HEIG VD
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Tuilage de données géospatiales pour le métaverse

Travail de Bachelor - Rapport intermédiaire

Département TIC

Filière Informatique et systèmes de communication

Orientation Informatique logicielle

Ian Escher

24.05.2024

Supervisé par :
Prof. B. Chapuis (HEIG-VD)

Hes·so
Haute Ecole Spécialisée de Suisse occidentale

Préambule

Ce travail de Bachelor (ci-après **TB**) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'École.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD
Le Chef du Département

Yverdon-les-Bains, le 24.05.2024

Authentification

Je soussigné, Ian Escher, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Ian Escher

Yverdon-les-Bains, le 24.05.2024

Résumé

Le travail devant être effectué durant ce travail consiste à utiliser la spécification 3D Tiles Next [1] de Cesium [2] pour *streamer* un rendu 3D de tuiles vectorielles à l'intérieur du framework Baremaps [3] existant. Le produit final sera un prototype du support de cette spécification en utilisant une base de données Postgis [4]. Les fonctionnalités offertes 3D Tiles Next seront pleinement utilisées pour produire un rendu de haute qualité et performant.

Les rendus 3D que propose Cesium peuvent être distribués en deux catégories :

1. Le terrain géographique
2. Les bâtiments

Produire le rendu du terrain ainsi que le rendu des bâtiments comporte chacun ses propres difficultés d'optimisation. Un système de *level of details* devra être implémenté pour maintenir de hautes performances, même avec un nombre important de géométries affichées à l'écran.

Cependant, dans le cadre de ce travail, seul l'affichage des bâtiments sera traité. Pour cela, la spécification 3D Tiles Next propose une solution d'optimisation interne à son fonctionnement avec Cesium. Néanmoins, beaucoup reste à être fait quant à l'affichage des bâtiments ainsi que pour créer un système permettant de *streamer* les informations de la base de données d'OpenStreetMap vers Cesium.

1. https ://cesium.com/blog/2021/11/10/introducing-3d-tiles-next/
2. https ://cesium.com/
3. https ://github.com/baremaps/baremaps
4. https ://postgis.net/

Table des matières

Table des figures

Chapitre 1

Introduction

Le but de cette première partie de travail de Bachelor était de prendre en main le framework Baremaps, de comprendre son fonctionnement et de commencer à l'adapter pour qu'il puisse être utilisé avec Cesium JS.

Dans ce rapport intermédiaire, je vais commencer par vous présenter rapidement la structure du framework Baremaps. Une fois cela fait, cela posera les bases pour que je puisse vous expliquer les différentes tâches que j'ai effectué jusqu'à présent ainsi que les tâches futures.

Chapitre 2

Présentation du framework Baremaps

Le framework Baremaps est constitué de plusieurs modules, certains utilisés par tous les projets utilisant le framework, d'autres n'étant utilisés que par certains projets. Parmi les modules principaux, nous avons les modules `baremaps-cli`, `baremaps-core`, `baremaps-server` ainsi que `basemap`. Parmi les différents points abordés par ce rapport intermédiaire, seuls ces modules seront concernés. Je ne vais donc pas m'attarder ici sur les autres modules du framework.

2.1 baremaps-cli

Le module `baremaps-cli` est un module permettant de lancer des commandes en ligne de commande pour effectuer des opérations générales avec le framework Baremaps. Ce module est utilisé pour lancer des commandes telles que l'import de données OpenStreetMap ou le lancement d'un serveur web pour afficher des données OpenStreetMap.

2.2 baremaps-core

Le module `baremaps-core` est un module regroupant la logique métier du framework Baremaps. Ce module est utilisé pour effectuer des opérations telles que la génération de tuiles vectorielles à partir de données OpenStreetMap. C'est dans ce module que la majorité du code qui me servira et que j'écrirai se trouve.

2.3 baremaps-server

Le module `baremaps-server` est le module qui se charge de lancer un serveur web pour afficher des données OpenStreetMap. Ce module comporte donc les ressources nécessaires pour construire la page web affichant les données OpenStreetMap. C'est dans ce module que Cesium JS sera utilisé et configuré.

