.y < markPx.y + 30)) {
        Toast.makeText(contexte, texteAAfficher, Toast.LENGTH_LONG).show();
        return true;
      }else
        return false;
    }
  }
```

Dans la méthode `draw()`, rien de spécial à détailler : celle-ci ne fait que dessiner le petit robot sur la carte (*cf.* figure 8.11).

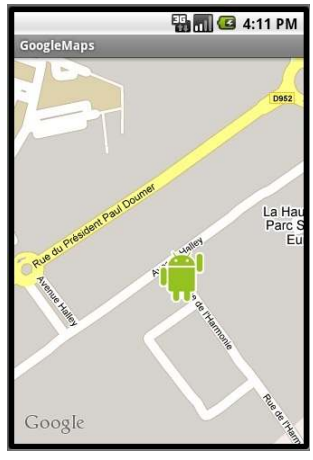

Figure 8.11 — Un marqueur sur une carte

La méthode `onTap()`se charge d'afficher un pop-up lorsque l'utilisateur sélectionne sur le marqueur. Ici, lorsque la méthode `onTap` est appelée, le texte passé en paramètre

du constructeur est affiché à l'écran et disparaît au bout de quelques secondes. Ce pop-up s'affiche grâce à l'appel à `Toast.makeText(...).show()` prenant en paramètres le contexte courant, le texte à afficher dans le pop-up et la durée d'affichage.

8.3.2 Changer le type de cartes

Le changement de type de cartes est très simple. Il suffit d'appeler la méthode correspondant à la vue Google Maps (*cf.* section 2.3.1.2) que vous souhaitez afficher.

Les méthodes disponibles sont les suivantes :

- `setSatellite(boolean)`,
- `setTraffic(boolean)`,
- `setStreetView(boolean)`.

8.3.3 Autres contrôles

Ajout d'un contrôle de zoom sur la carte

Il est très simple d'ajouter un zoom sur la carte. Il suffit d'appeler la méthode `setBuiltInZoomControls(true)` sur l'objet `MapView`.

Récupérer les coordonnées GPS à partir d'une adresse

Un objet très utile est présent dans l'API, il s'agit du `Geocoder`. Grâce à cet objet, il est possible de récupérer les coordonnées GPS correspondant à une adresse postale (*cf.* section 4.1).

Voici comment utiliser cet objet :

```
Geocoder geocoder = new Geocoder(activity) ;
List<Address> addresses = geocoder.getFromLocationName(
                          monAdresse,leNombreDeResultatsSouhaites);
```

Suite à l'appel de cette méthode, on obtient une liste d'adresses (de type `Address`) pour lesquelles il est possible de connaître leur latitude et longitude.

8.4 INTERACTION AVEC LE RÉCEPTEUR GPS

Après vous avoir vu comment manipuler les cartes Google Maps, nous allons voir comment interagir avec le téléphone et notamment avec son capteur GPS. Nous allons examiner dans cette section comment obtenir la position du GPS et comment l'utiliser dans notre projet.

8.4.1 La classe *LocationManager*

La classe mise en avant pour la réalisation de cette étape est la classe `LocationManager`. Celle-ci dispose de la méthode `getLastKnownLocation()`, qui a pour fonction de récupérer la dernière position connue par le GPS. Pour ce faire, nous allons créer une méthode dans la classe `GoogleMaps.java`. Nous l'appellerons `dernierePositionConnue()`, pour être original. Le code est le suivant :

```
public GeoPoint dernierePositionConnue(){
    LocationManager lm = (LocationManager)getSystemService(
                          Context.LOCATION_SERVICE);
    Location location = lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
    GeoPoint point = new GeoPoint((int)(location.getLatitude()*1E6),
                                  (int)(location.getLongitude()*1E6));
    return point;
  }
```

En plus du code, il faut ajouter des autorisations dans le fichier AndroidManifest.xml :

```
<uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION"></uses-permission>
<uses-permission
android:name="android.permission.ACCESS_MOCK_LOCATION"></uses-permission>
```

L'inconvénient de cette façon de faire est que nous ne pouvons pas savoir si la dernière position courante est récente ou non. En effet, si l'utilisateur du téléphone a coupé son GPS et a voyagé avec celui-ci, la dernière position connue peut être très différente du lieu où il se trouve réellement... Nous verrons une autre manière de procéder à la section 8.5.

8.4.2 Interaction entre la carte et le GPS

Nous savons maintenant récupérer la dernière position connue par le GPS ; voyons donc comment mettre à jour la carte une fois celle-ci récupérée.

Il est ici simple de modifier le code existant puisque dans notre exemple initial, nous créions une position. Il suffit juste de remplacer cette position par un appel à notre méthode créée précédemment. La classe `GoogleMaps.java` est donc maintenant la suivante :

```
public class GoogleMaps extends MapActivity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        MapView map = (MapView)findViewById(R.id.map);
        MapController mapController = map.getController();
        GeoPoint point = this.dernierePositionConnue();
        Overlay mark = new Marqueur(BitmapFactory.decodeResource(
              getResources(), R.drawable.androidmarker),"vous êtes ici",point,
```

```
                getBaseContext());
        map.getOverlays().add(mark);
        mapController.animateTo(point);
        mapController.setZoom(17);
        map.invalidate();
    }

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }
  public GeoPoint dernierePositionConnue(){
    LocationManager lm = (LocationManager)getSystemService(
                            Context.LOCATION_SERVICE);
    Location location = lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
    GeoPoint point = new GeoPoint((int)(location.getLatitude()*1E6),
                                  (int)(location.getLongitude()*1E6));
    return point;
  }
```

Cette nouvelle classe va donc récupérer la dernière position connue par le GPS, l'afficher sur une carte Google Maps et ajouter un marqueur sur celle-ci, comme pour la figure 8.11.

8.5 ÉVÉNEMENTS

Dans la section précédente, l'objectif était d'obtenir la dernière position courante du GPS et de l'afficher. Nous allons maintenant nous intéresser à ce qu'il faut faire pour mettre à jour la carte. Pour ce faire, une interface est présente dans le SDK, l'interface `LocationListener`.

Cette interface est à implémenter afin de pouvoir effectuer des changements de positions. Celle-ci permet de savoir que faire lorsque le GPS est activé/désactivé et lorsqu'un changement de position courante est notifié.

Pour cela, nous allons créer une classe `PositionCouranteListener` implémentant `LocationListener` afin de spécifier quoi faire dans chaque situation. Le code de la classe est le suivant :

```
public class PositionCouranteListener implements LocationListener {

  private GoogleMaps maps;

  public PositionCouranteListener(GoogleMaps maps) {
    this.maps = maps;
  }
  @Override
  public void onLocationChanged(Location location) {
    GeoPoint point = new GeoPoint((int)(location.getLatitude()*1E6),
                                  (int)(location.getLongitude()*1E6));
    maps.changerPosition(point);
  }
```

```
  @Override
  public void onProviderDisabled(String provider) {
    Toast.makeText(maps, "gps désactivé", Toast.LENGTH_LONG);
  }

  @Override
  public void onProviderEnabled(String provider) {
    Toast.makeText(maps, "gps activé", Toast.LENGTH_LONG);
  }

  @Override
  public void onStatusChanged(String provider, int status, Bundle extras) {

  }

}
```

Le constructeur est nécessaire afin d'avoir une référence sur la carte pour en modifier l'affichage. Lorsqu'un changement de position est notifié et que la méthode `onLocationChanged` est appelée, nous changeons la position courante de la carte. Pour cela, une méthode a été ajoutée dans la classe `GoogleMaps.java`. La méthode est la suivante :

```
public void changerPosition(GeoPoint point){
  map.getController().setCenter(point);
  map.invalidate();
}
```

Il suffit juste, de manière similaire à l'initialisation de la carte à une position, de récupérer le contrôleur de la carte et grâce à celui-ci, de changer la position. Et ensuite, de notifier à la carte de se rafraîchir. Pour que cette nouvelle méthode fonctionne, il faut également passer l'objet `MapView` en attribut de méthode.

Pour finir la modification du code, il faut que ce *listener* soit pris en compte et ajouté dans le code principal de la carte. Toujours dans la classe `GoogleMaps.java`, il faut donc créer une instance de notre *listener* et celui-ci vient s'ajouter au manager de localisation. Ainsi, la classe devient :

```
public class GoogleMaps extends MapActivity {
  private MapView map;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        map = (MapView)findViewById(R.id.map);
        MapController mapController = map.getController();
        PositionCouranteListener positionListener = new
                              PositionCouranteListener(this);
        LocationManager lm = (LocationManager)getSystemService(
                                Context.LOCATION_SERVICE);
        lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0,
                  positionListener);
```

```
        GeoPoint point = this.dernierePositionConnue();
        Overlay mark = new Marqueur(BitmapFactory.decodeResource(
                        getResources(), R.drawable.androidmarker),
                        "Vous êtes ici",point, getBaseContext());
        map.getOverlays().add(mark);
        mapController.animateTo(point);
        mapController.setZoom(17);
        map.invalidate();
    }

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }
  public GeoPoint dernierePositionConnue(){
    LocationManager lm = (LocationManager)getSystemService(
                          Context.LOCATION_SERVICE);
    Location location = lm.getLastKnownLocation(LocationManager.GPS_PROVIDER);
    GeoPoint point = new GeoPoint((int)(location.getLatitude()*1E6),
                                  (int)(location.getLongitude()*1E6));
    return point;
  }
  public void changerPosition(GeoPoint point){
    map.getController().setCenter(point);
    map.invalidate();
  }
}
```

L'ajout de ce *listener* a été réalisé par l'appel à la méthode `requestLocationUpdates` dont les paramètres sont le *provider* pour récupérer la position GPS, la durée minimale et la distance minimale entre deux notifications. Attention cependant au niveau de batterie en sélectionnant des durées et distances très courtes...

Vous savez maintenant afficher et manipuler les cartes Google Maps. Néanmoins pour des applications plus avancées, vous aurez besoin d'interagir avec un serveur afin d'envoyer et de recevoir des données. La section suivante est consacrée à l'étude de telles fonctions.

8.6 INTERACTION CLIENT/SERVEUR

Les échanges client/serveur sont courants au sein d'applications importantes, nous allons détailler ici deux manières de procéder. La première montre comment appeler des services web au sein d'une application et la seconde est basée sur le protocole XMPP.

8.6.1 Services web

Tout d'abord, voyons ce que sont les services web et en quoi ils nous seront d'une grande utilité dans notre projet.

Services web

Les services web (ou *Web Services* en anglais) sont des fonctionnalités exécutables à distance pouvant être développés dans divers langages comme Java, .NET, etc. Ces services peuvent être appelés à partir d'un langage différent de celui qui a été utilisé pour réaliser le service. Par exemple, il est possible d'appeler un service codé en Java au sein d'un code PHP.

Les fonctionnalités fournies par ces services (*cf.* figure 8.12) peuvent consister à interagir avec une base de données (1) ou à fournir des informations au programme appelant le service (2).

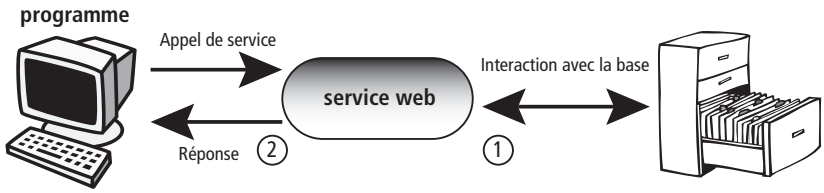

Figure 8.12 — Schéma fonctionnel d'un service web

Examinons un exemple concret pour mieux comprendre. Considérons un programme dont le but est de connaître le temps et la température dans une ville. Admettons que Météo France mette à disposition une WSDL avec une méthode `getTempsEtTemperature(String ville)`. Ici, le programme va appeler le service afin d'obtenir ces informations. Ce service a pour but de récupérer les informations relatives à la ville (certainement dans une base de données) et de les transmettre au programme appelant. Cette réponse se fait sous forme d'un fichier XML. Le programme devra donc parser ce fichier pour récupérer les informations nécessaires.

Nous allons supposer qu'une WSDL (*Web Services Description Language*, ou langage de description de services) est disponible à l'adresse http://mon-exemple-web-services/wsdl.WSDL, celle-ci proposant la méthode `ajouterCoordonnee(int latitude, int longitude)`. Nous allons utiliser également la bibliothèque kSoap2 disponible à l'adresse http://code.google.com/p/ksoap2-android/downloads/detail?name=ksoap2-android-assembly-2.4-jar-with-dependencies.jar, qui facilitera l'appel aux services web ainsi que la façon de parser la réponse d'un appel de service.

Dans notre cas, le service web va s'avérer très utile. Il va permettre de récupérer la dernière position connue par le GPS mais aussi d'insérer des données dans une base. Nous allons donc voir dans cette section comme appeler un service web et dans un second temps lire la réponse SOAP contenant des coordonnées.

Appel du service

L'appel d'un service est simple. Pour cela, nous allons créer une classe `AppelMethode` qui s'occupera de cette fonction. Le code de cette classe sera le suivant :

```
public class AppelService {

  private static final String NAMESPACE = "http://mon-site-web.fr";
  private static final String URL =
                      "http://mon-exemple-web-services/wsdl.WSDL";
  private static final String SOAP_ACTION = "ajouterCoordonnee";
  private static final String METHOD_NAME = "ajouterCoordonnee";
  private void ajouterCoordonnee(int latitude, int longitude) {
    try {
      SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);
      request.addProperty("latitude", latitude);
      request.addProperty("longitude", longitude);
      SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(
               SoapEnvelope.VER11);
      envelope.setOutputSoapObject(request);
      AndroidHttpTransport androidHttpTransport = new AndroidHttpTransport(
                                                        URL);
      androidHttpTransport.call(SOAP_ACTION, envelope);

