Cette étude se concentre uniquement sur la cartographie, l'utilisation du GPS du smartphone et la navigation qui s'en suit. La plupart des projets utilisent uniquement la partie **QML**, puisqu'il s'agit généralement de réglages associés à la vue avec une gestion événementielle adaptée.

Cartographie et géolocalisation

L'un des gros intérêt du smartphone est de pouvoir exploiter pleinement la mobilité avec des applications de géolocalisation avec une cartographie qui suit notre mouvement et nos déplacements. **Quick** dispose d'objets de très haut niveaux qui permettent de résoudre ces attentes de façon intuitive, concise et donc très facile à mettre en œuvre.

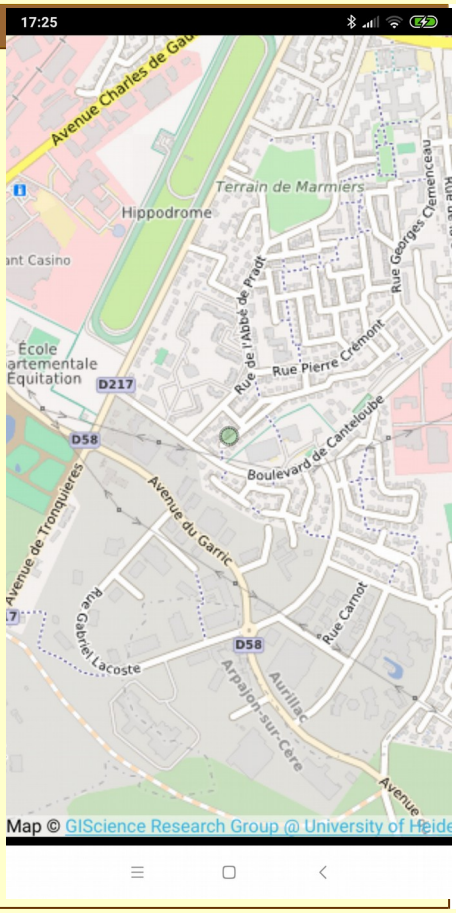
quick-map.pro

```
QT += quick location positioning
CONFIG += c++11

TARGET = Map-Manu

DEFINES += QT_DEPRECATED_WARNINGS

SOURCES += main.cpp
RESOURCES += qml.qrc
# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Vous devez intégrer deux modules supplémentaires à votre projet, « location » qui permet de géolocaliser à l'aide du capteur GPS, ainsi que « positioning » qui permet d'afficher la carte correspondant au lieu géolocalisé.

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtLocation 5.6
import QtPositioning 5.6

Window {
  width: 640
  height: 480
  visible: true

  Plugin {
    id: openstreetmap
    name: "osm" // OpenStreetMap
  }

  Map {
    id: carte
    anchors.fill: parent
    plugin: openstreetmap
    center {
      latitude: 44.92
      longitude: 2.44
    }
    zoomLevel: 16

    MapCircle {
      center: carte.center
      radius: 20
      color: 'green'
      border.width: 3
      opacity: 0.25
    }
  }

  PositionSource {
    updateInterval: 1000
    active: true
    onPositionChanged: carte.center = position.coordinate
  }
}
```



On ne peut pas faire plus court, surtout au regard du résultat. Quatre composants sont utilisés :

- ***Plugin*** *qui permet de choisir le système de cartographie, ici « **OpenStreetMap** »,*
- ***Map*** *qui affiche une carte suivant une longitude et une latitude. Vous pouvez choisir le niveau de précision (niveau de zoom). Sans réglage supplémentaire, vous pouvez tourner votre carte avec deux doigts, changer le facteur de zoom, déplacer la carte en glissant sur les différents côtés, etc.*
- ***MapCircle*** *qui permet d'afficher une puce circulaire juste sur l'endroit géolocalisé.*
- ***PositionSource*** *qui active le GPS et qui permet ainsi la géolocalisation avec une fréquence de rafraîchissement réglable, ici toutes les secondes.*

Ici, la géolocalisation se fait en continu. Il est tout à fait possible de le faire de façon ponctuelle lorsque nous demandons une adresse particulière, par exemple, ce qui sera fait par la suite.

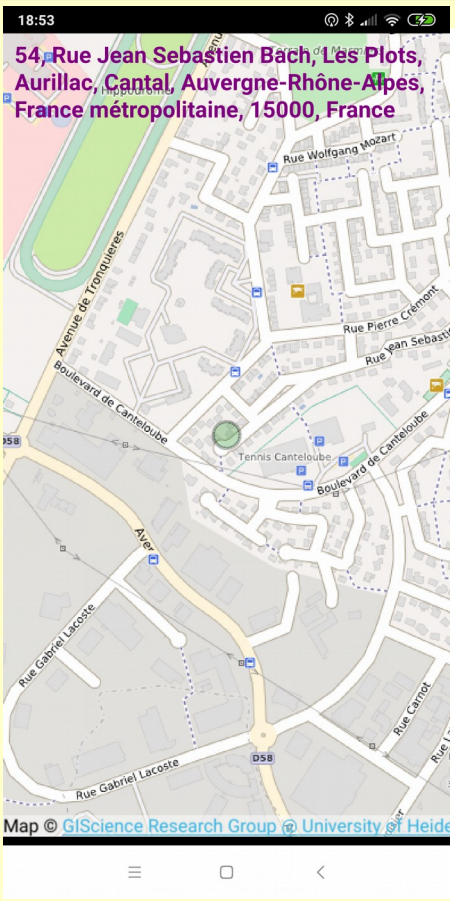
Actuellement, la puce suit le centre de la carte et ne garde pas la position géolocalisée. Nous verrons comment faire ultérieurement.

Retrouver l'adresse exacte

Dans le projet que nous venons de voir, très peux d'objets sont utilisés pour résoudre la géolocalisation avec la carte qui va avec. Un autre objet intéressant, « **GeocodeModel** », permet de connaître l'adresse complète d'une rue d'une ville à partir de ses coordonnées géographiques (l'inverse est aussi possible grâce à cette même classe – ***géocodage inverse***).

