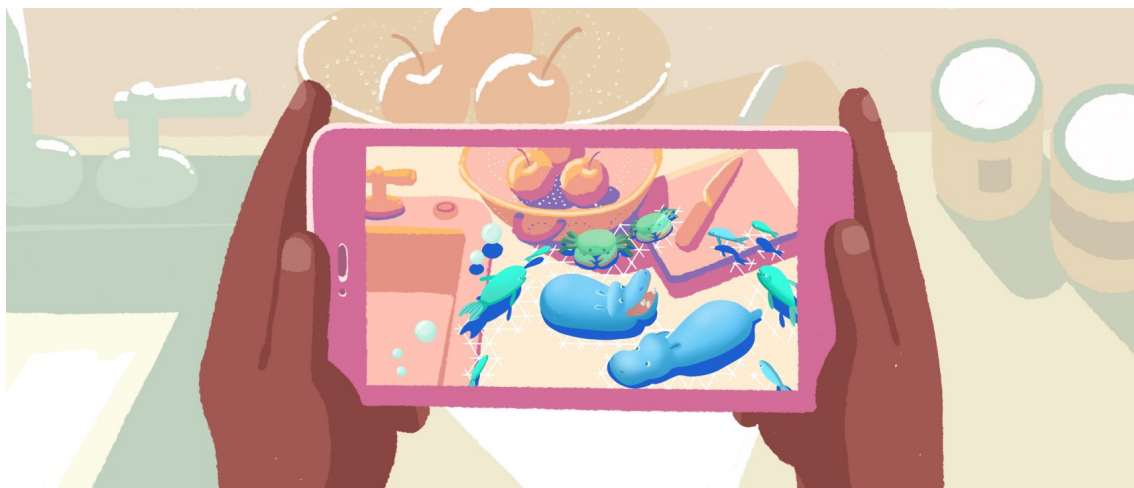




Éléments 3D géolocalisés en Réalité Augmentée

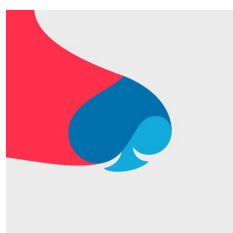
Tristan Cabantous

Stage d'Ingénieur chez Capgemini DEMS

Effectué du 11 Février au 09 Août 2019

Tuteur en entreprise : Raphaël Guenon

Tutrice à l'Université : Jessie Carbonnel



**UNIVERSITÉ
DE MONTPELLIER**

Master 2 Informatique - IMAGINA
Département Informatique - Université de Montpellier

Éléments cartographiques géolocalisés 3D en Réalité Augmentée

Tristan Cabantous

Stage d'ingénieur dans le cadre du Master 2 IMAGINA¹ dans la faculté des sciences de l'université Montpellier et effectué dans l'entreprise Capgemini DEMS², du 11 Février 2019 au 09 Août 2019.

Résumé

Mon stage de fin d'études du Master 2 IMAGINA de l'université de Montpellier a été réalisé dans l'entreprise **Capgemini DEMS** (anciennement Sogeti High Tech) du 11 février 2019 au 09 août 2019, sous la tutelle de Raphaël Guenon.

Mon objectif a été de **réaliser une application de réalité augmentée**³ permettant d'afficher des informations géographiques **géolocalisées**. Dans un premier temps, j'ai analysé les possibilités technologiques de réalité augmentée adaptées et déterminé un format de stockage pour les informations cartographiques. Dans un second temps, j'ai réalisé un prototype fonctionnel permettant d'afficher ces informations géolocalisées en réalité augmentée.

Mots-clés : *Réalité Augmentée Géolocalisée, Unity3D, C#, ARCore*

1. Images, Games et Intelligent Agents

2. Digital Engineering & Manufacturing Services

3. Superposition de la réalité et d'éléments graphiques ou sonores en temps réel

Déclaration

Le travail de ce rapport a pour base mes recherches personnelles menées dans le cadre de mes études au département informatique de la faculté de Montpellier et lors d'un stage dans l'entreprise Capgemini DEMS. Aucune partie de ce rapport n'a été publiée dans un autre cadre que celui évoqué au-dessus, ou pour tout autres diplômes ou qualifications. Le contenu de ce rapport est issu de mon propre travail exceptées les références citées dans mon écrit et servant à illustrer mon propos.

Copyright © 2019 by CABANTOUS Tristan.

"Les droits de copyright sur ce rapport doivent rester à leurs auteurs. Aucune citation en provenance de l'ensemble du rapport ne peut être publiée sans l'accord et le consentement de ses auteurs et l'information qui lui en est extraite doit être reconnue."

Remerciements

J'aimerais remercier dans un premier temps Jessie Carbonnel, tutrice à l'université, pour sa disponibilité et son écoute. Je remercie toute l'équipe de l'open-space A3 et tout particulièrement Raphaël Guenon, tuteur en entreprise, pour sa patience et sa bienveillance tout au long du stage ainsi que Nicolas Moreau pour son accueil et sa confiance. Enfin, j'adresse mes remerciements au corps enseignant du Master IMAGINA pour la qualité des enseignements que j'ai suivis et tout particulièrement à Monsieur Puech pour sa bienveillance.

Bonne lecture.

Tristan Cabantous

Table des matières

Résumé	3
Déclaration	4
Remerciements	5
1 Introduction	9
1.1 Présentation de la société	9
1.2 Description du projet	10
1.3 Choix du sujet de stage	14
1.4 Déroulement et gestion du projet	14
2 État de l'art	17
2.1 Applications existantes	17
2.1.1 AuGeo	17
2.1.2 Trimble Connect	18
2.1.3 Trimble SiteVision	18
2.2 Formats de stockage	19
2.2.1 Objectifs	19
2.2.2 Les formats existants intéressants	19
2.3 Méthode de projection	22
2.3.1 L'intérêt d'une projection	22
2.3.2 Les projections adaptées	22
2.4 Outils de Réalité Augmentée	25
2.4.1 Objectifs	25
2.4.2 Les outils envisageables	25
2.5 Aide à la précision	28
2.5.1 Objectifs et intérêts	28
2.5.2 Les outils potentiels	29

3	Rapport de conception	33
3.1	Choix des technologies	33
3.1.1	Réception et stockage des données	33
3.1.2	Gestion et architecture des données	35
3.1.3	Outils de Réalité Augmentée	39
3.1.4	Amélioration de la précision	40
3.1.5	Récapitulatif	41
3.2	Les fonctionnalités	42
3.2.1	Liste des fonctionnalités	42
3.2.2	Cas d'utilisation	42
3.3	Les modules de l'application	43
3.4	Chiffrage et planning prévisionnel	44
4	Développement d'un premier prototype	47
4.1	Un module de conversion	47
4.1.1	Présentation du module	48
4.1.2	Fonctionnement de la conversion	49
4.2	Mise en place du prototype	50
4.2.1	Linker	50
4.2.2	La conversion des coordonnées	51
4.3	La gestion des Layers	51
4.3.1	Création des Layers	52
4.3.2	Instanciation des layers	53
4.4	Définition de l'attitude de l'appareil	58
4.4.1	Implémentation du GPS et de la boussole Android	59
4.4.2	Interface de l'application	60
4.5	L'ajout de la réalité augmentée	61
4.5.1	Mise en place d'ARCore	62
4.5.2	Mise à jour des positions au cours du temps	63
4.6	Résultats et limites de la première version du prototype	64
4.6.1	Résultats obtenus	64
4.6.2	Limites	67
5	Vers un prototype fonctionnel	69
5.1	Amélioration de la géolocalisation et de l'orientation	70
5.1.1	Implémentation du Satlab SLC	70
5.1.2	Le Landmarks Orientation Provider	71

Table des matières	8
5.1.3 Modification de l'interface	72
5.2 Fonctions additionnelles et corrections	74
5.2.1 Gestion de l'altitude	74
5.2.2 Correction de la mise à jour des positions	75
6 Conclusion	76
6.1 Bilan technique	76
6.1.1 Résultats obtenus	76
6.1.2 Limites perceptibles	78
6.1.3 Fonctions additionnelles et pistes d'amélioration	79
6.1.4 Planning final	80
6.2 Bilan humain	82
6.2.1 Difficultés rencontrées	82
6.2.2 Enseignements tirés et compétences acquises	83
Bibliographie	85
Annexes	87
A Liste des fonctionnalités	87
A.1 Les fonctionnalités essentielles	87
A.2 Les fonctionnalités secondaires	88
B Cas d'utilisation	89
C Diagramme de classe	91
D Planning prévisionnel	92
E Planning final	94

Chapitre 1

Introduction

Ce stage a été réalisé lors du semestre 4 du Master IMAGINA de la faculté des sciences de Montpellier (34). L'objectif de ce parcours est de former des ingénieurs dans les industries de l'image, des jeux vidéos, des simulateurs et de l'informatique 3D. L'enseignement des modules de ce parcours est réalisé par les enseignants-chercheurs du LIRMM¹. Ce projet de fin d'études doit rentrer dans les enseignements du Master et nous permettre de développer nos compétences dans notre domaine de prédilection. J'ai pu réaliser ce stage dans l'entreprise Capgemini DEMS (anciennement appelé Sogeti High Tech) sur le site de Toulouse (31) *du 11 Février au 09 Août 2019* et sous la tutelle de Raphaël Guenon, responsable technique.

1.1 Présentation de la société

Créée en 1967 à Grenoble, Capgemini est actuellement **la première entreprise de services numériques en France** et la sixième mondiale.



FIGURE 1.1 – Logo de Capgemini

En 2017, Capgemini est présente dans plus de 40 pays et compte 200 000 collaborateurs. Son chiffre d'affaires est de 12,8 milliards d'euros.

1. Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier

Sogeti High Tech, aujourd'hui Capgemini DEMS, est la filiale du groupe Capgemini dédiée à l'ingénierie industrielle et à l'innovation technologique dans les secteurs de l'aéronautique, du spatial et de la défense. Il existe plus de 4000 collaborateurs dans ce groupe.

Mon stage s'est déroulé dans le pôle *3D/Réalité Augmentée/Réalité Virtuelle*² de l'unité *Market Unit System & Digital Engineering*. À Toulouse, le pôle développe surtout des applications pour des sociétés dans l'aéronautique, notamment pour **Airbus**³, mais mène aussi des projets internes comme **Andy3D** : application permettant de faire de l'assistance à distance grâce à la RA⁴/RV⁵ et l'interprétation en temps réel d'un modèle 3D.

Mon stage s'est déroulé sur le site Aéroport de Toulouse (3 *Chemin de Laporte, 31300 Toulouse*) dans l'open-space A3.

1.2 Description du projet

Mon objectif au sein du groupe a été de réaliser **une application de réalité augmentée permettant d'afficher des informations géographiques géolocalisées**.



FIGURE 1.2 – Logo du prototype Glare

On a par la suite **nommé** cette application **Glare : Geo Location Augmented Reality Experience**.

La description exacte de l'intitulé du stage était : *"Au sein de son Business Group RA/RV, Capgemini DEMS réalise des projets mettant en oeuvre des technologies émergentes. Capgemini DEMS développe actuellement une solution d'aide aux pilotes en environnement aéroportuaire embarquant une base de données contenant de nombreuses informations géographiques. Elle développe d'autre part plusieurs applications de réalité augmentée. Dans le cadre de ses activités R&D, Capgemini DEMS souhaite étudier la possibilité de faire cohabiter ces deux types d'application à travers une application de réalité augmentée permettant d'afficher des informations géographiques géolocalisées. Votre mission consistera à analyser les possibilités technologiques*

2. Simulation d'un utilisateur dans un monde virtuel

3. Constructeur aéronautique européen basé à Toulouse. Fabrique plus de la moitié des avions de ligne produits dans le monde.

4. Abréviation de Réalité Augmentée

5. Abréviation de Réalité Virtuelle

de réalité augmentée appropriées et à réaliser un couplage avec les différents types de données géographiques utilisés au sein du groupe."

La réalité augmentée géolocalisée

La réalité augmentée est une technologie visant à inclure des informations **en temps réel superposées à la réalité**. Ces informations peuvent être graphiques, comme des objets en deux ou trois dimensions ou sonores. Elle est le plus fréquemment utilisée par l'intermédiaire d'un casque spécialisé ou par un smartphone ou une tablette. Les caméras sur le téléphone ou le casque vont capturer l'environnement et l'application va analyser cet environnement et afficher l'information choisie. Il est important de prendre en compte qu'il est nécessaire d'avoir les informations de rotation du dispositif (gyromètre⁶, magnétomètre⁷ et/ou accéléromètre⁸) pour comprendre vers où l'utilisateur pointe sa caméra. Aujourd'hui, les plus grandes entreprises investissent dans ce domaine, notamment **Google et Apple**. Ils ont tous deux créé des outils permettant de faire de la RA, respectivement : *ARCore* (cf. page 26) et *ARKit* (cf. page 26).

D'après *Digi-Capital*, une entreprise de conseil en RA/RV de la Silicon Valley, le marché combiné de la réalité augmentée et de la réalité virtuelle pourrait dépasser les 108 milliards de dollars en 2021 [13].

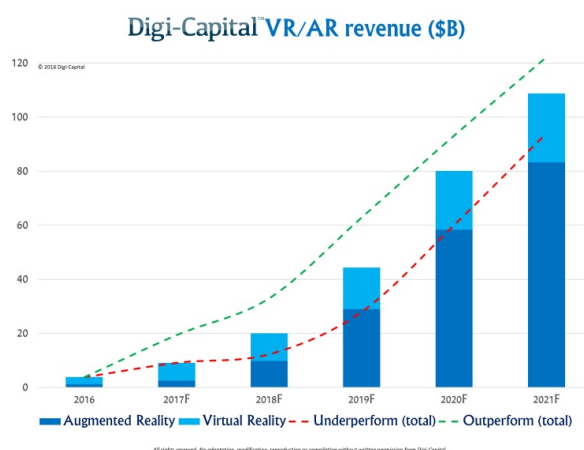


FIGURE 1.3 – Le marché de la RA/RV selon Digi-Capital

Dans le cas de **la réalité augmentée géolocalisée**, on cherche à afficher une information à un endroit donné, c'est-à-dire à **une position précise**. C'est une discipline assez

6. Mesure les changements d'orientation et les vitesses de rotation

7. Mesure les champs magnétiques pour agir comme une boussole

8. Mesure les accélérations

nouvelle et innovante en informatique où l'on fait correspondre la réalité augmentée avec une notion de géolocalisation. Les possibilités sont grandes et beaucoup d'entreprises s'y intéressent.

Les enjeux d'une telle application sont :

- **la précision du tracking**, c'est-à-dire la procédure d'analyse de l'environnement et l'élaboration d'une carte numérique de l'espace par l'application,
- **la précision de la géolocalisation**, la capacité de notre application à déterminer la position dans le monde de l'utilisateur et des objets à afficher
- et **la précision de l'orientation** pour déterminer vers où pointe la caméra de l'utilisateur.

En effet, l'intérêt d'une application de RA géolocalisée est d'avoir des informations à des positions précises. Si la précision n'est pas au rendez-vous, nos informations ne seraient pas bien représentées dans le monde réel et notre application ne serait pas fonctionnelle.

Informations cartographiques géolocalisées

Dans notre projet, les informations à afficher en réalité augmentée sont cartographiques et géolocalisées. Un certain objet devra être affiché à une certaine position dans l'espace 3D. Comme évoqué plus haut dans la description du sujet de stage, il a fallu réfléchir aux moyens de stocker et représenter ces informations.

Contexte du projet

L'idée de ce projet est venue suite à la réalisation de deux applications dans le pôle RA/RV de Capgemini DEMS toulousain.



FIGURE 1.4 – Mission +

La première est **Mission+**, un logiciel d'aide à la préparation des plans de vol aéronautique développé pour Navblue⁹. Dans un avion, on l'utilise dans une partie du cockpit qu'on appelle *Monde Ouvert*. Elle doit permettre au pilote de l'aider à naviguer sans pour autant être essentielle (elle n'est pas définie comme *critique*). Elle lui permet de visualiser un grand nombre de données importantes pour un pilote dans un environnement 3D.

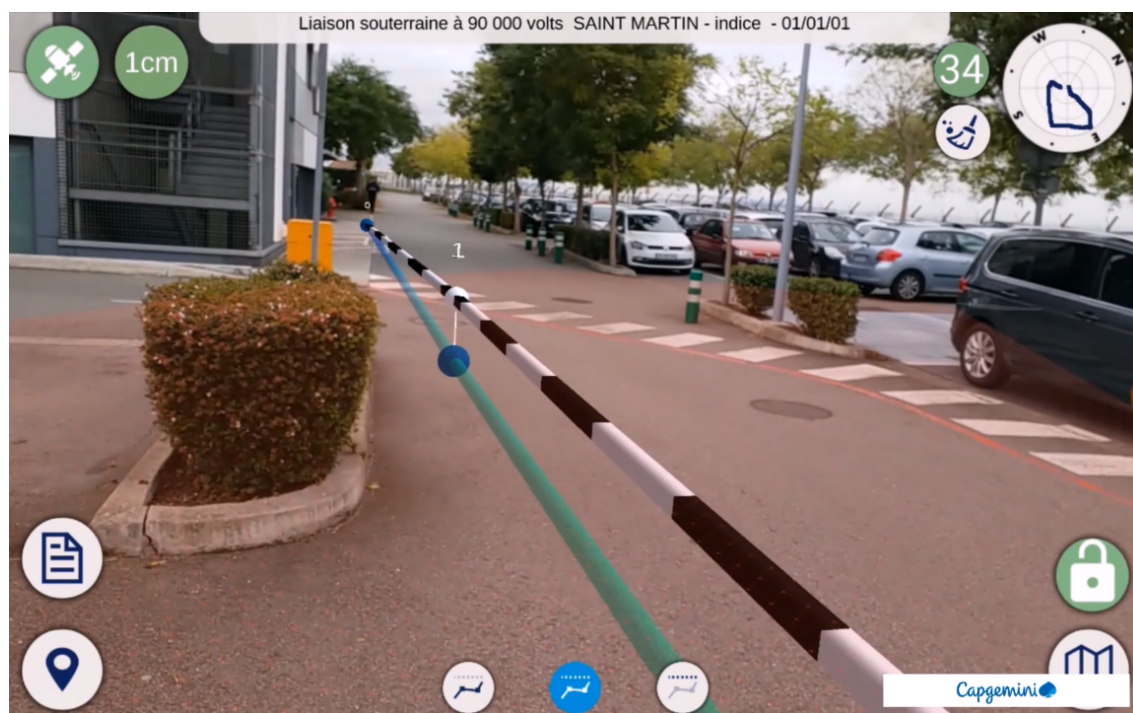


FIGURE 1.5 – RA-LS

La deuxième application est **RA-LS**, un projet [22] pour RTE¹⁰. L'objectif de cette application est d'afficher en réalité augmentée des lignes souterraines d'électricité. Ces liaisons souterraines sont enterrées et ne peuvent pas être détectées à moins de creuser sous plusieurs mètres. L'intérêt est de pouvoir afficher ces données plus précisément qu'avec une carte standard, en réalité augmentée et directement sur une tablette.

L'objectif de mon stage a donc été de mettre en relation ces deux applications afin de créer un prototype capable d'afficher des informations cartographiques en RA comme pour le projet RTE, de toute source, notamment de Mission+.

Ce stage rentre plutôt dans le cadre de la recherche et du développement. L'idée de mon application leur est venue suite au projet RA-LS développé par mon tuteur de stage. L'objectif étant dans un premier temps de remettre en cause les choix d'implémentation du projet et les technologies utilisées. Le second objectif était de permettre d'afficher tout type de

9. Compagnie d'Airbus spécialisée dans les logiciels d'exploitation des compagnies aériennes

10. Réseau de Transport d'Électricité

donnée, d'avoir une application qui ne soit pas limitée dans le choix des données à afficher. Comme décrit précédemment, la précision est une notion importante qui va définir les outils que l'on va utiliser. En partant de zéro, il a fallu dans un premier temps que je définisse les outils que j'allais utiliser dans l'état de l'art en recherchant des technologies adaptées à mon problème. Ensuite, j'ai défini comment hiérarchiser et organiser mes technologies entre elles pour arriver à un prototype permettant de faire de la réalité augmentée géolocalisée à partir de données provenant de plusieurs sources. Enfin, j'ai réalisé le prototype à partir des choix faits précédemment.

1.3 Choix du sujet de stage

Depuis que j'ai commencé mes études j'ai toujours été attiré par **l'interaction visuelle** et notamment par la réalité virtuelle et augmentée. En première année de DUT j'avais d'ailleurs fait un exposé sur les hologrammes et l'entreprise MagicLeap¹¹ que j'ai pu étudier dans l'état de l'art de ce stage.

Je voulais aussi développer mes compétences sur le moteur de jeu **Unity3D**. En effet, la majorité des projets de réalité virtuelle et augmentée dans le pôle est réalisée sur Unity. Il s'agit d'un moteur de jeu multiplateformes. C'est l'un des plus répandus dans l'industrie du jeu vidéo mais aussi dans le domaine de la simulation et de la RA/RV. Il a notamment été utilisé par *Blizzard*¹² pour développer *Hearthstone*¹³. Il était important pour moi d'approfondir mes connaissances sur ce moteur pour la suite de ma vie professionnelle.

Je voulais aussi me confronter au travail d'ingénieur dans une **grande entreprise**. J'ai par le passé travaillé plusieurs fois dans une petite entreprise et il était important pour moi d'élargir le champ de mes connaissances du monde de l'entreprise et de varier mes expériences professionnelles dans le secteur.

1.4 Déroulement et gestion du projet

Avant de développer le prototype, il était nécessaire de comprendre les enjeux de l'application et le domaine dans lequel j'allais travailler. De plus, il me fallait aussi définir les différents outils que j'allais utiliser. Un **état de l'art** a alors été réalisé à mon arrivée et ce

11. Startup américaine de réalité augmentée

12. Société américaine de développement et de d'édition de jeux vidéo

13. Jeu de cartes en ligne

pendant trois semaines (du 11 Février au 1^{er} Mars) pour définir :

— **le stockage des données**

- le format à utiliser pour afficher des informations cartographiques en Réalité Augmentée
- et la représentation des données dans Unity
 - la méthode de projection à utiliser
 - les conventions d'architecture des données à adopter

— **l'affichage en Réalité Augmentée**

- la technologie à utiliser
- l'appareil de Réalité Augmentée adapté
- et la gestion de l'apparence des informations à afficher

— **la précision**

- de la géolocalisation
- et de l'orientation.

Cette phase d'analyse a été suivie par une **phase de conception** de deux semaines (du 4 Mars au 15 Mars) me permettant de mieux envisager la suite du projet. Dans cette partie, j'ai défini la structure de l'application et découpé le prototype en plusieurs modules. La décomposition en module m'a permis ensuite de réaliser un chiffrage et de définir un planning prévisionnel.

Le suivi des avancées du projet a été fait selon **la méthode Kanban**.

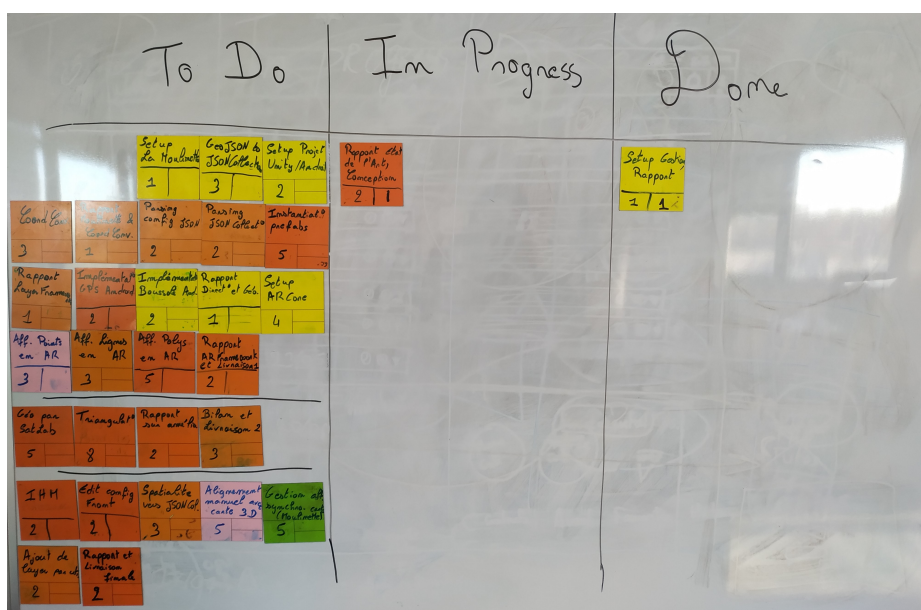


FIGURE 1.6 – Gestion du projet par la méthode Kanban

Il s'agit simplement d'un tableau à trois colonnes contenant la description et l'état des

tâches à réaliser. La première colonne est dédiée aux tâches à faire, la seconde à celles en cours et la dernière à celles qui sont terminées. Quotidiennement, je mettais à jour ce tableau en inscrivant le temps réel que je passais sur la tâche en cours.

La faisabilité des tâches est évaluée en jours et chaque tâche est définie par un intitulé.