    } catch (Exception e) {
      Log.e("ajouterCoordonnee", "", e);
    }
  }
}
```

Rien de spécial à retenir dans ce code, il vous faut juste remplacer les constantes par les vôtres, adaptées à votre WSDL et vos noms de méthodes et si nécessaire ajouter des paramètres à votre requête.

Cet appel de service ne nécessitait pas de valeur de retour, nous allons maintenant voir comment utiliser un service renvoyant une valeur.

Parser la réponse

Nous savons maintenant appeler un service ayant pour but d'ajouter une coordonnée dans la base. Néanmoins, il est souvent nécessaire de récupérer des données par un appel de service.

Considérons ici que nous souhaitons connaître la dernière position connue d'un de nos amis. Nous allons garder la même WSDL mais avec une nouvelle méthode `getDernierePositionConnue()`, renvoyant un objet du type `Position` contenant un entier pour la latitude et un entier pour la longitude.

L'appel de service sera toujours aussi simple, mais il faut maintenant parser la réponse qui se présente sous la forme d'un fichier XML.

Note : Les échanges entre le client et le serveur se font au travers de messages SOAP (*Simple Object Access Protocol*) au format XML. Les informations demandées

sont renvoyées dans ce format et il faut donc lire le fichier afin de pouvoir obtenir ces données.

Dans la classe `AppelService`, nous allons ajouter la méthode `getDernierePositionConnue` retournant un objet `Position` dont la classe sera la suivante :

```
public class Position {

  private int latitude;
  private int longitude;
  public Position(int latitude, int longitude) {
    this.latitude = latitude;
    this.longitude = longitude;
  }

  public void setLatitude(int latitude) {
    this.latitude = latitude;
  }

  public int getLatitude() {
    return latitude;
  }

  public void setLongitude(int longitude) {
    this.longitude = longitude;
  }

  public int getLongitude() {
    return longitude;
  }
}
```

La nouvelle méthode `getDernierePositionConnue` se présentera, elle, comme suit :

```
  private Position getDernierePositionConnue() {
    SoapObject positionSoap = null;
    Position position = null;
    try {
      SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME_POSITION);
      SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(
              SoapEnvelope.VER11);
      envelope.setOutputSoapObject(request);
      AndroidHttpTransport androidHttpTransport = new AndroidHttpTransport(
                                                          URL);
      androidHttpTransport.call(SOAP_ACTION_POSITION, envelope);
      positionSoap = (SoapObject)envelope.getResponse();
      position = this.parserReponse(positionSoap);

    } catch (Exception e) {
      Log.e("getDernierePositionConnue", "", e);
    }
    return position;
  }
```

Elle fonctionne de la même manière que la méthode `ajouterCoordonnees` sauf que dans cette méthode, nous faisons appel à la méthode `parserReponse` prenant en paramètre un objet `SoapObject` correspondant au fichier XML de réponse. Afin de récupérer la dernière position connue, il faut parser ce fichier, ce qui est fait dans cette méthode `parserReponse`.

Le fichier XML de réponse se présente de cette façon :

```
<position>
  <latitude>48512899</latitude>
  <longitude>2173944</longitude>
</position>
```

Il faut donc récupérer la propriété racine position et à partir de celle-ci, accéder aux propriétés latitude et longitude pour en obtenir les valeurs. Cela est fait au sein de la méthode `parserReponse` :

```
private Position parserReponse(SoapObject positionSoap) {
    SoapObject positionObjet = (SoapObject)positionSoap.getProperty(
                                "position");
    int latitude = Integer.parseInt(positionObjet.getProperty(
                                "latitude").toString());
    int longitude = Integer.parseInt(positionObjet.getProperty(
                                "longitude").toString());
    Position position = new Position(latitude,longitude);
    return position;
  }
```

Un simple appel à la méthode `getProperty` sur un objet `SoapObject` permet de récupérer un autre objet SOAP ou la valeur de la propriété.

Vous savez donc maintenant appeler un service web et parser le résultat.

8.6.2 XMPP

XMPP (*eXtensible Messaging and Presence Protocol*) est un protocole standard ouvert de messagerie instantanée, anciennement appelé Jabber.

Si le protocole est avant tout connu pour ses fonctionnalités d'échange de messages textuels et de notification de présence, comme tout bon *chat* sait le faire, les possibilités de XMPP sont bien plus larges et peuvent servir de base à un véritable MOM (*Message-Oriented Middleware*) en permettant à plusieurs applications d'interagir au travers de messages XMPP uniquement.

Note : Pourquoi XMPP peut-il être utilisé au sein d'Android ? Il est présent au sein de Gmail au travers de son service de *chat* GTalk. Néanmoins, ne faisant pas partie du SDK, il faut utiliser une bibliothèque externe, la bibliothèque Smack. Celle-ci est une implémentation Java du protocole XMPP. Elle a dû cependant être modifiée pour Android puisque souvenez-vous, la machine virtuelle d'Android n'est pas une machine virtuelle classique. La bibliothèque est disponible à l'adresse suivante : ftp://ftp-developpez.com/florentgarin/android/smack.jar.

Maintenant que nous savons ce qu'est XMPP et à quoi il sert. Voyons la partie qui nous intéresse dans notre projet, le code.

XMPP est relié à un compte Gmail et envoie en instantané un message à un destinataire, lui aussi possesseur d'un compte GMail. Dans notre exemple, nous allons envoyer notre position à un ami et coder ce que nous devons faire en cas de réception de la position de ce dernier.

```
public class XMPPPosition{
  private final static String SERVER_HOST = "talk.google.com";
  private final static int SERVER_PORT = 5222;
  private final static String SERVICE_NAME = "gmail.com";
  private final static String LOGIN = "xxxxx.xxxxx@gmail.com";
  private final static String PASSWORD = "xxxxx";

  private XMPPConnection connection;
    public XMPPPosition(Position position, String destinataire) {
    try {
      this.initialiserConnection();
    } catch (XMPPException e) {
      e.printStackTrace();
    }
    String dest = destinataire;
    String text = "latitude : "+position.getLatitude()+"/longitude :
                  "+position.getLongitude();

    Message msg = new Message(dest, Message.Type.chat);
    msg.setBody(text);
    connection.sendPacket(msg);
  }

  private void initialiserConnection() throws XMPPException {
    //Initialisation de la connexion
        ConnectionConfiguration config =
                new ConnectionConfiguration(SERVER_HOST, SERVER_PORT,
                                            SERVICE_NAME);
        connection = new XMPPConnection(config);
        connection.connect();
        connection.login(LOGIN, PASSWORD);
        Presence presence = new Presence(Presence.Type.available);
        connection.sendPacket(presence);
        //enregistrement de l'écouteur de messages
    PacketFilter filter = new MessageTypeFilter(Message.Type.chat);
    connection.addPacketListener(new PacketListener() {
        public void processPacket(Packet packet) {
          Message message = (Message) packet;
          if (message.getBody() != null) {
              Toast.makeText(maps, message, Toast.LENGTH_LONG);
          }
        }
      }, filter);
  }

}
```

La méthode `initialiserConnection()` permet de se connecter à un compte Google. L'objet `Presence` sert à indiquer votre disponibilité, comme sur tout bon *chat*. Le `PacketListener` sert à définir quoi faire dès réception d'un nouveau message. Dans notre cas, lors de la réception d'une position, nous affichons un pop-up indiquant les coordonnées GPS de notre ami.

Les constantes suivantes permettent de se connecter au compte GMail à partir du nom de serveur Google pour GTalk, le port du serveur, le nom du service ainsi que des identifiants de connexion :

```
private final static String SERVER_HOST = "talk.google.com";
private final static int SERVER_PORT = 5222;
private final static String SERVICE_NAME = "gmail.com";
private final static String LOGIN = "xxxxx.xxxxx@gmail.com";
private final static String PASSWORD = "xxxxx";
```

```
Message msg = new Message(dest, Message.Type.chat);
    msg.setBody(text);
    connection.sendPacket(msg);
```

Ce portion de code sert juste à créer le message et à l'envoyer, de manière très simple.

En résumé

Ce chapitre a permis de découvrir l'intégration de cartes Google Maps dans une application Android, de les manipuler et d'échanger des données avec un serveur, *via* des services web ou le protocole XMPP. Vous voici donc prêt à les utiliser pour vos propres applications !

ANNEXES

A

OpenLayers, une alternative à Google Maps

Objectifs

Bien que cet ouvrage soit consacré à Google Maps, nous avons souhaité vous offrir un aperçu des alternatives existantes, et notamment l'une des plus connues d'entre elles, OpenLayers. Cette annexe se veut une simple introduction afin de vous faire découvrir les potentialités de cette bibliothèque.

Ainsi, nous découvrirons l'origine de ce projet, puis nous étudierons les principales classes, ce qui nous permettra ensuite de réaliser notre première carte et enfin d'y ajouter nos propres données.

A.1 À LA DÉCOUVERTE D'OPENLAYERS

A.1.1 L'histoire d'OpenLayers

C'est lors de la conférence Where 2.0 de 2006, que l'idée de créer OpenLayers, une bibliothèque JavaScript *open source* comparable à Google Maps, a été émise par la société Metacarta[1].

1. http://www.metacarta.com/index.htm

Une première version du code est disponible dès juin 2006, suivie deux mois plus tard de la version 2.0. Très rapide au début, la publication de nouvelles versions tend aujourd'hui à se stabiliser à un rythme d'une tous les six mois.

La dernière version disponible lors de l'écriture de cet ouvrage est la 2.9.1. C'est celle-ci qui sera utilisée dans cette présentation.

A.1.2 Les ressources importantes

Avant de passer aux notions concrètes relatives à Openlayers, il nous a semblé intéressant de lister les ressources dont vous aurez besoin lors de votre découverte de cette bibliothèque.

Premièrement, il s'agit bien évidemment de savoir où la trouver. Deux modes d'utilisation sont possibles. En effet, vous pouvez soit accéder directement en ligne à la bibliothèque[1], ce que je vous déconseille, ou sinon la télécharger. Pour cela rien de plus simple, il vous suffit de vous rendre à l'adresse du site[2] où vous trouverez la dernière version au format compressé zip ou au format tar.gz.

Comme indiqué précédemment, il est préférable de télécharger l'intégralité de la bibliothèque. En effet, imaginez que le site d'OpenLayers soit indisponible : avec la version en ligne, vous ne pourriez plus continuer à travailler. De plus, le fait de la télécharger depuis un serveur distant va également ralentir l'exécution de vos pages web. Pour toutes ces raisons, je vous conseille dès à présent de travailler avec l'intégralité de la bibliothèque au sein de votre environnement de développement.

Pour le moment, laissons de côté l'archive d'OpenLayers et rendons-nous sur la page présentant l'interface de programmation[3](API). Vous y trouverez l'ensemble des classes, méthodes et attributs qu'il est possible d'utiliser.

Enfin, si vous souhaitez connaître ce qu'il est possible de faire avec OpenLayers vous trouverez sur le site de la bibliothèque de nombreux exemples[4]. Le code source de ces derniers est consultable, cela pourra donc vous servir lors de vos prochains développements. Le site gis.ibbeck.de[5] propose également une liste de réalisations mais le niveau technique est plus poussé.

A.1.3 OpenLayers comparé à Google Maps

Ces deux bibliothèques étant fortement similaires, vous vous demandez très certainement quelles sont les raisons qui vous pousseront à utiliser l'une ou l'autre. Sans entrer dans les détails, un élément majeur est à mettre en évidence, les conditions d'utilisation[6]. En effet, Google Maps impose certaines contraintes :

1. http://www.openlayers.org/api/OpenLayers.js
2. http://openlayers.org/
3. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers-js.html
4. http://openlayers.org/dev/examples/
5. http://gis.ibbeck.de/ginfo/apps/OLExamples/Index/index.html
6. http://code.google.com/intl/fr/apis/maps/terms.html

- La page doit être librement accessible. Cela implique que vous ne pouvez pas limiter l'accès à la carte à certaines personnes ou encore l'utiliser en intranet.
- Aucune activité commerciale de la carte ne peut être faite sans l'accord de Google ou la souscription d'une licence commerciale.
- Les données que vous publiez peuvent être intégralement utilisées par Google[1].

Note : Bien évidemment, cette question des conditions d'utilisation de Google Maps fait toujours débat. Afin de ne pas s'écarter de notre sujet principal, nous n'entrerons pas plus dans les détails. Néanmoins, nous vous conseillons la lecture de l'article d'Ed Parson[2] un blogueur très connu travaillant pour... Google. Bien qu'il soit écrit en anglais, les exemples et les explications sont faciles à comprendre.

Concernant OpenLayers, celle-ci est sous licence Open Source. De ce fait, les conditions d'utilisation sont beaucoup plus souples. Contrairement à Google Maps, vous pourrez librement l'utiliser au sein de vos projets commerciaux, la redistribuer ou encore l'améliorer. Néanmoins, du fait d'une popularité moindre et d'une API plus complexe, elle est également plus difficile à prendre en main. Elle bénéficie également d'une popularité moindre. C'est pourquoi, au cours des prochains chapitres, nous apprendrons les bases nécessaires afin que vous puissiez créer ensuite vos propres applications.

A.2 CRÉER UNE CARTE

Supplément Web : tous les exemples de ce chapitre se trouvent à l'adresse suivante : http://www.geotribu.net/dunod/ol.

A.2.1 Les classes principales

Avant de commencer la programmation, attardons-nous un peu sur les classes principales de l'interface de programmation (API). Pour commencer, vous aurez besoin uniquement de trois d'entre elles : `map`, `layer` et `control`. À partir de celles-ci, vous pourrez en effet construire et afficher votre première carte. Détaillons-les sommairement :

- `map` : c'est la classe principale de l'API, qui permet de définir tous les éléments relatifs à la carte (projection, unités de mesure, actions réalisables, couche à afficher, etc.). C'est donc tout naturellement le premier objet qui sera instancié dans nos scripts.

1. Voir section 11.1 http://code.google.com/intl/fr/apis/maps/terms.html# section_11_1
2. http://www.edparsons.com/2008/11/who-reads-the-terms-of-service-anyway/

- `layer` : avoir une carte c'est bien, pouvoir y afficher des données c'est mieux ! Comme vous l'avez certainement deviné, la classe `layer`[1] permet d'ajouter des couches cartographiques. Celles-ci peuvent se présenter sous la forme d'objets vecteurs ou d'images. Au total, il est possible d'utiliser plus d'une trentaine de formats de données.
- `control` : c'est à partir de cette classe que nous pourrons interagir avec la carte. Là aussi, plus d'une trentaine de contrôles sont disponibles. Vous pourrez par exemple afficher l'échelle de la carte, intégrer des outils de navigation ou encore permettre l'affichage et le masquage de vos couches.

Ces principales classes étant présentées, passons maintenant aux exemples.

A.2.2 Importer la bibliothèque

La première chose à faire est de créer le squelette de notre page puis d'importer notre bibliothèque. Le langage JavaScript étant interprété par votre navigateur, vous n'avez pas besoin de disposer d'un serveur web (Apache, IIS, etc.) pour réaliser les différents tutoriels de ce chapitre.

> **Note** : Je vous conseille très vivement d'installer un serveur web. En effet, sans celui-ci, vous ne pourrez utiliser aucun langage de script qui s'exécute côté serveur tel que par exemple le langage PHP. Vous serez alors très vite limité dans vos développements.

Revenons à nos moutons et à notre bibliothèque. Que vous disposiez ou non d'un serveur web, commencez par créer un nouveau dossier nommé ol-exemples. C'est dans celui-ci que nous enregistrerons nos différentes réalisations et surtout la bibliothèque OpenLayers (précédemment téléchargée). Récupérer l'archive et décompressez-la. Vous devriez alors avoir le chemin de dossier suivant : openlayers-exemples/OpenLayers-2.9.

Maintenant, ouvrez votre éditeur de texte favori et de la même manière créez un nouveau fichier que vous nommerez example1-carte_basique.html. Celui-ci aura la structure suivante :