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtLocation 5.6
import QtPositioning 5.6

Window {
  width: 640
  height: 480
  visible: true

  Plugin {
    id: openstreetmap
    name: "osm" // OpenStreetMap
  }

  Map {
    id: carte
    anchors.fill: parent
    plugin: openstreetmap
    zoomLevel: 16

    MapCircle {
      center: gps.position.coordinate
      radius: 20
      color: 'green'
      border.width: 3
      opacity: 0.25
    }
  }

  GeocodeModel {
    id: adresse
    plugin: carte.plugin
    onStatusChanged: {
      if (status == GeocodeModel.Error)
        info.text = "Il faut activer le WIFI ou\nLes données mobiles"
    }
    onLocationsChanged: {
      if (count>=1) info.text = get(0).address.text
    }
  }

  PositionSource {
    id: gps
    updateInterval: 1000
    active: true
    onPositionChanged: {
      adresse.query = position.coordinate.latitude+","+position.coordinate.longitude
      adresse.update()
      carte.center = position.coordinate
    }
  }

  Text {
    id: info
    width: parent.width-20
    anchors {
      left: parent.left
      leftMargin: 10
      top: parent.top
      topMargin: 7
    }
    text: "Info"
    font.pixelSize: 20
    font.bold: true
    color: 'purple'
    wrapMode: Text.Wrap
  }
}
```



La gestion dans GeocodeModel est également très concise. Pour connaître tous les renseignements concernant la localisation d'une rue, vous devez passer par Internet, soit par le WIFI en local, soit par l'activation des « données mobiles » que vous pouvez contrôler avec le « slot » de changement de status.

À chaque fois qu'une nouvelle position est enregistrée dans le GeocodeModel, vous pouvez vérifier qu'elle est bien prise en compte « count = 1 », et récupérer les informations grâce à la méthode get(0). À partir de là, vous pouvez sélectionner soit une information de coordonnées GPS (latitude et longitude), soit une adresse.

Dans ce dernier cas, il existe un composant spécialisé Address qui regroupe toutes les propriétés la concernant, une rue « street », une ville « city », un état « country ». Vous pouvez également récupérer l'ensemble de l'adresse complète grâce à la propriété « text », ce qui a été fait ici.

Pour intégrer une nouvelle position dans GeocodeModel, vous devez proposer une nouvelle requête « query », généralement sous forme de texte. Dans notre cas, vous devez d'abord spécifier la latitude et ensuite la longitude, les deux séparées par une virgule. Pour que cette requête soit pris en compte, vous devez rafraîchir le GeocodeModel au moyen de la méthode update().

Géolocaliser à partir d'une adresse

Toujours dans le domaine de la géolocalisation, il peut être intéressant de montrer un endroit précis de la carte en relation avec une adresse saisie au clavier en précisant le nom de la rue avec son numéro ainsi que le nom de la ville. Dans ce contexte, nous n'avons plus besoin du capteur **GPS** et donc du composant **PositionSource**.

Pour cela, nous allons modifier quelques lignes de codes par rapport au projet précédent. L'ossature générale est préservée sans donc la classe PositionSource. Nous avons besoin de deux zones de saisie avec un bouton de soumission.

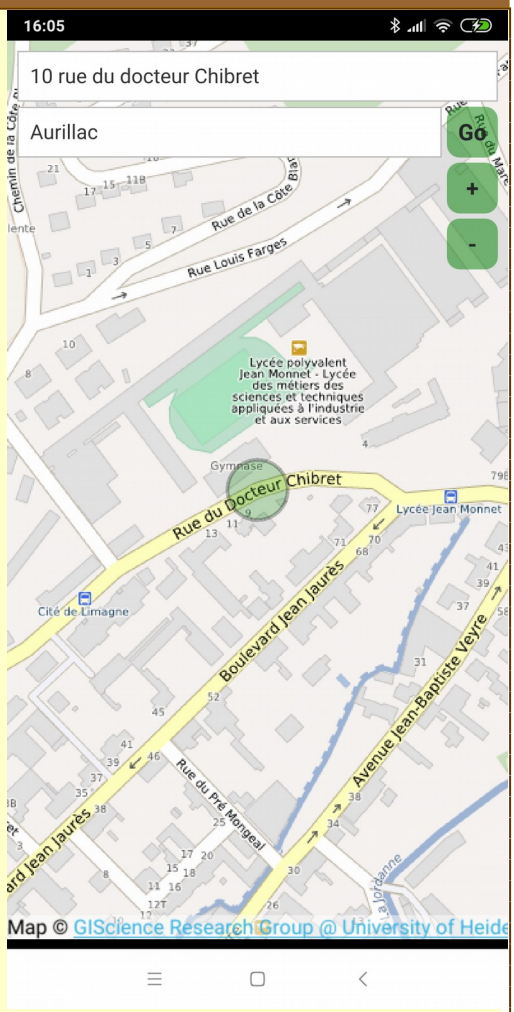
Nous profitons de l'occasion pour rajouter deux boutons supplémentaires qui vont nous permettre de régler le zoom à notre convenance. Tous ces boutons auront la même apparence et seront situés dans la même zone. Dans ce contexte, je propose donc de fabriquer un nouveau composant Bouton avec des réglages communs.

Bouton.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.4

Button {
  anchors {
    topMargin: 4
    right: parent.right
    rightMargin: 8
  }
  height: ville.height
  width: ville.height
  font.bold: true
  background: Rectangle {
    color: "green"
    opacity: 0.5
    radius: 10
  }
}
```

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtLocation 5.6
import QtPositioning 5.6
import QtQuick.Controls 2.4

Window {
  width: 640
  height: 480
  visible: true

  Plugin {
    id: openstreetmap
    name: "osm" // OpenStreetMap
  }

  Map {
    id: carte
    anchors.fill: parent
    plugin: openstreetmap
    center {
      latitude: 44.92
      longitude: 2.44
    }
    zoomLevel: 16

    MapCircle {
      id: puce
      radius: 20
      color: 'green'
      border.width: 3
      opacity: 0.25
    }
  }

  GeocodeModel {
    id: adresse
    plugin: carte.plugin
    onLocationsChanged: {
      if (count>=1) {
        carte.center = get(0).coordinate
        puce.center = get(0).coordinate
      }
    }
  }

  TextField {
    id: rue
    placeholderText: "Rue ou Avenue"
```



```
    anchors {
      top: parent.top
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: parent.right
      rightMargin: 8
    }
  }

  Bouton {
    id: soumettre
    anchors.top: rue.bottom
    text: "Go"
    onClicked: {
      adresse.query = rue.text+","+ville.text+",France"
      adresse.update()
    }
  }

  TextField {
    id: ville
    placeholderText: "Ville"
    anchors {
      top: rue.bottom
      topMargin: 4
      left: parent.left
      leftMargin: 8
      right: soumettre.left
      rightMargin: 4
    }
  }

  Bouton {
    id: plus
    text: '+'
    anchors.top: soumettre.bottom
    onClicked: carte.zoomLevel++
  }

  Bouton {
    id: moins
    text: '-'
    anchors.top: plus.bottom
    onClicked: carte.zoomLevel--
  }
}
```

Tracé d'un itinéraire entre deux points de géolocalisation

Je vous propose de revenir sur le sujet de la cartographie, mais cette fois-ci nous verrons comment tracer un itinéraire entre deux coordonnées **GPS**. Pour tester notre projet, et pour éviter d'avoir trop de lignes de code, nous allons placer directement dans le programme les coordonnées **GPS**. Nous verrons plus tard comment faire un itinéraire variable.

quick-itineraire.pro

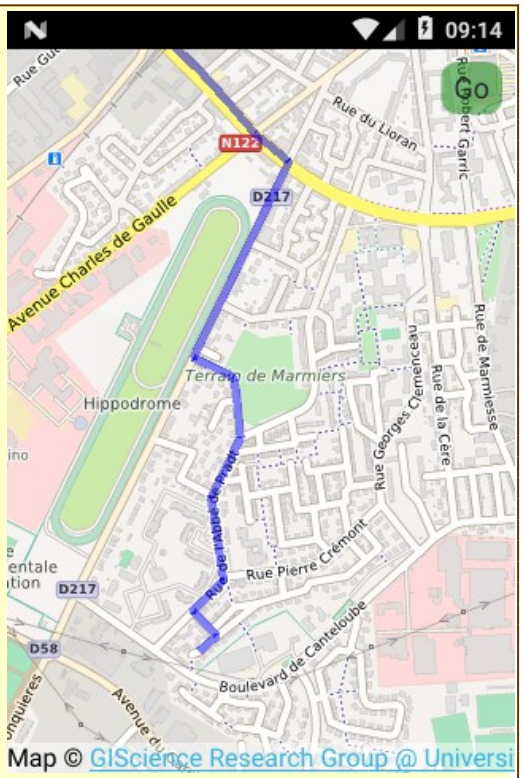
```
QT += quick location positioning
CONFIG += c++11