2.4 basemap

Le module `basemap` est le module permettant de'importer le fichier OpenStreetMap et de le transformer en une base de données PostgreSQL/PostGIS. À l'heure ou j'écris ce rapport, je n'ai pas encore eu à travailler sur ce module.

2.5 Fondements écrits par Antoine Drabble

Afin d'évaluer si le projet s'apprêtais à être utilisé dans le cadre d'un travail de Bachelor, M. Antoine Drabble a effectué un travail de recherche et a produit un prototype de visualisation de bâtiments 3D à l'intérieur du client Cesium JS sur lequel je vais m'appuyer. Ce prototype, bien qu'incomplet, pose les fondements de l'utilisation de Cesium JS, la spécification 3D Tiles et le framework Baremaps les uns avec les autres. Ce prototype est donc un bon point de départ pour mon travail.

Il n'est pas pertinent de détailler ici le travail effectué par M. Antoine Drabble fichier par fichier mais j'expliquerai au fur et à mesure de ce rapport ce qui a été fait et ce que je dois modifier ou ajouter pour remplir mes objectifs.

Chapitre 3

Utilisation de la base de donnée OpenStreetMap

Comme décrit en page 3 section 2.4, le module Basemap se charge d'importer le fichier OpenStreetMap et de le transformer en une base de données PostgreSQL/-PostGIS. Une fois la base de donnée générée, il faut maintenant pouvoir récupérer et traiter les données correctement.

3.1 Extraction des données et définition d'un bâtiment

La partie nous intéressant dans la base de données OpenStreetMap est la table `osm_nodes`. Cette table contient les données de tout les bâtiments. Ceux-ci ne contiennent généralement qu'une Géométrie, une structure de donnée contenant plusieurs points et formant un polygone. Parfois seulement, des Tags peuvent être ajoutés. Ces tags peuvent contenir des informations sur la hauteur du bâtiment, sa couleur, son matériau, etc.

La documentation OpenStreetMap [1] nous permet d'avoir la liste complète des tags existants et nous intéressant concernant les bâtiments.

Pour pouvoir récupérer toutes ces informations, j'utilise la requête SQL suivante :

```
SELECT st_asbinary(geom),
    tags -> building,
    tags -> height,
    tags -> building:levels,
    tags -> building:min_level,
    tags -> building:colour,
    tags -> building:material,
    tags -> building:part,
    tags -> roof:shape,
    tags -> roof:levels,
    tags -> roof:height,
    tags -> roof:color,
    tags -> roof:material,
    tags -> roof:angle,
    tags -> roof:direction,
    tags -> amenity
FROM osm_ways
WHERE (tags ? building or tags ? building:part) and
    st_intersects(geom, st_makeenvelope(%1$s, %2$s, %3$s, %4$s, 4326))
LIMIT %5$s;
```

`%1$s, %2$s, %3$s, %4$s, et %5$s` sont des espaces réservés qui seront remplacés par les coordonnées de la zone à charger lors de l'exécution de la requête.

1. https ://wiki.openstreetmap.org/wiki/Simple_3D_Buildings

3.2 Couleurs des bâtiments

Concernant la couleur des bâtiments, il est possible de récupérer la couleur de base du bâtiment en utilisant le tag `building:colour`. Cependant, ce tag ne donne que le mot en anglais de la couleur et non pas son code RGB, HEX ou HSL.

Pour remédier à cela, j'ai créé la classe `ColorUtils` [2] qui contient une map statique de toutes les couleurs selon le standard définit par W3Schools [3]. Cette classe permet de convertir le nom de la couleur en code HEX :

```
1  public class ColorUtility {
2    private static final Map<String, String> colorMap;
3
4    static {
5      colorMap = new HashMap<>();
6      colorMap.put("aliceblue", "f0f8ff");
7      // ...
8      colorMap.put("yellowgreen", "9acd32");
9    }
10
11   public static Color parseName(String color) {
12     // Calcul des valeurs RGB en fonction du code HEX.
13     return new Color(r, g, b);
14   }
15 }
```

`Color` [4] étant un record, il est possible de créer une nouvelle couleur en passant les valeurs RGB en paramètre.