```
<html>
  <head>
    <title>OpenLayers Example</title>
    <link rel="stylesheet" href="./OpenLayers-2.9/theme/default/style.css"
                           type="text/css" />
    <style type="text/css">
        #my_map{
            width : 100%;
            height : 100%;
            border : 1px solid black;
```

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Layer-js.html

```
        }
    </style>
    <script src="./OpenLayers-2.9/OpenLayers.js"></script>
    <script type="text/javascript">
        function init(){
            //Notre code viendra ici
        }
    </script>
  </head>
  <body onload="init()">
    <div id="my_map"></div>
  </body>
</html>
```

Ce premier squelette HTML n'a rien de bien compliqué. Nous nous contentons simplement d'importer la bibliothèque OpenLayers, de créer une balise `my_map` et de spécifier que la fonction `init` sera lancée une fois la page chargée. Cela se fait à partir de la balise `body` dans laquelle est spécifié l'argument `onload`.

Nous avons également spécifié grâce à CSS que la balise `my_map` prendra automatiquement toute la place disponible sur la page (`width` et `height` à 100 %). Dans les prochains exercices, vous pourrez vous amuser à changer cette valeur et ainsi voir les modifications (par exemple `width:500px` et `height:500px`).

A.2.3 Développement du script principal

Création de la structure

Comme nous l'avons spécifié précédemment, l'objet central de la bibliothèque est l'objet `Map`. Pour instancier celui-ci, un seul argument est obligatoire : le `div` qui contiendra notre carte. Il est également possible d'ajouter des arguments optionnels. Néanmoins, pour des raisons de simplicité nous n'aborderons pas cela immédiatement. Votre fichier example1-carte_basique.html devrait maintenant ressembler à ceci :

```
<script type="text/javascript">
function init(){
   var map = new OpenLayers.Map('my_map');
}
</script>
```

Ajout d'une couche de données

Si vous souhaitez afficher cet exemple dans votre navigateur, vous n'aurez rien d'autre qu'une page blanche. Décevant non ? Alors, ajoutons immédiatement notre première couche de données ! Toujours à partir de notre code, nous allons maintenant créer un objet `layer` :

```
var metacarta = new OpenLayers.Layer.WMS(
                    "OpenLayers WMS",
                    "http://labs.metacarta.com/wms/vmap0?",
                    {layers: 'basic'}
                );
```

Vous vous demandez certainement à quoi correspondent toutes ces lignes. Contrairement à Google Maps, il est possible d'utiliser différents formats de données. Dans l'exemple ci-dessus nous interrogeons un serveur cartographique en utilisant la norme WMS.

WMS

WMS (*Web Map Service Interface Standard*) est un protocole cartographique défini par l'Open Geospatial Consortium[1]. Il permet, à partir d'une requête normalisée, d'interroger un serveur cartographique qui renvoie alors l'image demandée. L'avantage d'utiliser cette norme est que quel que soit le serveur utilisé, si celui-ci implémente le protocole WMS, la requête sera alors standardisée.

Dans notre cas, nous interrogeons celui de Metacarta. Rappelez-vous, c'est l'entreprise qui a écrit les premières lignes de code d'OpenLayers.

Maintenant, regardons de nouveau ce que donne notre fonction `init` en entier :

```
<script type="text/javascript">
        function init(){
            var map = new OpenLayers.Map('my_map') ;
            var metacarta = new OpenLayers.Layer.WMS(
                    "OpenLayers WMS",
                    "http://labs.metacarta.com/wms/vmap0?",
                    {layers: 'basic'}
                );
            map.addLayer(metacarta);
            map.zoomToMaxExtent();
        }
    </script>
```

Comme vous pouvez le constater, en plus de spécifier la création d'une nouvelle couche (que nous avons nommée WMS) il est nécessaire d'ajouter celle-ci à notre carte. Cela se fait par la méthode `addLayer`. Ensuite, nous spécifions que l'affichage de la carte se fera au maximum de l'étendue possible grâce à la méthode `zoomToMaxExtent`.

Voilà comment en à peine une dizaine de lignes de code, il a été possible d'obtenir une interface similaire à celle présentée figure A.1.

Si l'objet `map` est instancié sans aucun autre argument que le `div` de la carte, OpenLayers ajoute par défaut le contrôle `Navigation`[2] et le contrôle `PanZoom`[3]. Cela explique pourquoi vous pouvez naviguer sur la carte. Nous verrons dans la prochaine section comment enlever tous les contrôles d'une carte, puis comment en ajouter.

1. http://www.opengeospatial.org/
2. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Control/Navigation-js.html
3. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Control/PanZoom-js.html

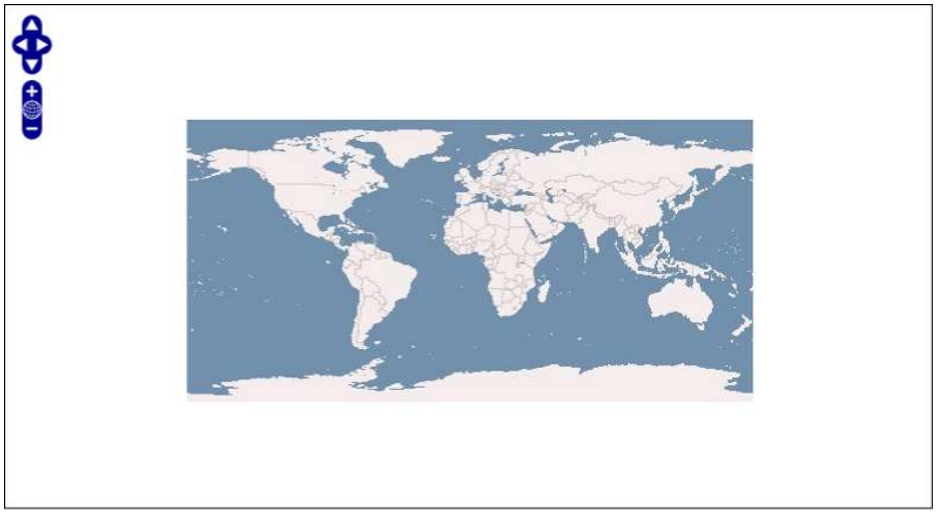

Figure A.1 — Carte basique avec OpenLayers

A.2.4 Interagir avec la carte grâce à la classe *control*

Instancier un objet control

La classe `control`[1] regroupe l'ensemble des interactions qu'il est possible de réaliser sur la carte ou sur les couches. Plus d'une trentaine d'interactions sont possibles, nous utiliserons les plus courantes d'entre elles dans les exemples ci-dessous.

Avant de passer aux exemples, commencez par sauvegarder votre travail précédent puis créez un nouveau fichier nommé example2-controls.html. Celui-ci reprendra la même structure que example1-carte_basique.html. Il suffit donc de copier et coller tout le contenu de celui-ci au sein du nouveau fichier.

Nous sommes maintenant parés. Commençons par étudier les différentes manières d'ajouter des contrôles à une carte. En effet, vous pouvez soit les spécifier lors de l'instanciation de l'objet `map`, ou alors durant le déroulement de votre script.

Dans le premier cas, cela se fait directement lors de la création de l'objet `map` :

```
// Instanciation à partir de l'objet map
var opt = {
    controls:[
          new OpenLayers.Control.Navigation()
    ]
};
var map = new OpenLayers.Map('my_map', opt) ;
```

Mais vous pouvez également les créer plus tard et les ajouter ensuite à la carte :

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Control-js.html

```
// Instanciation au cours du script
var zoomBar = new OpenLayers.Control.PanZoomBar()
map.addControl(zoomBar) ;
```

Ces deux portions de codes auront exactement les mêmes effets. Pour des questions de simplicité, nous utiliserons principalement la première méthode. Cela nous donnera donc le script suivant :

```
var opt = {
    controls:[
        new OpenLayers.Control.Navigation(),
        new OpenLayers.Control.PanZoomBar()
    ]
};
var map = new OpenLayers.Map('my_map', opt);
```

Explorons un peu plus en détail les contrôles qui sont à notre disposition. Bien évidemment, nous ne pourrons pas ici tous les présenter. Néanmoins, vous avez maintenant acquis suffisamment de connaissance pour le faire par vous-même.

La classe LayerSwitcher

Pour le moment nous n'utilisons qu'une seule source de données (Metacarta). Néanmoins, si nous disposions de plusieurs couches, comment pourrions-nous faire pour les afficher et les masquer ? Cela se fait grâce à la classe `LayerSwitcher`[1]. Ajoutons-la immédiatement à notre carte :

```
var opt = {
    controls:[
        new OpenLayers.Control.Navigation(),
        new OpenLayers.Control.PanZoomBar(),
        // Ajout du layer Switcher
        new OpenLayers.Control.LayerSwitcher()
    ]
};
```

Après avoir rafraîchi votre page, vous devriez avoir en haut et à droite de votre écran un onglet bleu cliquable où sont listées la ou les couches de la carte. Avec une seule couche de donnée, l'intérêt est donc limité ! Profitons en alors pour en ajouter une seconde. Cette fois-ci nous interrogerons le serveur WMS de la NASA. Vous verrez, la carte est bien plus jolie. Le code nécessaire est le suivant :

```
//Ajout d'une couche WMS provenant du serveur de la nasa
var nasa = new OpenLayers.Layer.WMS( "NASA WMS",
                "http://wms.jpl.nasa.gov/wms.cgi?",
                {layers: 'BMNG', format: 'image/png'},
                {isBaseLayer: false}
);
```

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Control/LayerSwitcher-js.html

```
//la couche a une opacité à 0.5
nasa.setOpacity(0.5);
 //on ajoute toutes les couches à la carte
map.addLayers([metacarta, nasa]);
```

Détaillons cette nouvelle portion de code. Comme vous pouvez le remarquer, nous avons repris la même structure que la couche « metacarta ». Même s'il a été bien évidemment nécessaire de modifier l'URL du serveur.

Comme, nous souhaitions que cette nouvelle source de données puisse s'afficher en même temps que la première, nous avons spécifié que l'attribut `isBaseLayer` prendrait la valeur `false`.

De plus, pour que nous puissions continuer à voir la couche principale, nous avons diminué l'opacité de notre nouvelle source de données. Pour cela, nous avons utilisé la méthode `setOpacity`. Cette dernière prend en argument un nombre décimal compris entre 0 et 1.

Note : OpenLayers distingue deux types de couches : les calques de base (`base layer`) et les calques simples (`overlay`). Dans le premier cas, il n'est possible d'afficher qu'une couche à la fois. Dans le second cas, les couches se superposent aux calques de base.

Enfin, plutôt que de répéter deux fois la méthode `addLayer` (au singulier), nous préférons utiliser la méthode `addLayers` (au pluriel). Celle-ci prend en argument un tableau contenant les différentes couches.

Avant de clore cette section, effectuons un dernier paramétrage. et essayons de centrer la carte sur l'Europe.

Cherchons dans l'API la méthode qui conviendrait le mieux. Logiquement, cette méthode se situe dans la classe `Map` puisque cette propriété s'applique à la carte en général. Il s'agit en fait de `setCenter`[1] qui prend en argument un objet de type `LonLat`[2] et un niveau de zoom. Revenons à notre code et appliquons les modifications nécessaires :

```
map.addLayers([metacarta, nasa]) ;
var lonlat = new OpenLayers.LonLat(13.798,48.515);
var nivZoom = 4;
map.setCenter(lonlat, nivZoom) ;
```

Comme vous pouvez le constater, l'ancienne méthode `zoomToMaxExtent` a été remplacée par `setCenter`. Une fois votre page rafraîchie, votre carte sera automatiquement centrée sur l'Europe.

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Map-js.html# OpenLayers.Map.setCenter
2. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/BaseTypes/LonLat-js.html

J'imagine déjà la question que vous vous posez ! Comment avoir les coordonnées de la zone que l'on désire ? Est-ce que je me trompe ? Pour cela rien de plus simple, OpenLayers dispose d'un contrôle `MousePosition`[1]. Celui-ci affiche les coordonnées de la carte en fonction de la position de votre souris. Comme vous savez créer de nouveaux contrôles, je vous laisse le soin de l'ajouter à la carte.

La classe OverviewMap

Continuons notre exploration des contrôles proposés par OpenLayers et ajoutons-en un très utile pour se repérer, une minicarte. Il suffit simplement d'ajouter à notre tableau de contrôles, la ligne suivante :