TARGET = Map-Manu
DEFINES += QT_DEPRECATED_WARNINGS

SOURCES += main.cpp
RESOURCES += qml.qrc
# Default rules for deployment.
qnx: target.path = /tmp/${TARGET}/bin
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Vous devez intégrer deux modules supplémentaires à votre projet, « location » qui permet de géolocaliser à l'aide du capteur GPS, ainsi que « positioning » qui permet d'afficher la carte correspondant au lieu géolocalisé.

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtLocation 5.9
import QtPositioning 5.8
import QtQuick.Controls 2.4

Window {
  visible: true
  width: 320
  height: 480
  title: qsTr("Calcul itinéraire")

  Plugin {
    id: openstreetmap
    name: "osm"
  }
```

```
  Map {
    id: carte
    anchors.fill: parent
    plugin: openstreetmap
    center {
      latitude: 44.92
      longitude: 2.44
    }
    zoomLevel: 15

    MapItemView {
      model: itinéraire
      delegate: MapRoute {
        route: routeData
        line.color: "blue"
        line.width: 5
        smooth: true
        opacity: 0.5
      }
    }
  }

  RouteModel {
    id: itinéraire
    plugin: carte.plugin
    autoUpdate: true
    query: RouteQuery {
      id: choixItinéraire
      travelModes: RouteQuery.CarTravel
      routeOptimizations: RouteQuery.ShortestRoute
    }
  }

  Button {
    anchors {
      right: parent.right
      rightMargin: 8
      top: parent.top
      topMargin: 8
    }
    font.pixelSize: 18
    text: "Go"
    background: Rectangle {
      color: "green"
      opacity: 0.5
      radius: 10
    }
    onClicked: {
      carte.center = QtPositioning.coordinate(44.904088, 2.4307575)
      choixItinéraire.clearWaypoints()
      choixItinéraire.addWaypoint(QtPositioning.coordinate(44.904088, 2.4307575))
      choixItinéraire.addWaypoint(QtPositioning.coordinate(44.9356277, 2.4521777))
      carte.update()
    }
  }
}
```



Nous retrouvons le même principe de fonctionnement que celui correspondant au GeocodeModel. Nous remplaçons ce composant par RouteModel à l'intérieur duquel nous plaçons le composant correspond à la requête demandée, savoir RouteQuery.

Celui-ci possède des réglages possibles, comme le choix du type de déplacement et si le trajet doit se faire au plus court ou au plus vite. Vous pouvez ajouter autant de point GPS que vous désirez. Pensez toutefois à nettoyer la liste des points au départ.

Si vous avez choisi un rafraîchissement automatique de votre modèle, c'est la carte que vous devez mettre à jour afin que le tracé ce réalise automatiquement lors du clic sur le bouton « Go ».

Pour le tracé de l'itinéraire, nous passons par le composant générique MapItemView qui est capable d'afficher une suite d'éléments grâce au modèle que vous proposez. Il suffit ensuite de préciser le type d'affichage que vous souhaitez, ici MapRoute qui est spécialisé sur le tracé d'un itinéraire (nous avons vu précédemment MapCircle pour le tracé de cercles).

Itinéraire en précisant le point de destination

Maintenant que nous connaissons bien le principe de fonctionnement, nous faisons évoluer le projet de telle sorte que nous puissions choisir la destination en précisant l'adresse complète et que l'itinéraire se trace en fonction de la position actuelle que nous donne le **GPS**.

Nous rajoutons également un petit marqueur pour bien visualiser la position à atteindre. Le cercle sera alors utilisé pour montrer la position actuelle. Nous ajoutons tous les composants que nous avons utilisés lors du projet concernant la géolocalisation par rapport à une adresse auquel nous rajoutons un bouton pour activer ou désactiver le GPS.

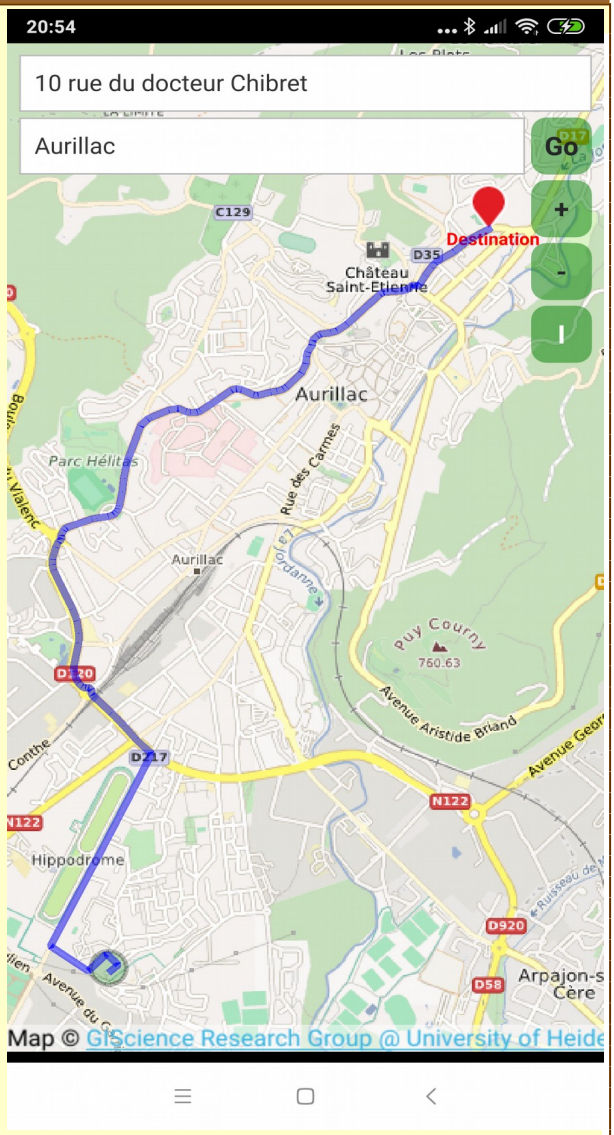
Bouton.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.4

Button {
  anchors {
    topMargin: 4
    right: parent.right
    rightMargin: 8
  }
```

```
  height: ville.height
  width: ville.height
  font.bold: true

  background: Rectangle {
    color: "green"
    opacity: 0.5
    radius: 10
  }
}
```