Une fois la couleur RGB récupérée, il est possible de l'utiliser dans la classe `GltfBuilder` [5] pour donner la couleur au bâtiment lors de sa modélisation.

2. baremaps-core/src/main/java/org/apache/baremaps/tdtiles/utils/ColorUtility.java
3. https ://www.w3schools.com/cssref/css_colors.php
4. baremaps-core/src/main/java/org/apache/baremaps/tdtiles/utils/Color.java
5. baremaps-core/src/main/java/org/apache/baremaps/tdtiles/GltfBuilder.java

3.3 Hauteur des bâtiments

Le calcul de la hauteur des bâtiments doit se faire en fonction de la définition [6] des bâtiments par OSM. Si aucun tags ne donne la hauteur du bâtiment, il aura par défaut une hauteur de 10 mètres.

Sinon, la hauteur du bâtiment est calculée en fonction des tags `height`, `building:levels`, `building:min_level`, `roof:height` et `roof:levels` :

- On commence avec une hauteur de bâtiment de 0 mètres.
- Si aucun tag n'est présent,
  - La hauteur du bâtiment est de 10 mètres.
- Sinon, si le tag `height` est présent,
  - La hauteur du bâtiment est égale à la valeur du tag `height`.
  - Si le tag `roof:height` est présent,
    - La hauteur du toit `roof:height` doit être soustraite de la hauteur du bâtiment
- Sinon,
  - Si le tag `building:levels` est présent,
    - La hauteur du bâtiment est égale à la valeur du tag `building:levels` multipliée par 3 mètres.
  - Si le tag `roof:levels` est présent,
    - La hauteur du toit `roof:levels` multipliée par 3 mètres doit être additionnée à la hauteur du bâtiment.
  - Si le tag `building:min_level` est présent,
    - La hauteur du vide en dessous du bâtiment est égale à la valeur du tag `building:min_level` multipliée par 3 mètres.

La hauteur connue, on peut la transmettre à la classe `GltfBuilder` [7] pour donner la hauteur au bâtiment lors de sa modélisation.

6. https ://wiki.openstreetmap.org/wiki/Simple_3D_Buildings
7. baremaps-core/src/main/java/org/apache/baremaps/tdtiles/GltfBuilder.java

3.4 Géométries des bâtiments

La classe `GltfBuilder` a donc accès à la couleur et à la hauteur du bâtiment. Ces informations ainsi que les autres tags lui sont transmis à travers le record `Building` [8] et `Roof` [9].

Outre ces informations, `Building` comporte aussi une `Geometry` [10] qui nous sert à déterminer le polygone au sol du bâtiment. Grâce à ce polygone, nous pouvons extruder le bâtiment en 3D. Cependant une `Geometry` ne contient qu'une liste de points et il faut une liste de triangles pour pouvoir modéliser le bâtiment en 3D. Pour cela, la méthode `DelaunayTriangulationBuilder` [11] était utilisée. Le problème était que la *Delaunay triangulation* [12] ne fonctionne pas avec des polygones concaves. Une version complémentaire de cette méthode est la *Constrained Delaunay triangulation* [13] qui fonctionne avec des polygones concaves en définissant des segments comme bords du polygone. Cette version de l'algorithme est aussi disponible dans la librairie JTS [14] mais elle propose aussi la classe `PolygonTriangulator` [15] qui permet aussi de faire une triangulation de polygones concaves de manière moins optimisée mais plus rapide que la *Constrained Delaunay triangulation*. C'est donc cette classe que j'ai utilisée pour trianguler les bâtiments.

Grâce à cette nouvelle triangulation, il est possible de modéliser les bâtiments en 3D correctement, même ceux comprenant des cours intérieures ou des trous dans leur structure.

La classe `DelaunayTriangulationBuilder` comportait cependant un paramètre `tolerance` qui permettait de définir la distance minimale entre deux points afin d'optimiser le coût de la triangulation. Cette possibilité n'est pas disponible dans la classe `PolygonTriangulator`, néanmoins j'implémente une alternative dans le chapitre suivant.

