

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)

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

A handwritten signature in black ink, appearing to read "I. 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

Préambule	i
Authentification	iii
Résumé	v
1 Introduction	1
2 3D Tiles	3
2.1 Spécification 3D Tiles Next	3
2.2 Implicit Tiling	4
2.2.1 Nouveautés de la version 1.1 de 3D Tiles	4
2.2.2 Avantages et inconvénients de l'implicit tiling	5
2.3 Level of Detail	6
2.4 3DTILES_bounding_volume_S2 extension	7
2.5 Database	10
2.5.1 Données OSM	11
2.5.2 Données générées	12
2.6 Création des fichiers glTF	12
2.6.1 Système de compression des géométries des bâtiments	12
2.6.2 Géométries des bâtiments	13
2.6.3 Hauteur des bâtiments	14
2.6.4 glTF par bâtiment ou par tuile	14
3 Subtrees	17
3.1 Introduction	17
3.2 Subtree Class	19
3.3 Morton Index	19
3.4 Availabilities	19
3.5 Transmission	19
4 Conclusion	21
Appendices	27

Table des figures

2.1	Exemple de Tileset contenant un arbre de Tiles [Cesa]	3
2.2	Division en quadtree par implicit tiling [Cesc]	4
2.3	Calcul du SSE [Cesc]	7
2.4	Tokens des 6 <i>root cells</i> [S2G]	10
2.5	Calcul du SSE [Cesc]	13
3.1	Exemple de hiérarchie de Subtrees	18

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

3D Tiles

2.1 Spécification 3D Tiles Next

La spécification **3D Tiles Next**¹ de **Cesium**² est une spécification open source permettant de visualiser des données géospatiales en 3 dimensions.

Pour pouvoir être utilisé, un dataset de données géospatiales doit être partitionné car il est souvent trop complexe pour nos ordinateurs de le traiter en entier. Cette spécification décrit comment le faire en organisant ces données en *Tiles* et en *Tilesets* pour pouvoir ensuite être fournies à un client tel que Cesium. Les Tiles, ou tuiles, contiennent toutes les informations nécessaires à décrire une portion d'une database dans un espace 3D comme par exemple un *bounding volume* qui décrit le volume 3D de la tuile ou encore son *content* qui contient l'objet 3D à afficher. Une Tile contient aussi une liste de Tile appelée *children*. Un Tilesets possède diverses informations comme la version de la spécification utilisée mais aussi une liste de Tile tout comme les Tiles. Cette structure permet de diviser les données en une hiérarchie de Tiles en arbre, ce qui facilite la recherche et le traitement des données.

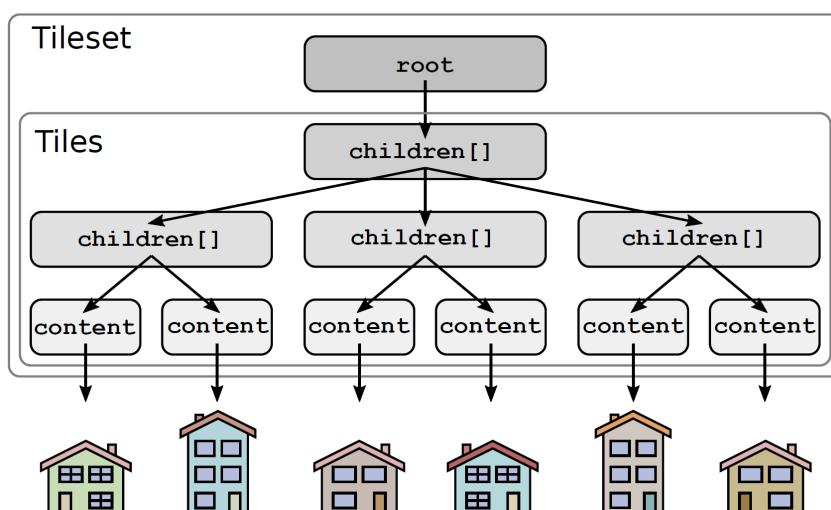


Figure 2.1 – Exemple de Tileset contenant un arbre de Tiles [Cesa]