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtLocation 5.6
import QtPositioning 5.6
import QtQuick.Controls 2.4

Window {
  width: 320
  height: 480
  visible: true
  title: qsTr("Calcul itinéraire")

  Plugin {
    id: openstreetmap
    name: "osm"
  }

  Map {
    id: carte
    anchors.fill: parent
    plugin: openstreetmap
    center {
      latitude: 44.92
      longitude: 2.44
    }
    zoomLevel: 17

    MapCircle {
      id: puce
      radius: 10
      color: 'green'
      border.width: 3
      opacity: 0.25
    }

    MapQuickItem {
      id: marqueur
      anchorPoint {
        x: image.width/2
        y: image.height
      }
      sourceItem: Column {
        Image { id: image; source: "icone/marqueur.png" }
        Text {
          text: "Destination"
          color: "red"
          font.pixelSize: 12
          font.bold: true
          x: -width/4
        }
      }
    }

    MapItemView {
      model: itinéraire
      delegate: MapRoute {
        route: routeData
        line.color: "blue"
        line.width: 5
        smooth: true
        opacity: 0.5
      }
    }
  }

  GeocodeModel {
    id: adresse
    plugin: carte.plugin
    onLocationsChanged: {
      if (count>=1) {
        choixItinéraire.clearWaypoints()
        choixItinéraire.addWaypoint(gps.position.coordinate)
        choixItinéraire.addWaypoint(get(0).coordinate)
        marqueur.coordinate = get(0).coordinate
        carte.update()
      }
    }
  }
```



```
  RouteModel {
    id: itinéraire
    plugin: carte.plugin
    autoUpdate: true
    query: RouteQuery {
      id: choixItinéraire
      travelModes: RouteQuery.CarTravel
      routeOptimizations: RouteQuery.ShortestRoute
    }
  }

  PositionSource {
    id: gps
    updateInterval: 1000
    active: true
    onPositionChanged: {
      carte.center = position.coordinate
      puce.center = position.coordinate
    }
  }

  TextField {
    id: rue
    placeholderText: "Rue ou Avenue"
    anchors {
      top: parent.top
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: parent.right
      rightMargin: 8
    }
  }

  Bouton {
    id: soumettre
    anchors.top: rue.bottom
    text: "Go"
    onClicked: {
      adresse.query = rue.text+","+ville.text+",France"
      adresse.update()
    }
  }

  TextField {
    id: ville
    placeholderText: "Ville"
    anchors {
      top: rue.bottom
      topMargin: 4
      left: parent.left
      leftMargin: 8
      right: soumettre.left
      rightMargin: 4
    }
  }

  Bouton {
    id: plus
    text: '+'
    anchors.top: soumettre.bottom
    onClicked: { carte.zoomLevel++;  puce.radius /= 2 }
  }

  Bouton {
    id: moins
    text: '-'
    anchors.top: plus.bottom
    onClicked: { carte.zoomLevel--;  puce.radius *= 2 }
  }

  Bouton {
    id: gpsactif
    text: 'A'
    anchors.top: moins.bottom
    checkable: true
    onCheckedChanged: {
      if (checked) { text = 'I';  gps.active = false  }
      else { text = 'A';  gps.active = true }
    }
  }
}
```

Il faut bien comprendre que tout ce qui fait parti de la carte comme artifice de visualisation doit impérativement être intégré dans l'objet relatif à Map. C'est le cas de MapCircle qui représente la puce que nous connaissons bien maintenant, mais aussi de MapQuickItem qui représente le marqueur d'arrivée avec une icône (marqueur.png) et un texte associé et enfin de MapItemView qui factorise l'ensemble des segments qui représente l'itinéraire entre la position actuelle et celle choisie par l'utilisateur.

Je le répète, tout se qui doit être visible et qui est associée à une position particulière sur la carte doit être introduite (composition) à l'intérieur d'un objet Map.

Navigation

Toujours à l'aide du projet précédent, nous pourrions aider l'utilisateur en indiquant ce qu'il doit faire à chaque changement de direction en précisant la distance avant d'effectuer le changement et qu'il direction il doit choisir. Comme tout le reste, nous faisons appel au compétence du « **Web Service** » associé à la cartographie choisie, sachant que généralement, la réponse est systématiquement en anglais.

Pour résoudre ce problème, je propose de rajouter un contrôleur qui permettra de générer un texte en français avec les instructions à suivre sur le premier segment de l'itinéraire (celui qui correspond à la prochaine échéance), sachant que ce genre d'information est implémentée par la classe RoutManeuver qui est capable de nous renseigner sur la distance avec le prochain segment, dans combien de temps nous pouvons l'atteindre et quelle direction nous devons choisir.

quick-itineraire.pro

```
QT += quick location positioning
CONFIG += c++11

DEFINES += QT_DEPRECATED_WARNINGS
TARGET = Itineraire
SOURCES += main.cpp navigation.cpp
HEADERS += navigation.h
RESOURCES += qml.qrc

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <navigation.h>
#include <QtQml>

int main(int argc, char *argv[])
{
  QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
  QGuiApplication app(argc, argv);

  Navigation navigation;
  QQmlApplicationEngine engine;
  engine.rootContext()->setContextProperty("nav", &navigation);
  engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

  if (engine.rootObjects().isEmpty()) return -1;
  return app.exec();
}
```

navigation.h

```
#ifndef NAVIGATION_H
#define NAVIGATION_H

#include <Qobject>

class Navigation : public QObject
{
  Q_OBJECT
public:
  explicit Navigation(QObject *parent = nullptr) : QObject(parent) {}
public slots:
  QString direction(int distance, int vers);
};