```
new OpenLayers.Control.OverviewMap() ;
```

Rafraîchissez votre page et vous devriez voir apparaître, en bas à droite, un onglet similaire à celui du `LayerSwitcher`. Une fois celui-ci ouvert, une seconde carte à plus petite échelle est disponible.

Si vous avez fait l'effort auparavant d'ajouter le contrôle `MousePosition`, vous avez certainement remarqué que celui-ci est recouvert quand la minicarte est affichée. C'est un peu embêtant et surtout inesthétique. Nous modifierons cela dans la prochaine section.

A.2.5 Modifier les styles par défaut d'OpenLayers

L'un des nombreux avantages d'OpenLayers est que vous êtes libre de créer ce que vous désirez. Pour vous le prouver, nous allons réaliser une interface cartographique composée de deux parties. Dans la première, qui sera la plus grande nous afficherons la carte. Dans la seconde, nous afficherons toutes les informations relatives à celle-ci.

Avant de commencer, tout comme précédemment, créez un nouveau fichier nommé example3-customStyle.html, qui reprend exactement la même structure que example2-controls.

Apportons-y maintenant quelques modifications. Tout d'abord, nous allons réduire légèrement la taille initiale de la carte à 95 %. Ensuite nous allons créer, en dessous de la balise `my_map`, un nouveau `div` ayant pour ID `infoMap` ainsi qu'une balise de type `span` ayant pour identifiant `customMousePosition`. Cela donne le squelette HTML suivant :

```
<body onload="init()">
    <div id="my_map"></div>
    <div id="infoMap">
        Position : <span id="customMousePosition"></span>
    </div>
</body>
```

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Control/MousePosition-js.html

Maintenant, nous allons spécifier que lors de la création du contrôle `mousePosition` l'élément à utiliser pour l'affichage des coordonnées est `customMousePosition`. Cela se fait de la manière suivante :

```
new OpenLayers.Control.MousePosition({div:$('customMousePosition')})
```

Enfin, apportons quelques touches de style en modifiant notre fichier CSS :

```
 #infoMap {
   border : 1px solid black;
   background-color : red;
   width : 100%;
   height : 20px;
}
#customMousePosition{
    color : white;
}
```

Eh voilà, en quelques lignes vous disposez d'une interface complètement personnalisée.

A.2.6 Comprendre la gestion des événements

Nous allons maintenant aborder un aspect particulièrement important d'OpenLayers, la gestion des événements. Ces derniers interviennent lorsque la carte ou un objet de la carte (couche, donnée) a été modifié.

Note : Les événements disponibles pour les différentes classes sont listés au sein de l'API. OpenLayers implémente déjà un très grand nombre d'événements. Uniquement pour la carte, plus d'une quinzaine sont disponibles[1] (changement de zoom, déplacement de la carte, ajout d'une couche, etc.)

Passons immédiatement à la pratique. Imaginons que nous souhaitions afficher le niveau de zoom actuel de la carte. Celui-ci changera à chaque fois que l'utilisateur effectuera un zoom avant ou un zoom arrière.

Pour cela, nous utilisons l'exemple précédent (example3-customStyle), et nous créons un nouveau document nommé example4-events. Ensuite, nous ajoutons un nouvel événement. Cela se fait de la manière suivante :

```
map.events.register(type, obj, listener) ;
```

Cette structure est la même pour tous les événements qui s'appliqueront aux objets de la bibliothèque (carte, couches, etc.). Quatre paramètres sont importants. Tout d'abord l'objet sur lequel nous souhaitons ajouter un événement. Dans notre cas, ce

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Map-js.html# OpenLayers.Map.EVENT_TYPES

sera l'objet `map`. Ensuite, il est nécessaire de définir le type d'événement, c'est-à-dire l'action que vous souhaitez intercepter. Puis, l'objet auquel il fait référence. Et enfin la fonction qui sera appelée chaque fois que l'événement se produit.

Comme nous souhaitons connaître lorsqu'un zoom est effectué sur la carte, nous utiliserons l'événement `zoomend`. En reprenant le code précédent, cela donne donc :

```
map.events.register("zoomend", map, zoomChanged) ;
```

Pour le moment la fonction `zoomChanged` n'existe pas encore. Créons-la immédiatement :

```
function zoomChanged(){
      var zoom = map.getZoom();
      document.getElementById("nivZoom").innerHTML = zoom;
}
```

Celle-ci, dans un premier temps, récupère le niveau de zoom actuel de la carte. Ensuite, l'élément `nivZoom` de la page est mis à jour grâce à la propriété `innerHTML`. `nivZoom` est un élément de type `span` que nous avons ajouté à notre structure HTML :

```
<div id="infoMap">
        Zoom : <span id="nivZoom"></span>
        Position : <span id="customMousePosition"></span>
</div>
```

Il ne reste plus maintenant qu'à apporter une dernière touche afin de finaliser notre application. En effet, en l'état, la fonction `zoomChanged` n'est appelée qu'à partir du moment où l'utilisateur change le niveau de zoom. Ce qui explique pourquoi, au départ, rien n'est affiché dans la barre inférieure. Pour remédier à cela, il suffit d'appeler cette fonction lors de la première exécution du code. Au final, vous devriez obtenir le code suivant :

```
function init(){
  ...
  map.setCenter(lonlat, nivZoom);
  zoomChanged()
 //create events zoom
  function zoomChanged(){
       var zoom = map.getZoom();
       document.getElementById("nivZoom").innerHTML = zoom;
  }
  map.events.register("zoomend", map, zoomChanged);
}
```

Si vous avez correctement effectué les modifications nécessaires, votre interface devrait maintenant ressembler celle présentée figure A.2.

Nous allons maintenant passer à un des aspects les plus intéressants d'OpenLayers, l'ajout de nouvelles sources de données telles que Google Maps, Bing Maps ou encore Yahoo Maps.

Figure A.2 — Personnalisation des contrôles d'OpenLayers

A.3 AJOUTER DES FONDS DE CARTES PROVENANT D'AUTRES PRODUCTEURS DE DONNÉES

A.3.1 Créer une carte avec un fond Google Maps, Bing Maps ou Yahoo Maps

Nous repartons de la structure de notre exemple numéro 1 (*cf.* section A.2) et nous nous contentons simplement d'ajouter de nouvelles couches.

En effet, ne serait-il pas agréable de pouvoir utiliser les données de producteurs tels que Google, Bing ou encore Yahoo ? Cela est tout à fait possible avec OpenLayers. Et comme toujours, cela se fait en à peine quelques lignes de code.

La première étape consiste à importer les bibliothèques des différentes API propriétaires, à savoir Google Maps[1], Yahoo Maps[2] et Bing Maps[3]:

```
<script src='http://maps.google.com/maps?file=api&amp;v=3'></script>
<script
src='http://dev.virtualearth.net/mapcontrol/mapcontrol.ashx?v=6.1'></script>
<script src="http://api.maps.yahoo.com/ajaxymap?v=3.0&appid=euzuro-
openlayers"></script>
<script src="/ol-exemples/OpenLayers-2.9/OpenLayers.js"></script>
```

Maintenant que nous disposons de tous les éléments nécessaires, nous pouvons nous attaquer au code. OpenLayers dispose, pour chacune de ces couches, d'une classe permettant d'afficher les données. Il nous suffit alors de les instancier de la manière suivante :

1. http://maps.google.fr/
2. http://maps.yahoo.com/
3. http://www.microsoft.com/maps/

```
function init(){
  ...
  var google = new OpenLayers.Layer.Google( "Google Hybrid" , {type:
G_HYBRID_MAP });
   var ve = new OpenLayers.Layer.VirtualEarth( "VE");
   var yahoo = new OpenLayers.Layer.Yahoo( "Yahoo");
   map.addLayers([google, ve, yahoo]) ;
   var lonlat = new OpenLayers.LonLat(13.798,48.515);
   var nivZoom = 4;
   map.setCenter(lonlat, nivZoom) ;
}
```

Bien que l'exemple soit relativement simple, détaillons-le succinctement. Dans un premier temps nous créons les objets `google`, `ve` et `yahoo`. Ces derniers permettent d'accéder aux données de chacun des producteurs. Ensuite, nous les ajoutons à la carte grâce à la méthode `addLayers`. Enfin, comme précédemment, nous effectuons un zoom sur l'Europe. Voici en image ce que vous devriez alors obtenir.

Figure A.3 — Affichage des données provenant de Google, Yahoo et Bing

A.3.2 Créer une carte avec un fond OpenStreetMap

Présentation d'OpenStreetMap

Les trois sources de données que nous venons de présenter (Google, Yahoo et Bing) s'accompagnent également de conditions d'utilisation, qui, dans certaines conditions peuvent devenir problématiques. C'est pourquoi nous vous présentons un projet de cartographie libre et communautaire, OpenStreetMap.

OpenStreetMap

OpenStreetMap est aux cartes ce que Wikipédia est aux encyclopédies. Imaginé en 2004 par Steve Coast, ce projet milite pour une production et une diffusion libres des données géographiques. D'abord limité à la thématique routière, OpenStreetMap

s'est considérablement étoffé jusqu'à proposer par exemple, l'emplacement des boîtes aux lettres, des points d'eau ou encore des toilettes publiques. Contrairement aux producteurs privés, les données sont enregistrées par toute une communauté à laquelle il est simple et facile d'adhérer. Aujourd'hui ce projet compte aujourd'hui plus de 280 000 contributeurs et près de 35,8 millions de kilomètres de route.

Contrairement aux sources de données propriétaires, la licence d'OpenStreetMap est beaucoup moins restrictive. Elle offre donc une plus grande souplesse d'utilisation.

Ajouter le fond de carte OpenStreetMap

Comme précédemment, nous utiliserons comme structure initiale l'exemple de la section A.2.

OpenLayers dispose là aussi d'une classe spécifique permettant l'affichage des données d'OpenStreetMap. Celle-ci n'est pas référencée dans l'API d'OpenLayers car en fait, elle est elle-même une déclinaison de la classe `XYZ`[1]. Mais son utilisation n'est pas différente. Il nous suffira simplement d'instancier celle-ci de la manière suivante :

```
function init(){
  ...
   var osm = new OpenLayers.Layer.OSM(
            "OSM Mapnik",
            "http://tile.openstreetmap.org/${z}/${x}/${y}.png"
   ) ;
   map.addLayers([osm]);
}
```

Comme vous avez pu le constater, si vous rafraîchissez votre page, vous vous retrouvez maintenant centré sur l'Afrique. La raison est simple, OpenStreetMap utilise une projection cartographique différente dont le code est 900913.

Projection cartographique

Une projection cartographique permet de représenter la surface ronde tridimensionnelle du globe terrestre sur une carte plane en deux dimensions. En effet, la terre étant une sphère, le passage à une surface plane entraine des déformations qu'il est nécessaire de corriger en utilisant des formules mathématiques. Celles-ci vont, en fonction de la projection choisie, préserver soit les angles soit la distance. Dans le domaine de la cartographie par Internet, la projection la plus utilisée est Universal Transverse Mercator (UTM) que vous retrouverez souvent abrégé par son code : 4326.

Afin de pallier cela, nous allons modifier la projection initiale de la carte et indiquer les bonnes coordonnées géographiques.

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Layer/XYZ-js.html

Cela nous donne l'occasion d'utiliser la classe `Projection`[1]. Néanmoins, nous aurons auparavant besoin de la bibliothèque JavaScript proj4js. En effet, OpenLayers s'appuie sur celle-ci afin d'effectuer la transformation des coordonnées d'un système de projection à un autre. Proj4js est disponible sur le site http://trac.osgeo.org/proj4js/. La dernière version disponible est la 1.0.1[2], c'est celle-ci que nous utiliserons. Ainsi après l'avoir téléchargée et placée dans votre répertoire de travail ol-exemples, nous allons l'inclure dans notre page :

```
<script src='./ol-exemples/proj4js/lib/proj4js-compressed.js'></script>
<script src="./ol-exemples/OpenLayers-2.9/OpenLayers.js"></script>
```

Nous disposons maintenant des bases nécessaires afin de passer d'un système de projection à un autre. Afin que notre projet fonctionne correctement, nous allons devoir modifier notre code à deux endroits.

Tout d'abord, nous allons ajouter aux options générales de notre carte les trois paramètres suivants : `projection`, `displayProjection` et `unit`. Le premier permet de remplacer la projection initiale de la carte qui est par défaut en 4326. Le second paramètre spécifie quelle sera la projection utilisée par les différents contrôles, celle-ci pouvant être différente de la projection initiale. Enfin, comme son nom l'indique, le paramètre `unit` définit l'unité de la carte. Appliquons immédiatement nos modifications :

```
var opt = {
     controls:[
          new OpenLayers.Control.Navigation(),
          new OpenLayers.Control.PanZoomBar(),
          new OpenLayers.Control.LayerSwitcher(),
          new OpenLayers.Control.MousePosition()
     ],
     projection: new OpenLayers.Projection("EPSG:900913"),
     displayProjection: new OpenLayers.Projection("EPSG:4326"),
     units: "m"
};
var map = new OpenLayers.Map('my_map', opt);
```

Allez, encore une dernière étape et nous touchons au but ! Nous allons maintenant modifier les coordonnées permettant de centrer notre carte sur l'Europe. En effet, n'oubliez que nous travaillons avec la projection 900913 : nos anciennes coordonnées ne sont donc plus bonnes. Nous allons avoir besoin de les adapter. Rassurez-vous, nul besoin d'être mathématicien, car c'est là qu'intervient Proj4js. Grâce à celle-ci et OpenLayers, nous allons, dans la méthode `transform`, spécifier la projection initiale et celle que nous souhaitons obtenir. Les coordonnées seront alors automatiquement transformées. Voici le code nécessaire :

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Projection-js.html
2. http://trac.osgeo.org/proj4js/attachment/wiki/Download/proj4js-1.0.1.zip