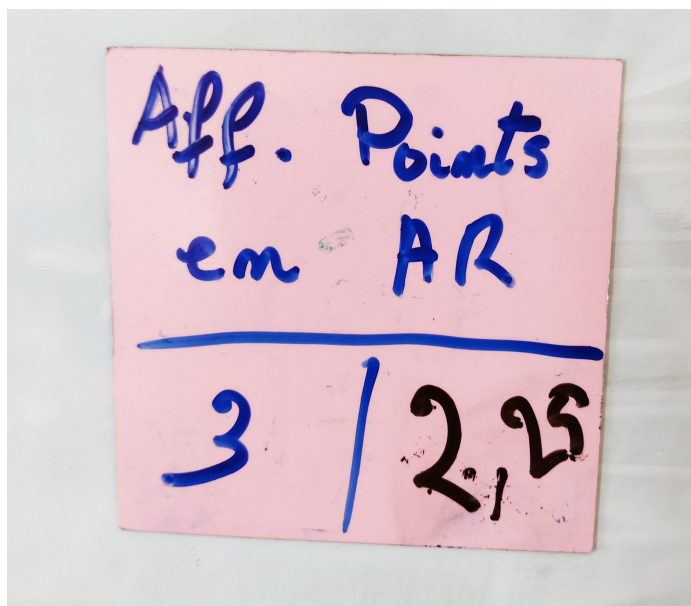


FIGURE 1.7 – Exemple de post-it

On indique l'intitulé de la tâche, le temps estimé pour la terminer (à gauche) et le temps passé dessus (à droite).

J'ai aussi utilisé en parallèle *Trello* et *TrelloGantt*. Il s'agit d'outils de gestion de projet en ligne. Ces deux outils permettent d'avoir une estimation des dates prévisionnelles et d'avoir un diagramme de Gantt¹⁴ facilement modifiable. Le diagramme prévisionnel est présenté dans la partie Chiffrage et planning prévisionnel (cf. page 44).

De plus, le suivi de mon projet par mon tuteur de stage s'est fait lors de réunions hebdomadaires où l'on échangeait sur les tâches en cours et à venir ainsi que sur les difficultés que je rencontrais.

14. Outil de gestion de projet permettant de visualiser les tâches dans le temps

Chapitre 2

État de l'art

Avant de commencer la conception et le développement de mon prototype, il était nécessaire que je comprenne les enjeux de la réalité augmentée géolocalisée. Notamment, il a fallu que je me renseigne sur les projets en réalité augmentée qui existaient et que je définisse les outils utiles à mon prototype.

2.1 Applications existantes

Dans un premier temps j'ai cherché à référencer les applications dont je pourrais m'inspirer. Cependant, la réalité augmentée géolocalisée étant une discipline nouvelle, il n'existe que très peu d'applications.

2.1.1 AuGeo

AuGeo [1] est une application mobile de Esri¹ Labs, une entreprise privée qui fournit des applications de systèmes géographiques (GIS²). L'application est disponible gratuitement sur iTunes et Google Play pour les utilisateurs d'ArcGIS, une suite de logiciels d'information géographique. L'application permet aux utilisateurs d'importer des données géographiques provenant d'ArcGIS et de les afficher en réalité virtuelle à partir d'un smartphone.

1. Environmental Systems Research Institute

2. Geographic Information System

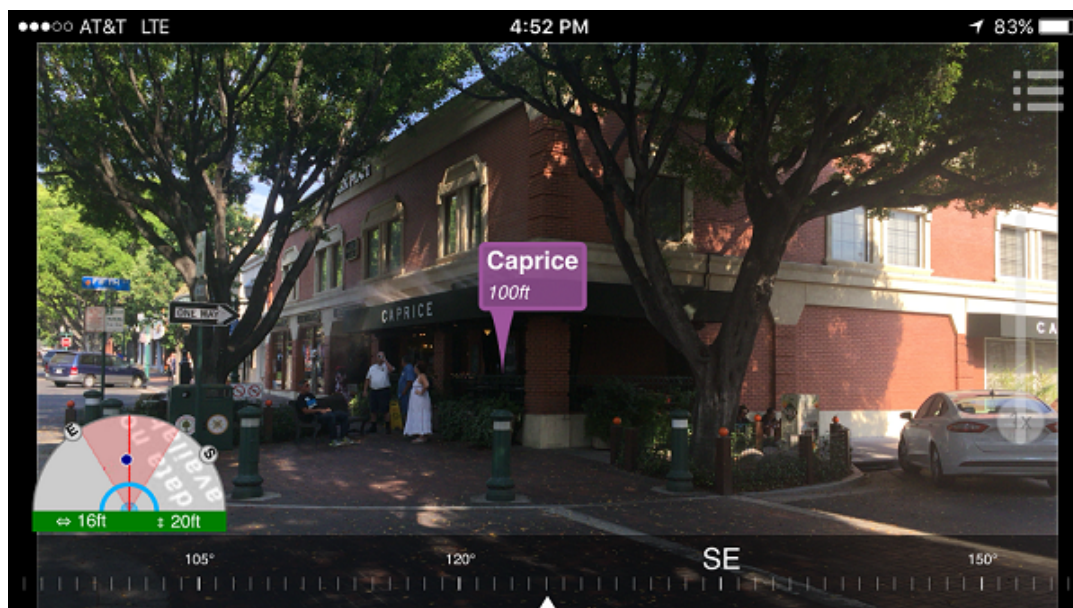


FIGURE 2.1 – AuGeo, exemple d'utilisation

2.1.2 Trimble Connect

Développé par Trimble (cf. page 31), Trimble Connect [2] est une application pour les utilisateurs d'Hololens (cf. page 28) et est utilisée par les entreprises de travaux. Le principe est d'utiliser la réalité mixte³ pour afficher des données 3D virtuelles sur un site permettant notamment aux ouvriers de comprendre et de mieux envisager les travaux à réaliser.

2.1.3 Trimble SiteVision

Toujours développée par Trimble, l'application Trimble SiteVision [3] est celle qui se rapproche le plus de notre projet. Les utilisateurs peuvent visualiser des informations cartographiques en extérieur avec une très haute précision de géolocalisation. Elle peut par exemple être utilisée pour visualiser des futurs bâtiments, des éléments souterrains, etc.

3. La fusion du monde réel et d'un monde virtuel en temps réel, où l'on peut interagir avec les objets virtuels 3D



FIGURE 2.2 – Exemple de Trimble SiteVision

L'application a été développée en utilisant ARCore (cf. page 26) et le Trimble Catalyst (cf. page 31) pour fournir une haute précision de géolocalisation.

2.2 Formats de stockage

2.2.1 Objectifs

On souhaite **définir un format de données généralistes** permettant d'afficher des données 3D géolocalisées. Ce format doit pouvoir **retranscrire des données de plusieurs sources**, de plusieurs bases de données. Par exemple, Mission+ utilise une base de données Spatialite (cf. page 21). L'objectif est d'avoir un format de fichier flexible pour retranscrire toutes les données possibles et léger pour pouvoir être facilement embarqué dans une application de RA.

2.2.2 Les formats existants intéressants

GeoJSON

Le GeoJSON [4] est un format de fichier d'encodage de données géospatiales qui utilise la norme *JSON*⁴. Il permet de décrire des données géographiques sous forme de *Feature* avec :

4. JavaScript Object Notation

- **Une géométrie** qui peut être de type *Point*, *LineString*⁵, *Polygone* ou un ensemble de ces 3 géométries
- **Un ensemble de propriétés** sous la forme *clé* et *valeur*.

Une géométrie peut être de deux dimensions (latitude et longitude) ou de trois dimensions en ajoutant une élévation (ou altitude).

C'est un format **facile à éditer**, il est souvent utilisé pour échanger entre différentes plateformes de traitement de données géographiques. Un autre avantage est qu'il existe déjà un parser⁶ sur Unity.

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [0.0, 0.0, 0.0]
      },
      "properties": {
        "cle" : "valeur"
      }
    }
  ]
}
```

SCRIPT 1 – Exemple de données GeoJSON

L'exemple ci-dessus décrit un point de coordonnées (0.0,0.0,0.0) qui a une propriété *cle* dont la valeur est *valeur*.

Keyhole Markup Language

Basé sur le *XML*⁷, le KML [5] (Keyhole Markup Language) est un format de fichier permettant de traduire des données géographiques dans le standard XML. Le format a été normalisé par l'OGC⁸. Il est notamment **utilisé par Google** dans *Google Earth* ou *Google Maps*. Le format est un peu plus lourd que le GeoJSON et il n'existe pas de parser dans Unity.

5. Un *LineString* aussi appelé en français **une ligne brisée** permet de définir une suite de segments reliant une suite de points

6. Analyseur syntaxique, formalisation du texte

7. Extensible Markup Language

8. Open Geospatial Consortium

PostGIS

PostGIS [6] est **une base de données spatiales** qui permet de stocker et manipuler des objets spatiaux. C'est une extension des bases de données PostgreSQL, un système de gestion de base de données relationnelles et objets. Elle permet donc de stocker des géométries (point, ligne, polygone, etc) où l'on peut indexer les données en fonction de leurs données spatiales et faire des requêtes pour accéder à des objets ou des relations dans notre base.

PostGIS permet donc d'avoir une structure flexible pour faire des analyses sur nos données. On peut ensuite accéder à notre base de données en utilisant *NPGsql* ou *PostgREST* dans Unity, mais l'implémentation est longue et pas forcément adaptée à une application portable comme une application de réalité augmentée.

Spatialite

Spatialite [7] est une extension de SQLite. C'est aussi une base de données géographiques à la différence qu'elle n'est pas basée sur une architecture client/serveur : le moteur SQL est directement **embarqué** dans l'application et la base de données correspond simplement à un fichier. Spatialite est beaucoup plus adaptée à une application portable comme la nôtre, cependant son implémentation dans Unity n'est pas gérée.

ShapeFile

Shapefile est un des formats les plus répandus pour traduire des formes géométriques. Il permet comme les précédents formats de fichiers de traduire tous les types de géométrie. Les principaux problèmes sont qu'il s'agit d'un format multi-fichiers et qu'on est rapidement limité par la taille des données. C'est un format vieillissant malgré le fait qu'il soit toujours l'un des plus utilisés aujourd'hui.

2.3 Méthode de projection

2.3.1 L'intérêt d'une projection

Étant donné que l'on souhaite travailler dans **un repère orthonormé**⁹ pour correspondre au repère Unity, on doit utiliser une projection. Travailler dans un repère orthonormé nous permet de facilement calculer des distances entre les points.

Il faut savoir que les coordonnées géographiques sont souvent données en degré notamment par le système GPS¹⁰ où la latitude représente le positionnement nord ou sud d'un point sur Terre par rapport à l'équateur et la longitude le positionnement Est ou Ouest par rapport à un méridien de référence. Cette projection s'appelle le **WGS84** et est très utilisée par les fournisseurs de données de géolocalisation.

Une projection peut nous permettre de travailler dans un **système décimal** (métrique) afin de faciliter les calculs par la suite. Il faut que la projection soit dite *conforme*, c'est-à-dire qu'elle respecte les angles pour éviter des artefacts au niveau de l'orientation entre nos données et notre caméra.

2.3.2 Les projections adaptées

Universal Transverse Mercator

Universal Transverse Mercator (UTM) [8] ou Transverse Universelle de Mercator est une projection dite **cylindrique** : on va chercher à projeter la surface de la terre sur un cylindre que l'on va ensuite dérouler pour former un plan en deux dimensions du globe.

9. un espace vectorielle constituée de vecteurs de norme 1 et orthogonaux

10. Global Positioning System, en français Système mondial de positionnement est le système de positionnement par satellites américains

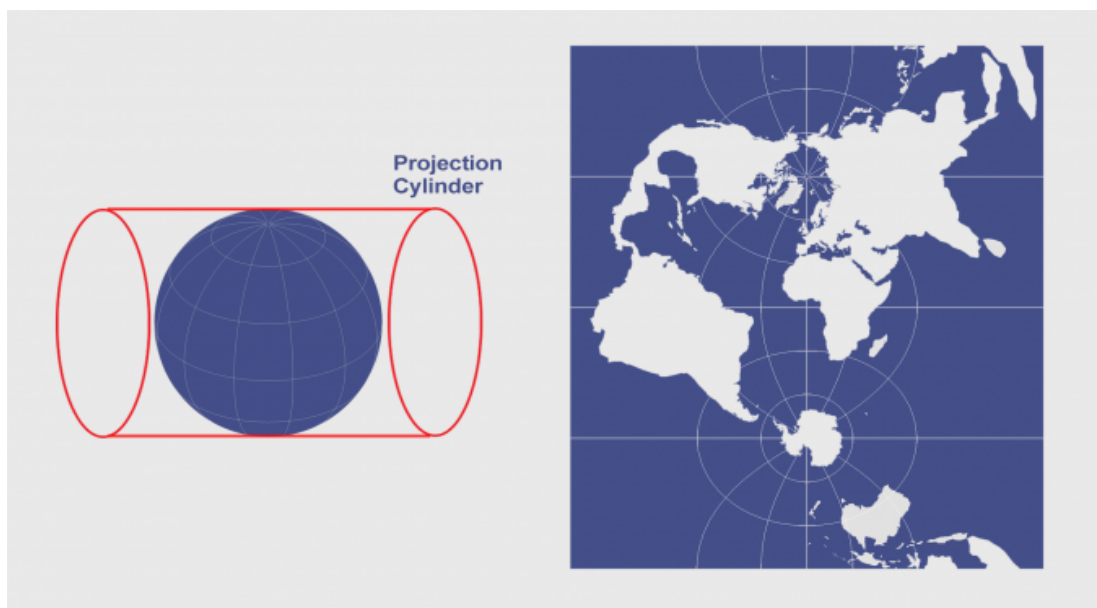


FIGURE 2.3 – La projection cylindrique UTM

Cette projection est conforme et possède un système métrique. Il s'agit de la projection **la plus utilisée dans le monde**. L'objectif est de découper le globe en 60 zones ou fuseaux de 6 degrés d'amplitude en longitude. La France est alors présente sur 3 fuseaux :

- **UTM Nord fuseau 30** : entre 6 degrés Ouest et 0 degré de Greenwich
- **UTM Nord fuseau 31** : entre 0 degré et 6 degrés Est de Greenwich
- **UTM Nord fuseau 32** : entre 6 degrés Est et 12 degrés Est de Greenwich

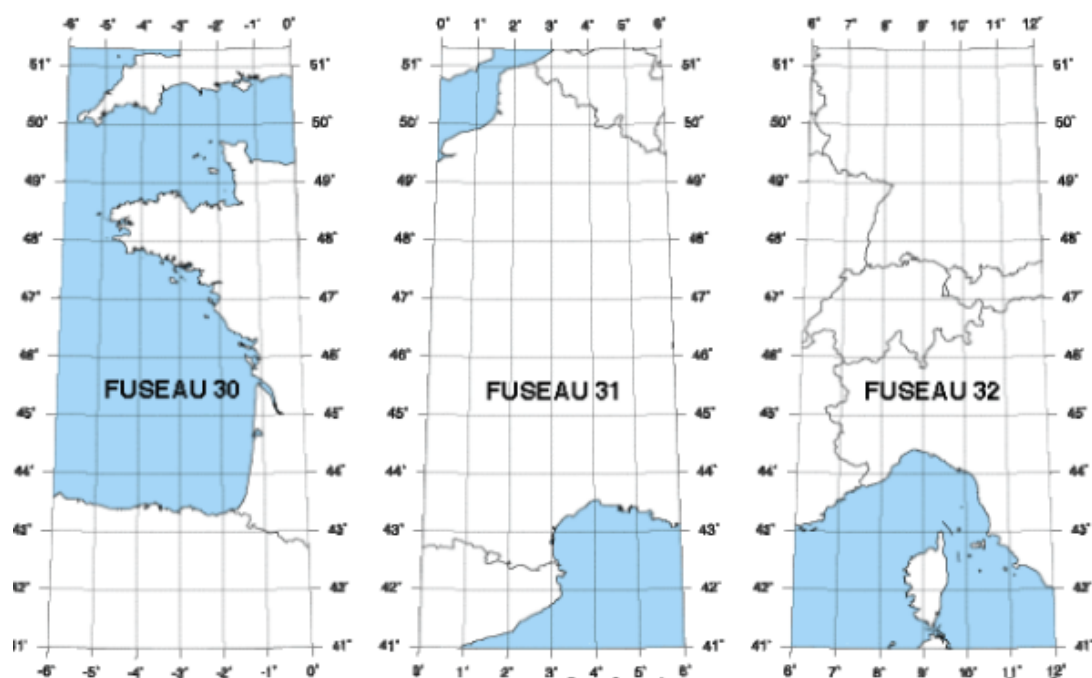


FIGURE 2.4 – La projection UTM en France

Un point en UTM est donné par 2 ou 3 valeurs :

- UTM E,
- UTM N,
- et l'altitude (facultatif).

Prenons l'exemple du Col de Louron qui a comme valeur (31 292 629, 4747 172, 2001). Le premier nombre de la première donnée (UTM E) représente le fuseau (ou zone) sur lequel on travaille, dans notre cas 31. Ensuite, on donne ce qu'on appelle les cadrants UTM : des carrés de 1 km x 1 km en partant du coin inférieur gauche du fuseau. Le premier nombre de l'UTM N (deuxième donnée) et le second de l'UTM E nous donnent les coordonnées du cadrant. Dans notre cas il s'agit du cadrant (292, 4747). Enfin les derniers nombres de l'UTM N et E nous donnent la position de notre col en mètres sur le cadrant : (629, 172). La dernière donnée nous donne l'altitude, dans notre cas le col fait 2001 mètres d'altitude.

Ce découpage permet d'avoir des erreurs d'échelle n'excédant pas 0.1% dans une même zone. Le plus gros problème est qu'on a de grosses erreurs de calculs à partir du moment où l'on change de zone. Étant donné que pour notre application nous ne travaillons que dans des repères locaux (maximum 1 kilomètre de rayon), UTM semble être une bonne solution à notre problème.

Lambert 93

La projection de Lambert [9] [10] est une projection dite conique conforme. On cherche cette fois à représenter la surface du globe sur un cône. Les méridiens sont des droites (longitude constante) et les parallèles sont alors des arcs de cercle (latitude constante).

C'est la projection officielle française, elle est aussi utilisée dans d'autres pays comme la Belgique et l'Estonie. Chaque projection est liée à **un système géodésique** : un système de référence des positions de la Terre. Ce système repose sur un ellipsoïde de révolution qui doit représenter la forme de la Terre. Le principal intérêt du Lambert 93 est son système géodésique RGF93 qui est compatible avec le WGS84, le système géodésique utilisé par le GPS.

Le problème de cette projection est **la déformation linéaire** qui est assez importante. On peut atteindre un mètre d'imprécision au Nord et au Sud de la France.

2.4 Outils de Réalité Augmentée

2.4.1 Objectifs

On souhaite étudier les différents outils disponibles pour faire de la réalité augmentée. Ces outils vont aussi souvent déterminer l'appareil qu'on va ensuite utiliser. On cherche un outil permettant de faire du **tracking** en réalité augmentée pour déterminer l'environnement dans lequel se trouve l'utilisateur. Le tracking nous permettra notamment de détecter le sol et déterminer où afficher nos objets. On cherche aussi un outil qui implémente des méthodes permettant de faire de la réalité augmentée géolocalisée. Étant donné que l'on n'a pas d'interaction avec les objets, l'appareil à utiliser ne doit pas être une contrainte. On peut tout autant envisager d'utiliser un casque qu'un smartphone ou une tablette. De plus la plateforme (Apple, Google ou Windows) n'est pas une contrainte non plus, on cherche à avoir **le meilleur résultat possible**. La seule contrainte que l'on impose est qu'il existe une implémentation dans Unity.

D'autres technologies de réalité augmentée ont été étudié lors de mon stage comme notamment le casque de Magic Leap mais ces technologies ne correspondaient pas au projet et n'ont pas été développé dans ce rapport.

2.4.2 Les outils envisageables

ARCore

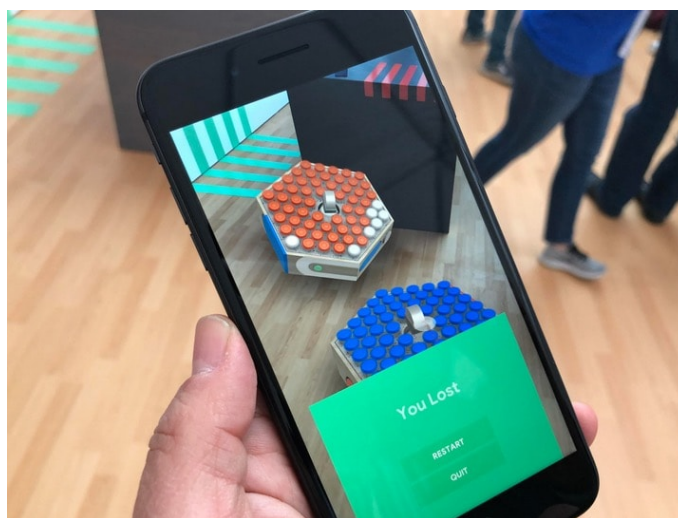


FIGURE 2.5 – Exemple d'application ARCore

ARcore [11] est le SDK¹¹ de réalité augmentée développé par **Google**. Il permet de développer des applications en RA pour les téléphones et tablettes Android¹² et iOS¹³ récents. Il implémente des outils de mapping et de détection de surface. De plus il existe une version pour Unity. L'utilisation de cet outil de Google implique la facilité d'implémentation des autres outils de Google comme celle de Google Maps qui permet de faire la cartographie.

ARKit

ARKit [14] est le SDK d'Apple de réalité augmentée.

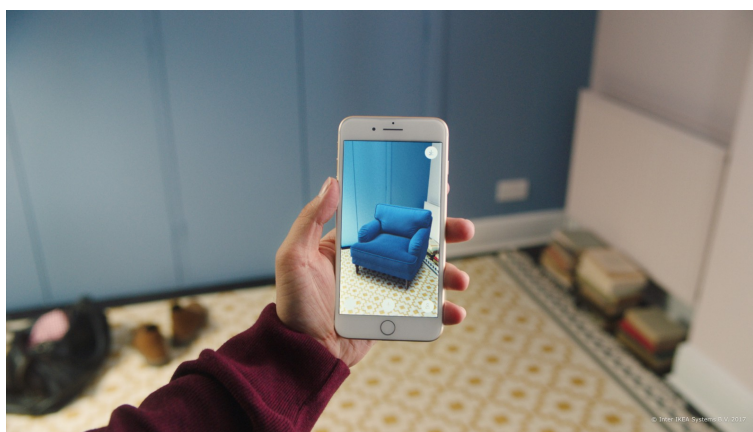


FIGURE 2.6 – Exemple d'utilisation d'ARKit

Il existe une version pour Unity et quelques projets [15] de réalité augmentée géolocalisée utilisant ce SDK ont été réalisés. Il n'existe pas non plus de méthodes ou d'outils permettant de faire de la RA géolocalisée.

Les deux SDK présentés ci-dessus sont souvent comparés et il est compliqué de déterminer lequel est le plus efficace. Cependant, il faut savoir qu'ARKit existe depuis plus longtemps et qu'il a donc été plus souvent utilisé que son concurrent de Google.

11. Software Development Kit, kit de développement logiciel. C'est un ensemble d'outils logiciels destinés aux développeurs pour faciliter le développement sur une certaine plateforme.

12. Système d'exploitation mobile de Google

13. Système d'exploitation mobile d'Apple

Wikitude

Le SDK Wikitude est l'un des seuls à implémenter des méthodes de RA géolocalisée. Il permet d'ajouter facilement des points d'intérêt dans notre application à une certaine géolocalisation [16]. Cependant, ces méthodes ne sont pas encore disponibles pour Unity et il coûte 3000€ par an.

Vuforia

Vuforia est un SDK très répandu dans le domaine de la réalité augmentée et l'entreprise est en partenariat avec Unity. Il implémente selon l'appareil qu'on utilise ARCore ou ARKit ou utilise son propre moteur d'AR si ni l'un ni l'autre n'est disponible sur l'appareil utilisé. L'outil est plutôt utilisé pour faire de la réalité avec marqueur : on va afficher un objet en réalité augmentée sur un marqueur.



FIGURE 2.7 – Vuforia : exemple de RA par marqueur

Ce marqueur est ajouté dans une base de données et analysé par Vuforia pour en extraire des points d'intérêt (points de Harris). La caméra va alors chercher ces points d'intérêt dans l'espace pour afficher un modèle 3D.

8th Wall

8th Wall est sensiblement la même chose que Vuforia à la différence qu'on peut directement l'implémenter pour une application Web. Il permet lui aussi de faire de la réalité

augmentée sans utiliser ARCore ou ARKit lorsqu'ils ne sont pas disponibles sur l'appareil que l'on utilise. Il existe aussi une méthode *WorldPoint* qui n'est pas très complète et qui n'est pas très développée.

KudanSLAM

KudanSLAM est le second SDK que j'ai pu étudier qui implémente des méthodes de réalité augmentée géolocalisée. Il existe un prototype qui implémente ces méthodes [17]. Cependant, Kudan n'est pas beaucoup utilisé et semble vieillissant.

Hololens

Hololens est très largement le casque le plus prometteur dans le domaine de la réalité augmentée.

Il a été créé par Windows et il existe aujourd'hui une version numéro 2 qui corrige la plupart des défauts de la version 1, notamment sur le champ de vision.

Un des réels intérêts du casque est son tracking et sa capacité **d'occlusion**, c'est-à-dire la capacité de déterminer s'il faut cacher ou montrer des objets virtuels devant ou derrière un plan (par exemple une personne qui traverse notre champ de vue). Hololens est

réellement intéressant pour faire de la **réalité mixte**, c'est-à-dire définir des interactions avec les objets, notamment avec les gestes de la main ou avec la voix. Il existe un outil nommé MRTK (Mixed Reality Tool Kit) qui permet de créer des applications pour Hololens sur Unity.



FIGURE 2.8 – Windows Hololens

2.5 Aide à la précision

2.5.1 Objectifs et intérêts

On souhaite réaliser une application de réalité augmentée géolocalisée, nous avons donc besoin d'outils permettant d'obtenir notre géolocalisation et l'orientation de notre caméra. Dans le cas notamment d'une application mobile, nous pourrions utiliser le GPS et la boussole du téléphone. Cependant, ces outils ne garantissent pas une bonne précision.

En effet, la précision du GPS d'un téléphone aujourd'hui est d'environ 5 mètres à l'horizontale et l'imprécision est trois fois plus importante verticalement. Une telle imprécision n'est pas envisageable dans le cas de notre application, nous avons donc besoin d'outils et de méthodes nous permettant d'assurer une meilleure précision.

2.5.2 Les outils potentiels

Real-Time Kinematic et Precise Point Positioning

RTK (Real-Time Kinematic ou Cinématique temps réel en français) et le PPP (Precise Point Positioning) sont des techniques de positionnement par satellite.

RTK va utiliser les mesures de phase des ondes envoyées par les satellites pour améliorer la précision de la géolocalisation en allant jusqu'à **une précision au centimètre**. Le signal envoyé par le satellite va être comparé à une copie interne de signal. Comme les horloges de notre application et du satellite sont synchronisées, il existe un décalage dû au temps de parcours de l'onde. RTK va utiliser ce décalage pour déterminer une géolocalisation plus précise.

Le PPP va lui aussi utiliser la mesure de phase et il va aussi utiliser une description précise des orbites et des horloges des satellites GNSS¹⁴, plus précise que celles définies dans le message de navigation. Son avantage est qu'il n'a pas besoin de station de base sur laquelle se référencer mais son temps de convergence est plus long. Le CNES¹⁵ a développé une méthode qui l'implémente nommée PPP-Wizard [18] et propose un package¹⁶ pour le développer.

Ces deux méthodes peuvent être implémentées sur Android [19] à l'aide de RTKLib mais l'implémentation demande du temps.

Mapbox

Mapbox est un outil cartographique, concurrent de Google Maps, utilisé la plupart du temps pour afficher des positions sur des cartes dans le cadre d'applications web. Cependant, Mapbox développe de nombreux outils sur Unity, notamment pour faire de la réalité augmentée [20] sur ARCore et ARKit. Il existe un projet permettant d'**aligner notre vision aux données cartographiques** de Mapbox. Ce projet permet alors d'améliorer la précision

14. Système de positionnement par satellites, Global Navigation Satellite System

15. Centre National d'Études Spatiales

16. Un ensemble de logiciels

de l'orientation de notre application. Deux méthodes sont développées, la première dite *manuelle* demande à l'utilisateur d'aligner sur son écran mobile la carte 3D à son environnement.



FIGURE 2.9 – Mapbox - Alignement manuel RA

La méthode manuelle est intéressante mais n'est pas très intuitive pour l'utilisateur et ne donne pas une précision très exacte. Elle demande aussi d'avoir des repères visuels autour de l'utilisateur pour pouvoir aligner son environnement à la carte.

La deuxième méthode, dite *automatique*, va placer des points où l'utilisateur est passé pendant l'exécution de l'application. À partir de ces points, on va pouvoir déterminer la différence d'orientation entre le monde réel et notre caméra dans Unity. Une méthode semblable a d'ailleurs été développée par mon tuteur de stage Raphaël Guenon pour le projet **RA-LS**.

En théorie cette méthode semble fonctionnelle, cependant elle demande plusieurs paramètres. Le premier est qu'il faut demander à l'utilisateur de se déplacer pour pouvoir poser des points dans l'espace, on ne peut pas utiliser cette méthode directement au lancement de l'application. Le second est qu'il faut une bonne précision de géolocalisation pour connaître exactement la position des points que l'on place. Cette méthode nécessite un certain nombre de points afin d'avoir une précision acceptable.

Mapbox est gratuit pendant un certain temps mais devient vite coûteux. Ces deux méthodes peuvent être implémentées à la main en dehors de Mapbox (en utilisant des données d'une autre source pour la méthode manuelle).

Satlab

Satlab est un fournisseur de solution de géolocalisation. Cette société vend du matériel permettant de fournir une localisation très précise. Parmi les outils proposés, elle vend un récepteur multi-GNSS qui implémente le **RTK** et compatible Android [21]. Il fournit les positions de **plusieurs constellations** satellites comme GPS, Galileo¹⁷ ou GLONASS¹⁸. Lorsque le RTK est disponible, il assure une précision au centimètre près. On peut ensuite accéder aux données de positionnement par bluetooth sur notre appareil Android. Il faut compter environ 4500€ pour acheter le produit plus un abonnement mensuel.

Ce produit a été utilisé pour le projet RTE par mon tuteur de stage. D'après son expérience, son implémentation a été complexe et fastidieuse.



FIGURE 2.10 – Satlab SLC

Trimble

Trimble est lui aussi un fournisseur de solutions de géolocalisation. Cette entreprise américaine spécialisée dans les logiciels géo-spatiaux développe beaucoup d'applications de réalité augmentée comme notamment les applications Trimble SiteVision (cf. page 18) ou Trimble Connect (cf. page 18). L'application SiteVision utilise le **Trimble Catalyst**.

Cet outil permet lui aussi de capter plusieurs constellations de satellites et implémente le RTK. Les avantages du Trimble Catalyst en comparaison avec Satlab sont les facilités d'implémentation et d'accès aux données sur Android. Le Catalyst se branche par



FIGURE 2.11 – Le Trimble Catalyst

17. Système de positionnement par satellites développé par l'Union européenne

18. Système de positionnement par satellites russes

USB à notre tablette ou smartphone Android et à partir des APIs¹⁹ fournis par Trimble, on accède aux données de géolocalisation. De plus le Catalyst coûte environ 350€ mais l'abonnement mensuel est un peu plus cher. Cependant, le Trimble Catalyst ne fournit pas encore une précision au centimètre près dans le sud de la France mais plutôt à 30 cm près. Ce service devrait être développé dans les mois, années à venir.

19. interface de programmation applicative

Chapitre 3

Rapport de conception

La conception est une partie essentielle dans la réalisation d'un prototype. J'ai défini les outils que j'allais utiliser pour mon prototype et les fonctionnalités de mon application pour ensuite pouvoir séparer l'application en modules. Chaque module de l'application est alors découpé en tâches que l'on va chiffrer pour déterminer un planning prévisionnel.

3.1 Choix des technologies

Avant de définir les modules et les fonctionnalités de notre application, il a fallu faire un choix des technologies à utiliser. Pour ça, j'ai réalisé une présentation le Mercredi 6 Mars en présence de mon tuteur de stage Raphael Guenon et de Nicolas Moreau, engagement manager chez Capgemni DEMS, une présentation de mes choix d'implémentation du prototype et des outils à utiliser et disponibles dans l'entreprise.

3.1.1 Réception et stockage des données

Pour déterminer le format de stockage j'ai répertorié les avantages et les inconvénients des trois formats qui me semblaient les plus intéressants : GeoJSON, Spatialite et PostGIS.

Format	GeoJSON	Spatialite	PostGIS
Descriptif	Fichier JSON permettant de stocker des données géographiques + un ensemble de propriétés	Extension de SQLite permettant de gérer des données géographiques. N'est pas basé sur une architecture client/serveur : Moteur SQL embarqué, et la BD dans un seul fichier.	Extension de PostgreSQL qui permet de stocker et manipuler des objets spatiaux.
Avantages	-Facile à intégrer dans d'autres systèmes. -Facile à parser et à intégrer dans Unity. -Rapide et léger.	-Pratique pour les systèmes embarqués. -Facile de visualiser les données. -Léger, pratique pour une seule appli/dispositif	-Permet de faire beaucoup de traitements, indexer etc -Structure flexible pour faire des analyses -On peut l'implémenter dans Unity avec NPGSql ou PostgREST
Désavantages	-Ne supporte pas les types Date. -Les positions n'ont pas d'attributs. -Pas adapté au streaming.	-Limité par la quantité et la taille -Dépendant d'autres librairies dans SQLite -Implémentation lourde dans Unity (déjà gérée par Edouard)	-Lourd -Pas forcément nécessaire pour notre application

FIGURE 3.1 – Tableau présentant les formats envisageables

Mon choix s'est alors porté vers le **GeoJSON** pour sa simplicité et sa légèreté. En effet, son intégration dans Unity est déjà géré par un parser existant [23] et il est adapté au type de projet qu'on souhaite réaliser.

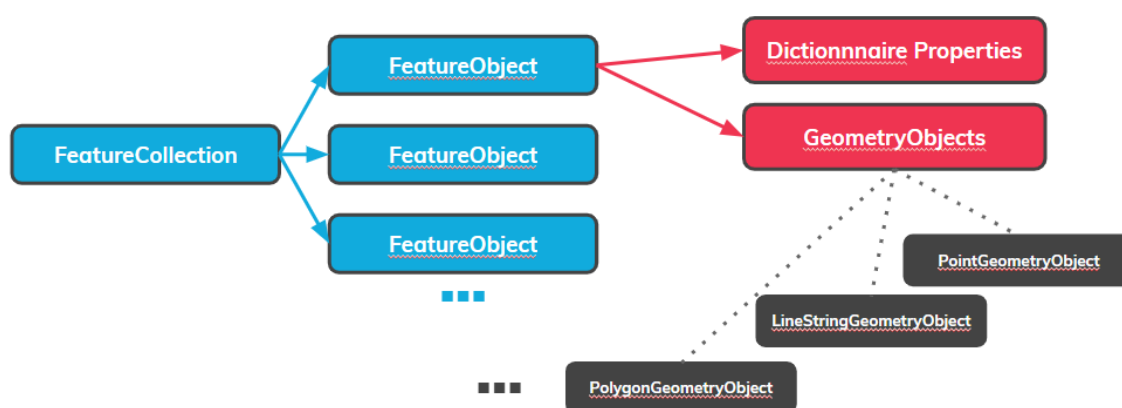


FIGURE 3.2 – Schéma de la structure des données dans Unity

Lorsqu'un GeoJSON est parsé, on obtient un objet *FeatureCollection*. Cette *FeatureCollection* contient des *FeatureObjects* qui représentent les différents objets de notre GeoJSON. Chaque *FeatureObject* possède un dictionnaire de *properties* qui représentent les propriétés de notre élément géolocalisé (cf. page 19) et un *GeometryObject* qui est soit de type *PointGeometryObject*, *LineStringGeometryObject*, etc... contenant la géométrie de notre élément

géolocalisé.

Il a fallu ensuite déterminer le modèle de projection à utiliser pour afficher nos données dans Unity. Étant donné que nous travaillons en réalité augmentée, notre utilisateur ne compte pas parcourir des kilomètres lors de l'utilisation de l'application. De plus, il est important d'utiliser un repère orthonormé pour pouvoir plus simplement calculer des distances et pour pouvoir l'intégrer dans Unity. En prenant en compte ces paramètres, mon choix de méthode de projection s'est alors porté vers l'**UTM**. Le premier avantage de l'UTM est qu'il est utilisable dans le monde entier (sauf sur les pôles). Le second est qu'il s'adapte parfaitement à notre type d'application et à notre environnement : Unity. En effet, l'UTM n'altère que très peu le positionnement lorsqu'on ne change pas de zone et les positions sont fournies en mètre, idéal pour l'implémentation dans Unity où l'on va toujours travailler en fonction de la position de l'utilisateur, dans un repère local.

3.1.2 Gestion et architecture des données

Pour cette application, il faut prendre en compte que plusieurs types de données seront présentes lors de l'affichage et qu'elles proviennent de plusieurs sources. Étant donné la flexibilité du GeoJSON, nous devons imposer des conventions pour permettre à notre application de pouvoir retrouver les informations nécessaires lors de l'insertion des données. Il a alors fallu se demander comment gérer ces couches d'informations et comment les représenter.

On définit alors la notion de **Layer**. Un Layer est **un fichier GeoJSON et l'apparence que l'on souhaite lui donner**. Par convention, un Layer ne contient qu'un seul type de géométrie : des points, des segments, des polygones, etc. Par exemple, on peut imaginer un Layer contenant les lignes souterraines RTE qui est lié à un fichier GeoJSON et qui contient des lignes.

On va alors renommer nos fichiers GeoJSON des **JSONCollection** : Un fichier GeoJSON contenant un seul type de géométrie et contenant des propriétés nous permettant de mieux envisager l'implémentation dans le projet, comme notamment des boîtes englobantes c'est-à-dire les coordonnées les plus petites et les plus grandes de la collection.