8. baremaps-core/src/main/java/org/apache/baremaps/tdtiles/building/Building.java
9. baremaps-core/src/main/java/org/apache/baremaps/tdtiles/building/Roof.java
10. org.locationtech.jts.geom.Geometry
11. org.locationtech.jts.triangulate.DelaunayTriangulationBuilder
12. https ://en.wikipedia.org/wiki/Delaunay_triangulation
13. https ://en.wikipedia.org/wiki/Constrained_Delaunay_triangulation
14. https ://locationtech.github.io/jts/
15. https ://locationtech.github.io/jts/javadoc/org/locationtech/jts/triangulate/polygon/PolygonTriangulator.html

Chapitre 4

Optimisation de l'affichage des bâtiments

Le plus gros problème lors de l'utilisation de l'application actuellement est que lorsque l'on déplace la caméra dans une nouvelle zone de la carte, les bâtiments prennent parfois plus d'une dizaine de secondes à s'afficher tout en bloquant le site web. De plus, les bâtiments apparaissent de manière à priori aléatoire et non au plus proche de la caméra.

Je vais, dans ce chapitre, aborder différentes possibilités d'optimisation.

4.1 Implicit tiling

Avant de chercher à optimiser l'affichage des bâtiments, il est important de comprendre comment les bâtiments sont actuellement affichés. La procédure entière comporte plusieurs étapes. Le client Cesium JS commence par charger le fichier `tileset.json`[1] qui contient les informations sur la manière dont il doit charger les tuiles. Parmi ces informations, on trouve les URI auxquelles il doit envoyer des requêtes pour obtenir les tuiles mais aussi les informations nécessaires à définir l'`implicit tiling` :

```
"implicitTiling" : {
    "subdivisionScheme" : "QUADTREE",
    "subtreeLevels" : 6,
    "availableLevels" : 18,
    "subtrees" : {
        "uri" : "/subtrees/{level}.{x}.{y}.json"
    }
}
```

L'Implicit tiling est une technique qui consiste à diviser la carte en tuiles de manière récursive. Tant que le `level` de division n'a pas atteint la valeur maximum `availableLevels`, on divise la tuile dans laquelle nous nous trouvons en quatre tuiles de même taille. On peut voir un exemple de cette division sur la figure 4.1.