#endif // NAVIGATION_H
```

navigation.cpp

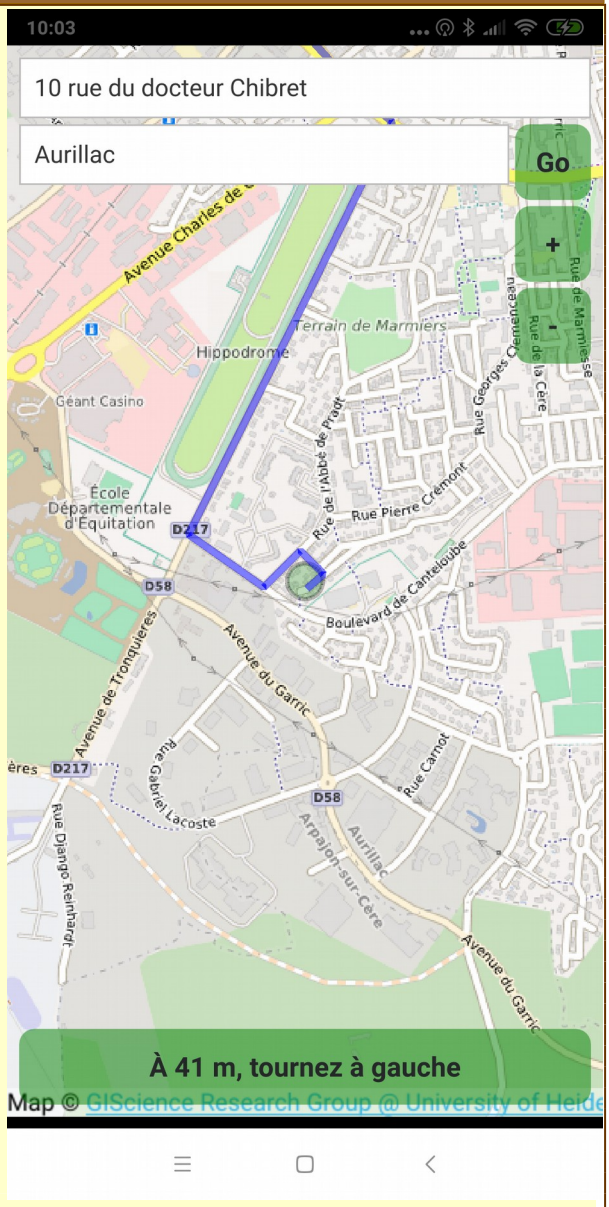
```
#include "navigation.h"

QString Navigation::direction(int distance, int vers)
{
  const QString choix[] = {"destination atteinte",
                           "tout droit",
                           "allez vers la droite",
                           "tournez légèrement à droite",
                           "tournez à droite",
                           "virage serré à droite",
                           "faites demi-tour à droite",
                           "faites demi-tour à gauche",
                           "virage serré à gauche",
                           "tournez à gauche",
                           "tournez légèrement à gauche",
                           "allez vers la gauche"};
  return QString("À %1 m, %2").arg(distance).arg(choix[vers]);
}
```

Bouton.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.4

Button {
  anchors {
    topMargin: 4
    right: parent.right
    rightMargin: 8
  }
  height: ville.height+10
  width: height
  font.bold: true
  background: Rectangle {
    color: "green"
    opacity: 0.5
    radius: 10
  }
}
```

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.3
import QtLocation 5.6
import QtPositioning 5.6
import QtQuick.Controls 2.4

Window {
  width: 320
  height: 480
  visible: true
  title: qsTr("Calcul itinéraire")

  Plugin {
    id: openstreetmap
    name: "osm"
  }

  Map {
    id: carte
    anchors.fill: parent
    plugin: openstreetmap
    center {
      latitude: 44.92
      longitude: 2.44
    }
    zoomLevel: 17

    MapCircle {
      id: puce
      radius: 10
      color: 'green'
      border.width: 3
      opacity: 0.25
    }

    MapQuickItem {
      id: marqueur
      anchorPoint {
        x: image.width/2
        y: image.height
      }
      sourceItem: Column {
        Image { id: image; source: "icone/marqueur.png" }
        Text {
          text: "Destination"
          color: "red"
          font.pixelSize: 12
          font.bold: true
          x: -width/4
        }
      }
    }

    MapItemView {
      model: itinéraire
      delegate: MapRoute {
        route: routeData
        line.color: "blue"
        line.width: 5
        smooth: true
        opacity: 0.5
      }
    }
  }

  GeocodeModel {
    id: adresse
    plugin: carte.plugin
```



```
    onLocationsChanged: {
      if (count==1) {
        choixItinéraire.clearWaypoints()
        choixItinéraire.addWaypoint(gps.position.coordinate)
        choixItinéraire.addWaypoint(get(0).coordinate)
        marqueur.coordinate = get(0).coordinate
        carte.update()
      }
    }
  }

  RouteModel {
    id: itinéraire
    property bool actif: false
    plugin: carte.plugin
    autoUpdate: true
    query: RouteQuery {
      id: choixItinéraire
      travelModes: RouteQuery.CarTravel
      routeOptimizations: RouteQuery.ShortestRoute
    }
    onStatusChanged: {
      if (status == RouteModel.Ready) {
        if (count>=1) {
          actif = true
          direction.text = nav.direction(get(0).segments[0].maneuver.distanceToNextInstruction,
                                         get(0).segments[0].maneuver.direction)
        }
      }
    }
  }

  PositionSource {
    id: gps
    updateInterval: 3000
    active: true
    onPositionChanged: {
      carte.center = position.coordinate
      puce.center = position.coordinate
      if (itinéraire.actif) adresse.update()
    }
  }

  TextField {
    id: rue
    placeholderText: "Rue ou Avenue"
    anchors {
      top: parent.top
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: parent.right
      rightMargin: 8
    }
  }

  Bouton {
    id: soumettre
    anchors.top: rue.bottom
    text: "Go"
    onClicked: {
      adresse.query = rue.text+","+ville.text+",France"
      adresse.update()
    }
  }

  TextField {
    id: ville
    placeholderText: "Ville"
    anchors {
      top: rue.bottom
      topMargin: 4
      left: parent.left
      leftMargin: 8
      right: soumettre.left
      rightMargin: 4
    }
  }

  Bouton {
    text: '+'
    anchors.top: soumettre.bottom
    onClicked: { carte.zoomLevel++;  puce.radius /= 2 }
  }

  Bouton {
    text: '-'
    anchors.top: plus.bottom
    onClicked: { carte.zoomLevel--;  puce.radius *= 2 }
  }
```