```
var point = new OpenLayers.Geometry.Point(13.798,48.515);
new OpenLayers.Projection.transform(
       point,
       new OpenLayers.Projection("EPSG:4326"),
       new OpenLayers.Projection("EPSG:900913")
);
var lonlat = new OpenLayers.LonLat(point.x,point.y);
var nivZoom = 4;
map.setCenter(lonlat, nivZoom) ;
```

N'hésitez pas à zoomer sur Paris, vous verrez qu'OpenStreetMap n'a rien à envier aux fournisseurs privés. De plus, vous pourrez l'utiliser à votre guise. En effet, pour l'affichage des données, seule la mention de la source est nécessaire.

Figure A.4 — Affichage des données d'OpenStreetMap

A.4 AJOUTER SES PROPRES DONNÉES

A.4.1 Ajouter une couche vectorielle

Maintenant que nous avons manipulé des données provenant de producteurs extérieurs, vous souhaitez certainement connaître comment créer les vôtres. OpenLayers dispose pour cela d'une classe spécifique, la classe `Feature`[1]. Schématiquement, il est nécessaire de comprendre qu'une *feature* n'est qu'une entité. Elle se rattache ensuite à une couche qui sera de type `vector`[2]. Passons immédiatement au code, vous y verrez bien plus clair...

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Feature-js.html
2. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Layer/Vector-js.html

Note : Dans cet exemple, nous allons simplement afficher un point sur la carte. Mais sachez qu'il est possible d'afficher des géométries plus complexes. Plus d'une dizaine de géométries sont disponibles (rectangle, courbe, ligne, etc.). Vous pouvez également, si vous le souhaitez, modifier le style par défaut et afficher par exemple, à la place du point, une image.

Pour cela, créez un nouveau fichier nommé example7-vector. Nous reprendrons en partie le code de l'exemple 6 (OpenStreetMap). Maintenant, ajoutons nos propres données. Nous allons commencer par créer une nouvelle couche qui sera de type `vector`. Cela se fait de la manière suivante :

```
var vector = new OpenLayers.Layer.Vector("Capitale");
map.addLayers([vector]);
```

Pour le moment, cette couche ne contient aucune donnée. Nous allons donc en créer une nouvelle grâce à la classe `vector`. Un seul argument est obligatoire, la géométrie de l'objet que vous souhaitez créer. Dans notre cas, il s'agira d'un point ayant pour coordonnées Paris. Cela donne donc :

```
var pointFeat = transformProj(new OpenLayers.Geometry.Point(2.351, 48.856));
var feat = new OpenLayers.Feature.Vector(pointFeat, {"name":"paris"});
vector.addFeatures([feat]);
```

Vous devez certainement vous demander à quoi sert cette fonction `transformProj`. En fait, elle évite d'avoir à écrire chaque fois la même transformation. Celle-ci est codée de la manière suivante :

```
function transformProj(pointProj){
      p = new OpenLayers.Projection.transform(
                pointProj,
                new OpenLayers.Projection("EPSG:4326"),
                new OpenLayers.Projection("EPSG:900913")
      );
      return p
}
```

Nous faisons de même pour les paramètres permettant de zoomer la carte sur Paris :

```
var point = transformProj(new OpenLayers.Geometry.Point(2.362,48.8285));
var lonlat = new OpenLayers.LonLat(point.x,point.y);
var nivZoom = 7;
```

Eh voilà, vous savez maintenant comment créer des objets géographiques. Avant de clore cette présentation d'OpenLayers, nous allons réaliser un dernier exemple permettant d'afficher une infobulle lorsque l'utilisateur clique sur le point représentant Paris.

A.4.2 Afficher des infobulles

Toujours à partir de l'exemple précédent, créez un nouveau fichier nommé example8-infobulle.html. Pour afficher une infobulle, OpenLayers dispose de la classe `Popup`[1]. Différents types d'infobulles sont disponibles. Nous utiliserons la classe `FramedCloud`. Celle-ci prend plusieurs arguments : l'identifiant de l'infobulle, sa position géographique, sa taille, son contenu (qui peut être en HTML), un objet auquel il pourra s'accrocher, l'affichage ou non d'un bouton permettant de fermer l'infobulle et enfin une fonction qui sera déclenchée à sa fermeture.

On obtient donc le code suivant :

```
map.setCenter(lonlat, nivZoom);
popup = new OpenLayers.Popup.FramedCloud(
     "Paris",
            new OpenLayers.LonLat(feat.geometry.x, feat.geometry.y),
            null, feat.attributes.name,
            null,false, null
);
map.addPopup(popup);
```

La figure A.5 montre le résultat produit par ce code.

Figure A.5 — Affichage d'une infobulle

1. http://dev.openlayers.org/releases/OpenLayers-2.9.1/doc/apidocs/files/OpenLayers/Popup-js.html

En résumé

Cette annexe nous a donné l'occasion de découvrir les principes généraux de la bibliothèque OpenLayers. Bien évidemment, nous avons dû nous limiter car elle pourrait faire l'objet d'un ouvrage entier... Néanmoins, nous espérons vous avoir apporté un peu de diversité dans l'univers de la cartographie web.

B

API statiques

Objectifs

Cette annexe présente l'API (version 2) permettant de produire des cartes statiques par le moyen de simples requêtes HTTP. Elle détaille les paramètres entrant en jeu dans la création de ces requêtes et montre des exemples de réalisation.

B.1 INTRODUCTION

Les API cartographiques statiques (ou *Static Maps API* dans la terminologie Google) permettent de générer des cartes à l'aide de simples URL, envoyées à un serveur Google, et décrivant toutes les caractéristiques souhaitées pour ces cartes : localisation, format, zoom, type, etc. Elles sont doublement « statiques » :

- aucune programmation JavaScript n'est requise, et donc aucune exécution dynamique de code ;
- les cartes produites sont de simples images, et sont donc privées de tous les outils cartographiques Google Maps (réglage du zoom, choix du type de carte, etc.).

Les cartes statiques répondent aux besoins suivants :

- le principal, inclure une carte bitmap dans une page web ;
- accessoirement, produire une carte bitmap pour tout autre usage (privé, car l'utilisation d'une telle carte en dehors du Web n'est pas autorisée par Google).

La figure B.1 montre un exemple d'intégration d'une carte statique dans une page web. Elle se fait à l'aide d'une balise `<IMG>` dont l'attribut `SRC` est valué avec la requête HTTP exprimant la carte souhaitée :

```
<IMG
SRC="http://maps.google.com/maps/api/staticmap?sensor=false&maptype=roadmap
  &center=1+place+de+la+Bastille,Paris&zoom=16&size=300x300&format=png"
  BORDER=2 ALT="carte">
```

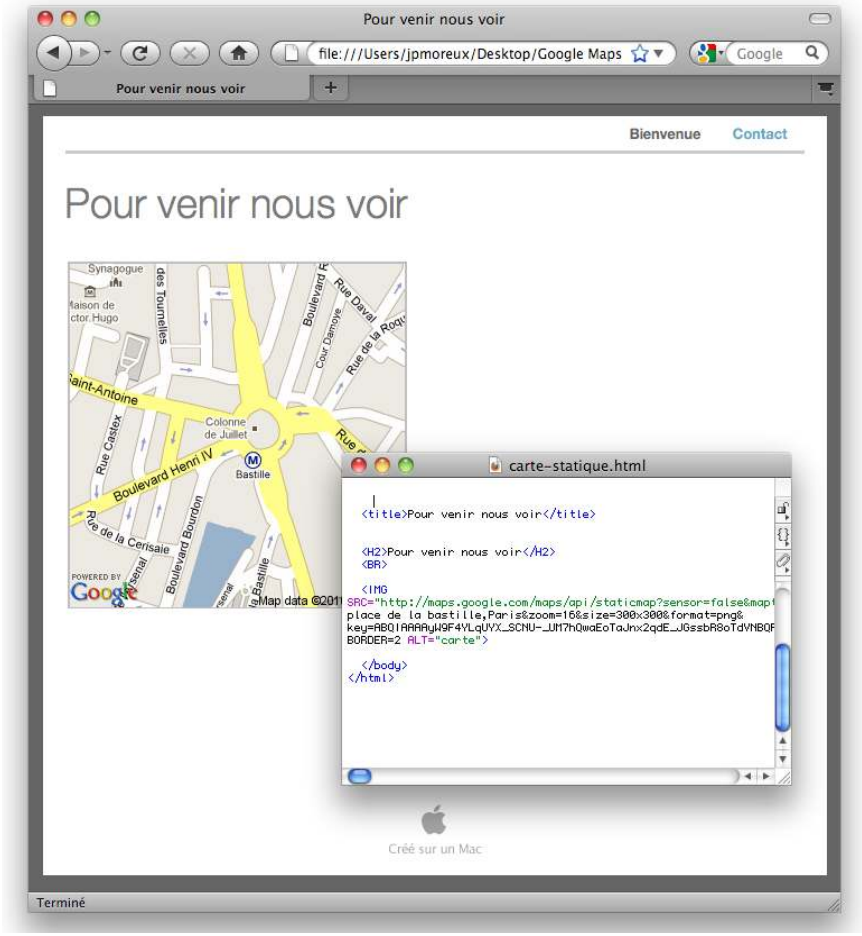

Figure B.1 — Une carte statique incluse dans une page web

La même URL peut être ouverte directement dans une page vierge d'un navigateur web, ce qui conduira bien sûr à l'affichage de la même carte (figure B.2).

```
http://maps.google.com/maps/api/staticmap?sensor=false&maptype=roadmap
  &center=1+place+de+la+Bastille,Paris&zoom=16&size=300x300&format=png
```

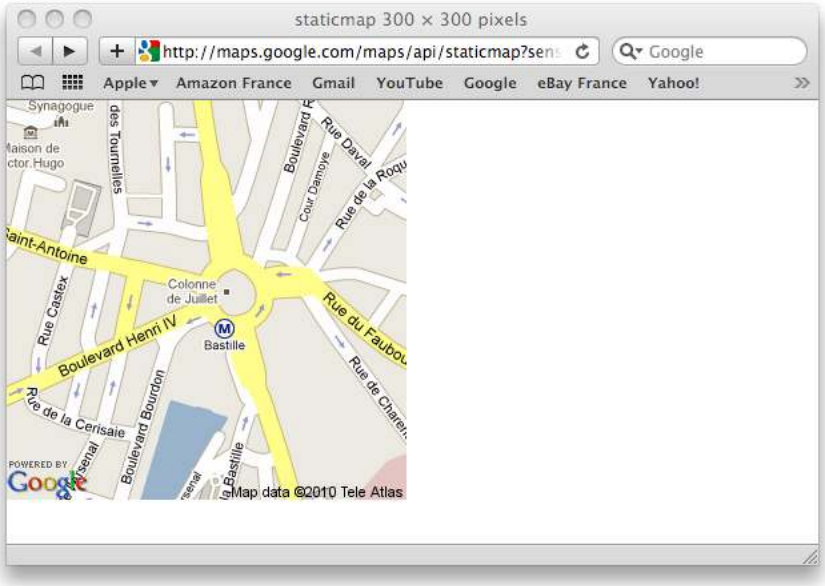

Figure B.2 — Une carte statique affichée dans un navigateur

Limitations

La première limite imposée par Google concerne la fréquence d'utilisation du service, qui est de 1 000 requêtes par jour et par utilisateur. Les images identiques ne sont pas comptabilisées, seule la première requête pour une image donnée l'est. Si cette limite est atteinte, la requête renvoie une icône d'alerte et non la carte demandée.

La deuxième limitation concerne la longueur de la requête, qui est fixée à 2 048 caractères.

D'autres limitations techniques concernent notamment les formats d'image utilisés (GIF, JPEG et PNG uniquement) et la taille de l'image (640 x 640 pixels maximum).

B.2 CONSTRUCTION D'UNE REQUÊTE

B.2.1 Syntaxe

Toute URL soumise au service de cartes statiques doit respecter la syntaxe suivante :

```
http://maps.google.com/maps/api/staticmap?paramètres
```

où `paramètres` est une liste d'étiquettes de paramètre et de leurs valeurs spécifiant la carte désirée. Ils sont séparés par le caractère `&` :

```
http://maps.google.com/maps/api/staticmap?paramètre=valeur&paramètre=valeur...
```

- Certains paramètres sont requis, d'autres sont optionnels, comme nous le verrons dans les sections suivantes. En cas de mauvaise expression de la requête, la carte n'est pas générée et un message d'erreur est renvoyé et affiché dans le navigateur (si la requête a été émise depuis une fenêtre de navigateur). Ce message peut prendre plusieurs aspects, selon la nature de l'erreur : syntaxe de la requête ou valeur d'un paramètre (figure B.3).

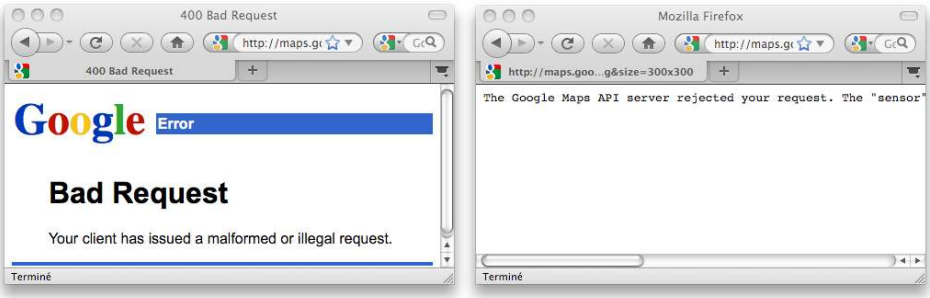

Figure B.3 — Messages d'erreur de syntaxe

Note : Google propose un service en ligne permettant de construire une requête interactivement, à l'aide d'une page web proposant des champs pour valuer la plupart des paramètres. La carte et la requête associée sont automatiquement mises à jour, en fonction des choix de l'utilisateur. Ce service est disponible à l'adresse suivante : http://gmaps-samples.googlecode.com/svn/trunk/simplewizard/makestatic map.html.

B.2.2 Paramètre de *reporting*

Ce paramètre définit le contexte dans lequel le service de cartes statiques Google Maps est utilisé :

- `sensor` (requis) indique si l'application utilise ou non un senseur de localisation, tel un GPS, afin de déterminer la localisation de l'utilisateur de l'application. Ce paramètre est notamment utile dans le cas d'applications dédiées aux périphériques mobiles (voir la section 2.2.2). Il est de type booléen et prend donc pour valeur `true` ou `false`.

API Static Maps Version 2

La version 1 de l'API incluait un autre paramètre, `key` qui permettait de spécifier la clé du domaine web duquel étaient émises les requêtes. Ce paramètre n'est plus nécessaire dans la version 2. Seuls les utilisateurs ayant souscrit à un service « Google Maps API Premier » doivent utiliser les paramètres `client` et `signature` afin de s'authentifier, en leur donnant pour valeur l'ID client et la signature cryptographique fournis par Google :