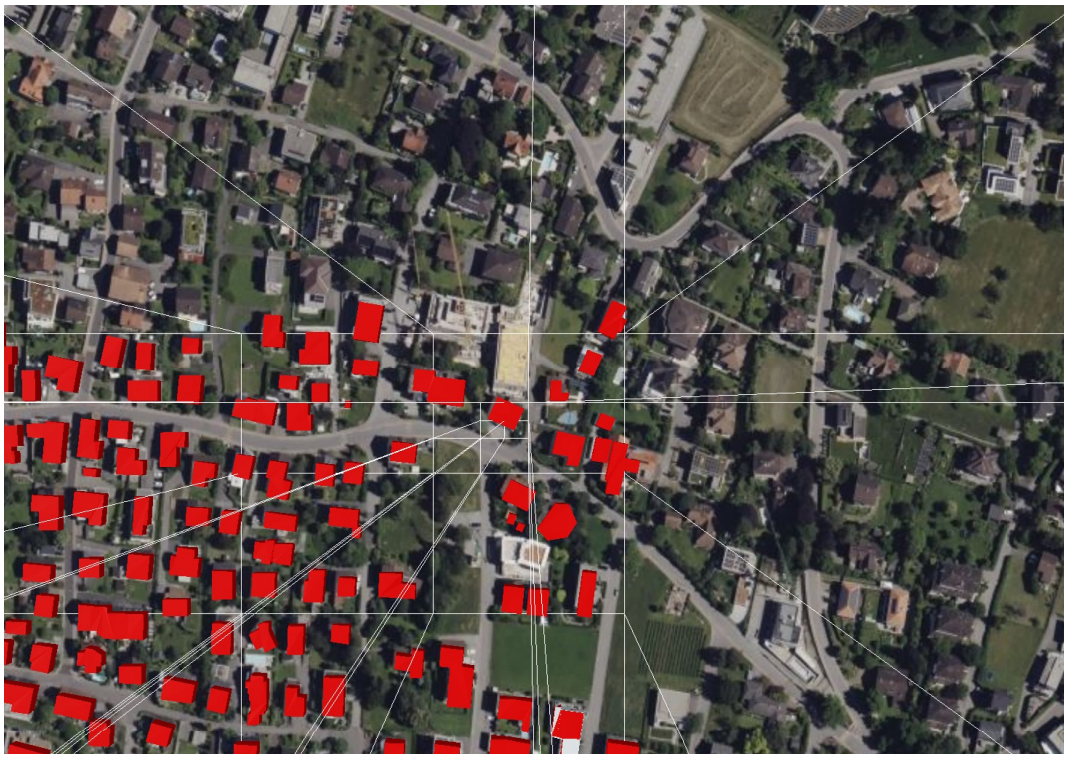

Figure 4.1 – *Exemple de quadtree utilisé par Cesium JS*

1. baremaps-server/src/main/resources/tdtiles/tileset.json

La notion de `levels` est directement liée au `quadtree` lui même. À chaque division, le `level` augmente de 1. Ainsi, une tuile de `level` 0 est la tuile de base, une tuile de `level` 1 est une tuile résultant de la division de la tuile de `level` 0, etc. Nous trouverons donc les tuiles de `level` le plus élevé à l'endroit où nous nous trouvons.

Sachant cela, nous pouvons maintenant comprendre comment les bâtiments sont affichés. Lorsque le client Cesium JS charge une tuile, il envoie une requête au serveur Baremaps pour obtenir les bâtiments de cette tuile. Le serveur Baremaps va alors vérifier que le `level` de la tuile est suffisamment élevé pour que les bâtiments soient affichés. Si tel est le cas, il va alors effectuer la requête SQL vu au chapitre 3.1 avec néanmoins une limite au nombre de bâtiments retournés (ce nombre dépendant du `level`). Les bâtiments retournés sont alors convertit en `glTF` et envoyés au client Cesium JS qui les affiche.

Cette approche est la source du problème causant les bâtiments à s'afficher de manière ressemblant à de l'aléatoire. Si nous imposons une limite de bâtiments par tuile, alors la requête SQL ne retournera que les premiers bâtiments dans sa liste, peu importe qu'ils soient proches de la caméra ou non.

J'ai donc modifié ce fonctionnement pour que lorsqu'une tuile soit demandée à être affichée, le serveur Baremaps retourne tous les bâtiments de cette tuile. Néanmoins, un changement est donc aussi nécessaire quand à quelle tuile doit être affichée ou non. Générer tous les bâtiments d'une tuile trop grande ne fera que bloquer le client Cesium JS le temps quel la tuile lui soit fournie. En augmentant le `level` à partir duquel les bâtiments sont affichés, nous pouvons garantir que tous les bâtiments affichés sont proches de la caméra tout en évitant de prendre trop de temps à les générer.

4.2 Profiling

Afin de voir exactement quelle partie du code bloque lorsque l'on déplace la caméra, j'ai utilisé l'outil de `profiling` lié à l'IDE IntelliJ. Cet outil permet de voir le temps passé dans chaque méthode du code. J'ai donc lancé le `profiler` et j'ai déplacé la caméra dans une zone de la carte où les bâtiments mettent du temps à s'afficher. Les résultats sont visibles sur la figure 4.2.