-
1. <https://cesium.com/blog/2021/11/10/introducing-3d-tiles-next/>
 2. <https://cesium.com/>

D'autres éléments sont présents dans les Tiles et Tilesets. Certains seront abordés plus tard dans ce rapport, sinon, pour plus d'informations sur la spécification 3D Tiles, vous pouvez consulter les [documents de références proposés par Cesium](#)³.

2.2 Implicit Tiling

2.2.1 Nouveautés de la version 1.1 de 3D Tiles

Avec la nouvelle version 1.1 de la spécification 3D Tiles, il a été introduit une nouvelle fonctionnalité appelée *Implicit Tiling*. Elle permet de diviser implicitement un dataset en tuiles de mêmes tailles sans avoir à les définir explicitement.

Pour cela, l'implicit tiling divise 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 tuiles de même taille.

Deux méthodes de division sont actuellement disponibles : QUADTREE et OCTREE. La première reste sur un concept de tuiles en deux dimensions, tandis que les octrees permettent de diviser les tuiles en rajoutant une notion de hauteur, donc en 3 dimensions. Dans mon cas, j'utilise une division en quadtree puisque tout les bâtiments que je dois traiter se trouvent sur un plan 2D.

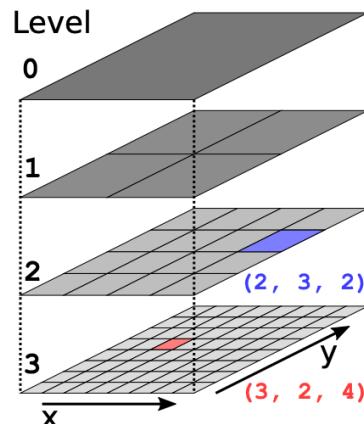


Figure 2.2 – Division en quadtree par implicit tiling [Cesc]

Le niveau 0 englobe l'entièreté du Tileset, dans notre cas la planète entière, puis, à chaque division, chaque tuile correspond non pas à une profondeur supplémentaire, mais à une portion de plus en plus petite du Tileset. 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. Plus d'informations quand à l'indexation des tuiles sont disponibles dans la section 3.3.

Pour définir un implicit tiling, il faut en premier lieu créer un Tileset hôte qui définira entre autre son `bounding volume`. L'implicit tiling va ensuite définir les tuiles de ce Tileset hôte en fonction de son `subdivisionScheme`, de son `subtreeLevels` et de ses `availableLevels`, tout en prenant comme taille initiale le `bounding volume` du Tileset. Enfin, il est possible de définir les adresses auxquelles le client doit envoyer des requêtes pour obtenir les tuiles ainsi que les *Subtrees*. Ces derniers seront discutés

3. <https://github.com/CesiumGS/3d-tiles/tree/main/reference-cards>

2.2. IMPLICIT TILING

dans leur chapitre dédié section 3.1. Selon la spécification, le client Cesium s'attend à recevoir des fichiers glb (glTF binary) pour les tuiles. Ci-dessous un exemple de fichier JSON définissant un Tileset avec un implicit tiling :

```
{  
    "asset" : {  
        "version" : "1.1"  
    },  
    "geometricError": 1200000,  
    "root" : {  
        "boundingVolume": {  
            "region": [-3.14, -1.57, 3.14, 1.57, 0, 60]  
        },  
        "refine": "REPLACE",  
        "geometricError": 1200000,  
        "content": {  
            "uri" : "/content/content_glb_{level}__{x}_{y}.glb"  
        },  
        "implicitTiling" : {  
            "subdivisionScheme" : "QUADTREE",  
            "subtreeLevels" : 4,  
            "availableLevels" : 19,  
            "subtrees" : {  
                "uri" : "/subtrees/{level}.{x}.{y}.subtree"  
            }  
        }  
    }  
}
```

En plus de l'implicit tiling, la version 1.1 de 3D Tiles apporte le support des fichiers glTF ainsi que deux nouvelles extensions : [EXT_mesh_features](#)⁴ et [EXT_structural_metadata](#)⁵. Cela introduit beaucoup de nouvelles possibilités par rapport à l'ancien système, notamment à des *metadata* très précises par objet glTF.