```
http://maps.google.com/maps/api/staticmap?center=Paris&zoom=12&size=400x300
  &sensor=false&client=gme-votre_ID_client&signature=votre_signature
```

Les anciennes requêtes V1 incluant ce paramètre `key` continuent néanmoins de fonctionner avec la nouvelle version V2.

La page web « Static Maps API v2 Upgrade Guide » (http://code.google.com/intl/fr/apis/maps/documentation/staticmaps/upgrade.html) décrit les principales différences entre ces deux versions et pourra donc aider à mettre à jour un site utilisant les API originelles (dépréciées depuis août 2009).

B.2.3 Paramètres de localisation

Les paramètres de localisation sont au nombre de deux. Ils définissent le contenu de la carte :

- `center` (requis) permet de spécifier le centre de la carte, en fournissant soit des coordonnées géographiques, sous la forme d'une paire de valeurs numériques (latitude, longitude), soit une adresse explicite, en toutes lettres ;
- `zoom` (requis) définit le niveau de zoom avec lequel sera générée la carte.

La documentation Google Maps indique que ces deux paramètres sont obligatoires, mais il apparaît qu'il n'en est rien. Ainsi, la requête minimale suivante :

```
http://maps.google.com/maps/api/staticmap?sensor=false&size=300x300
```

affiche la totalité de la planisphère, au plus faible niveau de zoom.

Centre

Un point quelconque du monde est identifié de manière unique par une *localisation géocodée*, qui, on l'a vu, peut être exprimée en termes de coordonnées géographiques ou d'adresses.

Dans le premier cas, il faut fournir une paire de nombres réels exprimant le positionnement d'un point *via* :

- sa latitude, dans la direction nord-sud, mesurée par rapport au plan de l'équateur, où elle vaut 0° (et donc 90° au pôle nord et –90° au pôle sud) ;
- sa longitude, dans la direction est-ouest, qui est une mesure angulaire prise par rapport à un méridien de référence, celui de Greenwich. Elle a pour étendue l'intervalle [180°,–180°].

Ces coordonnées doivent être exprimées en notation décimale (et non en degrés sexagésimaux), sous la forme d'une paire de nombres réels séparés par une virgule, la partie décimale ayant une précision maximale de six chiffres :

```
latitude.dddddd,longitude.dddddd
```

À titre d'exemple, les coordonnées de Paris sont :

- 48° 51′ 24″ Nord
- 2° 21′ 07″ Est

Converties en notation décimale, on obtient :

- 48 + 51/60 + 24/3600 = 48,856667
- 2 + 21/60 + 7/3600 = 2,351944

L'URL pour obtenir une carte centrée sur Paris sera donc :

```
http://maps.google.com/maps/api/staticmap?sensor=false
  &center=48.856667,2.351944&...
```

Attention : Donner des valeurs hors limites aux paramètres latitude et longitude ne conduit pas à l'affichage d'un message d'erreur.

Dans le cas des adresses, il suffit de fournir toute adresse valide, identifiant une adresse urbaine, un lieu remarquable, une région, etc. Cette adresse sera convertie par Google Maps en ses coordonnées géographiques, par le moyen d'un service de géocodage (lequel peut être utilisé pour tester une adresse donnée : http://gmaps-samples.googlecode.com/svn/trunk/geocoder/singlegeocode.html).

La syntaxe générale à respecter est la suivante :

```
adresse,ville,pays
```

Notons que l'adresse doit être encodée en vue d'une utilisation sous forme d'URL, en remplaçant les éventuelles espaces présentes dans les toponymes par des caractères + :

```
La+Bourboule
```

Fournir une adresse inconnue ou erronée conduira à l'affichage d'une carte centrée sur un toponyme phonétiquement approchant ou au pire, sur le point de coordonnées 0.0,0.0.

Zoom

Ce paramètre a pour valeur un nombre entier, dans l'intervalle [0,21], 0 correspondant au zoom le plus faible (le monde peut être vu sur la carte, voir figure B.4) et 21 le zoom

Figure B.4 — Le monde affiché au zoom 0

le plus fort. L'augmentation du niveau de zoom d'un pas correspond au doublement de l'échelle pixel/terrain, dans les deux directions.

Attention : L'homogénéité des informations portées sur les cartes à l'échelle de la Terre, pour un niveau de zoom donné, n'est pas garantie. En effet, elle dépend de la granularité des données modélisées et de la topographie spécifique à chaque ville. La figure suivante montre Paris et Tokyo (zoom 15). On constate, par exemple, que les informations de circulation routière sont traitées différemment.

Figure B.5 — Zoom et granularité des informations

De plus, certains niveaux de zoom ne sont pas définis pour certaines localisations. Une requête portant sur de tels couples localisation-zoom aura pour résultat une carte vide.
Enfin, donner une valeur négative conduit à une carte de zoom 0 et une valeur supérieure à 21 une carte de zoom 21.

B.2.4 Paramètres de la carte

Les paramètres de la carte en définissent les caractéristiques de présentation. Ils concernent notamment son apparence :

- `size` (requis) permet de spécifier les dimensions de la carte, exprimée en pixels, sous la forme d'une paire d'entiers (largeur × hauteur) ;
- `format` (optionnel) définit le format d'image utilisé pour générer la carte bitmap, le format par défaut étant PNG ;
- `maptype` (optionnel) définit le type de carte, parmi les différents types disponibles : carte routière, satellite, mixte et terrain ;
- `mobile` (optionnel) indique si la carte sera visualisée sur un périphérique mobile ou non ;
- `language` (optionnel) définit la langue utilisée pour afficher les toponymes sur la carte.

Taille

Le paramètre `size` définit la taille de la carte produite, exprimée en pixels, sous la forme d'une paire de valeur `largeurxhauteur` :

```
http://maps.google.com/maps/api/staticmap?sensor=false&size=300x300...
```

La taille maximale autorisée est un carré de 640 × 640 pixels, et la taille minimale 1 × 1.

Attention : Fournir une dimension nulle ou négative conduira à l'affiche d'un message d'erreur ; mais si la dimension est supérieure à la taille maximale, cette dernière sera utilisée.

Format d'image

Le paramètre `format` indique le format d'image utilisé pour générer la carte bitmap, choisi parmi les formats bitmap suivants :

- `png8` ou `png` (par défaut) : le format PNG 8 bits (256 couleurs) ;
- `png32` : le format PNG 32 bits (plus de 4 millions de couleurs) ;
- `jpg` : le format JPEG progressif, ou « interlacé », qui permet un affichage en plusieurs passes, à mesure que les données de l'image sont téléchargées par un navigateur web ;

- `jpg-baseline` : le format JPEG non progressif, qui est plus largement supporté que sa variante progressive ; il sera notamment préféré si la carte n'est pas destinée à être utilisée sur le Web ;
- `gif` : le format GIF.

Les formats PNG et GIF offrent une compression sans perte de qualité. *A contrario*, les formats JPEG et JPEG progressif offrent des tailles de fichier inférieures, au prix d'une perte de qualité.

Attention : Il n'est pas possible de spécifier le paramètre de compression du format JPEG. À l'usage, on constate une réelle perte de qualité, comparée à un format sans perte tel le GIF (voir figure B.6). Pour l'exemple fourni, la taille d'une carte de 400 × 400 pixels est de 76 Ko en GIF, 52 Ko en JPEG et 68 Ko en PNG 8 bits.

Figure B.6 — Qualité d'image des formats GIF (à gauche) et JPEG (à droite)

Le nombre de couleurs autorisé par les formats GIF et PNG 8 bits est largement suffisant pour les cartes routières. Pour les cartes satellitaires (voir section suivante), on pourra éventuellement lui préférer un format 32 bits.

Types de carte

Le paramètre `maptype` indique le type de carte souhaité, choisi parmi les formats suivants :

- `roadmap` (par défaut) : une carte routière classique, de celles qui sont fournies par le site Google Maps ;
- `satellite` : une carte obtenue par imagerie satellitaire ;
- `terrain` : une carte fournissant des informations de relief, d'hydrographie, de végétation, ainsi que les principaux axes routiers ;
- `hybrid` : une carte combinant informations routières et imagerie satellitaire.

La figure B.7 montre ces derniers trois types de carte, en prenant pour exemple la ville de Paris.

Figure B.7 — Cartes de type satellite, terrain et hybride, de gauche à droite

Périphérique mobile

Une carte destinée à être utilisée sur un périphérique mobile de type *smartphone* peut gagner à être générée avec le paramètre `mobile=true`. Ce faisant, Google garantit qu'elle sera adaptée aux écrans de taille réduite. Il convient cependant de faire des tests, notamment en fonction du type ou de la gamme des périphériques concernés.

Langue

Le paramètre `language` définit la langue utilisée pour afficher les toponymes présents sur la carte. Cette fonctionnalité est disponible pour certaines paires pays/langue, dont la liste n'est pas documentée… Les langues disponibles, ainsi que leurs codes, sont donnés à cette adresse : http://spreadsheets.google.com/pub?key=p9pdwsai2hDMsLkXsoM05KQ&gid=1. Des tests pour la France montrent par exemple que le Russe (`ru`) et le Japonais (`ja`) sont opérationnels.

B.2.5 Paramètres de personnalisation

L'API offre plusieurs paramètres (optionnels) permettant de personnaliser une carte :

- `markers` : ajout d'icônes de localisation de points remarquables ;
- `path` : ajout de tracés vectoriels ;
- `visible` : calcul automatique d'emprises.

Marqueurs

Comme on l'a vu au chapitre 3, les marqueurs permettent de placer des symboles sur une carte, à l'aide du paramètre `markers`, qui définit un ensemble d'un ou plusieurs marqueurs doté(s) d'un style particulier (taille, couleur, légende, icône). La syntaxe est la suivante :

```
markers=style|localisation|localisation...
```

Les localisations sont séparées à l'aide du caractère barre verticale | (ou pipe Unix).

Attention : La définition de style doit être placée en premier ; elle est ensuite suivie de la localisation du ou des marqueurs. Si l'on souhaite placer des marqueurs de différents styles, il faut écrire autant de paramètres `markers` que de styles.

Les styles de marqueurs sont créés à l'aide de plusieurs descripteurs, également séparés par le caractère |, et définissant leurs attributs visuels :

- `size` (optionnel) : la taille du ou des marqueur(s), choisie entre `tiny`, `small` et `mid` (par ordre de taille croissante). La valeur par défaut est une taille légèrement supérieure à `mid` ;
- `color` (optionnel) : la couleur du ou des marqueur(s), choisie parmi les couleurs prédéfinies {`black`, `brown`, `green`, `purple`, `yellow`, `blue`, `gray`, `orange`, `red`, `white`} ou spécifiée sous la forme d'une valeur 24 bits exprimant les composantes RVB de la couleur (par exemple 0x00FF00 pour un vert pur). La valeur par défaut est `red` ;
- `label` (optionnel) : la légende du ou des marqueur(s), constituée d'un seul caractère alphanumérique choisi parmi {A-Z, 0-9}. En l'absence du paramètre `label`, ou si la légende n'est pas un caractère alphanumérique, un point noir sera affiché en lieu et place de la légende.

Attention : Seules les tailles par défaut et `mid` permettent d'afficher une légende, et les caractères minuscules ne sont pas autorisés.

Ces descripteurs doivent être séparés de leurs valeurs par un caractère deux-points :

```
label:B
```

Les localisations sont exprimées en termes de coordonnées ou d'adresses (voir la section B.1.1). Ainsi, un marqueur est créé sur la Bastille à l'aide de la requête suivante :

```
http://maps.google.com/maps/api/staticmap?sensor=true&center=paris&zoom=13
  &size=400x400&markers=label:B|place+de+la+bastille
```

Une localisation par coordonnées géographiques est bien sûr possible. La requête suivante produira la même carte :

```
http://maps.google.com/maps/api/staticmap?sensor=true&center=paris&zoom=13
  &size=400x400&markers=label:B|48.852878,2.36944848.852878,2.369448
```

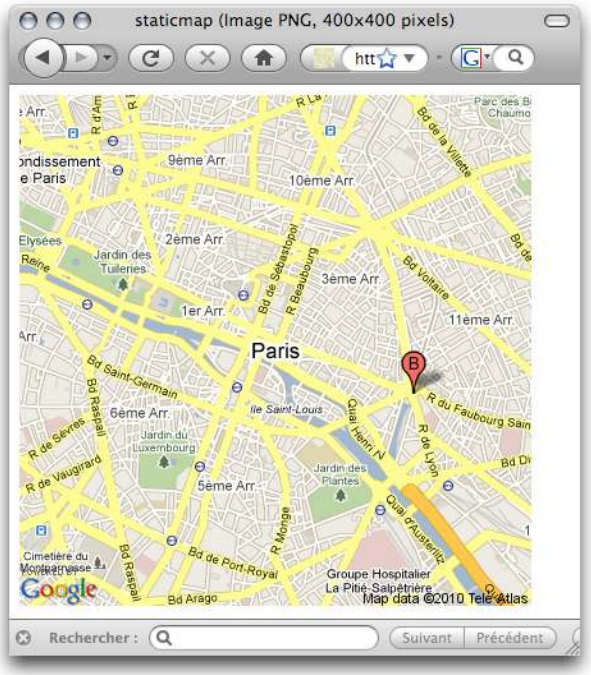

Figure B.8 — Création d'un marqueur avec légende

Attention : Si la localisation d'un marqueur n'est pas incluse dans l'emprise de la carte (telle que spécifiée avec les paramètres `center` et `zoom`), le marqueur ne sera bien sûr pas affiché. Par contre, si ces paramètres `center` et `zoom` sont omis, la carte sera centrée sur le marqueur (figure B.9) :

```
http://maps.google.com/maps/api/staticmap?sensor=true&size=400x400
 &markers=label:B|place+de+la+bastille
```

Dans le cas de plusieurs marqueurs (dans cet exemple, localisés place de la Bastille et sur l'opéra Garnier), l'emprise de la carte sera automatiquement choisie afin de les contenir tous (figure B.10).