```
  Bouton {
    id: direction
    anchors {
      bottom: parent.bottom
      bottomMargin: 8
      left: parent.left
      leftMargin: 8
    }
  }
}
```

Centres d'intérêt

Dans le système de géolocalisation, il peut être intéressant de pouvoir retrouver où se situe les boulangeries les plus proches, les station services, les pizzerias, etc., ce que nous appelons les centres d'intérêt. Encore une fois, **QML** dispose d'un composant de très niveau adapté à cette situation là. Il s'agit de la classe **PlaceSearchModel**, qui renvoie l'ensemble des items associés à la recherche effectuée.

Je vous propose de réaliser un projet qui nous donne ce genre d'informations, suivant deux cas de figure. Soit nous proposons la ville où doit être effectuée la recherche, soit suivant l'endroit où nous nous trouvons par géolocalisation GPS.

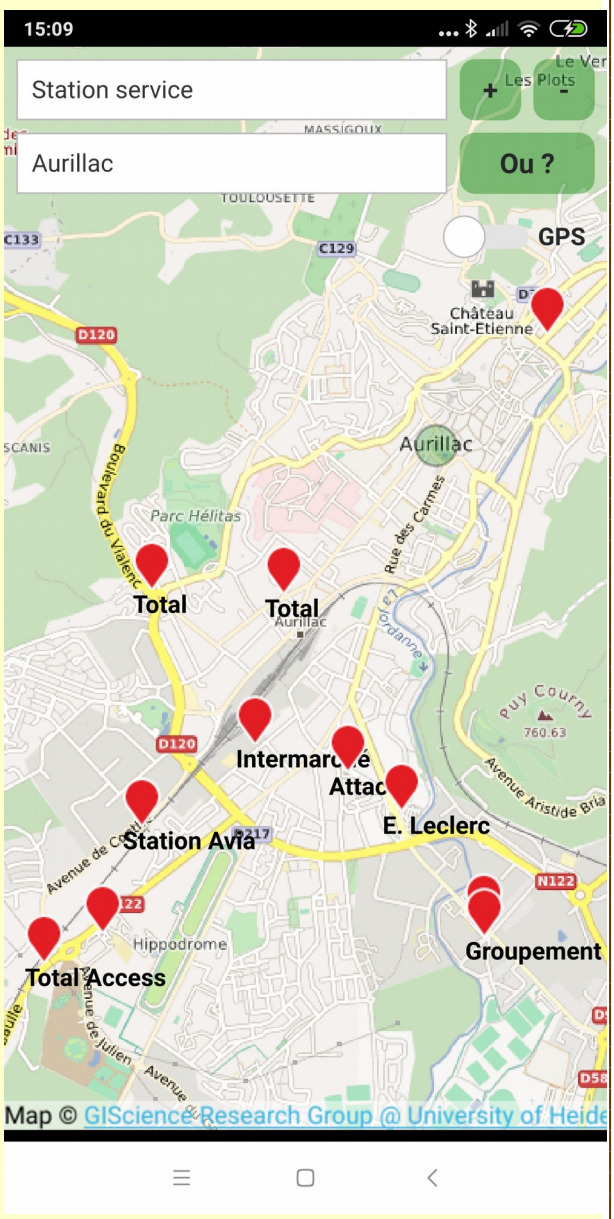
Bouton.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.4

Button {
  anchors {
    topMargin: 8
    right: parent.right
    rightMargin: 8
  }
  height: ville.height
  width: height
  font.bold: true
  background: Rectangle {
    color: "green"
    opacity: 0.4
    radius: 10
  }
}
```

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtPositioning 5.8
import QtLocation 5.9
import QtQuick.Controls 2.4

Window {
  visible: true
  width: 640
  height: 480
  title: qsTr("Centres d'intérêt")

  Plugin {
    id: openstreetmap
    name: "osm"
  }

  Map {
    id: carte
    anchors.fill: parent
    plugin: openstreetmap
    center {
      latitude: 44.92
      longitude: 2.44
    }
    zoomLevel: 14

    MapCircle {
      id: puce
      radius: 90
      color: 'green'
      border.width: 2
      opacity: 0.25
    }

    MapItemView {
      model: recherche
      delegate: MapQuickItem {
        coordinate: place.location.coordinate
        anchorPoint.x: image.width * 0.5
        anchorPoint.y: image.height
        sourceItem: Column {
          Image { id: image; source: "qrc:/icone/marqueur.png" }
          Text { text: title; font.bold: true }
        }
      }
```

```
  }
}

PlaceSearchModel {
  id: recherche
  plugin: carte.plugin
}

GeocodeModel {
  id: localisation
  plugin: carte.plugin
  onLocationsChanged: {
    if (count>=1) {
       carte.center = get(0).coordinate
       puce.center = get(0).coordinate
       recherche.searchTerm = intérêt.text
       recherche.searchArea = QtPositioning.circle(carte.center, 10000)
       recherche.update()
    }
  }
}

PositionSource {
  id: gps
  updateInterval: 10000
  onPositionChanged: {
    carte.center = position.coordinate
    puce.center = position.coordinate
  }
}

Address {
  id: adresse
  street: "1"
  country: "France"
}

Bouton {
  id: moins
  text: '-'
  anchors.top: parent.top
  onClicked: { carte.zoomLevel--; puce.radius*=2 }
}

Bouton {
  id: plus
  text: '+'
  anchors.top: parent.top
  anchors.right: moins.left
  onClicked: { carte.zoomLevel++; puce.radius/=2 }
}

TextField {
  id: intérêt
  placeholderText: "Centre d'intérêt"
  anchors {
    top: parent.top
    topMargin: 8
    left: parent.left
    leftMargin: 8
    right: plus.left
    rightMargin: 8
  }
}

TextField {
  id: ville
  placeholderText: "Ville"
  anchors {
    top: intérêt.bottom
    topMargin: 8
    left: parent.left
    leftMargin: 8
    right: soumettre.left
    rightMargin: 8
  }
}

Bouton {
  id: soumettre
  anchors {
    left: plus.left
    top: plus.bottom
   }
  text: "Ou ?"

  onClicked: {
    if (choixGPS.checked) {
      recherche.searchTerm = intérêt.text
      recherche.searchArea = QtPositioning.circle(carte.center, 10000)
      recherche.update()
    }
```



```
      else {
        adresse.city = ville.text
        localisation.query = adresse
        localisation.update()
      }
    }
  }