```

{
  "type": "FeatureCollection",
  "ellipsoide": "IAG GRS 80",
  "coord_type": "WGS84",
  "first_coord_lat": "0.0",
  "first_coord_long": "0.0",
  "last_coord_lat": "45.0",
  "last_coord_long": "45.0",
  "features":
  [
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [[0.0, 0.0, 0.0], [33.0, 32.0, 0.0], [45.0, 45.0, 0.0]]
      },
      "properties": {
        "first_coord_lat": "0.0",
        "first_coord_long": "0.0",
        "last_coord_lat": "45.0",
        "last_coord_long": "45.0"
      }
    },
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [[43.21, 2.34, -1], [44.91, 8.07, -1], [14.60, 47.12, -1]]
      },
      "properties": {
        "first_coord_lat": "14.60",
        "first_coord_long": "2.34",
        "last_coord_lat": "43.21",
        "last_coord_long": "47.12"
      }
    }
  ]
}

```

SCRIPT 2 – Un fichier GeoJSON type appelé JSONCollection

Les boîtes englobantes sont définies à l'aide des valeurs de *first_coord_lat*, *first_coord_long*, *last_coord_lat* et *last_coord_long*. On va aussi stocker le type de coordonnées utilisé dans les données et l'ellipsoïde associé.

Pour définir son apparence, on va définir un *Prefab* à notre Layer. Un *Prefab* est un objet de Unity (GameObject) qui contient différents arguments et fonctions, et qui peut être instancié dans une scène Unity. On va alors définir un *Prefab* au préalable contenant l'apparence et les arguments que l'on souhaite donner à notre Layer et instancier nos *Prefabs* selon les positions qui sont contenues dans notre *JSONCollection*.

Pour gérer nos différents Layers dans notre application, on va utiliser un ou plusieurs fichiers de configuration qui contiendront les informations nécessaires. Un fichier de confi-

guration est un fichier JSON qui peut contenir un ou plusieurs Layers.