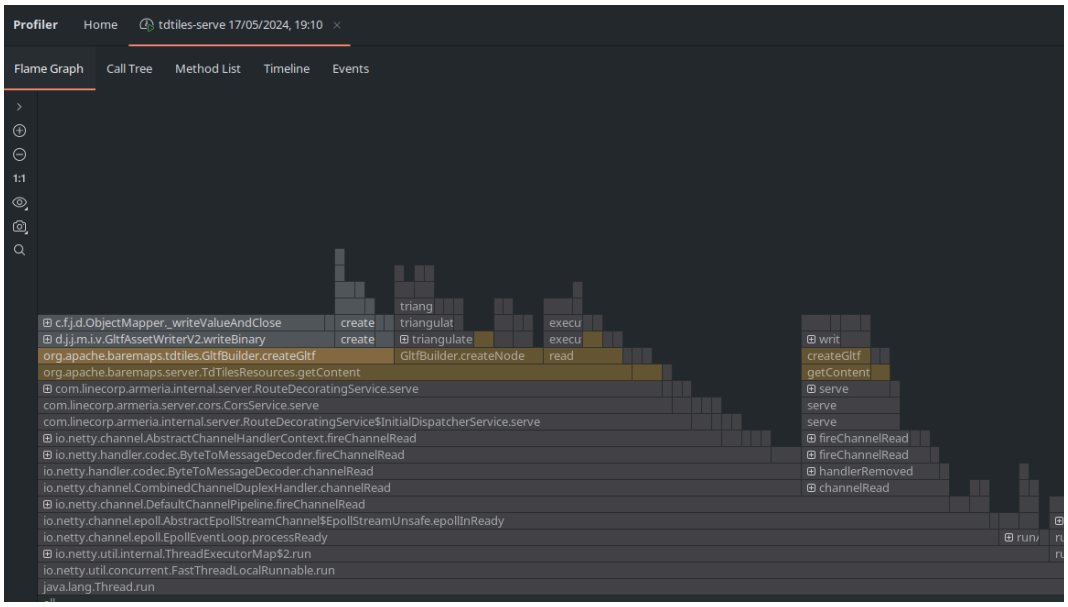

Figure 4.2 – *Résultats du profiling*

Grâce à ces résultats, nous pouvons observer que c'est l'écriture en fichiers binaire `glTF` des bâtiments qui prend le plus de temps. Malheureusement, cette fonction `writeBinary` [2] ne peut pas être plus optimisée que ce qu'elle est déjà. Je parle néanmoins dans la section suivante d'une manière de lui alléger la tâche.

On remarque aussi que la méthode `createNode` [3] prend aussi beaucoup de temps. Ici il est possible de *multithreader* cette méthode pour gagner du temps. J'ai donc écrit la méthode `createNodes` [4] qui permet de le faire en prenant en compte le nombre de CPUs disponibles.

2. de.javagl.jgltf.model.io.v2.GltfAssetWriterV2
3. baremaps-core/src/main/java/org/apache/baremaps/tdtiles/GltfBuilder.java
4. baremaps-server/src/main/java/org/apache/baremaps/server/TdTilesResources.java

4.3 Système de compression des géométries des bâtiments

Comme vu à la section précédente, l'écriture des bâtiments en fichiers binaires `glTF` est une des étapes les plus longues. Une manière de réduire le temps passé dans cette étape est de simplement réduire la quantité de donnée que la fonction prend en paramètre. Pour cela, j'utilise un système de *Level of details* (LOD) pour les géométries des bâtiments.

Ce système se base sur les `levels` des tuiles discutés au chapitre 4.1. Plus le `level` est élevé, plus les bâtiments sont proches de la caméra et donc plus les bâtiments doivent être détaillés. En fonction du `level` de leur tuile, les bâtiments subissent jusqu'à 3 niveau de compression :

- `level` 0 : Les bâtiments sont affichés en 3D avec toutes les géométries.
- `level` 1 : Les bâtiments sont affichés en 3D avec une géométrie simplifiée.
- `level` 2 : Les bâtiments sont affichés en 2D.
- `level` 3 : Les bâtiments sont affichés en 3D avec toutes les géométries uniquement si ils ont des caractéristiques spécifiques enregistrées dans la base de données OpenStreetMap.