  Switch {
    id: choixGPS
    text: "GPS"
    font.bold: true
    anchors {
      top: soumettre.bottom
      topMargin: 8
      right: parent.right
      rightMargin: 8
    }
    onCheckedChanged: {
      ville.enabled = !checked
      gps.active = checked
    }
  }
}
```

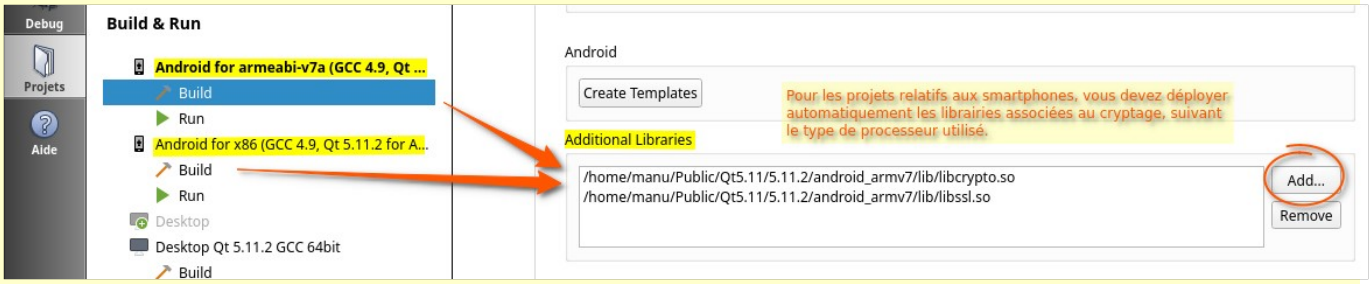
Nous utilisons cette fois-ci, le composant Address à la place d'un simple texte lorsque nous composons la requête pour le géocodage, qui nous permet de cibler la bonne adresse. Lorsque vous mettez le chiffre « 1 » tout seul dans l'identification de la rue, le système prend alors automatiquement le centre de la ville (ce qui est bien sympathique).

Pour le système de recherche de centres d'intérêt, représenté donc par la classe PlaceSearchModel, vous devez préciser le type « d'intérêt » avec la propriété « searchTerm » et la zone de couverture « cercle limite autour du point de géolocalisation » grâce à la propriété « searchArea » mais aussi grâce à l'appel de la méthode circle() de QtPositioning, dont le deuxième argument nous donne le rayon de couverture en mètres, ici 10km.

Conclusion : communication avec les services de géolocalisation

Attention, pour que l'ensemble des projets puissent fonctionner correctement lors d'une communication avec les services de géocodage, de routage ou de recherche de centres d'intérêt, vous devez intégrer les librairies de cryptage **OpenSsl**, puisque tous ces échanges sont normalement protégés. Vous avez ci-dessous la procédure à suivre pour cela dans le cas particulier de l'utilisation de l'application avec un smartphone.

Pour les projets clients relatifs aux smartphones Android, il faut que la librairie correspondant à OpenSSL puissent être déployée automatiquement avec le projet. Pour cela, vous devez récupérer les fichiers binaires correspondant au smartphone réel et au smartphone émulé. Ces fichiers sont disponibles sur Internet. Deux sont nécessaires « libcrypto.so » et « libssl.so ». Vous allez ensuite dans la rubrique « Projets » de QtCreator et plus précisément dans la zone « Build Android APK ».



Dans certains cas, il peut être nécessaire de placer ces deux fichiers dans le répertoire du projet d'étude (il faut tester).

Plusieurs services de géolocalisation avec plusieurs types de cartes

Je vous propose d'étoffer le projet précédent afin que nous puissions avoir plusieurs types de cartes, à visualiser, comme par exemple, le mode cycle, ou bien une carte topographique, la vue satellitaire, etc. et de choisir également d'autres services de géolocalisation, tout ceci dans la même application.

Ainsi, en plus du système cartographique « openstreetmap », je propose de prendre « mapbox », « here » et « esri ». Pour chacun de ces services, vous devez vous enregistrer sur les sites respectifs en prenant des comptes gratuits (ils possèdent alors des limitations du nombre d'accès dans le mois – pour une seule personne, cela devrait suffire).

Vous découvrirez alors que suivant les comptes, vous obtiendrez des réponses plus ou moins importantes avec le nom des éléments de recherche ou pas.

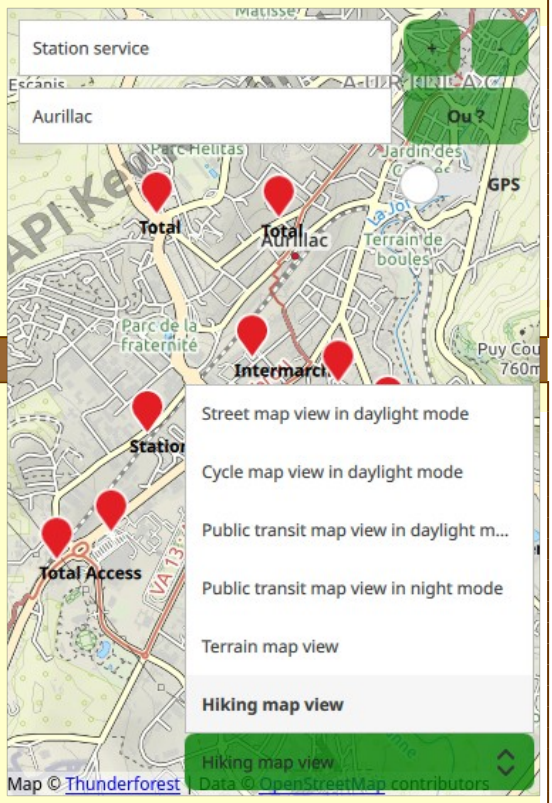
Pour ce projet, nous utilisons un « SwipeView » afin que nous puissions choisir très simplement le fournisseur avec lequel vous désirez faire votre recherche. Si vous appuyez longuement votre doigt et que vous le déplacez, vous déplacez alors votre carte. Si par contre vous faites un glissé rapide latéral, vous changez alors de système cartographique.

Dans notre projet, nous reprenons notre composant « Bouton » et nous rajoutons un nouveau composant « Carte » qui aura la même présentation générale quelque soit le système cartographique choisi.

Bouton.qml

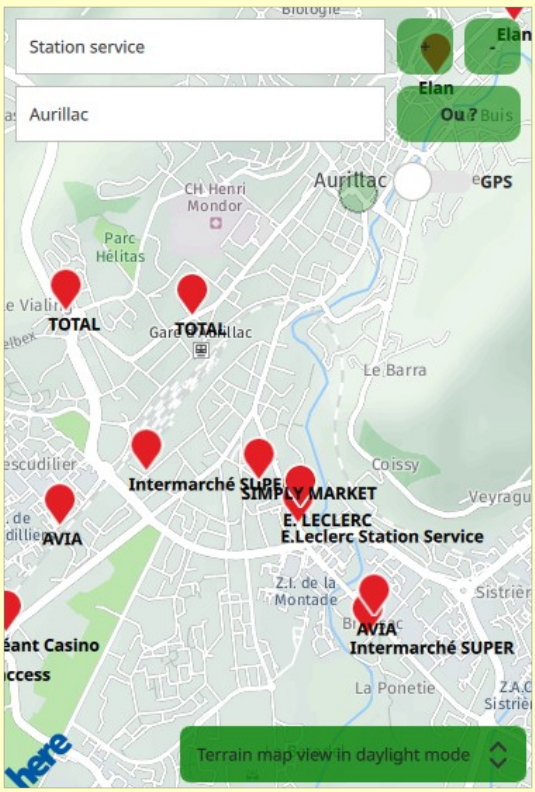
```
import QtQuick 2.0
import QtQuick.Controls 2.4
```