Bien que la génération de fichiers glTF sera discuté dans la section 2.6, je ne parlerai pas des extensions dans ce rapport puisqu'elles n'ont pas été utilisées dans mon projet.

2.2.2 Avantages et inconvénients de l'implicit tiling

Comme son nom l'indique, cette technique de division de tuiles est implicite, ce qui signifie que l'on ne peut pas définir nous même les caractéristiques des tuiles. Cela peut être un avantage pour des datasets très grands, comme une planète entière, où il serait difficile de définir explicitement les tuiles, mais cela peut poser problème lorsque le **bounding volume** d'une tuile contenant une petite maison de campagne sera le même que celui de la tuile qui contiendra l'Empire State Building. Un autre point où il est bénéfique de pouvoir définir explicitement par tuile sont les niveaux de détails. En reprenant le même exemple, il serait intéressant de pouvoir définir que la tuile contenant l'Empire State Building doit contenir plusieurs niveaux de détails tandis que la tuile contenant la maison de campagne n'en a besoin que d'un seul.

4. https://github.com/CesiumGS/glTF/tree/3d-tiles-next/extensions/2.0/Vendor/EXT_mesh_features

5. https://github.com/CesiumGS/glTF/tree/3d-tiles-next/extensions/2.0/Vendor/EXT_structural_metadata

En utilisant l'implicit tiling, une tuile ne contiendra forcément qu'un seul niveau de détail définit implicitement. Une tuile définit explicitement peut quant à elle contenir plusieurs niveaux de détails sous la forme d'une chaîne de tuiles (liées entre elles par leur champ `children`). Plus d'informations sur les niveaux de détails et les `geometric error` sont disponibles dans la section 2.3.

Du bon côté, l'implicit tiling permet d'effectuer tous ces calculs de `bounding volume` et de `geometric error` de manière extrêmement rapide en ne se basant que sur la tuile parente plutôt que de devoir analyser la database et calculer ces valeurs en fonction des objets qu'elle contient.

Un entre-deux serait d'utiliser l'implicit tiling pour les tuiles de plus haut niveau, puis de définir explicitement les tuiles de plus bas niveau ou alors des tuiles spécifiques qui nous intéresseraient principalement. Cela permettrait de profiter des avantages des deux méthodes. Actuellement, il est possible de fournir des nouveaux Tilesets à la place de fichier glb à un client qui effectuerait des requêtes sur la base d'un implicit tiling. Cette fonctionnalité n'est néanmoins absolument pas décrite dans la spécification 3D Tiles. Il ne faut donc actuellement pas la considérer comme une fonctionnalité stable ou supportée par Cesium.

J'ai cependant fait quelques tests en l'utilisant, initialement par accident. Je n'ai malheureusement pas réussi à obtenir un résultat satisfaisant. Les `bounding volumes` des tuiles générés par mes soins n'étaient pas du tout pris en compte par le client Cesium, ce qui posait de gros problèmes lors du calcul du niveau de détail à afficher. De plus, lorsque la caméra avait certains angles de vue, les tuiles étaient déchargées et enlevées de l'affichage. Je suspecte que cela soit partiellement dû aux mauvais `bounding volumes` attribués aux tuiles. Une différence entre une vue vers le Nord et une vue vers le Sud affectait aussi la visibilité des tuiles, ce qui me laisse penser que d'autres paramètres doivent aussi entrer en jeu.