```
[
  {
    "name" : "Lignes souterraines RTE de 150kV",
    "filePath" : "lignes-souterraines-rte-150kV",
    "maxRenderDistance" : 500,
    "isActive" : true,
    "geometryType" : "LineString",
    "prefabName" : "CylinderWithPoints"
  },
  {
    "name" : "Aéroport de Mission Plus",
    "filePath" : "missionplus_airport",
    "maxRenderDistance" : 500,
    "isActive" : true,
    "geometryType" : "Point",
    "prefabName" : "GreenSphere"
  },
  {
    "name" : "Enceintes de poste RTE",
    "filePath" : "enceintes-poste-rte",
    "maxRenderDistance" : 500,
    "isActive" : false,
    "geometryType" : "Polygon",
    "prefabName" : "BasicPolygon"
  }
]
```

SCRIPT 3 – Exemple de fichier de configuration

Chaque élément de notre JSON contient une couche d'informations, soit un Layer. Il a comme propriétés :

- *name* : le nom que l'on souhaite lui donner, afficher,
- *filePath* : le nom du fichier JSONCollection associé,
- *maxRenderDistance* : la distance maximale à partir de laquelle l'utilisateur peut voir les objets du Layer,
- *isActive* : si l'on souhaite l'activer, c'est à dire l'afficher ou non,
- *geometryType* : le type de géométrie dont il s'agit,
- *prefabName* : et le nom du prefab que l'on va charger dans la scène.

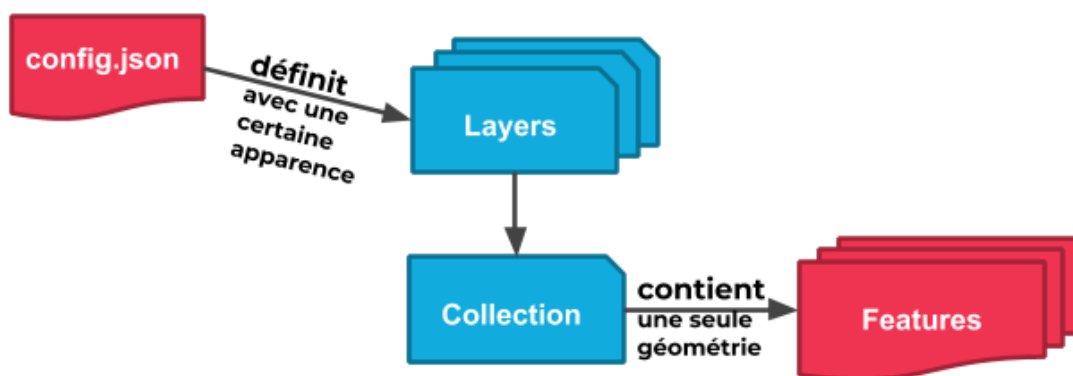


FIGURE 3.3 – Processus de gestion des données

Pour résumer, on part du fichier de configuration *config.json* qui contient un tableau : chaque élément du tableau contient une définition d'un Layer, c'est-à-dire : Le nom du fichier JSON-Collection à charger contenant les éléments cartographiques et l'apparence qu'on souhaite lui donner. Un Layer ne doit traduire qu'un seul type de géométrie.

Les données et les fichiers de configuration sont stockés dans le dossier *Assets/Resources* de notre projet Unity. Il s'agit d'un dossier spécifique de Unity qui permet d'accéder à des données (*Assets*) pendant l'exécution de notre application avec la fonction *Resources.Load*. Je sépare les données, les apparences et les configs dans des sous-répertoires comme traduit dans le schéma ci-dessous.

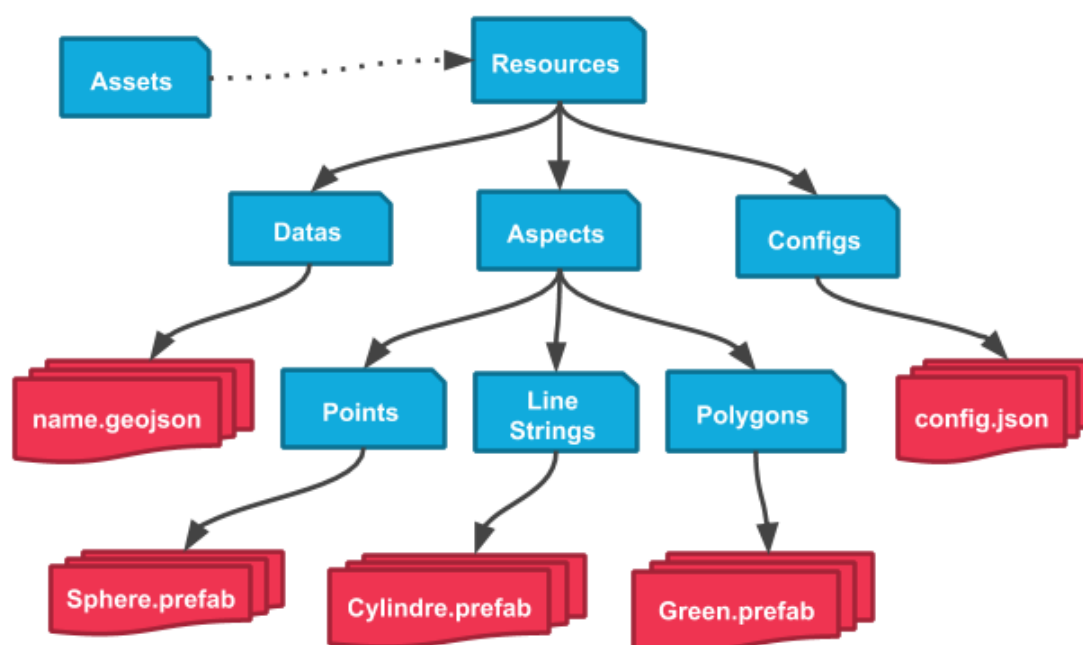


FIGURE 3.4 – Architecture des données

3.1.3 Outils de Réalité Augmentée

Parmi les outils de réalité augmentée présentés dans l'état de l'art, 4 ont été envisagés : ARCore (cf. page 26), ARKit (cf. page 26), Wikitude (cf. page 27) et Hololens (cf. page 28). Pour choisir celui que j'allais utiliser j'ai alors cherché à les comparer selon 4 critères :

- la précision du tracking,
- s'il implémente un module de réalité augmentée géolocalisée,
- s'il existe une implémentation dans Unity
- et son coût.

		Précision	geoAR	Unity	Coût
ARKit		-	Non prévue	Plugin	Prix de l'appareil
ARCore		+	Non prévue	SDK	Prix de l'appareil
Wikitude		-	Prévue	Pas GeoAR	3000 € / yr
Microsoft HoloLens		++	Non prévue	MRTK	Prix de l'appareil

FIGURE 3.5 – Tableau comparatif des outils de RA

Pour la précision du tracking, on peut facilement dire que Hololens fournit un meilleur service. Wikitude implémente ARKit ou ARCore selon l'appareil qu'on utilise cependant il est toujours mieux d'utiliser ces technologies natives. La comparaison du tracking entre ARCore et ARKit est compliquée : ARKit est plus ancien et était au début disponible sur plus d'appareils qu'ARCore (environ 450 millions pour Apple et 100 millions pour Google en 2017). Il y a donc en Septembre 2018 plus d'utilisateurs d'ARKit que d'ARCore. De plus, le nombre de projets sur Github¹ et le nombre de questions sur Stackoverflow², qui sont des indicateurs de développement important, sont supérieurs [12] pour ARKit en 2018. Cependant, il reste difficile de déterminer lequel des deux SDK est le plus performant.

Certains articles [24] définissent ARCore comme meilleur par notamment sa capacité à gar-

1. Service d'hébergement et de gestion de développement en ligne

2. Site de question/réponse sur la programmation informatique

der en mémoire des grandes images qui lui permet de moins perdre le tracking. Un autre avantage est qu'ARCore est multi-plateforme (on peut l'utiliser sur un appareil Android et un appareil iOS adapté).

Le seul SDK qui implémente un module de RA géolocalisée est Wikitude. Cependant, il n'est pas disponible sur Unity pour l'instant et le SDK est aussi le seul payant (3000€ par an).

Après cette démarche, mes choix se sont portés sur le **Microsoft Hololens et ARCore**. Pour essayer de déterminer sur lequel des deux je souhaitais travailler, je me suis alors demandé lequel des deux appareils (casque ou tablette / smartphone) était le mieux adapté pour faire de la réalité augmentée géolocalisée et lequel des deux me prendrait le moins de temps pour développer mon prototype. Dans ma conception de la réalité augmentée géolocalisée et en vue de la flexibilité de notre projet, un appareil mobile comme une tablette me semble plus adapté de part sa facilité de transport. En effet, une application de réalité augmentée comme la nôtre demande à l'utilisateur d'être à l'extérieur (notamment pour la géolocalisation) et de se déplacer dans son environnement pour visualiser les objets 3D. On a pu voir cependant que des projets similaires comme le Trimble Connect (cf. page 18) font la même chose en utilisant Hololens. Mais le réel intérêt d'utiliser un casque dans ce projet était de permettre notamment aux ouvriers de libérer l'usage de leurs mains. Étant donné que notre prototype propose simplement une visualisation et pas d'interaction de réalité mixte, on privilégie un usage pratique et léger. De plus, l'implémentation d'ARCore me semble de manière générale moins complexe.

3.1.4 Amélioration de la précision

Pour la géolocalisation, 2 solutions sont possibles : utiliser un appareil externe qui nous fournira des données sur notre position ou développer une méthode dans notre application avec la librairie RTKLib (cf. page 29). Pour choisir la solution on va évaluer sa précision, le temps de convergence pour arriver à cette précision, le temps de développement, son coût et sa maniabilité.

	Précision	Convergence	Temps	Coût	Maniabilité
Développement d'une solution	30 cm	+15min	--	0	Mobile ou Tablette
Appareil externe	1-5 cm	+	+	Prix du device	Mobile ou Tablette + Device

FIGURE 3.6 – Tableau comparatif des solutions de géolocalisation

En développant notre propre solution, nous n'aurions besoin que de notre appareil (tablette ou smartphone) et le coût serait de zéro. Cependant, la précision et le temps de convergence seraient moins bons et le temps de développement beaucoup plus long. Nous avons donc fait le choix d'utiliser un appareil externe. Étant donné que le projet RA-LS a été développé avec le Satlab SLC (cf. page 31), j'ai pu développer une solution de géolocalisation à partir de cet appareil.

Pour l'orientation, l'objectif est d'implémenter les deux méthodes développées par Mapbox à la main (sans utiliser Mapbox), c'est-à-dire proposer à l'utilisateur la possibilité de s'aligner manuellement à une carte 3D (cf. page 29) et la seconde méthode dite automatique qu'on appellera *Landmarks Orientation Provider* (cf. page 30).

3.1.5 Récapitulatif

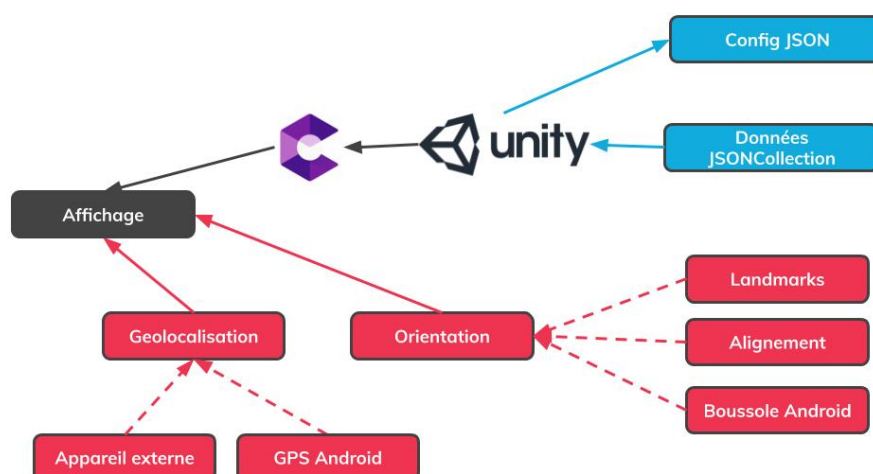


FIGURE 3.7 – Récapitulatif des solutions choisies

Pour résumer, on va utiliser pour stocker nos données le format **GeoJSON** qu'on va structurer et appeler **JSONCollection**. Il faudra pour cela définir un module permettant de passer d'un fichier GeoJSON à une **JSONCollection**. Pour gérer nos couches d'information, on utilisera des fichiers de configuration JSON. **ARCore** est la technologie choisie pour afficher nos objets en Réalité Augmentée sur le moteur de jeu Unity3D. De plus, pour obtenir une meilleure précision que la boussole et le GPS Android, on va utiliser **un appareil externe** qui nous permettra d'assurer une meilleure géolocalisation et **Landmarks Orientation Provider** pour l'orientation. On offrira aussi la possibilité à l'utilisateur de **s'aligner manuellement avec une carte 3D**.

3.2 Les fonctionnalités

On cherche à définir ce que doit faire l'application et ce qu'elle doit permettre à l'utilisateur de voir et de faire. Pour ça, on va établir une liste des fonctionnalités afin de déterminer ensuite des cas d'utilisation.

3.2.1 Liste des fonctionnalités

On sépare en deux parties la liste des fonctionnalités (cf. en annexe page 87). La première est la liste des fonctionnalités essentielles pour obtenir une application qui affiche nos objets en réalité augmentée selon une certaine géolocalisation. La seconde liste décrit des fonctionnalités qui donnent à l'utilisateur plus de possibilités, notamment pour la gestion des données.

On sépare la liste des fonctionnalités en deux parties : une partie pour l'application et une partie pour le module externe qui permettra de passer de n'importe quel fichier à un fichier **JSONCollection**.

3.2.2 Cas d'utilisation

On obtient ensuite le cas d'utilisation en annexe (cf. en annexe page 89) qui répertorie les interactions possibles pour l'utilisateur. Les ellipses représentent des actions et les traits les différentes interactions possibles pour l'utilisateur.

3.3 Les modules de l'application

L'objectif ensuite a été de séparer les fonctionnalités dans des modules de l'application. Chaque module doit pouvoir être séparé du reste de l'application. Cette séparation permet au prototype d'être plus facilement extensible : on peut facilement remplacer un module sans devoir modifier les autres modules.

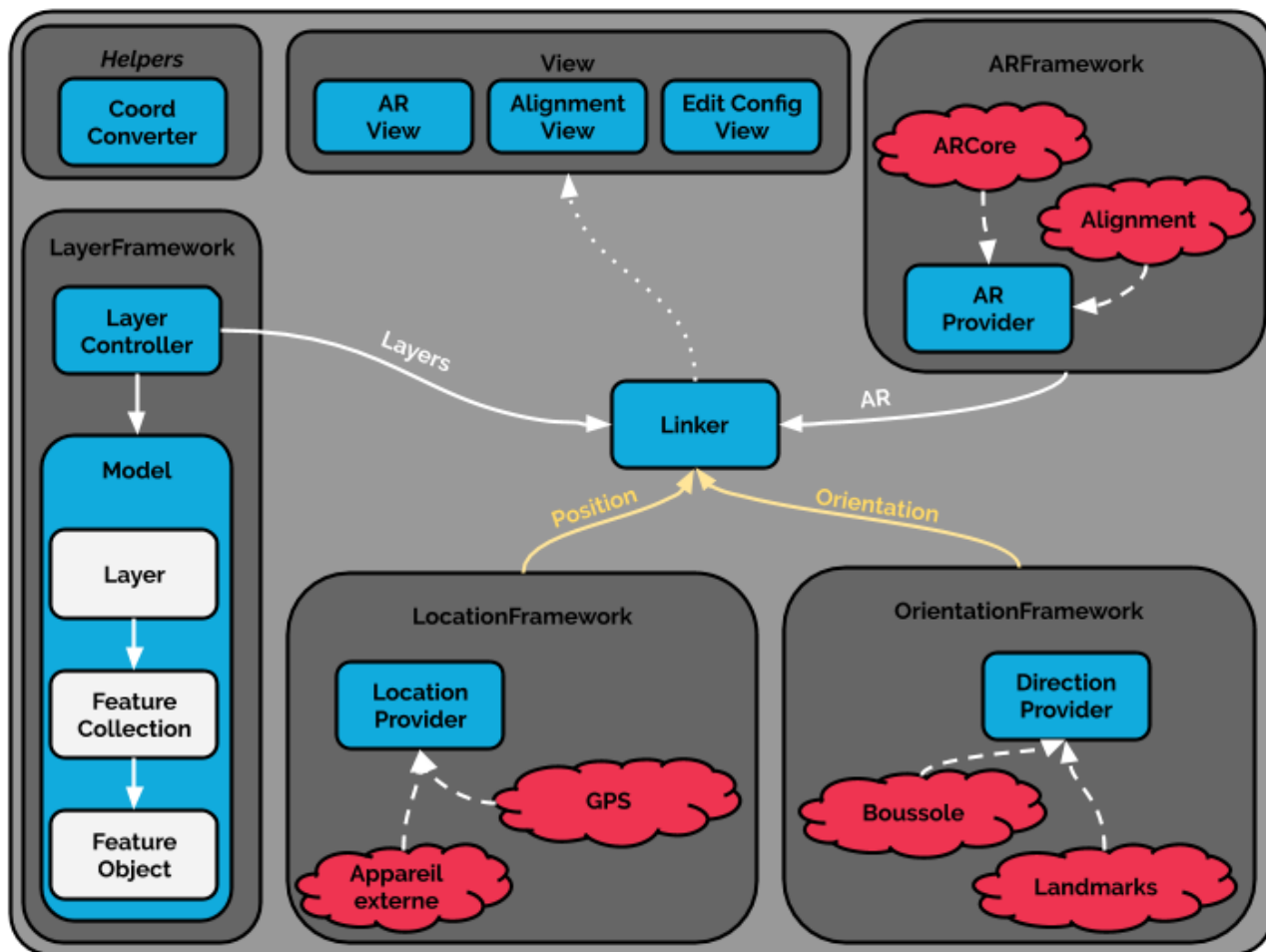


FIGURE 3.8 – Structure globale de l'application

On peut observer sur le diagramme ci-dessus les différents modules, appelés *Frameworks*, de notre application. On retrouve :

- le **LayerFramework** qui va s'occuper des Layers,
- l'**ARFramework** qui va implémenter ARCore, s'occuper de l'affichage des données et d'implémenter l'alignement manuel,
- le **LocationFramework** qui s'occupe de fournir la géolocalisation par l'appareil externe ou par le GPS Android,
- et l'**OrientationFramework** qui s'occupe de fournir l'orientation de la caméra par

la boussole Android et par les Landmarks.

Tous les modules sont liés à **Linker**. C'est celui qui va faire le lien entre les modules et gérer nos scènes Unity, c'est-à-dire nos vues (*Views* en anglais). Il va obtenir les différentes informations dont il a besoin pour afficher les données. Les liens jaunes signifient qu'il existe une *Interface* entre les deux entités. L'interface va permettre de définir les méthodes accessibles de nos *Frameworks* depuis le Linker. Elles vont permettre de rendre les modules extensibles et facilement modifiables et changeables. On va y définir les fonctions dont on a besoin dans nos modules pour que l'application fonctionne.

Le LayerFramework va fonctionner selon la **méthode MVC** : Modèle - Vue - Contrôleur. Le modèle (*Model* en anglais) contient les données à afficher, la vue (*View* en anglais) est la représentation graphique et le contrôleur (*Controller* en anglais) va gérer les actions à effectuer sur nos données et faire le lien vers la vue. Il va suivre le diagramme de classe en annexe (cf. page 91). Le Modèle comprend les 3 classes *Layer*, *FeatureCollection* et *FeatureObject*. Dans la classe *Layer* on va retrouver les arguments se trouvant dans notre fichier de configuration. Pour initialiser nos Layers, le constructeur du LayerFramework va :

- Instancier des *ConfigObjects*
 - à partir d'un chemin vers les fichiers de configuration *configPath*
 - en parsant les fichiers JSON de configuration
- à partir des *ConfigObjects*, instancier des Layers

Un Layer lorsqu'il est instancié va alors parser les fichiers *JSONCollection* pour définir la *FeatureCollection*. Le Linker pourra alors instancier dans la Vue nos Layers.

Enfin, on retrouve dans notre structure globale de l'application, un module **Helpers** qui contient les outils utiles dans notre application. Ces outils sont accessibles à tous les autres modules. Un des outils évident est le **CoordConverter**, qui permet de gérer les différents types de coordonnées et de passer de l'un à l'autre.

3.4 Chiffrage et planning prévisionnel

Après le découpage en modules, la définition des tâches était toute trouvée. Il a fallu ensuite définir la priorité de chacune et le temps de réalisation estimé. Après ça, j'ai pu fixer un planning prévisionnel sous la forme d'un diagramme de Gantt (cf. en Annexe page 92).

Fonctionnalité	Priorité	Temps
Rapport + Gestion de projet	1	3
Module de conversion	2	4
Linker et Helper	3	5
LayerFramework	4	10
LocationFramework et DirectionFramework	5	5
ARFramework	6	17
Total à ce stade (Soit le 23 Mai)		44
Amélioration de la géolocalisation et de l'orientation	7	15
Bilan	8	3
Total à ce stade (Soit le 20 Juin)		62
Fonctions additionnelles	9	21
Total (Soit le 17 Juillet)		83

FIGURE 3.9 – Tableau prévisionnel des tâches

Le tableau ci-dessus présente les modules ou fonctionnalités principales en fonction de leurs priorités et du temps de développement (en jour). La priorité est notée de 1 à 9, avec 1 le plus important et 9 le moins important. *Il présente seulement des groupes de tâches. Les tâches sont détaillées dans le diagramme de Gantt.* L'ordre de priorité va permettre de structurer le développement du projet dans le temps et me permettre de toujours connaître ce que je dois faire après chaque tâche.

Selon le planning prévisionnel, un premier prototype fonctionnel devrait être disponible en 44 jours à compter de la fin de la conception (soit disponible le 23 Mai). Cette application doit pouvoir afficher en réalité augmentée des éléments cartographiques provenant d'un fichier GeoJSON à l'aide de la boussole et du GPS Android. 18 jours après, soit le 20 Juin, le prototype devrait implémenter le Landmarks Orientation Provider pour l'orientation et la géolocalisation par un appareil externe. Enfin, au bout de 83 jours à compter du début du développement, l'application devrait implémenter toutes les fonctionnalités.

Si l'on analyse la répartition du temps des tâches ci-dessus on peut alors tracer le diagramme ci-dessous. Ce diagramme présente en pourcentage le temps chiffré entre le début du stage (le 11 Février) et la fin du chiffrage (83 jours après la réalisation de l'état de l'art et de la conception, soit le 17 Juillet).

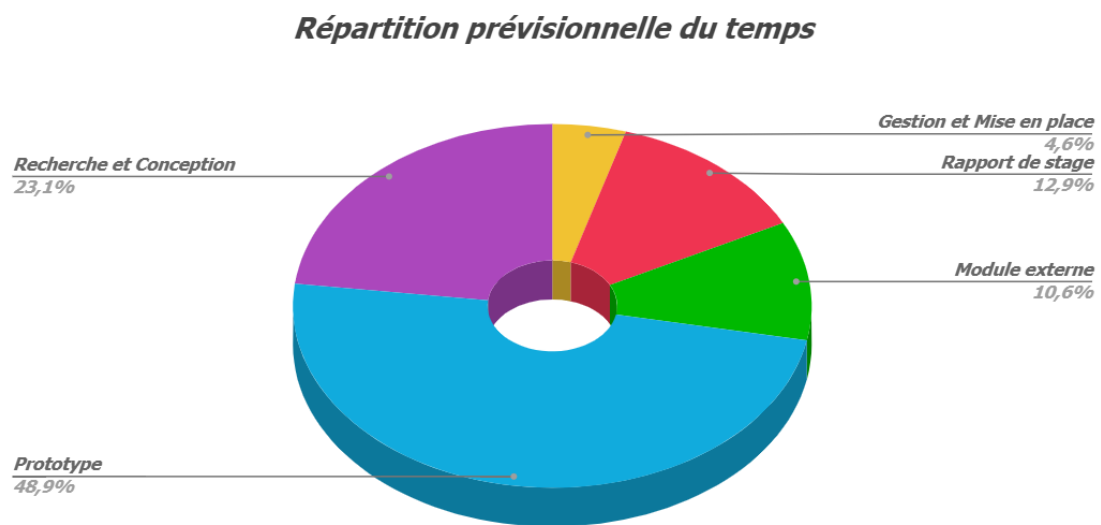


FIGURE 3.10 – Répartition prévisionnelle du temps

On peut voir notamment que l'on a passé pratiquement un quart du temps sur l'état de l'art et la conception (23,1%). On note aussi que l'on passe seulement la moitié du temps (48,9%) à la réalisation du prototype.

Chapitre 4

Développement d'un premier prototype

Le développement du prototype suit le planning prévisionnel fixé lors de la conception.

Dans un premier temps on cherche à obtenir les données à afficher. Pour ça, on va commencer par convertir nos données dans notre format *JSONCollection* avec le **module de conversion** pour pouvoir ensuite les récupérer dans le **LayerFramework** de notre projet Glare. On aura besoin d'un **outil de conversion de coordonnées** et d'implémenter notre **Linker** qui fera le lien vers tous les modules.

Dans un second temps on souhaite obtenir notre première version du prototype en utilisant le *GPS Android* et la *boussole Android*. Pour ça, on va développer une première version du **LocationFramework** et du **DirectionFramework**.