```
Button {
  anchors {
    topMargin: 8
    right: parent.right
    rightMargin: 8
  }
  height: ville.height
  width: height
  font.bold: true
  background: Rectangle {
    color: "green"
    opacity: 0.4
    radius: 10
  }
}
```

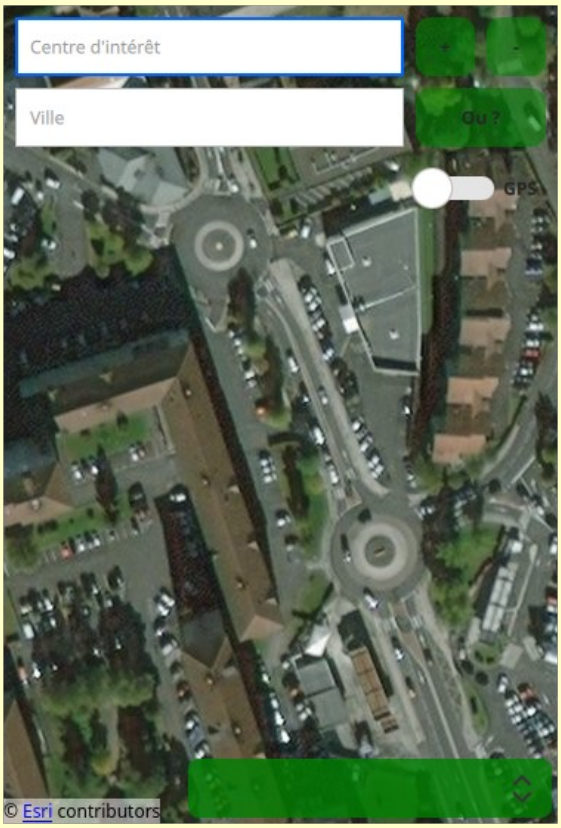


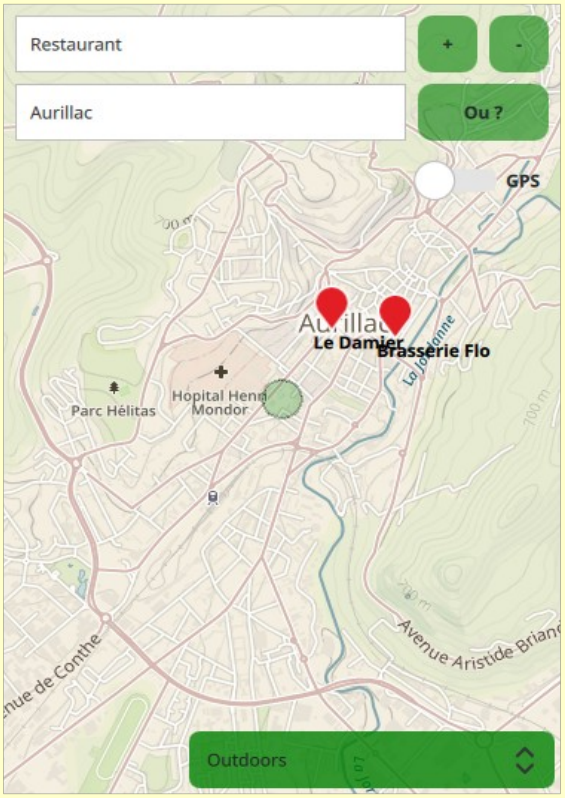
Carte.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.4
import QtPositioning 5.8
import QtLocation 5.9

Page {
  property alias plugin: carte.plugin

  Map {
    id: carte
    anchors.fill: parent
    center {
      latitude: 44.92
      longitude: 2.44
    }
    zoomLevel: 14

    MapCircle {
      id: puce
      radius: 90
      color: 'green'
      border.width: 2
      opacity: 0.25
    }

    MapItemView {
      model: recherche
      delegate: MapQuickItem {
        coordinate: place.location.coordinate
        anchorPoint.x: image.width * 0.5
        anchorPoint.y: image.height
        sourceItem: Column {
          Image { id: image; source: "qrc:/icone/marqueur.png" }
          Text { text: title; font.bold: true }
        }
      }
    }
  }

  PlaceSearchModel {
    id: recherche
    plugin: carte.plugin
  }

  GeocodeModel {
    id: localisation
    plugin: carte.plugin
    onLocationsChanged: {
      if (count>=1) {
         carte.center = get(0).coordinate
         puce.center = get(0).coordinate
         recherche.searchTerm = intérêt.text
         recherche.searchArea = QtPositioning.circle(carte.center, 10000)
         recherche.update()
      }
    }
  }

  PositionSource {
    id: gps
    updateInterval: 10000
    onPositionChanged: {
      carte.center = position.coordinate
      puce.center = position.coordinate
    }
  }

  Address {
    id: adresse
    street: "1"
    country: "France"
  }
```



```
Bouton {
  id: moins
  text: '-'
  anchors.top: parent.top
  onClicked: { carte.zoomLevel--; puce.radius*=2 }
}

Bouton {
  id: plus
  text: '+'
  anchors.top: parent.top
  anchors.right: moins.left
  onClicked: { carte.zoomLevel++; puce.radius/=2 }
}

TextField {
  id: intérêt
  placeholderText: "Centre d'intérêt"
  anchors {
    top: parent.top
    topMargin: 8
    left: parent.left
    leftMargin: 8
    right: plus.left
    rightMargin: 8
  }
}

TextField {
  id: ville
  placeholderText: "Ville"
  anchors {
    top: intérêt.bottom
    topMargin: 8
    left: parent.left
    leftMargin: 8
    right: soumettre.left
    rightMargin: 8
  }
}

Bouton {
  id: soumettre
  anchors {
    left: plus.left
    top: plus.bottom
   }
  text: "Ou ?"
  onClicked: {
    if (choixGPS.checked) {
      recherche.searchTerm = intérêt.text
      recherche.searchArea = QtPositioning.circle(carte.center, 10000)
      recherche.update()
    }
    else {
      adresse.city = ville.text
      localisation.query = adresse
      localisation.update()
    }
  }
}

Switch {
  id: choixGPS
  text: "GPS"
  font.bold: true
  anchors {
    top: soumettre.bottom
    topMargin: 8
    right: parent.right
    rightMargin: 8
  }
  onCheckedChanged: {
    ville.enabled = !checked
    gps.active = checked
  }
}

ComboBox {
  id: choixVue
  anchors {
    right: parent.right
    rightMargin: 4
    bottom: parent.bottom
    bottomMargin: 4
    left: parent.left
    leftMargin: parent.width/3
  }
  height: soumettre.height
  model: carte.supportedMapTypes
  textRole: "description"
```





```
    background: Rectangle {
      color: "green"
      opacity: 0.8
      radius: 7
    }
    onCurrentIndexChanged: carte.activeMapType = carte.supportedMapTypes[currentIndex]
  }
}
```

Il est possible de choisir le type de carte que vous désirez visualiser grâce à la propriété activeMapType de la classe Map en spécifiant son choix parmi la liste (supportedMapTypes) données par le fournisseur cartographique qui peut alors être plus ou moins étoffée.

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.4
import QtPositioning 5.8
import QtLocation 5.9

Window {
  id: vue
  visible: true
  width: 640
  height: 480
  title: qsTr("Centres d'intérêt")

  SwipeView {
    anchors.fill: parent
    Carte {
      plugin: Plugin {
        locales: "fr_FR"
        name: "osm"
        PluginParameter {
          name: "apikey"
          value: "votre clé"
        }
      }
    }

    Carte {
      plugin: Plugin {
        locales: "fr_FR"
        name: "here"
        PluginParameter {
          name: "here.app_id"
          value: "votre identifiant"
        }
        PluginParameter {
          name: "here.token"
          value: "votre clé"
        }
      }
    }

    Carte {
      plugin: Plugin {
        locales: "fr_FR"
        name: "esri"
        PluginParameter {
          name: "token"
          value: "votre clé"
        }
      }
    }

    Carte {
      plugin: Plugin {
        locales: "fr_FR"
        name: "mapbox"
        PluginParameter {
          name: "mapbox.access_token";
          value: "votre clé"
        }
      }
    }
  }
}
```

Vous avez besoin de placer vos différentes clés, données lors de votre inscription, en spécifiant les bons paramètres à l'aide de la classe adaptée PluginParameter. (Je n'ai malheureusement pas réussi à bien régler la clé pour « OpenStreetMap », et du coup, vous voyez apparaître le message « API KEY REQUIRED » et pourtant je suis bien enregistré, peut être quelqu'un connaît la solution).