```
http://maps.google.com/maps/api/staticmap?sensor=true&size=400x400
  &markers=label:B|place+de+la+bastille&markers=label:O
  |opera+garnier
```

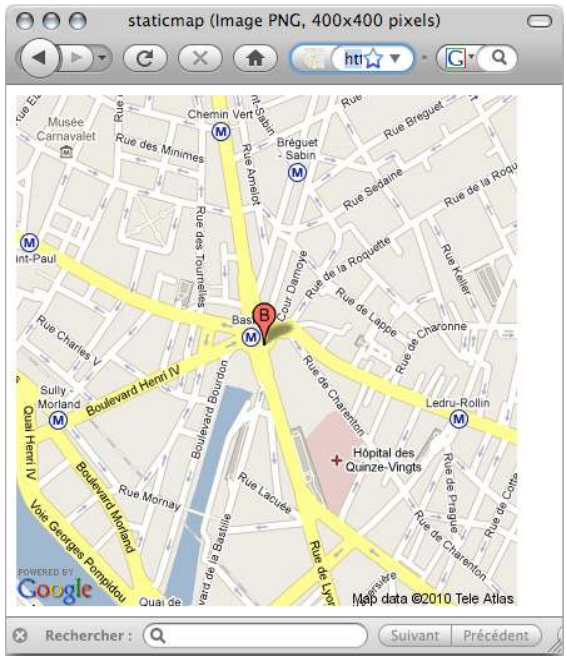

Figure B.9 — Carte automatiquement centrée sur le marqueur

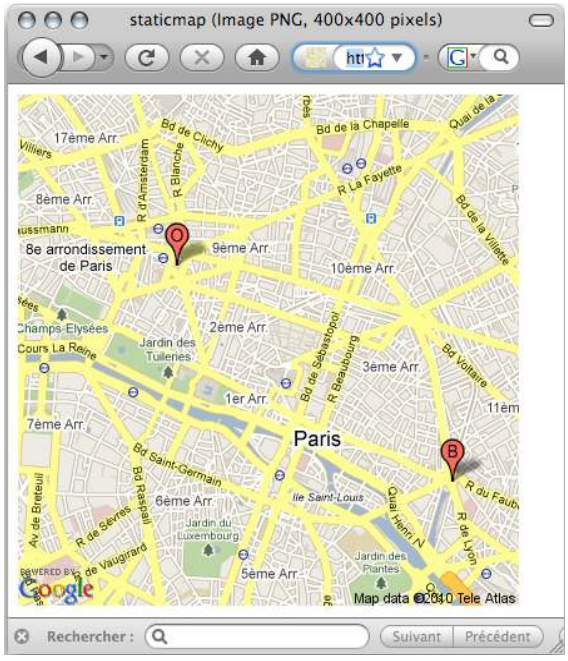

Figure B.10 — Carte automatiquement centrée sur plusieurs marqueurs

Icônes personnalisée

Il est possible de demander que les marqueurs soient affichés à l'aide d'une icône particulière, dont le fichier doit être fourni à l'aide du descripteur `icon`, valué avec une URL (encodée) pointant vers un fichier graphique au format GIF, JPEG ou PNG. En poursuivant l'exemple précédent, voici la requête permettant d'afficher les deux marqueurs à l'aide du fichier `icone.png` stocké sur le site www.monsite.fr (figure B.11) :

```
http://maps.google.com/maps/api/staticmap?sensor=true&size=400x400
  &markers=icon:http://www.monsite.fr/icone.png|place+de+la+bastille
  |opera+garnier
```

Cette fonctionnalité est limitée à des icônes de taille maximale de 4 096 pixels (64 × 64) et à cinq icônes *différentes* par requête (une même icône peut servir pour plusieurs marqueurs).

Figure B.11 — Marqueurs dotés d'une icône personnalisée

Le descripteur `shadow` permet d'indiquer si l'on souhaite ou non qu'une ombre portée soit ajoutée à l'icône. Il vaut `true` par défaut, comme on peut le constater sur la figure B.11.

Le service « Google Chart API » (http://groups.google.com/group/google-chart-api/web/chart-types-for-map-pins?pli=1) propose des icônes personnalisables. Ainsi, la requête suivante génère une icône de type « gare ferroviaire » sur fond blanc (voir figure B.12) :

```
http://chart.apis.google.com/chart?chst=d_map_pin_icon&chld=train|FFFFFF
```

De nombreuses icônes prédéfinies sont disponibles : parking, restaurant, aéroport, poste, etc.

Attention, les caractères spéciaux (`&` et `=`) de cette URL doivent être encodés et le caractère `|` doit être double-encodé avant d'être utilisé au sein d'une requête, puisque ce caractère a également un rôle dans la syntaxe du paramètre `markers`, comme on l'a vu :

```
http://maps.google.com/maps/api/staticmap?sensor=true&size=400x400
  &markers=icon:http://chart.apis.google.com/chart%3Fchst
  %3Dd_map_pin_icon%26chld=train%257CFFFFFF|gare+austerlitz
```

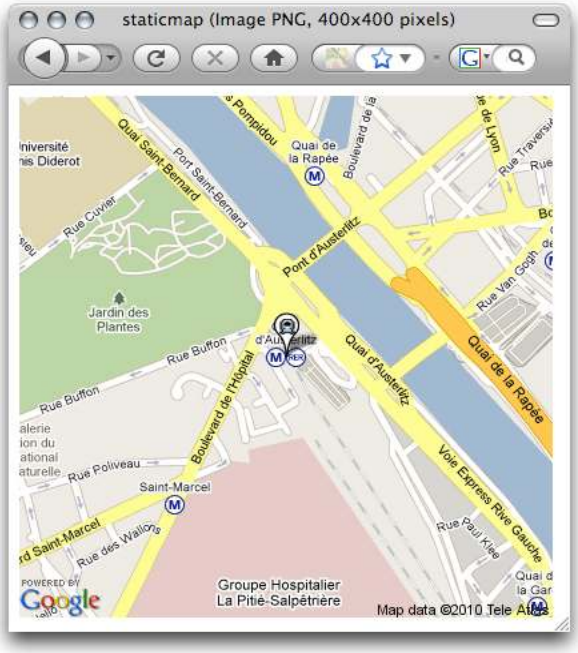

Figure B.12 — Icône de marqueur créée avec le service Google Chart

Chemins

Un chemin (*path*) permet de définir un tracé vectoriel sur une carte, au moyen d'un descripteur de style et d'un ensemble de points constitutifs du tracé. La syntaxe est la suivante :

```
path=style|localisation|localisation...
```

Les points sont séparés à l'aide du caractère barre verticale |, comme pour les marqueurs. De même, la définition de style doit être placée en premier puis être suivie de la définition du ou des points.

Les styles de chemins sont créés à l'aide de plusieurs descripteurs optionnels, séparés par le caractère |, et définissant leurs attributs visuels :

- `weight` (optionnel) : l'épaisseur du tracé, exprimée en pixels. La valeur par défaut est 5 ;
- `color` (optionnel) : la couleur du tracé, choisie parmi les couleurs prédéfinies {`black`, `brown`, `green`, `purple`, `yellow`, `blue`, `gray`, `orange`, `red`, `white`}, ou spécifiée sous la forme d'une valeur 24 bits ou 32 bits exprimant les composantes RVB de la couleur, ainsi que sa transparence dans le cas d'une valeur 32 bits (les deux derniers chiffres exprimant alors le degré de transparence, de total, `00`, à opaque, `FF`) ;
- `fillcolor` (optionnel) : la couleur de remplissage du tracé (qu'il n'est pas nécessaire de fermer).

La figure B.13 montre un exemple de chemin élémentaire, sans définition de style, puis un exemple plus complexe, avec un tracé d'épaisseur nulle et rempli d'une couleur semi-opaque.

```
http://maps.google.com/maps/api/staticmap?sensor=true&zoom=15&size=400x400
  &path=place+de+la+bastille,paris|metro+ledru+rollin,paris
http://maps.google.com/maps/api/staticmap?sensor=true&zoom=15&size=400x400
  &center=cathedrale+notre+dame,paris&path=color:0x000000|weight:0|fillcolor:
  0xFF000066|quai+de+horloge,paris|place+du+pont+neuf,paris|quai+des+orfevres,
  paris|quai+du+marche+neuf,paris|pont+au+double,paris|memorial+martyrs,paris|
  quai+aux+fleurs,paris|1+pont+arcole,paris|quai+de+la+corse,paris
```

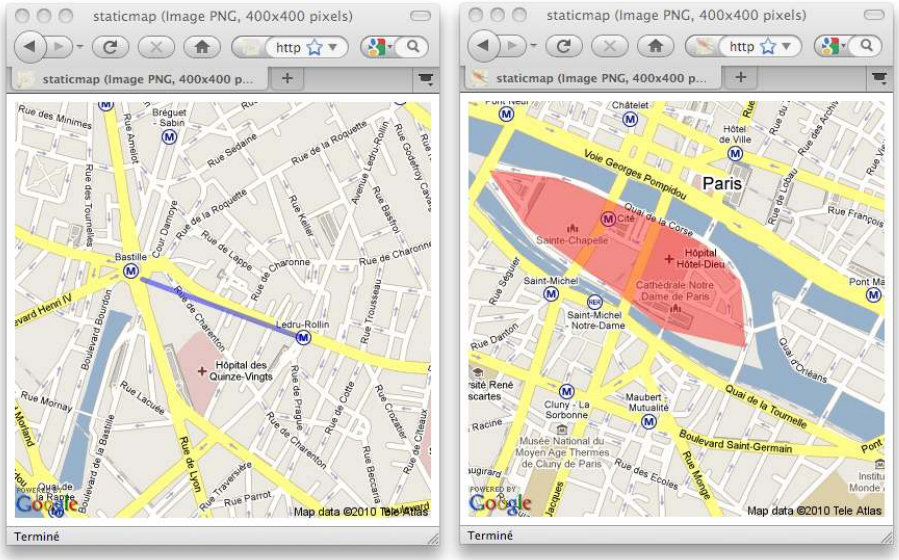

Figure B.13 — Exemples de chemins

Attention : De même que pour les marqueurs, si la localisation d'un chemin n'est pas incluse dans l'emprise de la carte (telle que spécifiée avec les paramètres `center` et `zoom`), le chemin ne sera bien sûr pas affiché. Par contre, si ces paramètres `center` et/ou `zoom` sont omis, la carte sera centrée sur le chemin.

Un chemin composé d'un seul point ne sera bien sûr pas affiché...

Les points définissant un tracé peuvent être fournis sous trois formes :

- des adresses, comme dans les exemples précédents ;
- des coordonnées géographiques (voir section B.1.1.1) ;
- des polylignes encodées. Dans ce cas, il suffit de faire précéder les données de la polyligne par le préfixe `enc` :

```
path=color:0x0000FF|weight:2|enc:données
```

L'utilitaire Interactive Polyline Encoder Utility (http://code.google.com/intl/fr/apis/maps/documentation/utilities/polylineutility.html) permet de définir interactivement de telles polylignes. L'exemple suivant reprend celui de la figure B.13, le contour de l'île de la Cité étant cette fois codé sous la forme d'une polyligne à l'aide de l'utilitaire (voir figure B.14). Il suffit alors de copier-coller le résultat fourni dans le champ « Encoded polyline » après le préfixe `enc` :

```
http://maps.google.com/maps/api/staticmap?sensor=true&zoom=15&size=400x400
  &center=sainte-
chapelle,paris&path=color:0x000000|weight:0|fillcolor:0xFF000066|enc:ooeiHq{...
```

Emprise

Le paramètre `visible` permet de définir l'emprise d'une carte en indiquant des localisations que l'on souhaite voir apparaître sur la carte. Les paramètres `center` et `zoom` ne sont alors plus nécessaires, puisque l'emprise est automatiquement calculée. Une marge est ajoutée autour des localisations.

La figure B.15 montre une carte centrée sur Paris et affichant également Caen et Troyes, puis une carte non centrée affichant Rennes et Nantes :