Pour terminer on aura évidemment besoin d'implémenter *ARCore* dans notre **ARFramework**.

4.1 Un module de conversion

Dans un premier temps on cherche à afficher les données venant de la base RTE. Ces données sont disponibles sur une plateforme web [25]. On peut notamment les télécharger en GeoJSON. On va alors commencer par implémenter la conversion du GeoJSON vers notre format *JSONCollection*. Cette conversion est aussi intéressante car la plupart des données géographiques peuvent facilement être représentées en GeoJSON. On pourrait alors utiliser cette conversion pour d'autres conversions.

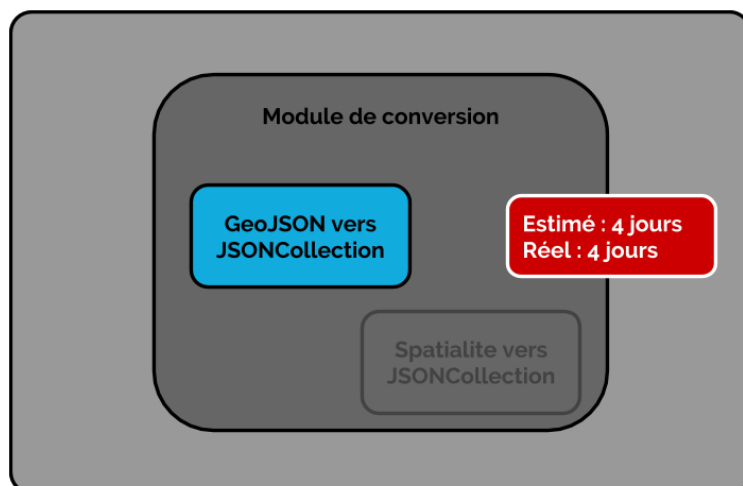


FIGURE 4.1 – Étape 1 - GeoJSON vers JSONCollection

Pour le module de conversion, on va créer un projet Unity séparé de notre prototype. L'intérêt de Unity est d'utiliser le parser GeoJSON déjà implémenté [23] présenté précédemment (cf. page 34) et permettre de facilement implémenter une interface utilisateur.

4.1.1 Présentation du module

Le module permet de convertir un fichier GeoJSON en JSONCollection (il doit par la suite implémenter la conversion d'une base Spatialite vers des fichiers JSONCollection).

Un fichier GeoJSON, pour être converti, doit cependant respecter plusieurs conventions :

- il doit ne comporter qu'un seul type de géométrie (Point, Ligne, Polygones, etc),
- ses coordonnées doivent être en UTM, WGS84 ou Lambert93. Les autres types de coordonnées ne seront pas gérés par l'application,
- et l'extension du fichier doit être *.geojson*.

Lors du lancement de l'application, l'utilisateur doit définir plusieurs paramètres dans une interface :

- le répertoire contenant les fichiers à convertir,
- le répertoire de destination des fichiers convertis
- le type de coordonnées utilisées dans ces fichiers,
- l'ellipsoïde utilisé,
- et s'il souhaite écraser les fichiers de même nom dans le répertoire de destination.

JSONCollection Converter

☒ **Override datas**
Defind if u want to override ResultFile. If not, converter will create new files with the name : fileName + (i) with i the nb files with the name fileName

Ellipsoïde :	WGS84, RGF93	StreamingAssets/InputFiles/	Directory to convert...
Projection type :	WGS84, UTM or Lambert	StreamingAssets/ResultFiles/	Directory converted files...

Start

Quit

FIGURE 4.2 – Interface du module de conversion

Au cours de la conversion, l'utilisateur retrouve l'évolution des conversions et les différentes erreurs possibles.

4.1.2 Fonctionnement de la conversion

L'objectif est toujours le même, rendre l'application la plus extensible possible. Pour cela, on donne la possibilité de facilement intégrer une nouvelle conversion dans l'application.

On a alors un objet **JSONCollectionConverter** qui va récupérer les paramètres de l'utilisateur, faire les tests de validité (l'existence des répertoires, la validité du type de coordonnées, etc) et appeler les autres conversions lors du lancement par l'utilisateur.

Le script **GeoJSONToJSONCollection** va alors se lancer et pour chaque fichier GeoJSON du répertoire donné, le convertir en JSONCollection. On va alors utiliser le parser pour ajouter les informations nécessaires. Le script va alors :

- vérifier la validité des fichiers GeoJSON,
- parser notre fichier,
- ajouter nos informations de boîte englobante pour chaque *FeatureObject* (c'est à dire chaque géométrie comprise dans le fichier),
- ajouter la boîte englobante globale de la *FeatureCollection* (toutes les géométries

du fichier),

- ajouter les informations concernant le type de coordonnées,
- créer nos fichiers *JSONCollection* sous forme de fichier *.json* (la lecture d'un fichier json est plus facile par la suite dans l'application) dans le répertoire de destination.

4.2 Mise en place du prototype

Avant d'implémenter nos différents modules, il nous faut développer les différents outils dont on aura besoin.

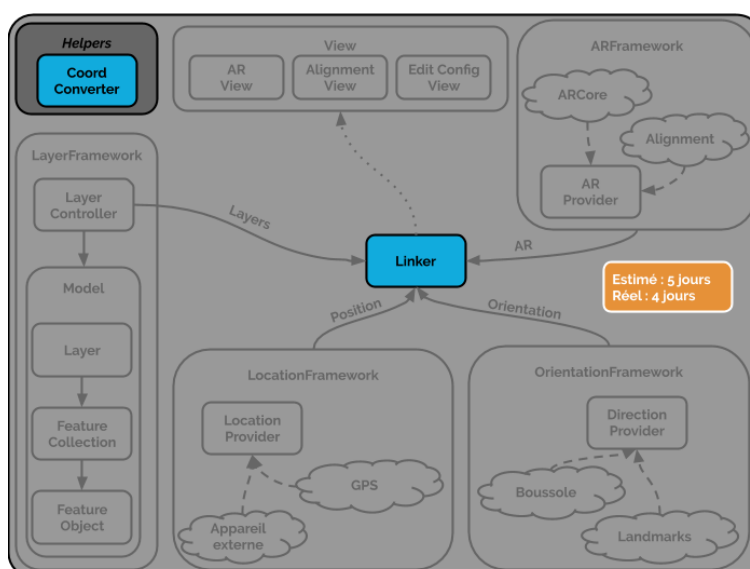


FIGURE 4.3 – Étape 2 - Mise en place du prototype

Le **Linker** est celui qui va faire le lien entre les différents modules de notre application et récupérer les informations nécessaires pour faire de la réalité augmentée géolocalisée. Par la suite, nous aurons besoin d'un outil de conversion de coordonnées. Le **CoordConverter** doit être accessible par tous les modules de l'application. Il va nous être utile pour traduire notre géolocalisation dans le repère Unity, pour instancier nos données, etc.

4.2.1 Linker

L'objectif du Linker est de récupérer les informations nécessaires au cours de l'utilisation de notre application, de l'évolution du comportement de l'utilisateur. Pour ça, il hérite de la classe **MonoBehaviour**, la classe de base de Unity. Cette classe permet à un GameOb-

ject d'évoluer dans une scène Unity et d'utiliser les fonctions de bases de Unity. Elle permet par exemple d'affecter à un `GameObject` une action à réaliser tout au long de l'application. Lorsqu'une classe hérite de `MonoBehaviour` elle peut alors être attachée à un `GameObject` (c'est à dire une entité de notre scène, de notre application) sous forme de composant de celui-ci. On peut notamment implémenter dans notre classe les méthodes :

- **Start()** : exécuter au lancement de l'application
- et **Update()** qui va être exécuté à chaque frame (selon un certain nombre d'images par seconde).

L'objectif du `Linker` va être d'initialiser notre scène dans la fonction *Start* en récupérant les données à afficher, la caméra ou encore notre géolocalisation et de mettre à jour les informations à afficher sur l'écran de notre utilisateur en fonction de son comportement dans la fonction *Update*. Le `Linker` devra donc faire le lien entre nos différents modules c'est à dire :

- `ARFramework`,
- `OrientationFramework`,
- `LocationFramework`
- et le `LayerFramework`.

Il doit ensuite retranscrire les informations nécessaires dans la scène.

4.2.2 La conversion des coordonnées

Pour pouvoir comparer nos données, notre position ou pour calculer les distances entre deux objets, nous avons besoin de travailler dans le même repère. Nous avons fait le choix lors de la conception de travailler en UTM permettant d'évoluer dans un système métrique et orthonormé (cf. page 35).

Un outil avait été développé par mon tuteur de stage Raphaël Guenon lors de la réalisation du projet RTE qui permet de passer d'une donnée WGS84 vers de l'UTM ou du Lambert 93 et réciproquement. J'ai pu réutiliser le composant dans mon projet pour implémenter la conversion vers l'UTM.

4.3 La gestion des Layers

La gestion des Layers est la troisième étape de la réalisation du prototype. Dans un premier temps on cherche à récupérer les données de nos `JSONCollections` pour ensuite les afficher dans notre scène Unity. Le temps de réalisation a été plus court que prévu,

notamment parce que le parsing des données avait été partiellement implémenté dans le module externe.

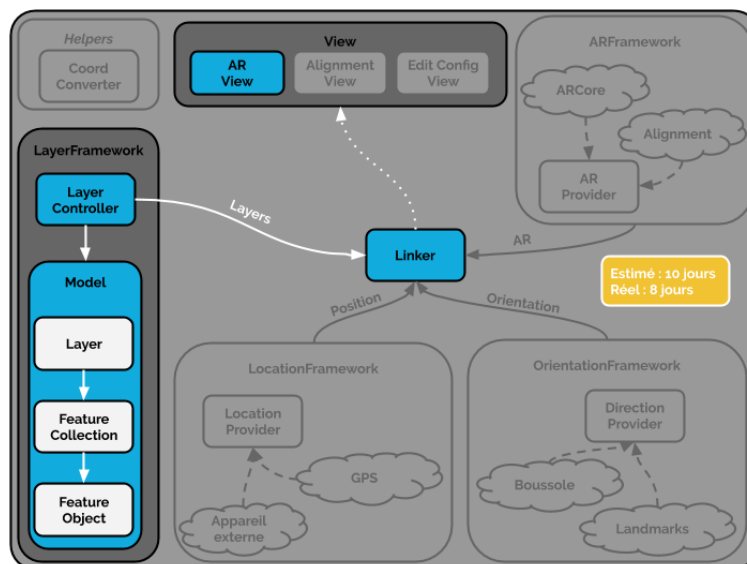


FIGURE 4.4 – Étape 3 - Gestion des Layers

4.3.1 Création des Layers

Pour arriver à l'obtention des Layers plusieurs étapes sont nécessaires. Dans un premier temps, il faut interpréter les fichiers de configuration pour connaître les données à afficher et comment les afficher. Dans un deuxième temps, il faut lire les fichiers *JSONCollection* pour obtenir nos données géolocalisées.

Pour rappel, un fichier de configuration est un fichier JSON contenant la définition de un ou plusieurs Layers (cf. page 36). Pour lire nos fichiers de configuration, on va utiliser une méthode de Unity qui permet de transcrire un fichier JSON en instance d'une classe (cette méthode s'appelle *JSON.Serialization()*). Dans notre cas, la classe à instancier est *ConfigObject* (défini dans le diagramme de classe lors de la conception, cf. page 91) qui contient :

- le nom du Layer,
- le chemin vers le fichier de données *JSONCollection*,
- la distance maximale d'affichage,
- si on souhaite l'activer,
- si on souhaite l'afficher,
- le type de géométrie,
- et le nom du prefab à instancier.

À partir de nos instances de *ConfigObject* dans le *LayerController* on va ensuite construire notre liste de Layer. Il nous suffit de lire le fichier *JSONCollection* correspondant pour en ex-

traire la *FeatureCollection* en utilisant le parser **GeoJSON** (cf. page 34).

L'implémentation de la création des Layers a été plus rapide que prévu. De plus, le diagramme de classe réalisé lors de la conception est fidèle à la réalité.

Quelques méthodes ont été rajoutées dans le *LayerController* pour améliorer la structure de notre projet comme une méthode permettant de vérifier la validité des données contenues dans les fichiers de configuration (vérifier que le prefab à instancier existe, etc) et une méthode qui permet d'obtenir la liste des Layers qui sont près de l'utilisateur. Cette méthode s'appelle ***GetSurroundingLayers()***. Elle va, en fonction d'une position donnée, qui représente la position de notre utilisateur dans le monde en UTM, retourner les Layers que l'on va afficher dans notre scène et les convertir en UTM. Pour ça, la méthode va utiliser les boîtes englobantes contenu dans les *FeatureCollections* pour voir si notre utilisateur se trouve proche ou non d'un certain Layer. On va ensuite **utiliser le *CoordConverter* développé** précédemment pour convertir nos données en UTM pour ensuite pouvoir **les instancier dans notre scène Unity**.

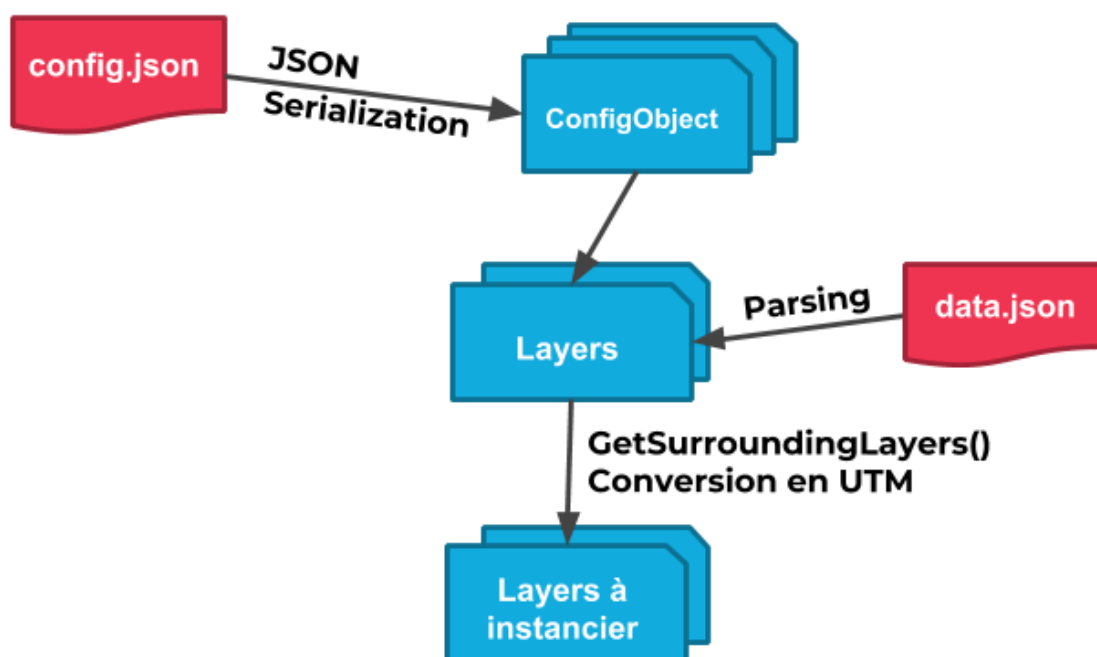


FIGURE 4.5 – Schéma récapitulatif de la création des Layers

4.3.2 Instanciation des layers

À partir de nos Layers à instancier (*SurroundingLayers*), on va afficher nos objets 3D dans notre scène Unity à des positions précises. Le *LayerController* va, pour chaque Layer s'il est défini comme actif dans le fichier de configuration, instancier un prefab spécifique pour

chaque Feature de la FeatureCollection du Layer à des positions données.

Il faut prendre en compte que les positions des objets dans la scène sont relatives à la position de l'utilisateur. La position de l'utilisateur dans la scène à l'initialisation sera (0,0,0) et les objets seront instanciés selon leurs distances à l'utilisateur dans le monde réel. Imaginons que l'utilisateur se trouve dans le parking de Capgemini à l'Aéropark de Toulouse. Sa position en UTM est (356 090.3, 4 802 825.0). Un objet qui se trouve en (356 091.3, 4 802 825.0) sera représenté dans notre scène en (1.0, 0.0). Cette définition doit être faite pour la suite du projet pour ARCore et nous permet aussi de gérer des nombres plus petits, d'être plus précis.

Selon notre conception, l'objectif de notre implémentation telle qu'elle a été imaginée était de pouvoir afficher tous les Layers selon nos choix d'apparence. Il suffit de changer le prefab à utiliser dans notre fichier de configuration pour changer la manière dont on l'affiche. Cependant, il faut que le prefab existe mais il faut aussi qu'il respecte certaines conventions. Pour qu'un prefab puisse être instancier dans notre application, il faut qu'il possède un composant qui hérite de notre classe **AbstractAspect**. Il s'agit d'une classe abstraite¹ qui va définir les fonctions et arguments à implémenter dans nos prefabs. Elle va notamment définir qu'un prefab est lié à une Feature et que chaque prefab a une méthode dans son composant qui permet de définir sa position lorsqu'on va l'instancier.

On va alors créer 3 prefabs par défaut avec chacun un composant différent qui hérite de notre classe *AbstractAspect* et qui va représenter les trois types de géométrie basique : un point, un segment et un polygone.

Affichage d'un point

L'affichage d'un point est l'affichage le plus simple des trois géométries. En effet, j'ai créé le prefab en utilisant l'implémentation de la sphère de Unity. On a ensuite affecté un Material² à notre sphère pour lui donner une certaine apparence.

1. Une classe qui n'est pas instanciable. Elle sert de base à d'autres classes dérivées

2. Élément qui va gérer l'apparence d'un GameObject en donnant une couleur ou une texture

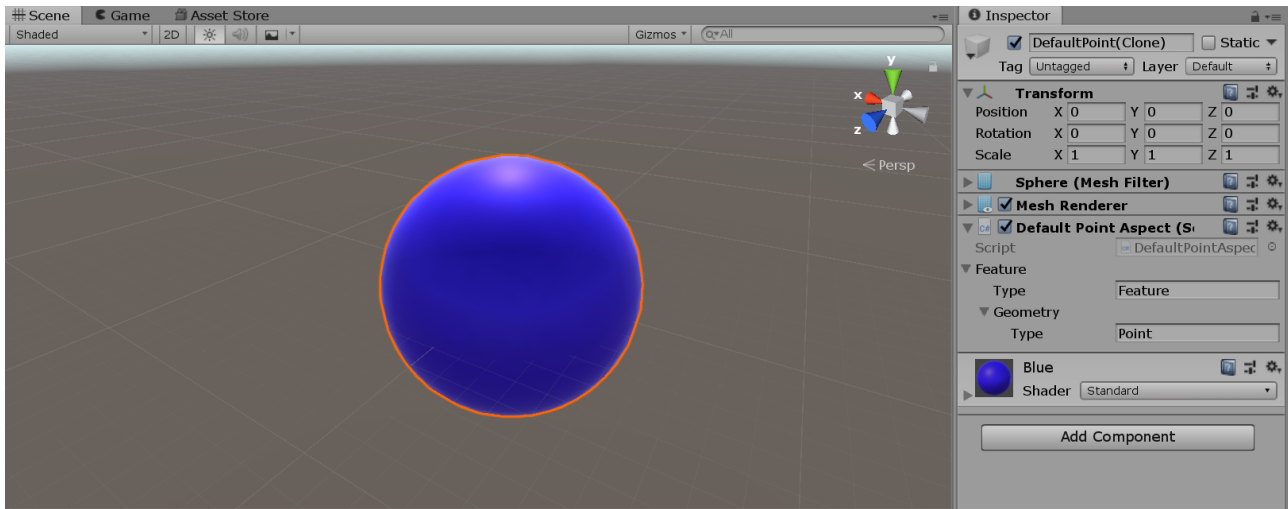


FIGURE 4.6 – Rendu d'un point dans notre scène Unity

On peut voir sur l'image ci-dessus le rendu d'un point par défaut. On retrouve dans l'Inspector (fenêtre à droite qui répertorie les caractéristiques d'un GameObject sélectionné) les composants de notre GameObject. On retrouve :

- le *MeshFilter* qui définit le squelette de notre objet, dans notre cas une sphère,
- le *MeshRenderer* qui va faire le lien avec le Material,
- et le *DefaultPointAspect*, le composant qui hérite de *AbstractAspect* va définir où afficher l'objet dans la scène et faire le lien avec la Feature associée.

Il a fallu ensuite définir la position de la sphère. Dans Unity, la position d'un GameObject est défini par un *Vector3* : 3 valeurs décimales représentant les coordonnées dans chacun des axes de Unity (x,y,z). Dans notre cas, on souhaite afficher l'objet par rapport à l'utilisateur. On a fait le choix de projeter nos objets en utilisant la projection UTM. On va alors donner à nos objets les positions en UTM par rapport à la position de l'utilisateur en UTM. Pour ça nos coordonnées vont prendre les valeurs :

- x va représenter pour nous la valeur de l'UTM Est de l'objet par rapport à l'utilisateur (c'est à dire la différence),
- y représente l'altitude de l'objet par rapport à l'utilisateur,
- et z va représenter pour nous la valeur UTM Nord de l'objet par rapport à l'utilisateur.

Étant donné qu'un point dans un *JSONCollection* (ou *GeoJSON*) n'est composé que d'une seule coordonnée il suffit simplement de donner à notre prefab la position décrite ci-dessus.

Affichage d'une ligne

Une ligne dans un *JSONCollection* est un tableau de plus de deux coordonnées. Il s'agit simplement d'un ensemble de points par où passe la ligne.

Pour gérer l'affichage par défaut d'une ligne on va alors utiliser un composant de Unity qui s'appelle **LineRenderer**. Il suffit simplement de lui donner notre liste de points contenue dans notre Feature (avec les coordonnées relatives à la position de l'utilisateur comme pour le point) et une couleur. Il va ensuite tracer notre ligne entre nos points.

On observe ci-dessous le rendu d'une ligne entre deux points (un segment) dans notre scène. Dans le composant LineRenderer on retrouve la liste des positions qu'on donne pour positionner notre segment dans la scène. Dans l'exemple ci-dessous on a : deux points, l'un en (0,0,0) et l'autre en (8.125, 0, -0.5).

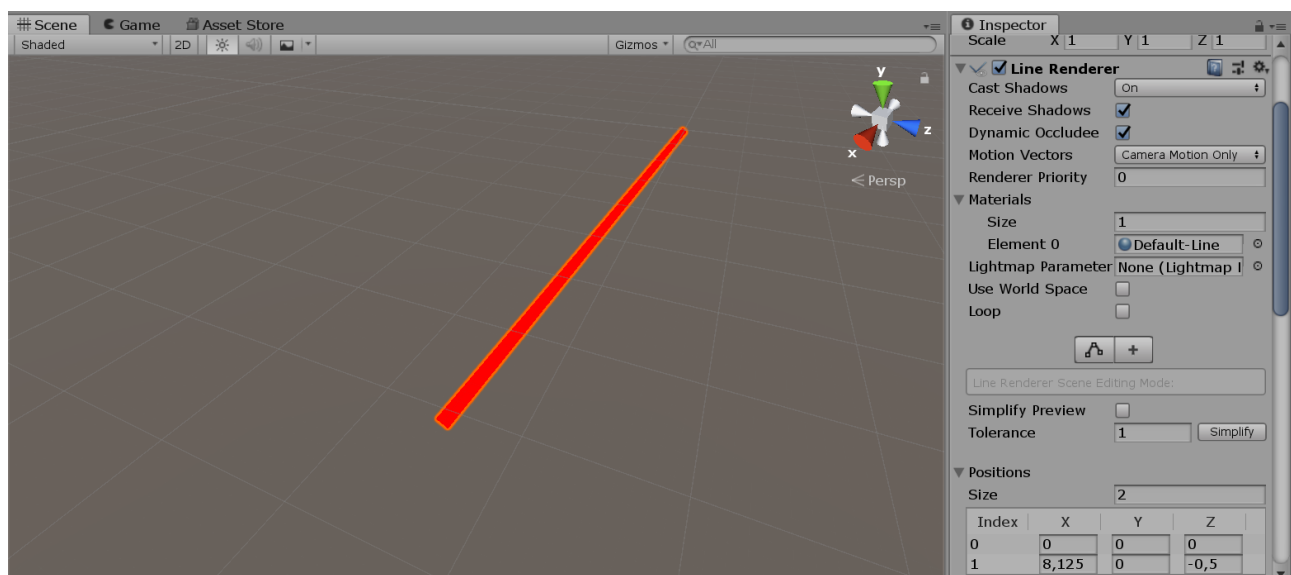


FIGURE 4.7 – Rendu d'un segment dans notre scène Unity

Affichage d'un polygone

L'affichage d'un polygone a été la partie la plus complexe de la gestion des Layers. Un polygone selon la convention GeoJSON [4] est décrit par un *LinearRing* : une ligne fermée qui traduit le contour des polygones. Il s'agit d'un tableau de tableau de points : le premier tableau définit le contour du polygone où le premier et le dernier élément sont équivalents. Les tableaux suivants vont définir les *trous* se trouvant dans ce périmètre (le contour du polygone définit par le premier tableau).

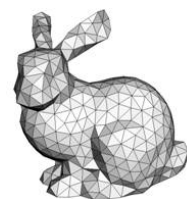


FIGURE 4.8 – Un maillage