2.3 Level of Detail

Les *Level Of Details*, appellés aussi LOD, est une technique qui permet de définir plusieurs niveaux de détails pour un même objet 3D. Cela permet de charger des objets plus ou moins détaillés en fonction de la distance à laquelle ils se trouvent de la caméra. Grâce à cela, il est possible de réduire la charge de travail de l'ordinateur en ne chargeant que le niveau de détail nécessaire pour garder un rendu qui, à l'œil humain, semble identique à celui d'un contenant tous les objets entièrement détaillés.

Pour calculer le bon niveau de détail à afficher, Cesium utilise le `bounding volume` ainsi que le `geometric error` d'une tuile. Le concept est simple, plus un objet occupe une partie importante de la fenêtre de rendu, plus il doit être détaillé. Le *Screen Space Error* ou SSE est une valeur qui permet de déterminer l'imprécision du rendu d'un objet en Pixels. Grâce à elle, il nous est possible de déterminer un seuil SSE auquel Cesium devra changer de LOD. Pour déterminer ce seuil, il nous est possible de définir un `geometric error` pour chaque tuile. Ce `geometric error` est une valeur qui permet de déterminer la distance maximale entre la géométrie réelle et la géométrie simplifiée en mètres. Plus cette valeur est grande, plus la géométrie simplifiée sera éloignée de la géométrie réelle. Avec le `geometric error`, le `bounding volume` et la distance entre la caméra et l'objet, il est possible de déterminer le SSE d'une tuile selon la formule suivante :

2.4. 3DTILES_BOUNDING_VOLUME_S2 EXTENSION

$$SSE = \frac{geometricError}{distance} \times \frac{screenHeight}{\tan(fovy/2)} \quad (2.1)$$

Où `screenHeight` est la hauteur de la fenêtre de rendu en pixels et `fovy` est l'angle de vue de la caméra en radians sur l'axe des y.

Pour plus d'informations, vous pouvez consulter la spécification 3D Tiles [Cesc]. Des graphiques tels que celui de la figure 2.3 sont disponibles pour mieux comprendre le concept.

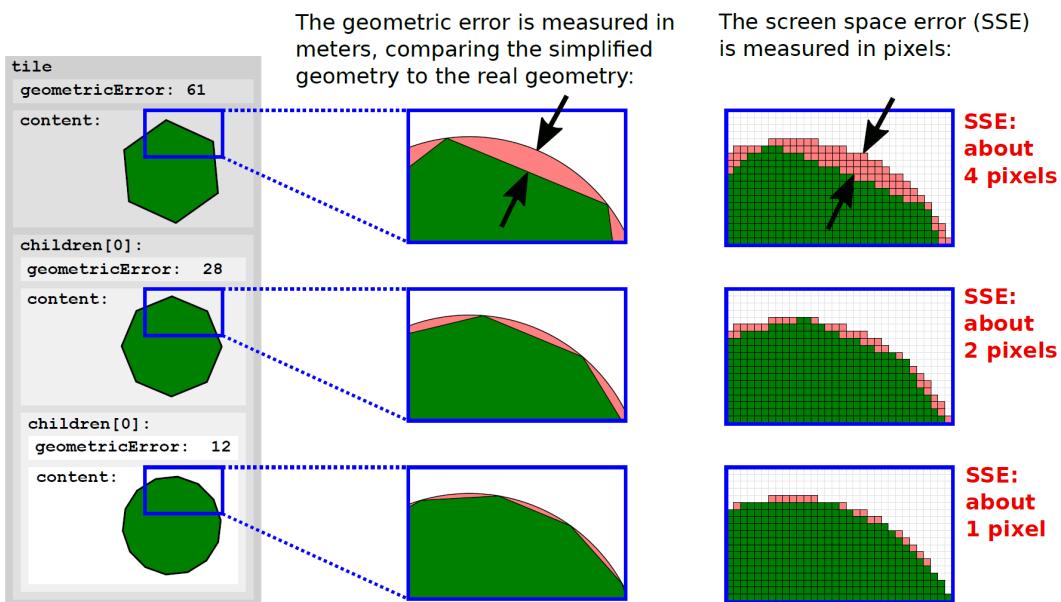


Figure 2.3 – Calcul du SSE [Cesc]