```
http://maps.google.com/maps/api/staticmap?center=paris&visible=caen|troyes
  &size=400x400&sensor=false
http://maps.google.com/maps/api/staticmap?visible=nantes|rennes
  &size=400x400&sensor=false
```

Attention : Comme on l'a vu aux sections et B.1.1.8 et B.1.1.9, l'utilisation de marqueurs ou de chemins en l'absence de paramètres `center` et `zoom` déclenche le même processus de calcul automatique de l'emprise.

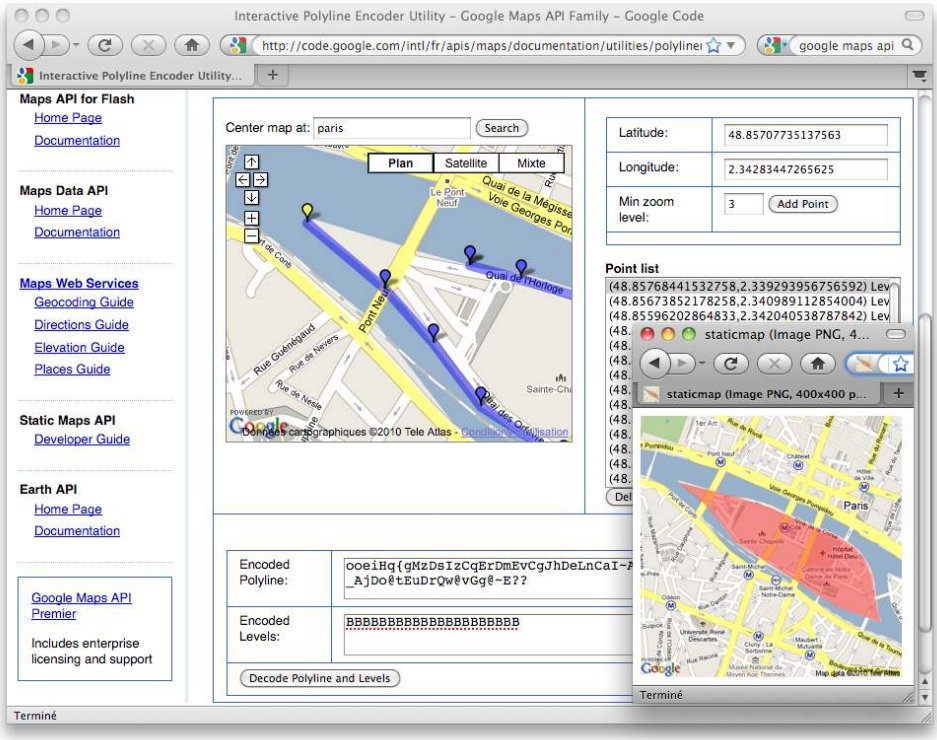

Figure B.14 — Un exemple de polyligne encodée

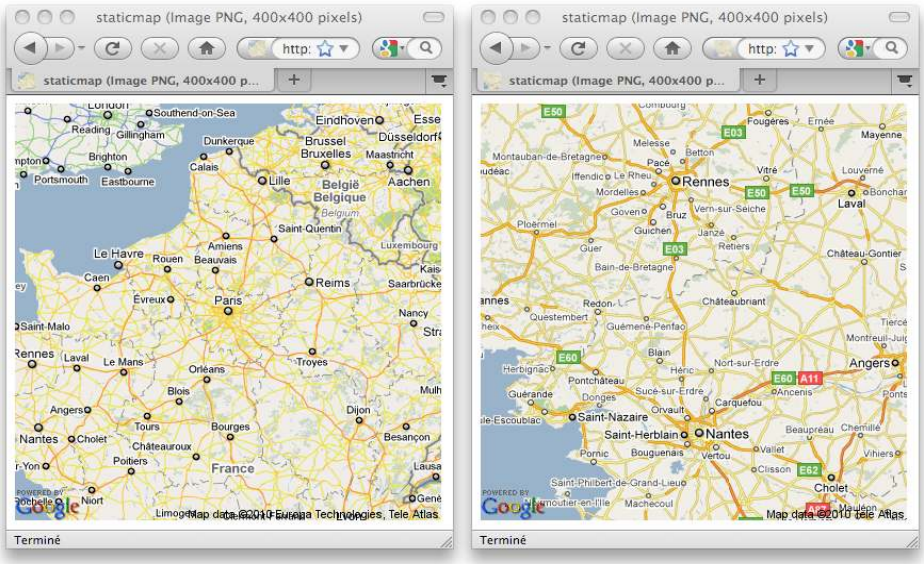

Figure B.15 — Emprises automatiques

B.3 EXEMPLES D'UTILISATION

Les API statiques, de par leur souplesse, permettent de créer des cartes sur mesure, répondant à un besoin particulier. Le premier exemple montre la création d'un plan d'accès, qui pourra par exemple être joint à des courriers électroniques afin d'aider des visiteurs à se rendre à une adresse. Le deuxième exemple utilise un langage de script automatisant la génération d'une série de cartes, paramétrée selon divers critères.

B.3.1 Création d'un plan d'accès

Le plan d'accès présente les moyens de transport les plus proches, ainsi que l'adresse elle-même (figure B.16).

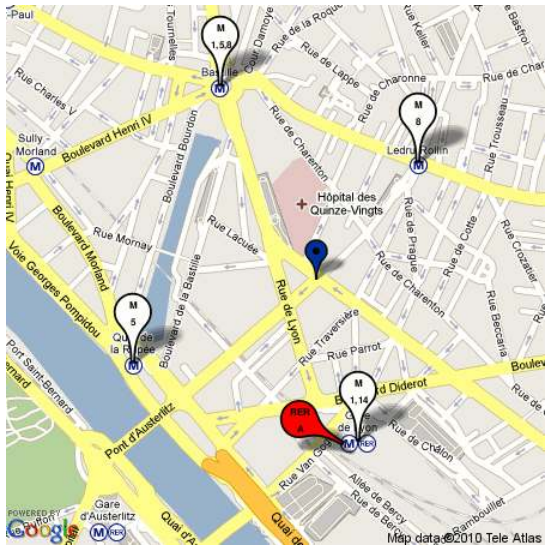

Figure B.16 — Le plan d'accès avec ses six marqueurs

La requête est construite autour de :

- quatre marqueurs utilisant les API du service Google Chart localisent les métros les plus proches. Ces marqueurs sont créés selon la syntaxe suivante :

```
http://chart.apis.google.com/chart?chst=d_map_spin&chld=échelle|rotation
  |couleur|corps du texte|gras|texte|texte|...
```

Ainsi, un marqueur sur fond blanc, réduit de 10 %, incluant une légende sur deux lignes composées en gras avec un corps de 8 points est créé à l'aide de :

```
http://chart.apis.google.com/chart?chst=d_map_spin&chld=0.9|0|FFFFFF|8|b|M|1,5,8
```

- un marqueur construit à l'aide du même service, servant à localiser une station RER. La couleur de fond est rouge (`FF0000`), et une rotation lui est appliquée afin qu'il ne recouvre pas le marqueur blanc localisé sur la même gare :

```
http://chart.apis.google.com/chart?chst=d_map_spin&chld=0.9|60|FF0000|8|b|RER|A
```

- un marqueur classique est utilisé pour localiser l'adresse, le service Google Chart étant limité à cinq icônes différentes :

```
color:0x003399|paris,68+av+ledru+rollin
```

Finalement, la requête complète est constituée par la concaténation des paramètres `markers` (en prenant soin de les encoder au préalable) :

```
http://maps.google.com/maps/api/staticmap?sensor=true&size=450x450&zoom=15
  &markers=color:0x003399|paris,68+av+ledru+rollin
  &markers=icon:http://chart.apis.google.com/chart%3Fchst%3Dd_map_spin%26chld
  %3D0.9%257C0%257CFFFFFF%257C8%257Cb%257CM%257C1,5,8|paris,metro+bastille
  &markers=icon:http://chart.apis.google.com/chart%3Fchst%3Dd_map_spin%26chld
  %3D0.9%257C0%257CFFFFFF%257C8%257Cb%257CM%257C8|paris,metro+ledru+rollin
  &markers=icon:http://chart.apis.google.com/chart%3Fchst%3Dd_map_spin%26chld
  %3D0.9%257C0%257CFFFFFF%257C8%257Cb%257CM%257C1,14|paris,gare+de+lyon
  &markers=icon:http://chart.apis.google.com/chart%3Fchst%3Dd_map_spin%26chld
  %3D0.9%257C0%257CFFFFFF%257C8%257Cb%257CM%257C5|paris,metro+quai+de+la+rapee
  &markers=icon:http://chart.apis.google.com/chart%3Fchst%3Dd_map_spin%26chld
  %3D0.9%257C61%257CFF0000%257C8%257Cb%257CRER%257CA|paris,28+rue+van+gogh
```

Comme on l'a vu à la section B.1.1.8, la carte est automatiquement centrée afin de faire apparaître l'ensemble des marqueurs spécifiés. Seuls le facteur de zoom et la taille de la carte sont précisés.

Cette carte peut ensuite être capturée dans le navigateur puis insérée dans une signature de courrier électronique. La figure suivante présente l'édition d'une telle signature dans Microsoft Entourage. Elle sera automatiquement ajoutée à la fin de tous vos courriers électroniques et donc envoyée à vos correspondants (attention, il faut veiller à composer les courriers en mode HTML pour que l'image soit expédiée avec le courrier lui-même).

B.3.2 Génération automatique de cartes

Les API statiques se prêtent fort bien à être utilisées au sein d'un programme ou d'un script. L'exemple suivant s'appuie sur AppleScript, mais tout autre langage de script permettant de piloter un navigateur web conviendra.

L'objectif est de générer des cartes routières de plusieurs dizaines de capitales du monde entier. Ces cartes doivent servir à tester différents cadrages et échelles (communs à la série de cartes), en vue de réaliser ultérieurement des illustrations basées sur ces cartes Google Maps. Ce processus d'automatisation a servi lors de la phase préparatoire de réalisation de l'exposition « Suspendu » de l'artiste Mona Hatoum (2010, Mac/Val, Vitry-sur-Seine, http://www.macval.fr).

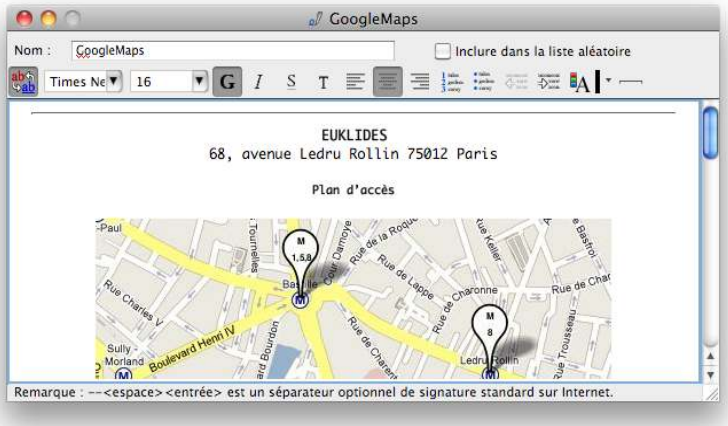

Figure B.17 — La signature avec le plan d'accès

L'intérêt de l'automatisation est donc double :

- aide à la définition des réglages optimaux, en permettant le test de nombreuses configurations des paramètres ;
- génération des cartes finales, une fois les paramètres validés.

La liste des villes est stockée dans un simple fichier texte tabulé, selon le format :

```
Ville centrage de la carte
...
```

Si le champ « centrage » n'est pas renseigné, la carte sera centrée par défaut sur le centre de la ville. Voici un extrait du fichier :

```
Alger gare de Agha
Beyrout
Buenos+Aires
...
```

Pour chaque ligne du fichier, les deux champs sont lus et déformatés puis transmis à une fonction `generer` qui a pour effet de générer la requête HTTP et de l'ouvrir dans le navigateur Safari :

```
on generer(lieu)
  global taille  -- paramètre size
  global zoom    -- paramètre zoom
  global format  -- paramètre format
  global GoogleURL
  if lieu's length = 1 then  -- pas de champ  centrage
    set monURL to GoogleURL & item 1 of lieu
  else
    set monURL to GoogleURL & item 2 of lieu & "," & item 1 of lieu
  end if
  set monURL to monURL & "&zoom=" & zoom
  set monURL to monURL & "&size=" & taille
```

```
    set monURL to monURL & "&format=" & format
    tell application "Safari" to open location monURL
end generer
```

Les variables globales `taille`, `zoom`, et `format` définissent respectivement la taille en pixels des cartes, leur zoom et le format d'image souhaité. La variable `GoogleURL` contient le début de la requête :

```
property GoogleURL : "http://maps.google.com/maps/api/staticmap?sensor=false
  &maptype=roadmap&center="
```

auquel sont ensuite concaténés les différents paramètres de la carte (localisation et autres), à l'aide de l'opérateur AppleScript `&`. Puis la carte est affichée dans le navigateur, à l'aide de la fonctionnalité de pilotage applicatif d'AppleScript :

```
tell application "Safari" to open location monURL
```

Enfin, la carte est capturée, à l'aide de techniques sortant du propos de cet ouvrage, et enregistrée sous le nom de la ville. La fenêtre du navigateur est ensuite fermée et le processus reprend avec une autre ville.

En quelques secondes, ce script génère donc autant d'images (figure B.18) que de villes présentes dans le fichier texte tabulé. Images qu'il est aisé de régénérer en changeant un ou plusieurs paramètres et en exécutant de nouveau le script.

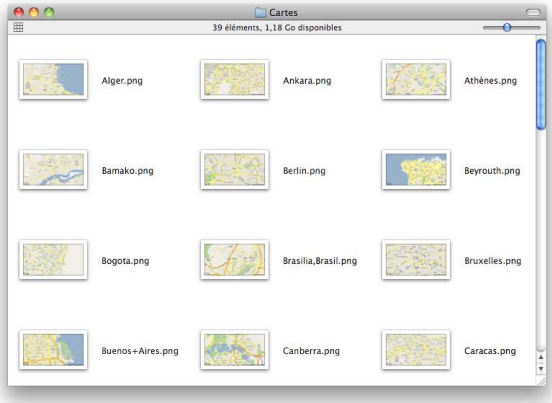

Figure B.18 — La série de cartes

En résumé

Cette annexe a présenté l'API permettant de produire des cartes statiques Google Maps. Elle ne possède pas la puissance et l'interactivité de sa cousine dynamique, mais elle pourra répondre à des besoins particuliers de production de cartes bitmap.

Index

D

E

F

G

H

I

N

O

P

R

S

T

U

V

W

X