Or, nos objets 3D doivent être sous la forme de **maillages** : une liste de triangles (ou polygones) décrivant la surface d'un objet. Pour **convertir nos Linear Rings en triangles**, on va utiliser une triangulation de Delaunay : décomposition d'un polygone en un ensemble fini de triangles. La triangulation va partitionner le polygone en créant des sommets permettant ensuite de former des triangles. Plusieurs librairies en C#³ permettent de faire de la triangulation. Après en avoir testé plusieurs qui me semblaient trop complexes pour notre application, j'ai utilisé un algorithme [26] adapté à notre projet et simple d'utilisation. Il va en fonction des points de notre line rings construire le mesh de notre objet.

On obtient alors l'objet ci-dessous en ne donnant seulement que les 6 points de contours.

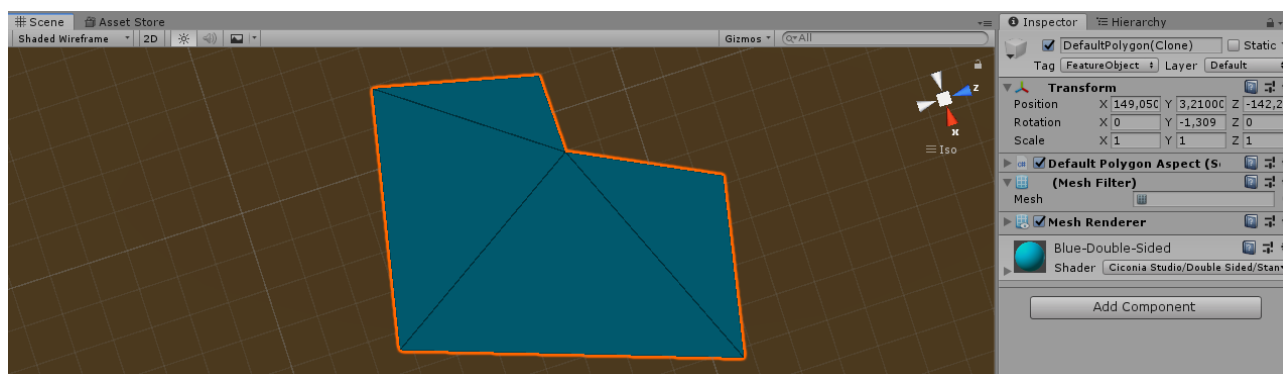


FIGURE 4.9 – Rendu d'un polygone dans notre scène Unity

Pour définir la position de l'objet on va simplement calculer **le barycentre du polygone**. On peut le comparer à un centre de gravité. On va calculer la moyenne arithmétique de nos points et associer cette moyenne à la position de l'objet (relative à la position de l'utilisateur).

Récapitulatif

Pour récapituler, à partir des Layers à instancier depuis les fichiers JSONCollections, on va instancier des prefabs définis auparavant en fonction du type de géométrie. On a créé trois prefabs par défaut pour traduire les trois types de géométrie possibles. Chaque prefab contient un composant hérité de *Abstract Aspect* qui va lors de l'instanciation définir comment afficher l'objet 3D et à quelles coordonnées.

3. Le langage de programmation utilisé dans Unity commercialisé par Microsoft

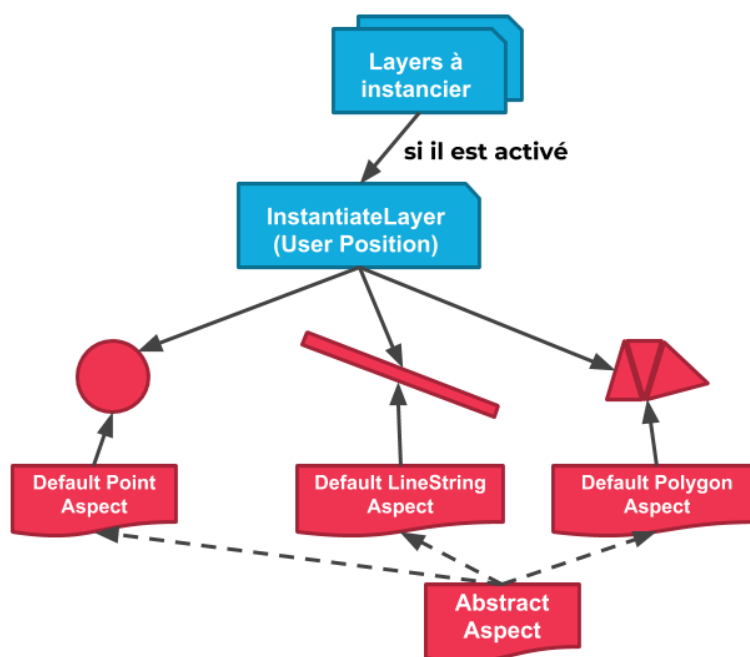


FIGURE 4.10 – Schéma récapitulatif de l'instanciation des Layers

4.4 Définition de l'attitude de l'appareil

L'objectif de la quatrième étape de l'implémentation de mon prototype est d'**obtenir les données GPS Android et la boussole Android**. Ces outils nous permettront de tester notre prototype en sachant bien que leur précision n'est **pas idéale**. Pour le GPS d'Android, on peut atteindre une précision horizontale de trois mètres (trois fois moins précis à la verticale). Quant à la boussole, on a une précision d'environ quinze degrés.

Cette étape s'est avérée plus rapide que prévu, notamment par la facilité d'implémentation des outils Android dans Unity. Il a fallu cependant créer une interface à notre application pour pouvoir observer les résultats de position et d'orientation obtenus.

Lors de ma conception, je comptais réaliser cette étape avant d'ajouter la réalité augmentée (en l'occurrence ARCore) dans l'application. Cependant, ces deux étapes ont été réalisées en parallèle pendant environ une semaine pour pouvoir tester les changements que j'effectuais directement sur mes objets en réalité augmentée.

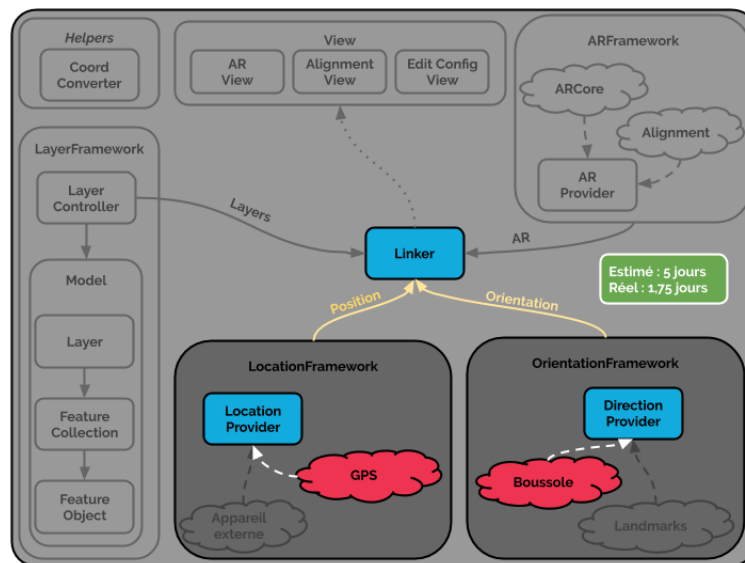


FIGURE 4.11 – Étape 4 - Définition de l'attitude de l'appareil

4.4.1 Implémentation du GPS et de la boussole Android

Toujours dans l'objectif de rendre l'application extensible, on va implémenter deux interfaces *ILocation* et *IOrientation* pour implémenter notre GPS et notre boussole Android. Une interface permet de définir les fonctions qu'on va appeler pour demander la position ou l'orientation de nos deux classes. Elles vont permettre par la suite de pouvoir changer de fournisseur de géolocalisation ou de direction sans avoir à toucher au coeur de l'application.

Le GPS et la boussole Android sont facilement implémentables à l'aide des fonctions de Unity. On va, au lancement de l'application, demander à notre appareil l'accès à la géolocalisation, et, au cours de l'exécution à chaque frame (dans une fonction *Update()*), demander notre position et notre orientation.

Cependant, il faut prendre en compte plusieurs paramètres. Le premier est que le GPS Android ne va pas fournir une nouvelle donnée à chaque frame. **La fréquence de réception est variable**, en fonction de la météo et du lieu où l'on se trouve. Pour donner un ordre d'idée, lorsque je testais mon application dans le parking de Capgemini DEMS, dans de mauvaises conditions météorologiques, je recevais six ou sept données en trois minutes d'utilisation, soit environ une donnée toutes les trente secondes. Dans des conditions idéales, on peut recevoir environ une donnée par seconde.

Contrairement au GPS, la boussole d'Android fournit beaucoup de données : **environ soixante par secondes**. Cependant, les données fournies sont peu précises, on a une précision d'environ quinze degrés. Il a alors fallu traiter les données que l'on reçoit pour *débruiter* la boussole. On va chercher à **lisser les données**. Pour ça, on va faire une moyenne des dix

dernières données. Étant donnée qu'il s'agit d'angles, une moyenne classique n'est pas utilisable. En effet, dans le cas où l'on pointe précisément le Nord, notre boussole va nous envoyer des données qui oscillent entre $(0^\circ, 10^\circ)$ et $(350^\circ, 360^\circ)$. Notre moyenne va alors être égale à environ 180° (qui signifie que l'appareil pointe vers le Sud) alors que notre appareil pointe vers le Nord. Pour ça, on va, pour chaque angle donné, calculer les coordonnées (x, y) sur le cercle trigonométrique (où x est alors le cosinus de l'angle donné et y le sinus). On va ensuite calculer la position moyenne des coordonnées (x, y) et retourner l'angle donnée par l'arc tangente⁴ de la position moyenne.

Il ne reste qu'à faire le lien avec les autres modules. Dans le *Linker*, on va à l'instant initial (dans la fonction *Start()*) instancier deux objets dans notre scène : l'un contenant le composant *GPSAndroid* (qui implémente l'interface *ILocation*) et l'autre *CompassAndroid* (boussole Android qui implémente l'interface *IOrientation*). Ces deux objets vont fournir au cours du temps les données d'attitude que l'on pourra utiliser pour instancier nos *Layers* et définir leurs positions dans la scène.

4.4.2 Interface de l'application

Pour obtenir des données GPS, on est obligé de sortir du bâtiment. L'objectif était alors de **connaître les données que je recevais directement sur mon application**. Pour ça, on va créer une interface. Elle va être modifiée par un manager *IMHManager* qui va mettre à jour les données à afficher. Le *Linker* va faire le lien entre les données qu'il reçoit et l'*IMHManager*.

On va aussi utiliser un outil de l'Asset Store⁵ *InGameDebugConsole* qui permet d'afficher la console⁶ Unity dans notre application.

4. Donne l'angle sur un cercle trigonométrique de la tangente aux coordonnées données

5. Le magasin de Unity où l'on peut trouver différents scripts ou prefabs

6. Une console nous retourne/affiche les erreurs, les avertissements ou les variables que l'on souhaite connaître lors de l'exécution d'un script



FIGURE 4.12 – Mise en place de l'interface

On retrouve trois boutons sur le haut : ils permettent de quitter l'application et plus tard de s'aligner avec une carte 3D ou de modifier nos données.

En bas à droite on affiche nos données d'orientation. On va implémenter une boussole virtuelle qui indiquera le Nord. On affiche aussi l'angle entre la direction de notre caméra et le Nord. Cet angle nous sera utile pour mettre à jour la position de nos objets.

En bas à gauche, on retrouve nos données GPS. Elles sont exprimées en *WGS84*, ce sont les données que nous envoie le GPS Android. On décide d'afficher la précision du GPS, une donnée qui sera importante pour la suite.

4.5 L'ajout de la réalité augmentée

Cette partie a été **la plus longue** depuis que j'ai commencé le développement de mon projet. En effet, étant donné que je n'avais jamais fait de réalité augmentée avant, il m'était difficile de réaliser cette partie et de la chiffrer. Elle a tout de même été moins longue que prévu. Cette différence s'explique par une surévaluation de ma part lors du chiffrage. En effet je pensais l'implémentation d'ARCore dans Unity plus complexe. Cependant, avec les projets que fournissent ARCore il est facile de s'en inspirer pour créer notre propre projet.

De plus, j'ai eu quelques difficultés lors de l'ajout de la réalité augmentée, notamment sur la mise à jour des positions des données et Raphaël Guenon a su me guider et m'orienter pour m'aider à avancer.

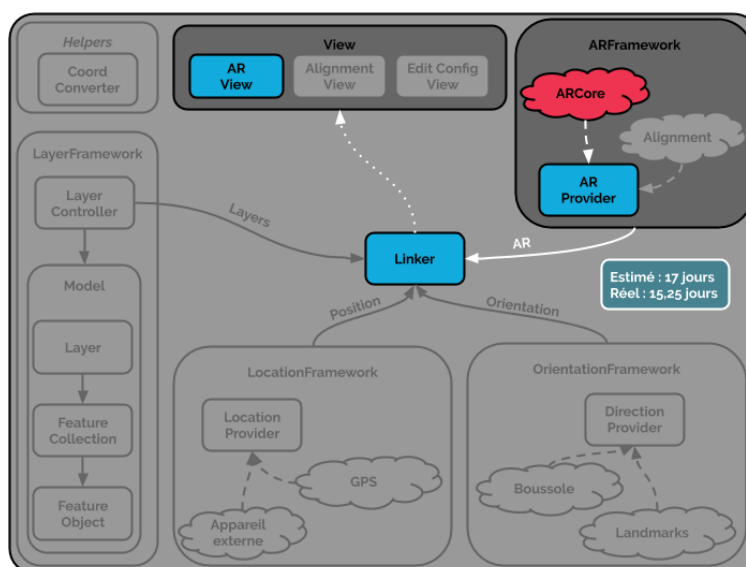


FIGURE 4.13 – Étape 5 - L'ajout de la réalité augmentée

Dans un premier temps j'ai mis en place ARCore dans le prototype. Dans un second temps j'ai développé la mise à jour des positions des objets instanciés.

4.5.1 Mise en place d'ARCore

Pour mettre en place ARCore et pour le lier à mon projet, je suis parti de la scène de test fournie par Google. Cette scène permet de poser un objet 3D en réalité augmentée lorsqu'on clique sur l'écran.

Elle comprend notamment deux *GameObjects* : l'un qui gère la caméra Android et l'autre qui s'occupe de la détection de surface : deux éléments dont nous avons besoin pour notre prototype. Le *GameObject* qui gère la caméra va faire en sorte d'utiliser la caméra de l'appareil Android et gérer son déplacement dans la scène. L'autre *GameObject* est celui qui va analyser notre environnement pour détecter des plans. Il va extraire de ce que voit la caméra, des points d'intérêt, pour en extraire des surfaces. Elles vont nous servir plus tard à déterminer où se situe le sol par rapport à l'utilisateur, pour ajuster l'altitude de l'utilisateur.

4.5.2 Mise à jour des positions au cours du temps

Comme décrit précédemment (cf. page 54), il faut **placer les objets en fonction de la caméra**. On ne peut pas bouger la caméra pour qu'elle s'adapte au monde réel, on doit **ajuster tous les objets pour qu'ils s'adaptent à la position de la caméra**. Seulement, la précision du tracking d'Arcore et la précision du GPS nous obligent à mettre à jour la position des objets tout au long de l'exécution du programme. En effet, lorsque, par exemple, on avance d'un mètre dans la réalité, la caméra dans Unity n'avance pas forcément d'un mètre. De plus, étant donné qu'à l'initialisation le GPS a souvent une précision très faible (à titre d'exemple, dans le parking de Capgemini, le GPS à l'initialisation de l'application nous fournit des données avec une précision de plus de 9 mètres en moyenne), la position des objets à l'instant zéro n'est pas fiable.

Pour mettre à jour nos positions, on va calculer deux choses :

- Ce qu'on appelle **deltaPosition** : la différence entre la position en UTM de l'utilisateur dans le monde et la position de la caméra dans Unity
- et le **deltaAngle** : la différence entre l'orientation de la caméra par rapport au Nord dans le monde réel et l'orientation de la caméra dans Unity par rapport à l'axe Z.

Le processus pour obtenir nos deux variables est simple.

Pour déterminer le *deltaAngle*, on va faire la soustraction de l'angle entre la caméra dans Unity et l'axe Z (le vecteur *oz*) et l'angle donné par la boussole (c'est à dire l'orientation de notre tablette dans le monde réel par rapport au Nord). Ce calcul va se faire à chaque fois qu'on reçoit une donnée de la boussole, c'est à dire à chaque frame.

Pour déterminer le *deltaPosition*, on va suivre le même processus : on va faire la soustraction de la position de la caméra dans Unity et de la dernière position donnée par le GPS convertie en UTM. Ce calcul se fait à chaque fois qu'on reçoit une donnée du GPS.

A l'aide de ces deux variables, on va pouvoir faire une **projection**. Une projection dans notre cas est finalement un **changement de repère** où on va essayer de projeter le monde réel vers notre scène Unity.

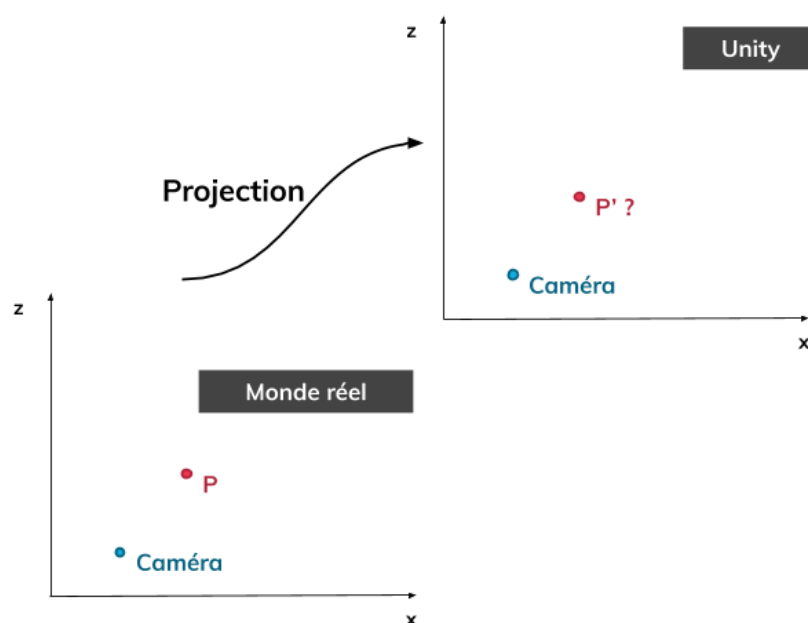


FIGURE 4.14 – Schéma de notre projection

En effet, comme décrit sur le schéma ci-dessus, l'objectif est de **connaître la position de l'objet P dans le monde Unity** en connaissant sa position dans le monde réel, la position de la caméra dans le monde réel et la position de la caméra dans le monde Unity.

À partir du *deltaPosition* et du *deltaAngle* on va pouvoir pour chaque *FeatureObject*, c'est à dire chaque élément géolocalisé que l'on affiche, mettre à jour sa position. Pour connaître la position de P dans Unity, on va alors traduire notre objet de sa position dans le monde réel plus le *deltaPosition*. On va ensuite pivoter l'objet autour de l'axe (Oy) (c'est à dire sur le plan (ox, oz)) de l'angle *deltaAngle*.

4.6 Résultats et limites de la première version du prototype

4.6.1 Résultats obtenus

À l'aide de cette première implémentation on obtient notre premier prototype. Avec ARCore, le GPS Android et la boussole Android on peut alors afficher des éléments cartographiques (géolocalisés) en réalité augmentée sur une tablette en partant de fichiers *JSONCollection*.

Pour pouvoir tester, j'ai placé différents objets dans le parking de Capgemini DEMS.



FIGURE 4.15 – Plan du parking et la position des différents objets

J'ai alors créé 3 fichiers GeoJSON :

- l'un contenant 3 points : les cercles bleus sur la carte,
- un second pour définir les 2 lignes : les lignes rouges sur la carte
- et un dernier pour définir les 2 polygones.

On a ensuite converti nos fichiers en JSONCollection à l'aide de notre *JSONCollection Converter* et on a défini un fichier de configuration pour dire à l'application de **charger nos 3 fichiers en utilisant les prefabs par défaut** définis précédemment.



FIGURE 4.16 – Vue d'ensemble des objets en RA dans le premier prototype



FIGURE 4.17 – Vue d'un segment en RA dans le premier prototype



FIGURE 4.18 – Vue d'un polygone en RA dans le premier prototype

Sur la première image ci-dessus on se situe à l'Ouest du parking et on pointe notre caméra vers l'Est. On peut alors voir les 3 points et au loin un des deux segments.

Sur la deuxième image on est à l'Est du parking et on pointe notre caméra vers l'Est. On peut alors observer le segment vu précédemment de plus près.

Sur la dernière image on regarde le polygone se trouvant le plus à l'Est du parking.

4.6.2 Limites

L'imprécision

Avec ce premier prototype on peut nettement constater l'importance d'avoir une bonne précision d'orientation et de géolocalisation. On peut voir sur les images 4.17 et 4.18 que **les objets sont mal placés**, surtout dû à l'imprécision du GPS : 3 mètres de précision. Notamment pour la représentation du segment 4.17 où l'on peut voir que le segment doit être plus à l'Ouest.

Cependant le plus marquant est l'imprécision de la boussole. On observe très nettement cette imprécision sur la première image 4.16. On voit notamment au loin que le segment n'est pas du tout à sa place. Plus les objets sont éloignés de la caméra, plus **l'imprécision va être visible**. On a en moyenne une imprécision de la boussole d'environ quinze degré. Plus la distance entre l'observateur et le point observé est importante, plus cette angle d'erreur de la boussole entraîne un décalage tangentiel.

Un autre problème dû à la boussole est que **les objets bougent, tremblent**. Ce phénomène est causé par le nombre de données que nous envoient la boussole et l'imprécision de ces données. Malgré le débruitage de la boussole (cf. page 59) ce phénomène reste présent.

Ces problèmes devraient être résolus avec l'implémentation de l'appareil externe pour la géolocalisation et le *Landmarks Orientation Provider* que l'on va développer par la suite.

L'altitude