2.4 3DTILES_bounding_volume_S2 extension

Lors de la section sur l'implicit tiling section 2.2, j'ai mentionné que l'implicit tiling créait les `bounding volumes` et les `geometric errors` de manière automatique. Ce n'est pas vraiment le cas. Par défaut, ils ne sont pas définis par l'implicit tiling, mais il est possible de rajouter l'extension `3DTILES_bounding_volume_S2`⁶ qui permet de compléter les tuiles générées par l'implicit tiling avec des `bounding volumes` et des `geometric errors` calculés automatiquement.

Sans trop rentrer dans les détails, l'extension utilise une `courbe de Hilbert`⁷ pour partitionner et indexer le plan en plusieurs niveaux de tuiles. Ce sujet rentre la thématique des `Binary Space Partitioning`⁸. Ce sujet sera abordé plus en profondeur lorsque je parlerai de la génération des subtree et de leurs indexes dans la section section 3.3.

Pour utiliser cette extension, il suffit de rajouter les champs `extensionsUsed` et `extensionsRequired` dans le fichier JSON du Tileset. Puis, il faut rajouter une

6. https://github.com/CesiumGS/3d-tiles/tree/main/extensions/3DTILES_bounding_volume_S2

7. https://en.wikipedia.org/wiki/Hilbert_curve

8. https://en.wikipedia.org/wiki/Binary_space_partitioning

CHAPITRE 2. 3D TILES

tuile dans le champ `children` du fichier JSON du Tileset. Cette tuile doit contenir un champ `boundingVolume` avec un champ `extensions` contenant les informations nécessaires pour l'extension.

Remarquez le champ `token` dans l'exemple ci-dessous. Il s'agit d'un identifiant unique correspondant à la *root cell* (la cellule de plus haut niveau) de la courbe de Hilbert. Selon l'extension, le globe terrestre a été divisé en 6 *root cells* comme si l'on avait projeté les 6 faces d'un cube sur la terre. Par exemple, l'Antarctique se trouve dans la cellule "5".

2.4. 3DTILES_BOUNDING_VOLUME_S2 EXTENSION

Voici un exemple de fichier JSON utilisant l'extension 3DTILES_bounding_volume_S2 :

```
{
  "asset" : {
    "version" : "1.1"
  },
  "geometricError": 1200000,
  "extensionsUsed": [
    "3DTILES_bounding_volume_S2"
  ],
  "extensionsRequired": [
    "3DTILES_bounding_volume_S2"
  ],
  "root" : {
    "boundingVolume": {
      "region": [-3.14159, -1.57079, 3.14159, 1.57079,
                 0, 60
      ]
    },
    "refine": "REPLACE",
    "geometricError": 1200000,
    "children": [
      {
        "boundingVolume": {
          "extensions": {
            "3DTILES_bounding_volume_S2": {
              "token": "1",
              "minimumHeight": 0,
              "maximumHeight": 20
            }
          }
        },
        "refine": "REPLACE",
        "geometricError": 120000
      },
      {
        "content": {
          "uri" : "/content/content_glb_{level}__{x}_{y}.glb"
        },
        "implicitTiling" : {
          "subdivisionScheme" : "QUADTREE",
          "subtreeLevels" : 4,
          "availableLevels" : 19,
          "subtrees" : {
            "uri" : "/subtrees/{level}.{x}.{y}.subtree"
          }
        }
      }
    ]
  }
}
```