Pour visualiser les différences entre les niveaux de compression, je vous invite à regarder la figure 4.3 où chaque niveau de détail est simbolisé par une couleur :

- `level` 0 : rouge
- `level` 1 : vert
- `level` 2 : bleu
- `level` 3 : blanc

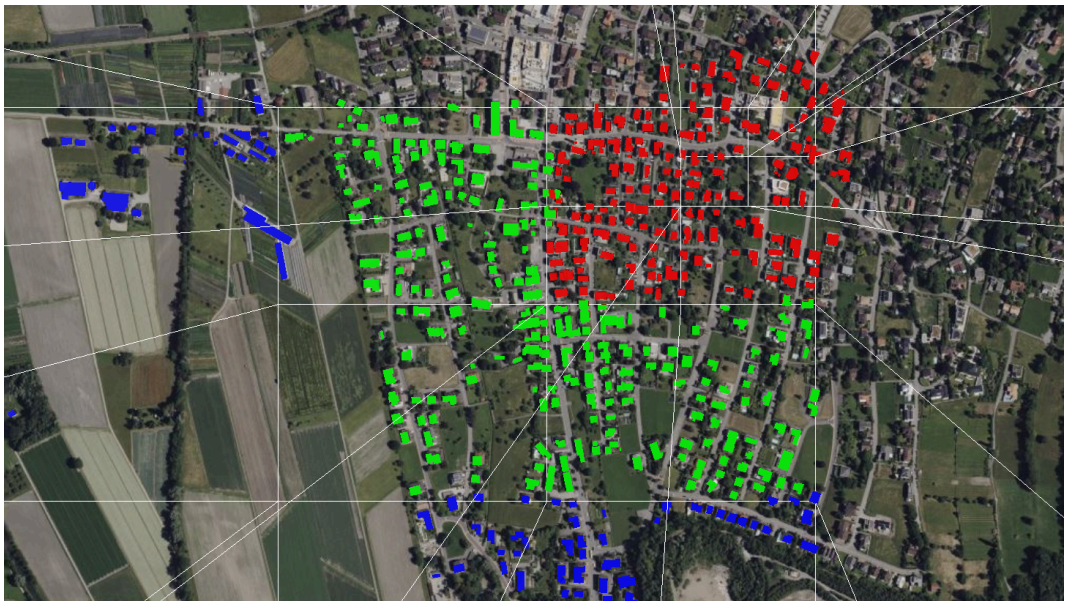

Figure 4.3 – *Niveaux de détails des bâtiments*

Faire ainsi permet de réduire drastiquement le temps passé à générer ces bâtiments tout en gardant une qualité d'affichage correcte.

Chapitre 5

Conclusion

Quelques obstacles sont survenus durant ce travail. Certains ont pu être traités dans ce rapport mais d'autres restent encore à résoudre. De plus, une majorité des *features* listées dans le cahier des charges sont encore à implémenter.

L'état de l'avancée du projet est néanmoins selon moi dans la bonne direction. Dans cette partie de mon TB, j'ai pu comprendre les fonctionnements de tous les outils et technologies utilisés. J'ai pu également commencer à les adapter pour qu'ils fonctionnent ensemble.

Ian Escher

Annexes

Glossaire

géométrie Une structure de donnée contenant plusieurs points et formant un polygone. 6

implicit tiling Une technique qui consiste à diviser la carte en tuiles de manière récursive.. 12

tags Objets contenus par une entité de la base de donnée OpenStreetMap servant à stocker des informations supplémentaires. Non obligatoires.. 6

Colophon :

La qualité de cet ouvrage repose que le moteur LaTeX. La mise en page et le format sont inspirés d'ouvrages scientifiques tels que le modèle de thèse de l'EPFL et celui des publications O'Reilly.

Les diagrammes et les illustrations sont édités depuis l'outil en ligne draw.io. Certaines illustrations ont été reprises dans Adobe Illustrator. Les représentations 3D sont exportées de SolidWorks et certains graphiques sont générés à la volée depuis un code source Python.

L'auteur fictive de ce document *Maria Bernasconi* est un nom emprunté, par amusement, aux spécimens publiés par Postfinance.

Ce document a été compilé avec XeLaTeX.

La famille de police de caractères utilisée est *Computed Modern* créée par Donald Knuth avec son logiciel METAFONT.

Le Colophon est le dernier élément d'un document qui contient des notes de l'auteur concernant la mise en page et l'édition du document : il est parfaitement optionnel.