Un autre problème évident à la vue des trois images précédentes est le problème d'altitude. En effet, l'altitude donnée par le GPS est très variable et trois fois moins précise que la précision horizontale. On a alors des **problèmes de perspective** que l'on peut observer sur l'image du polygone 4.18 où le polygone se trouve en dessous du sol. On a alors l'impression que le polygone est très petit par rapport au bâtiment qu'il doit représenter. Il était aussi difficile pendant mes différents tests de voir où se situaient exactement mes objets

quand ils étaient trop hauts du sol.

Ce problème d'altitude sera corrigé avec l'implémentation de l'appareil externe qui nous permettra d'avoir une précision de la géolocalisation au centimètre. Cependant, nous pourrions envisager d'implémenter d'un bouton nous permettant de **poser les objets sur le sol** (à la même altitude que nous). Cette fonctionnalité permettrait de se rendre compte du véritable positionnement des objets sur le plan (Nord, Est) (cf. page 74).

L'ergonomie

Un autre problème que j'ai eu est l'ergonomie de l'application. Pour pouvoir **modifier mes données** et définir quels objets je voulais afficher il me fallait modifier les fichiers de configuration dans Unity. Il n'y aura plus ce problème lorsque la modification des configurations sera possible directement dans l'application.

De plus, il est difficile pour l'utilisateur de visualiser et de comprendre où se situent les objets par rapport à lui-même sans pivoter la tablette jusqu'à trouver l'objet qu'il souhaite visualiser. **Une mini-carte** dans l'interface permettrait de mieux envisager les objets qui se trouvent autour de lui.

Chapitre 5

Vers un prototype fonctionnel

Après les différentes limites qu'on a pu observer sur la première version du prototype il était primordial d'améliorer notre application. Les premiers axes d'amélioration sont ceux de la géolocalisation et de l'orientation. J'ai ensuite modifié certaines implémentations pour optimiser l'application et ajouter des fonctionnalités permettant à notre projet d'être plus ergonomique ou plus fonctionnel.

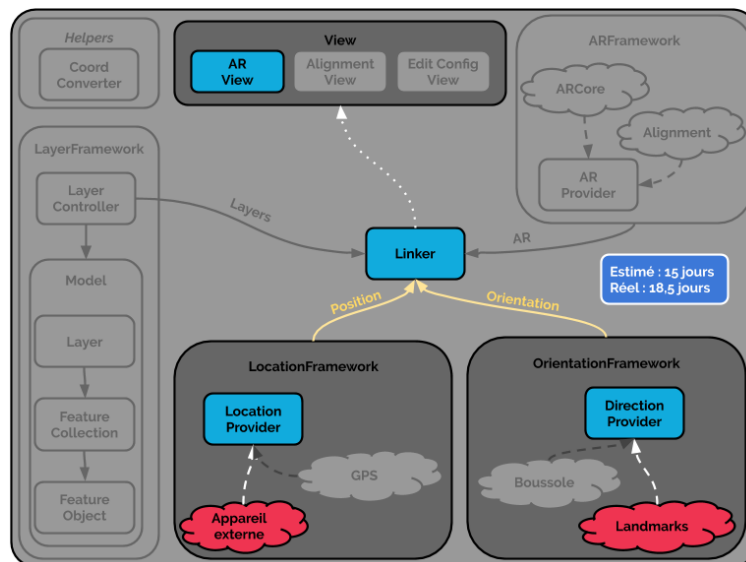


FIGURE 5.1 – Tablette Samsung avec le Satlab SLC

Cette partie devait durer 15 jours cependant avec l'ajout de nouvelles fonctionnalités et les corrections effectuées elle a duré trois jours et demi de plus.

5.1 Amélioration de la géolocalisation et de l'orientation

5.1.1 Implémentation du Satlab SLC

Avec les problèmes que nous avons eus sur le premier prototype je me suis rendu compte de l'importance d'avoir une bonne précision. Pour obtenir une meilleure solution de géolocalisation, on avait prévu lors de la conception d'utiliser un appareil externe nous permettant d'avoir **une précision au centimètre près**. On a alors implémenté le Satlab SLC (cf. page 31), un appareil que l'on peut fixer sur la tablette et qui va transmettre des données de géolocalisation par bluetooth.



FIGURE 5.2 – Tablette Samsung avec le Satlab SLC

Le SLC a déjà été utilisé par Raphaël Guenon pour le projet RA-LS. Etant donné que son

implémentation est fastidieuse et pas forcément intéressante il m'a fourni quelques fichiers me permettant de plus facilement l'implémenter comme notamment le plugin Android décrit par la suite.

Pour implémenter l'appareil, il faut lire les messages qu'il nous envoie. Ces messages sont sous la norme NMEA, une spécification utilisée par les équipements marins dont les outils de navigation GPS [5]. On va créer **un plugin Android** qui permet de décrypter des messages NMEA. Il va aussi nous envoyer **la liste des appareils bluetooth disponibles** et nous permettre de se connecter à l'un d'eux. On va alors demander à notre plugin les informations qu'il reçoit et d'effectuer les actions bluetooth depuis notre projet Unity pour implémenter le Satlab.

On va alors ajouter dans notre interface plusieurs informations :

- la liste des appareils disponibles en bluetooth, lorsqu'on va cliquer sur l'un d'eux on s'y connecte,
- un bouton nous permettant d'afficher ou non cette liste,
- le statut du Satlab (si on reçoit bien des données, si la correction RTK est disponible, etc)
- et un champ texte permettant d'ajuster l'altitude. En effet, Satlab ne donne pas l'altitude dans le même référentiel que le GPS Android. À Toulouse, il faut ajouter environ 50 mètres pour combler l'écart entre les deux référentiels. Cet écart varie en fonction de là où l'on se situe sur le globe. L'utilisateur pourra alors changer cet écart dans l'interface.

Ensuite on va **structurer les données reçues** pour implémenter notre interface *ILocation* utilisée précédemment pour implémenter le GPS Android. Dans le *Linker*, on va donner la priorité à Satlab. Si l'appareil est disponible et qu'il nous donne une valeur avec une meilleure précision que la précédente on va **utiliser cette donnée pour calculer le *deltaPosition***.

Le processus pour utiliser les données de Satlab est d'afficher les appareils bluetooth disponibles, de se connecter au Satlab SLC lorsque l'utilisateur clique sur lui dans l'interface. Ensuite, si le Satlab nous envoie des données, avec une meilleure précision que le GPS Android on utilise ces données pour calculer le **deltaPosition**.

5.1.2 Le Landmarks Orientation Provider

L'objectif de cette partie était d'**implémenter la méthode automatique de Mapbox** (cf. page 30). Comme décrit précédemment l'objectif est de placer des repères (*Landmarks*) dans l'espace pour calculer la différence entre l'orientation de la caméra dans Unity et dans

le monde réel : c'est à dire **un *deltaAngle***.

Il y a deux étapes pour obtenir notre *deltaAngle*. La première est l'ajout d'un Landmark. **Un Landmark est défini par :**

- une position de la caméra dans Unity,
- la géolocalisation au même moment
- et la précision de la géolocalisation.

On va, à chaque fois qu'on reçoit une géolocalisation, essayer d'ajouter un *Landmark*. On ajoute un *Landmark* si sa distance avec le plus proche existant est inférieure à la somme des deux précisions. Cette condition nous permet d'amenuiser l'imprécision.

La seconde est le calcul du *deltaAngle*. Pour ça, on va sélectionner deux *Landmarks* aléatoirement. On va à l'aide de ces deux *Landmarks* calculer deux vecteurs :

- Le vecteur dans le monde réel entre les deux *Landmarks*
- et le vecteur dans Unity entre les deux *Landmarks*.

On calcule ensuite **l'angle entre ces deux vecteurs**. On fait cette étape trente fois pour ensuite faire une moyenne des angles obtenues. Cette moyenne définit notre *deltaAngle*.

On va privilégier le *Landmarks Orientation Provider* lorsqu'on a au moins 5 *Landmarks*. On stocke au maximum 50 *Landmarks*. On remplace à chaque fois le *Landmark* ayant la précision la plus mauvaise lorsqu'on arrive à 50. Il faut prendre en compte qu'il est difficile d'ajouter beaucoup de *Landmarks* lorsqu'on a que le GPS de disponible. En effet, en prenant en compte notre condition pour ajouter des *Landmarks* et étant donné que la précision du GPS est au mieux de trois mètres, on doit parcourir à chaque fois six mètres (trois plus trois) pour ajouter un *Landmark*.

5.1.3 Modification de l'interface

Avec l'implémentation de ces fournisseurs d'information il a fallu modifier la précédente interface. On souhaite avec cette nouvelle interface donner **plus de retour à l'utilisateur sur l'état de l'application et les informations qu'on reçoit**.



FIGURE 5.3 – Nouvelle interface de GLARE

On retrouve en bas à gauche (de gauche à droite) les différentes informations de géolocalisation :

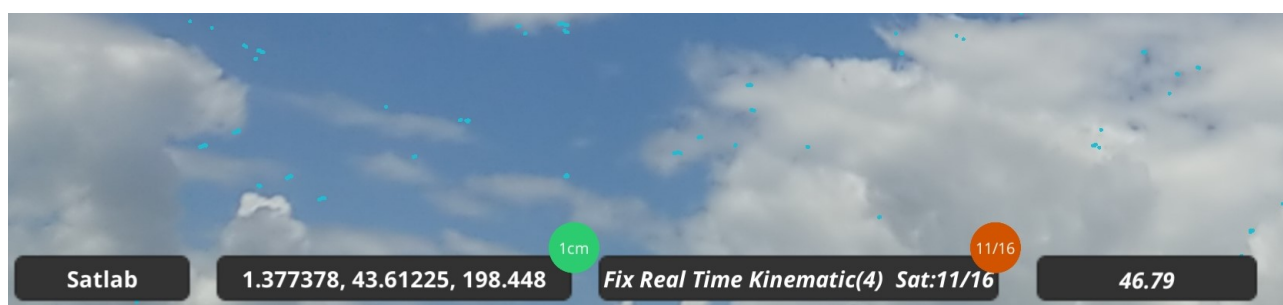


FIGURE 5.4 – Nouvelle interface de GLARE - Position

- Le fournisseur de la dernière géolocalisation reçue,
- la dernière position reçue,
- la précision de cette position, la couleur passe du rouge au vert plus la position est précise,
- les informations de Satlab : dans l'image précédente on peut voir que la correction RTK est disponible,
- le nombre de satellites vus par rapport au nombre de satellites disponibles (tra-

- ckés)
- et le champ texte pour ajuster l'altitude.

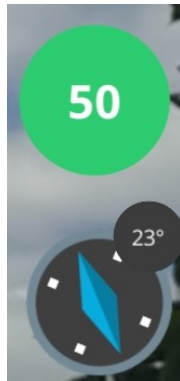


FIGURE 5.5 – Nouvelle interface de GLARE - Orientation

En bas à droite, on a la boussole et le nombre de *Landmarks* : Plus il y en a, plus le fond passe du rouge au vert.

Enfin, sur le haut de l'interface à droite, on va retrouver nos boutons (de gauche à droite) :

- un bouton nous permettant de poser les objets sur le sol, cette fonction est décrite par la suite (cf. page 74),
- le bouton pour passer sur la scène qui nous permettra de gérer les données,
- celui qui permettra d'utiliser la méthode manuelle de Mapbox,
- le bouton pour afficher la liste des appareils bluetooth disponibles
- et le bouton pour quitter l'application.

5.2 Fonctions additionnelles et corrections

5.2.1 Gestion de l'altitude

Pour nous permettre de mieux comprendre où se situent les objets dans l'espace, pour **éviter les différents problèmes de perspective** que l'on avait sur le premier prototype, on va donner la possibilité à l'utilisateur de plaquer au sol les objets. Pour ça, un bouton a été rajouté dans l'interface. Lorsque l'utilisateur clique dessus, tous les objets auront pour altitude zéro. Cette fonctionnalité est utilisée seulement pour avoir une notion du placement des objets en deux dimensions c'est à dire par rapport au Nord et à l'Est. On perd alors l'altitude réelle des objets.

Un autre problème se pose. La caméra dans Unity est défini à zéro lorsqu'on lance notre

application, pourtant nous tenons notre tablette à environ un mètre quarante du sol. Pour afficher parfaitement nos objets sur le sol il faut définir leur altitude à moins un mètre quarante. Pour l'instant, cette valeur est définie manuellement, on calculera par la suite automatiquement la hauteur de notre tablette par rapport au sol.

5.2.2 Correction de la mise à jour des positions

Après l'implémentation de Satlab je me suis heurté à une erreur de conception pour la mise à jour des positions. En effet, la position des objets dans Unity est définie en *Vector3* : les coordonnées (x,y,z) de l'objet dans Unity définies en *float*¹. Lorsque Satlab a été implémenté les résultats n'étaient pas excellents. Ce problème était du **aux limites du *float*** et à ma précédente conception. En effet, étant donné que la valeur de notre géolocalisation en UTM est très grande (de l'ordre du dix puissances six) et la valeur de la position de la caméra dans Unity est très petite, **on perd la précision au centimètre près** et la précision de la position de notre caméra dans Unity lorsqu'on calcule notre *deltaPosition*.

On va alors procéder différemment pour mettre à jour la position de nos objets. Dans les composants de nos objets qui héritent de la classe abstraite *AbstractAspect* on va directement envoyer la dernière géolocalisation et la position de la caméra. On va affecter à notre objet comme position celle de la caméra dans Unity. On pivote ensuite notre objet du *deltaAngle*. On va ensuite calculer **le vecteur de translation *T*** entre la position de l'objet dans le monde réel et la position de la caméra dans le monde réel (c'est à dire la dernière géolocalisation reçue). Il ne nous reste qu'à translater notre objet du vecteur *T*.

1. Type de donnée permettant de stocker une valeur réelle à virgule d'une taille de 32 bits

Chapitre 6

Conclusion

6.1 Bilan technique

6.1.1 Résultats obtenus

Après les modifications décrites précédemment, on peut clairement observer une modification sur le placement de nos objets. On a avec cette deuxième version, **un prototype fonctionnel**, capable d'**afficher en réalité augmentée toutes les géométries possibles avec précision**. Avec Glare, un utilisateur peut afficher ses données GeoJSON avec une précision au centimètre près. Pour présenter mes résultats, j'ai réutilisé le jeu de données présentées précédemment (cf. page 65) pour nous permettre de voir les différences entre les deux prototypes.

Sur la première image ci-dessous on se situe à l'Ouest du parking et on pointe notre caméra vers l'Est. On peut alors voir les 3 points.

Sur la deuxième image on est à l'Est du parking et on pointe notre caméra vers l'Est. On peut alors observer un des deux segment.

Sur la dernière image on regarde le polygone se trouvant le plus à l'Ouest du parking, c'est à dire le polygone sensé représenter le garage à vélo.



FIGURE 6.1 – Vue d'un point en RA dans le second prototype

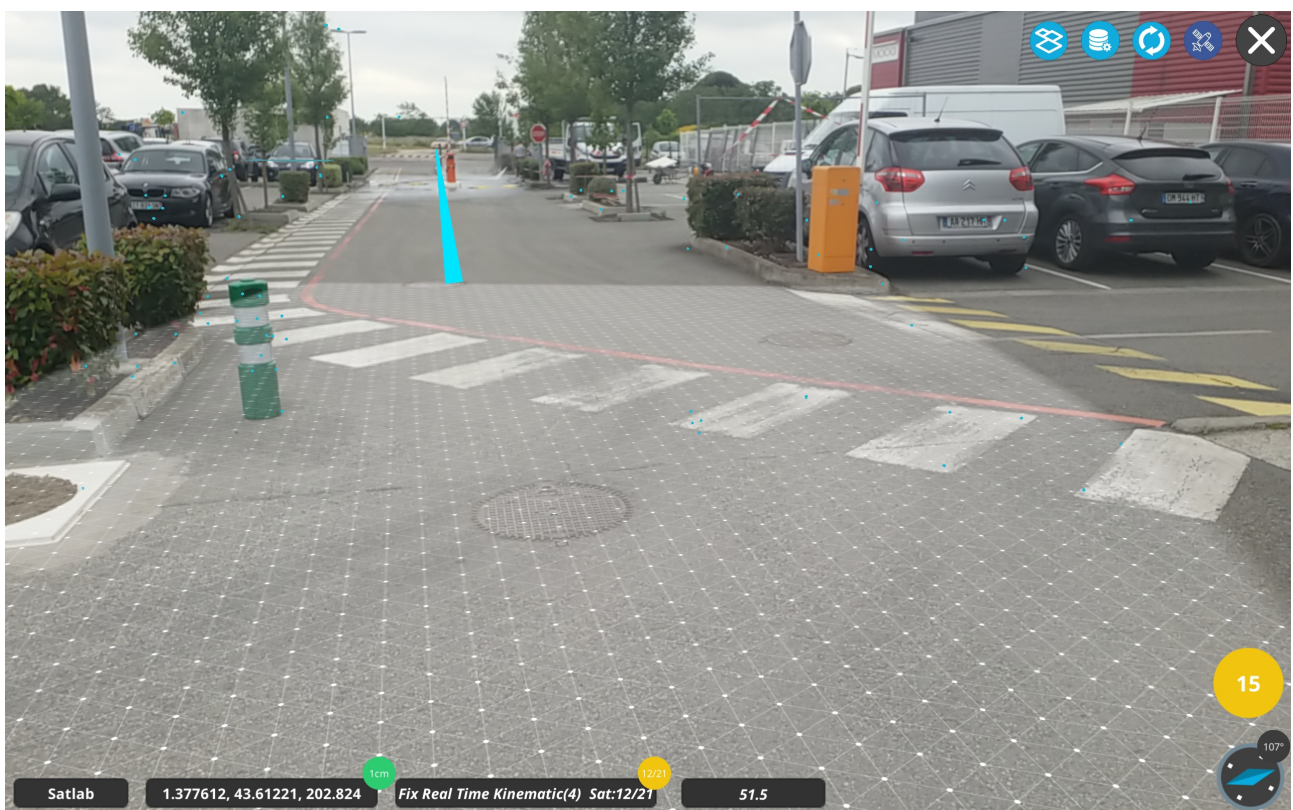


FIGURE 6.2 – Vue d'un segment en RA dans le second prototype



FIGURE 6.3 – Vue d'un polygone en RA dans le second prototype

On peut observer sur ces trois images l'importance de pouvoir plaquer au sol nos objets et de pouvoir jouer avec l'altitude. L'image la plus représentative de la précision de notre positionnement est la seconde image, celle qui présente un des segments (cf. page 77). On a à ce moment là un centimètre de précision pour la géolocalisation et quinze *Landmarks*. On peut alors constater que le segment est **parfaitement aligné et parfaitement placé**.

6.1.2 Limites perceptibles

Les principales limites de notre projet ne viennent pas du rendu mais plutôt du module externe, causées par certaines implémentations.

La première limite que l'on peut voir est que le deuxième objectif de mon stage n'est pas réalisé. En effet, **il n'est pas possible d'afficher les données Mission +** en réalité augmentée. Pour l'instant le module externe ne permet que de convertir des bases de données SpatiaLite (le système utilisé par Mission +). De plus, en testant de convertir des données RTE, je me suis rendu compte que le module externe n'arrive pas à parcourir toutes les données pour calculer les boîtes englobantes.

Un autre problème évident sur les données est que la plupart du temps l'altitude n'est pas fournie, c'est notamment le cas pour les données RTE. Aujourd'hui, les objets n'ayant pas

d'altitude définie sont affichés à hauteur de l'utilisateur. Cependant, cette définition **n'est pas valable pour la plupart des données géolocalisées**.

L'ergonomie de *Glare* pose aussi problème. Comme décrit précédemment (cf. page 68), il n'est pas possible d'éditer les fichiers de configuration directement dans l'application et il est difficile pour l'utilisateur de comprendre où se situent les objets dans son environnement.

Malgré les améliorations sur le placement de nos objets, on peut quand même observer des imperfections, surtout pour le placement des polygones (cf. page 78).

Enfin, notre amélioration de l'orientation (l'implémentation du *Landmarks Orientation Provider*) est dépendante de notre géolocalisation (plus d'informations page 72). En effet, si la géolocalisation est mauvaise nous ne pouvons pas poser assez de *Landmarks* pour fournir une bonne orientation. Nous avons donc **des imprécisions de géolocalisation et d'orientation lorsque Satlab (cf. page 31) n'est pas disponible**.

6.1.3 Fonctions additionnelles et pistes d'amélioration

Les fonctions additionnelles prévues lors de la conception vont vraiment apportées une plus-value au projet et répondre à la plupart des limites décrites précédemment.

La première est l'implémentation de la conversion du *Spatialite* (cf. page 21) vers le *JSONCollection* qui nous permettra d'**afficher les données provenant de Mission+**.

De plus, **une nouvelle tâche a été ajouté pour corriger les problèmes de blocage du module externe** lorsqu'on essaye de convertir un fichier trop volumineux. Je vais ajouter deux nouveaux champs : l'un pour préciser où l'on compte utiliser l'application (c'est à dire une géolocalisation) et un autre pour définir le rayon à partir de cette position. On va alors convertir uniquement les données comprises dans ce périmètre.

De plus, pour répondre au problème lié à la non-définition de l'altitude dans les données d'entrée, nous avons lors de la conception prévu de synchroniser les données d'entrée avec une carte d'altitude. Cette carte nous permettra de connaître pour chaque objet, son altitude précise.

Enfin les problèmes liés au rendu seront partiellement corrigés. Nous pourrons à la fin de la réalisation des fonctionnalités additionnelles ajouter des *Layers* et éditer les fichiers de configuration directement dans l'application. L'ajout des *Layers* a été prévu en chargeant des fichiers *JSONCollections* dans l'application. Cependant, on pourrait envisager de pouvoir ajouter des objets en **dessinant directement sur une carte**. Cette piste d'amélioration ajouterait à *Glare* plus d'autonomie et une grande plus-value.

Tous les problèmes d'ergonomie ne seront pas corrigés. Il n'est pas prévu dans les fonctions

additionnelles d'ajouter une mini-carte à notre interface. C'est une des pistes d'amélioration de *Glare*.

Comme autre piste d'amélioration, on pourrait imaginer pouvoir afficher des objets nous permettant d'indiquer une direction (avec un point de départ et un point d'arrivée).

6.1.4 Planning final

À la finalisation du rapport, c'est à dire le 14 Juin, on a dressé le planning final (cf. page 94). Étant donné que toutes les tâches n'ont pas été terminées, on estime que les tâches restantes seront réalisées dans le temps chiffré.

Fonctionnalité	Priorité	Temps	Réel
Rapport + Gestion de projet	1	3	4
Module de conversion	2	4	4
Linker et Helper	3	5	3
LayerFramework	4	10	8
LocationFramework et DirectionFramework	5	5	1,75
ARFramework	6	17	15,25
Total à ce stade		44	36
Amélioration de la géolocalisation et de l'orientation	7	15	18,5
Bilan	8	3	2,5
Total à ce stade		62	57
Amélioration et correction du module externe	9	0	2 (Estimation)
Fonctions additionnelles	10	21	21 (Estimation)
Total		83	80 (Estimation)

FIGURE 6.4 – Tableau final des tâches

À ce jour, malgré les trois jours et demi passés en plus sur l'amélioration de la géolocalisation et de l'orientation, j'ai **5 jours d'avance sur le temps chiffré**.

Malgré l'ajout de correction sur le module externe (cf. page 78), on estime que l'on aura 3 jours d'avance sur la fin du projet. Avec les 4 jours d'absence décrits dans le Gantt final, le projet finirait 1 jour plus tard que prévu c'est à dire le 18 Juillet (en sachant que mon stage se termine le 8 Août).

On va chercher à voir où j'ai gagné du temps par rapport au temps estimé, du premier jour de mon stage à la livraison du rapport, soit le 14 Juin. Pour ça, on cherche à comparer le

temps estimé et le temps réel sur les grands axes de mon stage (la réalisation du rapport de stage, le développement du prototype, etc).

Comparaison de la répartition prévisionnelle et réelle du temps de réalisation des grands axes de mon stage du premier jour (11/02) à la livraison finale (14/06)

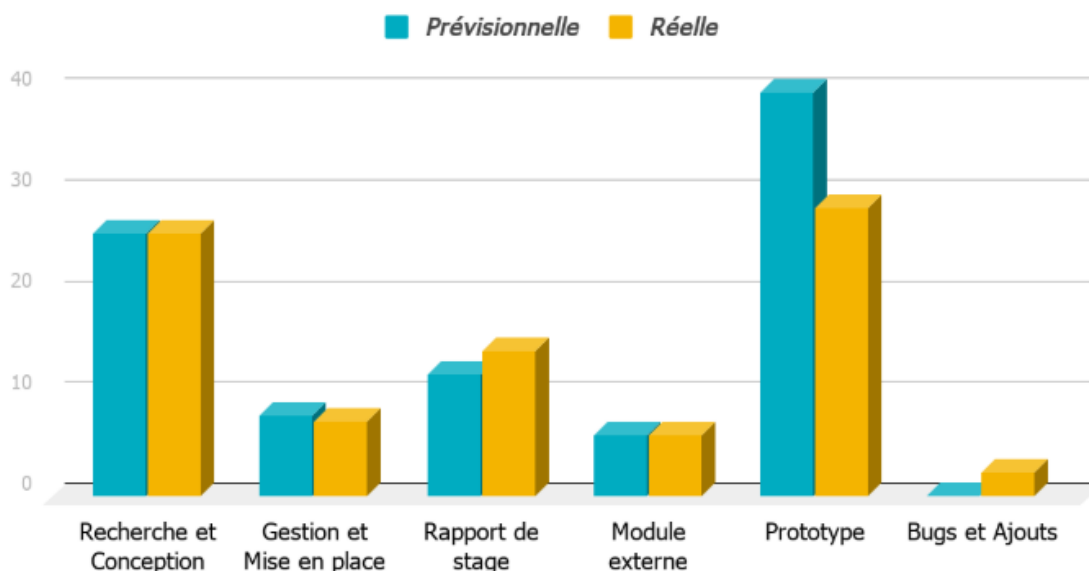


FIGURE 6.5 – Comparaison de la répartition du temps prévisionnel et final sur le stage

On se rend compte avec la figure ci-dessus que j'ai passé plus de temps que prévu sur le rapport de stage : **14 jours et demi au lieu de 12 prévus**. En effet, j'ai rédigé mon rapport tout au long de mon stage. Finalement, j'ai du revenir sur des parties que j'avais précédemment rédigées et je n'ai pas été **aussi efficace que prévu sur la rédaction**, notamment sur les parties *État de l'art* et *Conception*. Pendant le développement du projet, la plupart des tâches ont été réalisées **plus rapidement que prévu** comme par exemple l'intégration de la boussole et du GPS Android. J'ai finalement passé **28 jours et demi sur la réalisation du prototype au lieu de 40 estimés**. On peut voir que malgré l'ajout de fonctionnalités (comme le plaquage au sol des objets ou la réalisation de l'IHM) et certains imprévus (comme la correction de la mise à jour des positions) on a su tenir les délais de livraison.

On va maintenant chercher à voir quels ont été les modules de *Glare* où l'on a passé le plus de temps (de développement et de mise en place).

Jours de développement passés sur chaque module de l'application

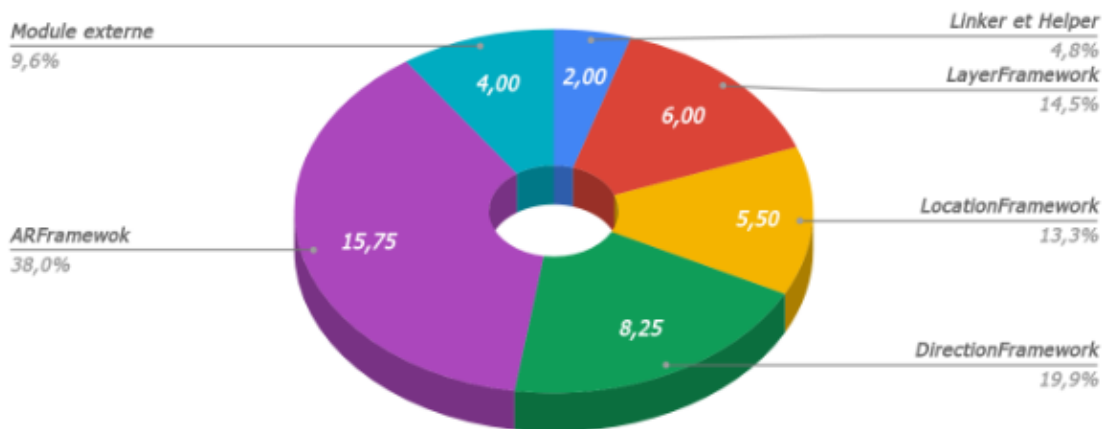


FIGURE 6.6 – Diagramme représentant le nombre de jours passés sur chaque module

L'intégration et la mise en place de la réalité augmentée dans notre projet a été la partie la plus longue de notre projet : **38%** du temps total. En effet, c'était la partie ayant le plus d'inconnus pour moi mais aussi le module demandant le plus de travail avec par exemple la création des *Prefabs* pour chaque géométrie. On voit aussi que le *DirectionFramework* a pris plus de temps que le *LocationFramework*.

6.2 Bilan humain

6.2.1 Difficultés rencontrées

J'avais précédemment réalisé un projet de recherche dans le cadre de mon Master 1 et nous nous étions confrontés au même problème : il est difficile dans un projet d'innovation de déterminer quand notre projet fonctionne. **Quand est-ce-qu'on définit que nos objets sont parfaitement placés?** Cette incertitude a entraîné beaucoup de remises en question et quelques retours en arrière dans mon intégration.

De plus, travailler dans un projet innovant implique que l'on a peu d'information. J'ai pu m'inspirer des différents projets et outils que j'avais analysé dans l'*État de l'art* mais j'ai surtout pu compter sur l'aide de mon tuteur en entreprise Raphaël Guenon qui a su me guider lorsque j'avais des interrogations. J'ai toujours pu **confronter mes résultats à ceux du projet RA-LS** ce qui m'a permis de me rendre compte de l'état de mon application.

J'ai cependant éprouvé quelques difficultés lorsque j'ai commencé à m'attaquer à des notions et algorithmes plus géométriques. Pour la mise à jour des positions, j'ai pris **beaucoup**

de temps à comprendre et à intégrer la projection du monde réel dans le référentiel Unity (cf. page 63). J'ai aussi eu beaucoup de mal à comprendre le fonctionnement de la méthode développée pour améliorer l'orientation (cf. page 30). Mais c'est quand je me suis confronté à ces difficultés que **j'ai appris le plus**, autant sur les bonnes pratiques de codage que sur la technique.

Un autre problème que j'ai rencontré, qui est lié à la nature même du projet, était les difficultés rencontrées pour tester mon application. Pour éventuellement voir les modifications effectuées, il me faut dans un premier temps *Build* l'application Android, c'est à dire demander à Unity de me fournir l'exécutable Android pour pouvoir ajouter l'application sur ma tablette : cette partie peut prendre jusqu'à une minute. Il me faut ensuite sortir du bâtiment pour obtenir les données GPS et voir les modifications apportées sur mes objets augmentés et éventuellement attendre que Satlab me fournisse une position. Il peut me falloir **jusqu'à 10 minutes pour tester mon application**.

6.2.2 Enseignements tirés et compétences acquises

Un des premiers enseignements que j'ai pu tirer de mon stage de fin d'études est **la nécessité de faire un état de l'art et une conception** avant le développement d'un projet. En effet, ces deux parties m'ont permis de déterminer et de comprendre les outils et algorithmes que j'allais développer. La conception m'a permis d'ordonner mon développement et ma gestion de projet, d'anticiper mon travail. Avec la méthode Kanban et la liste des tâches que j'avais défini lors de la conception, j'ai facilement pu suivre l'avancée de *Glare* et j'ai toujours su **qu'est-ce qu'il me restait à faire**.

Raphaël a toujours essayé de me donner des pistes sans me donner directement la solution. Cette démarche m'a **poussé à la réflexion** et m'a permis de toujours chercher les solutions **par moi-même**.

L'objectif du projet était souvent de **remettre en question les choix d'implémentation** et les solutions utilisées dans RA-LS. J'ai donc toujours remis en question mes méthodes en cherchant à fournir un travail meilleur et plus optimisé. Cependant, les algorithmes ou les technologies utilisés dans *Glare* sont pratiquement les mêmes que sur le projet RA-LS. Peut-être que j'ai été trop influencé par le projet, même si les différentes recherches que j'avais fait dans l'état de l'art m'ont poussé à faire ces choix.

On peut tout de même noter les différences entre les deux projets. Tout d'abord *Glare* permet d'afficher n'importe quelles données provenant d'un fichier GeoJSON et bientôt d'une base Spatialite. Le prototype offre aussi la possibilité d'afficher avec l'apparence que l'on souhaite les trois types de géométries possibles. Enfin, l'application a été conçue pour être

le plus extensible possible. On pourrait facilement changer de fournisseur de géolocalisation ou exporter le projet sur un appareil Apple par exemple.

Glare a été très enrichissant sur le plan technique. J'avais déjà réalisé un projet en réalité virtuelle sur Unity mais je n'avais jamais fait de réalité augmentée. J'ai appris énormément sur la RA et plus précisément sur **ARCore** mais aussi sur le **développement pour Android**. J'ai aussi pu mettre en pratique certaines notions trigonométriques et mathématiques apprises en cours comme la projection, même si leurs implémentations ont été compliquées pour moi.

Je n'avais aucune connaissance dans les systèmes d'information géographique (**SIG**) et dans les systèmes de projection cartographique. C'était très intéressant de travailler sur ces sujets et ces notions vont sûrement m'être utiles pour la suite de ma vie professionnelle à Toulouse.

Travailler dans une grande entreprise était totalement nouveau. J'avais quelques aprioris mais je voulais m'y confronter pour comprendre la vie en entreprise. Je me suis finalement épanoui chez Capgemini où l'on m'a laissé réaliser mon projet et m'organiser seul. J'ai apprécié **la confiance que l'on m'a donné** et la liberté que j'ai eu sur la réalisation de *Glare*.

Bibliographie

- [1] ARCGis Blog de ESRI, *L'application AUGeo*.
<https://www.esri.com/arcgis-blog/products/3d-gis/3d-gis/ar-for-your-gis/>
- [2] Site de Trimble, *Trimble Connect*.
<https://mixedreality.trimble.com/>
- [3] Site de Trimble, *Trimble SiteVision*.
<https://sitevision.trimble.com/>
- [4] Internet Engineering Task Force (IETF) , *Format GeoJSON*.
<https://tools.ietf.org/html/rfc7946>
- [5] OpenGeospatial (OGC) , *Format KML*.
<http://www.opengeospatial.org/standards/kml/>
- [6] PostGIS.fr , *Définition de PostGIS*.
<http://www.postgis.fr/chrome/site/docs/workshop-foss4g/doc/>
- [7] Wikipédia , *Définition de SpatiaLite*.
<https://fr.wikipedia.org/wiki/SpatiaLite>
- [8] Bel Horizon , *Définition de l'UTM*.
<http://www.bel-horizon.eu/la-cartographie/notions-de-cartographie/projections-utm-et-variantes.html>
- [9] IGN , *Lambert 93*.
<https://geodesie.ign.fr/contenu/fichiers/documentation/rgf93/Lambert-93.pdf>
- [10] Wikipédia , *Projection Conique conforme de Lambert*.
<https://fr.wikipedia.org/>
- [11] Google , *SDK ARCore*.
<https://developers.google.com/ar/>
- [12] Medium , *Comparing ARCore and ARKit*.
<https://medium.com>

- [13] Viuz, *Le marché potentiel de l'AR supérieur à la VR*.
<https://viuz.com/>
- [14] Apple, *ARKit 2*.
<https://developer.apple.com/arkit/>
- [15] Github/ProjectDent, *ARKit-CoreLocation*.
<https://github.com/ProjectDent/ARKit-CoreLocation>
- [16] Wikitude, *Location-based Augmented Reality*.
<https://www.wikitude.com/geo-augmented-reality/>
- [17] Youtube, *Unity 5 : Location-Based markerless Augmented Reality Tutorial*.
<https://www.youtube.com/>
- [18] CNES, *PPP-Wizard*.
<http://www.ppp-wizard.net/>
- [19] Github/eltorio, *RtkGps+*.
<https://github.com/eltorio/RtkGps>
- [20] Mapbox, *Location-based AR Platform*.
<https://www.mapbox.com/augmented-reality/>
- [21] Satlab, *SLC Multi-Purpose GNSS Receiver*.
https://www.satlab.com.se/product_uri/satlab-slc-multi-purpose-gnss-receiver/
- [22] Vimeo, *Projet RTE développé par Capgemini*.
<https://vimeo.com/301155823/4811a182f8>
- [23] Github/timokorkalainen, *Unity-GeoJSONObject*.
<https://github.com/timokorkalainen/Unity-GeoJSONObject>
- [24] iFlexion, *ARCore vs ARKit*.
<https://www.iflexion.com/blog/arcore-vs-arkit-better-building-augmented-reality-apps/>
- [25] Réseaux énergie, *Données RTE*.
<https://opendata.reseaux-energies.fr/pages/accueil/>
- [26] Github, *Triangulation de Delaunay sur Unity*.
<https://github.com/mattatz/unity-triangulation2D>
- [27] Wikipédia, *Norme NMEA*.
<https://fr.wikipedia.org/>

Annexe A

Liste des fonctionnalités

A.1 Les fonctionnalités essentielles

L'application doit permettre :

1. Afficher en RA des informations cartographiques
 - (a) Obtenir les données et leurs apparences
 - i. Parser des fichiers de configuration JSON
 - ii. Parser des JSONCollection
 - (b) Instancier les données cartographiques selon les paramètres choisis dans les configurations
 - i. Conversion des coordonnées en UTM
 - (c) Affichage en RA
 - i. Implémentation d'ARCore
 - ii. Afficher un Point en RA
 - iii. Afficher une Ligne en RA
 - iv. Afficher un Polygone en RA
 - (d) Localiser l'utilisateur et les données
 - i. Géolocalisation
 - A. Par le GPS Android
 - B. Par un appareil externe
 - ii. Orientation

A. Par la boussole Android

B. Par les Landmarks (Landmarks Orientation Provider)

Le module externe doit :

1. Standardiser des informations cartographiques 3D de toutes sources
 - (a) GeoJSON vers le JSONCollection

A.2 Les fonctionnalités secondaires

Dans un second temps, l'application doit permettre de :

1. S'aligner avec une carte 3D manuellement
2. Éditer l'apparence des informations cartographiques
 - (a) Éditer les fichiers de configuration
3. Ajouter des données JSONCollection
 - (a) Ajouter des Layers dans un fichier de configuration
 - (b) Sélectionner des prefabs parmi ceux disponibles dans le dossier Resources

Enfin, le module externe doit ensuite :

1. Standardiser des informations cartographiques 3D de toutes sources
 - (a) Spatialite vers JSONCollection
 - (b) Synchroniser avec une carte d'altitude lorsque l'altitude n'est pas donnée dans les données en entrée

Annexe B

Cas d'utilisation

Les cas d'utilisation du module externe permettant de convertir des données géospatiales vers du JSONCollection.

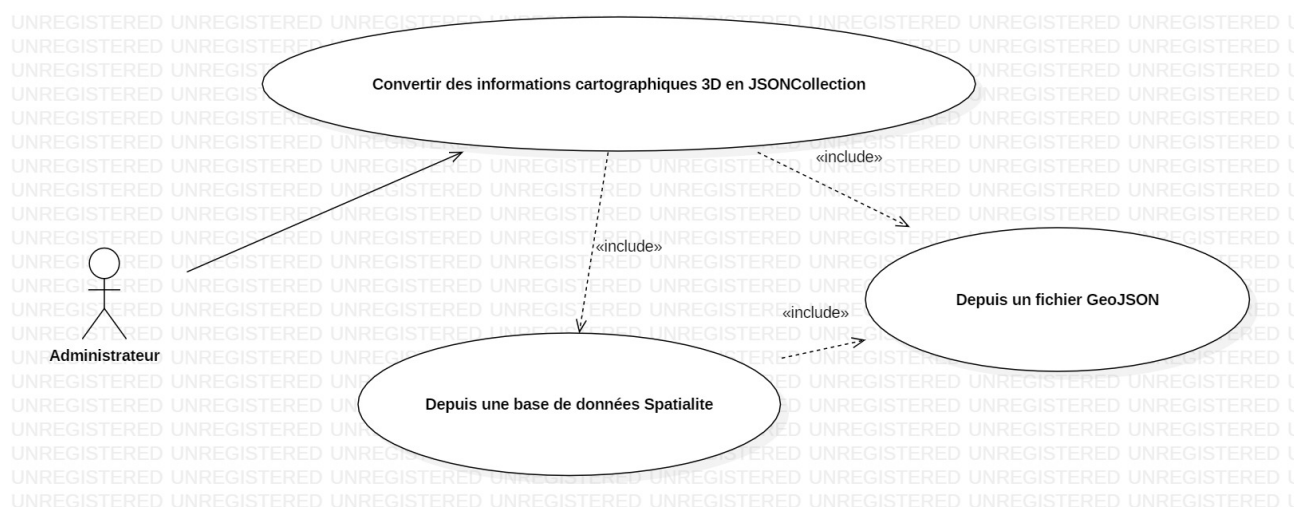


FIGURE B.1 – Cas d'utilisation du module externe

Les cas d'utilisation de l'application de Réalité Augmentée Géolocalisée.

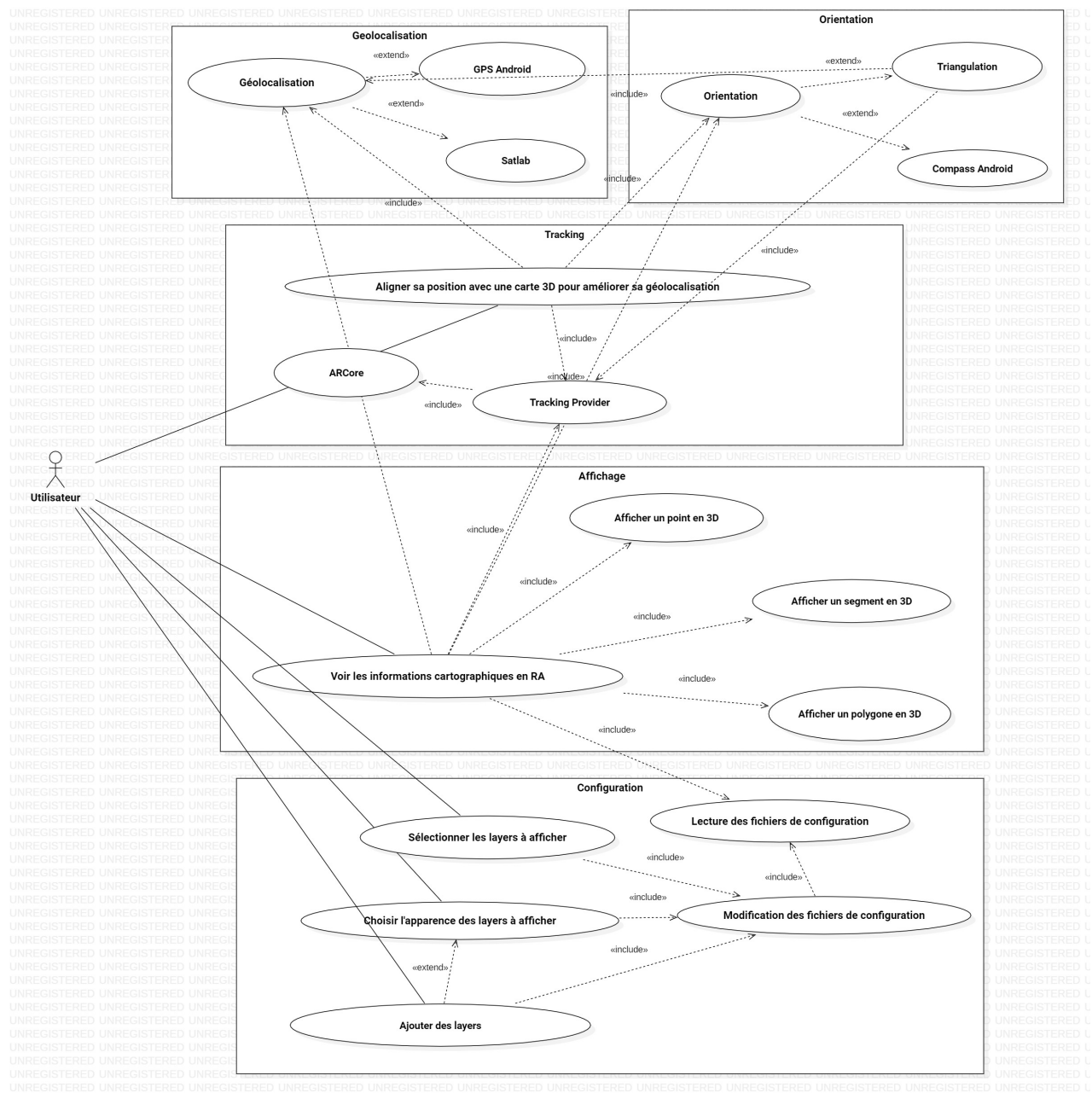


FIGURE B.2 – Cas d'utilisation de l'application

Annexe C

Diagramme de classe

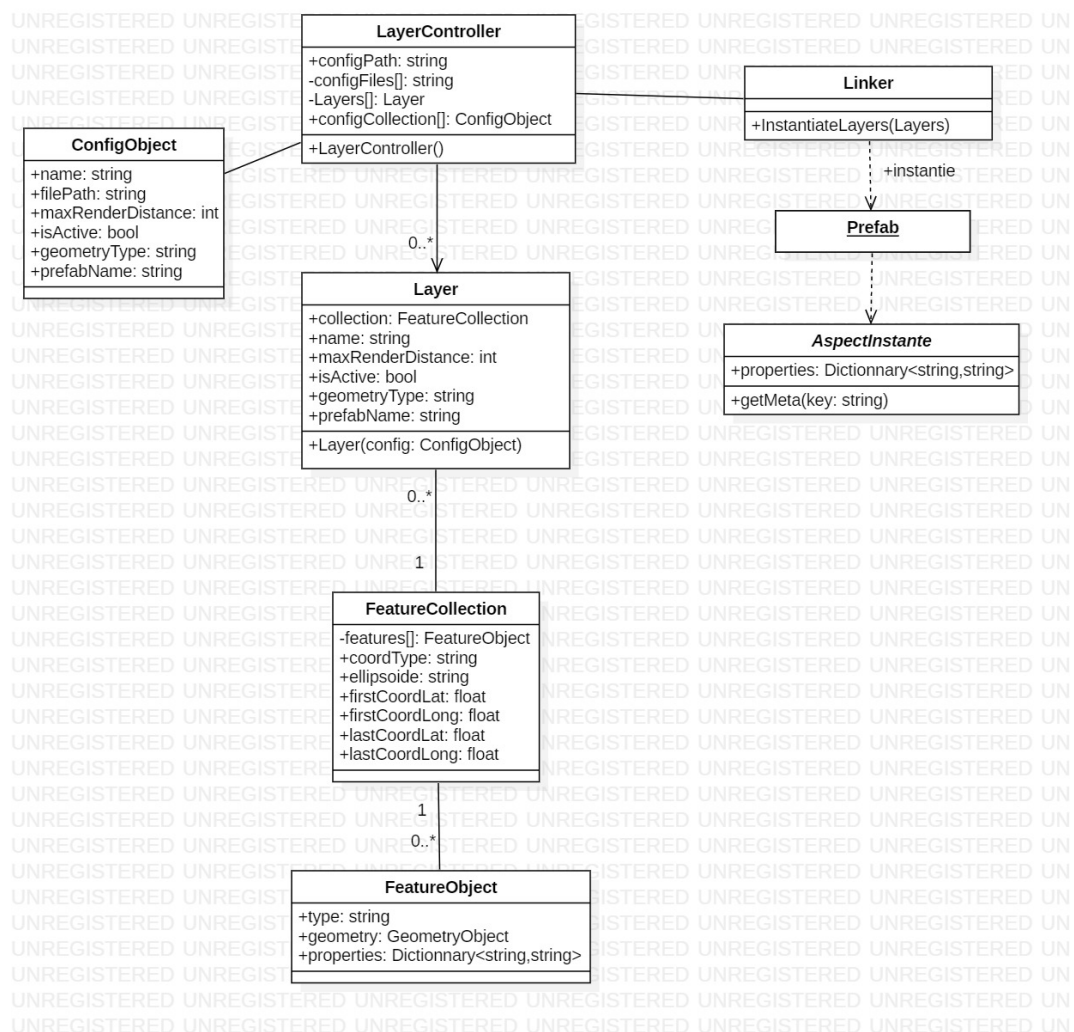


FIGURE C.1 – Diagramme de classe du LayerFramework

Annexe D

Planning prévisionnel

Le planning prévisionnel de la réalisation de l'application sous forme de Diagramme de Gantt.

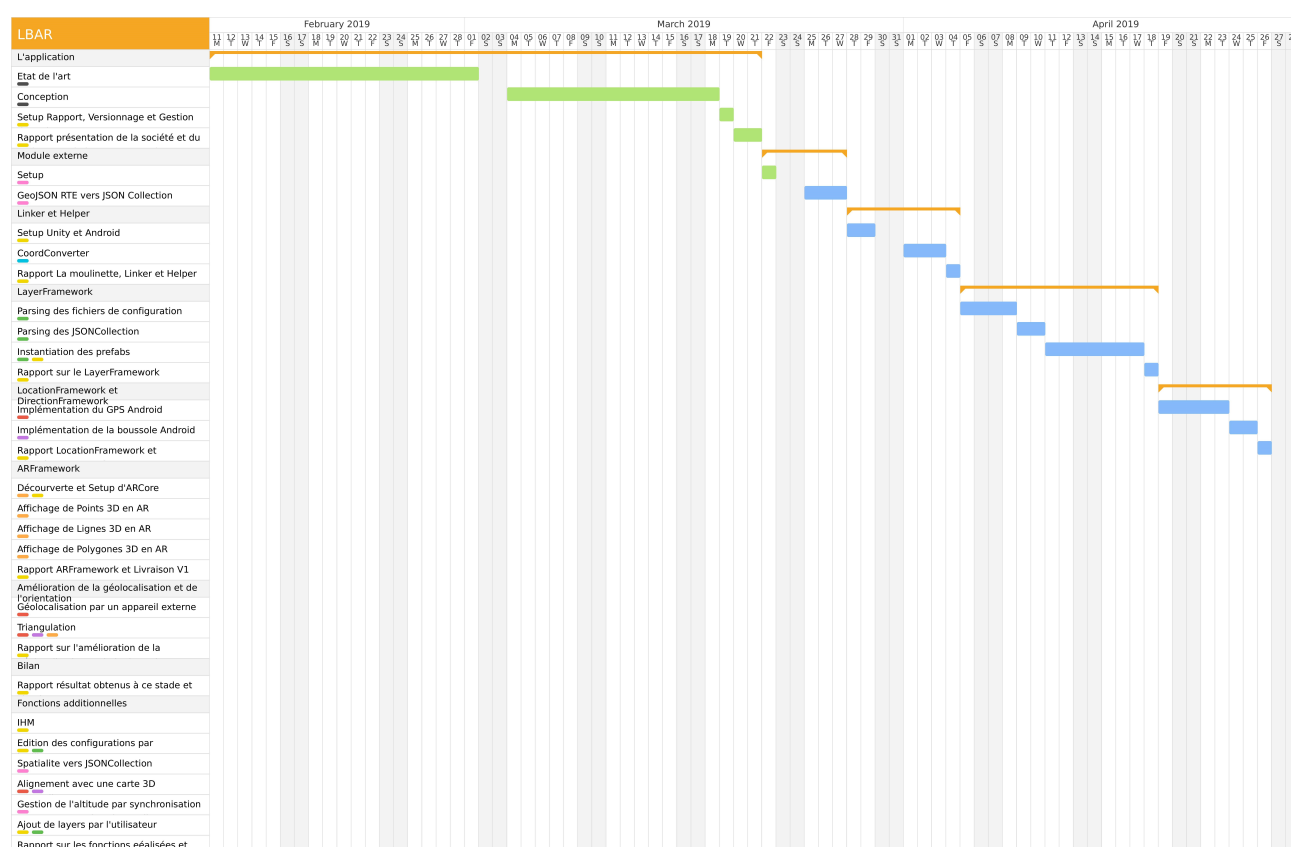
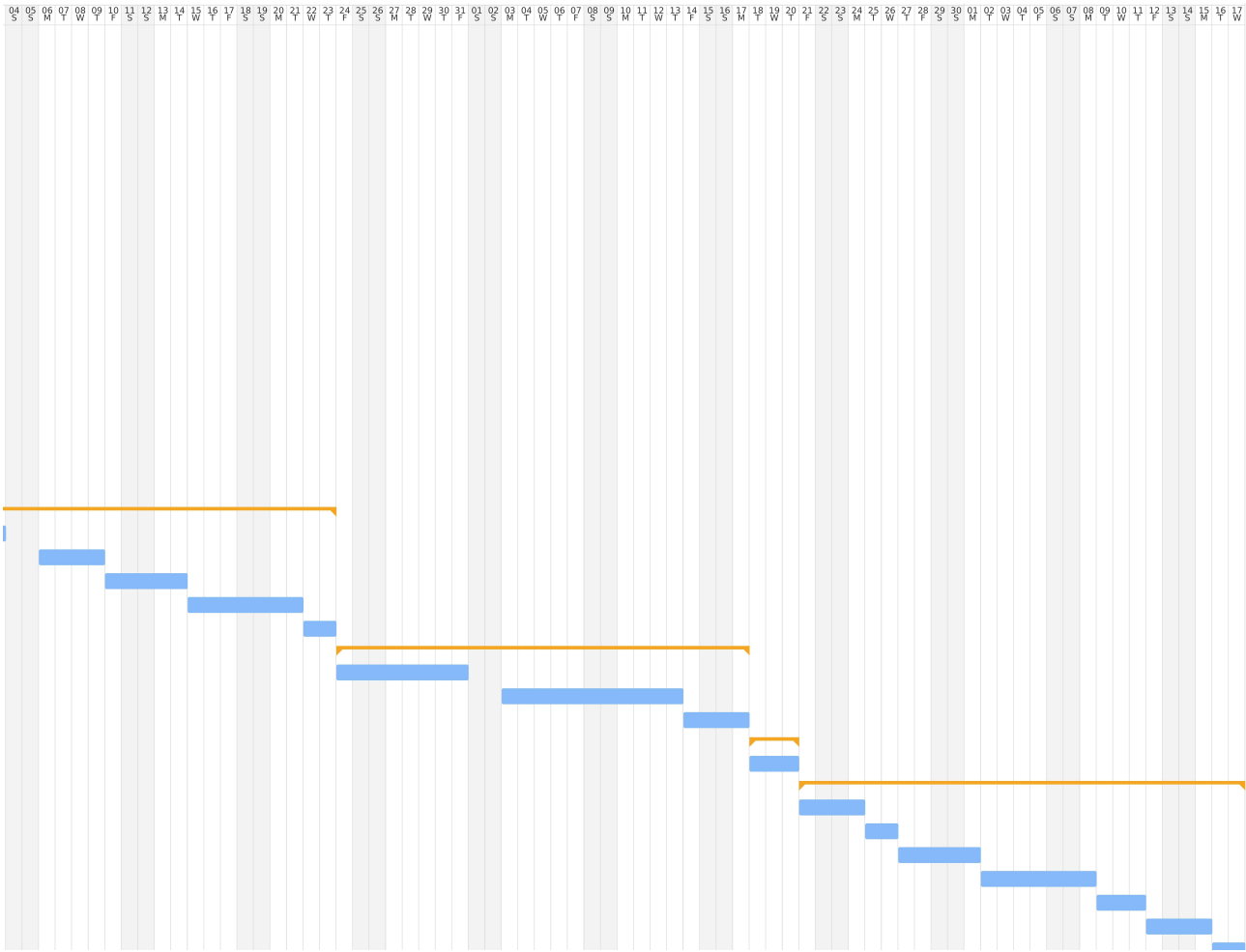


FIGURE D.1 – Planning prévisionnel - Diagramme de Gantt (Partie 1)



Annexe E

Planning final

Le planning final de la réalisation de l'application sous forme de Diagramme de Gantt.

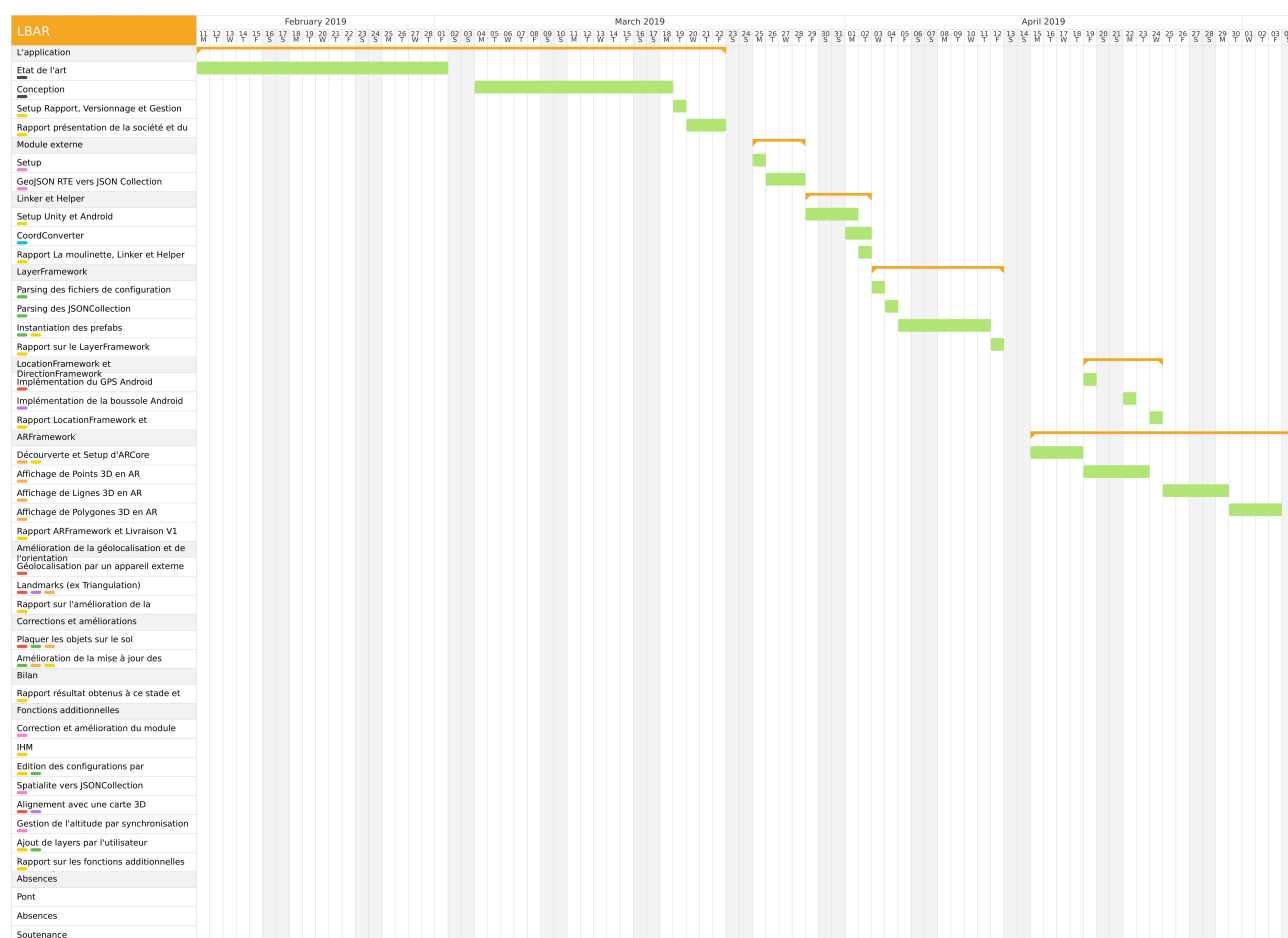


FIGURE E.1 – Planning final - Diagramme de Gantt (Partie 1)

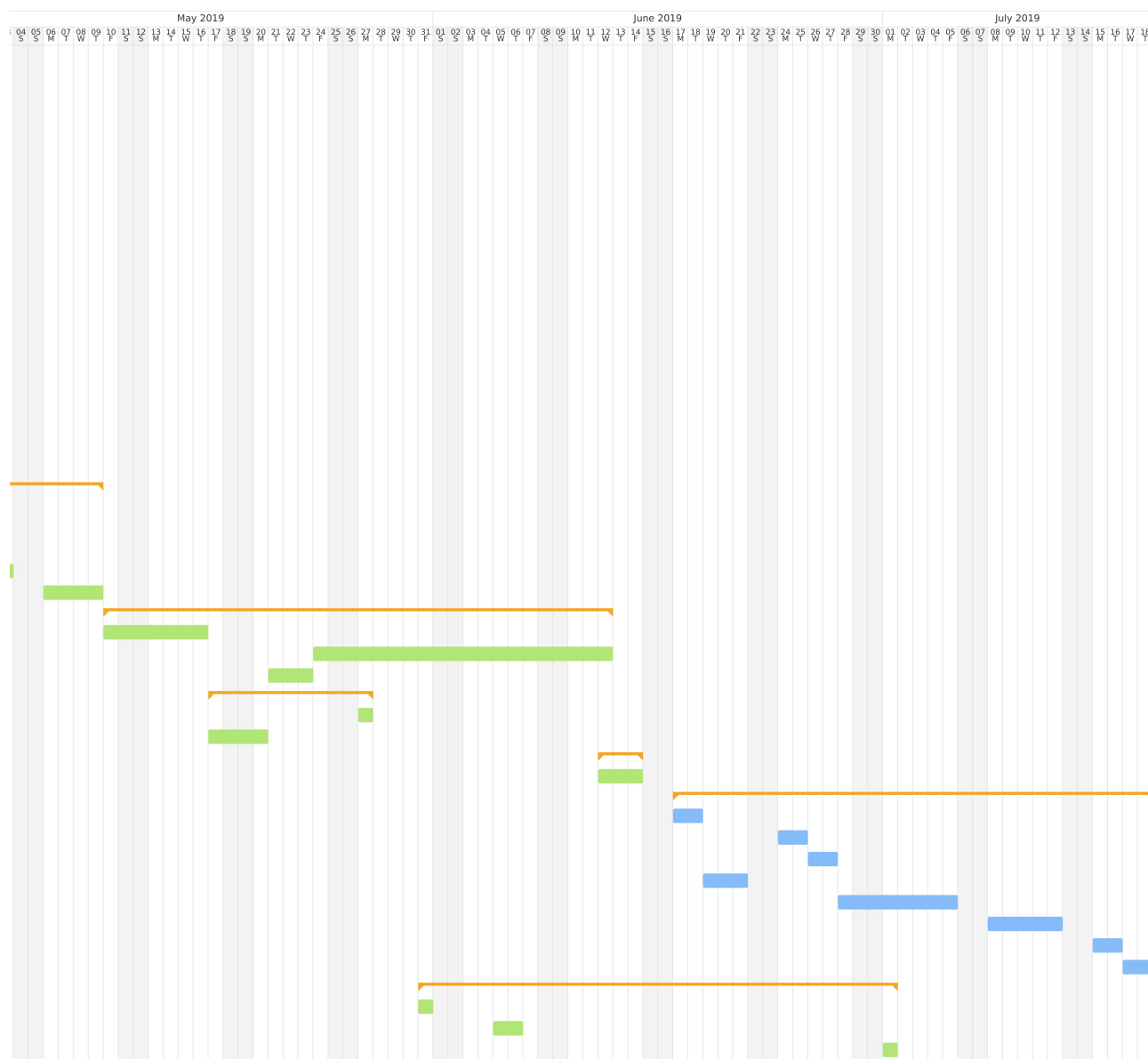


FIGURE E.2 – Planning final - Diagramme de Gantt (Partie 2)