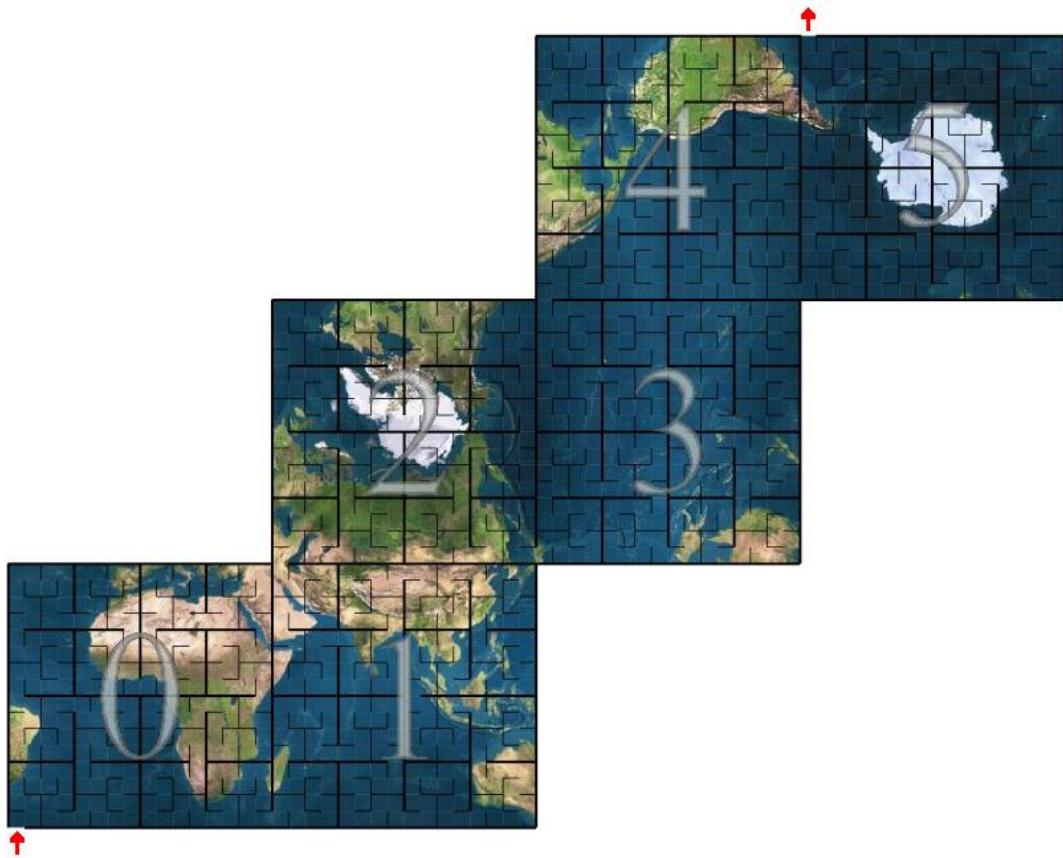


Figure 2.4 – Tokens des 6 root cells [S2G]

Notez aussi que dans l'exemple JSON ci-dessus, qu'une seule *root cell* est définie. Dans mon cas, comme j'aimerai pouvoir utiliser l'entièreté de la terre, j'ai défini les 6 *root cells* en tant que 6 **children** différents avec chacun leur propre **token**.

Cette manière de partager l'espace de la terre ne correspond cependant pas à la méthode qu'utilise l'implicit tiling. Les calculs nécessaires pour convertir les indexes de l'implicit tiling vers ceux des courbes de Hilbert ne sont néanmoins pas à effectuer par nos soins, l'extension ce charge de cela.

2.5 Database

La base de donnée que j'utilise se distingue par deux catégories de tables différentes. Premièrement, il y a les tables qui ont été générées par l'import de données OSM. Ces tables contiennent les informations des éléments OSM tels que les bâtiments, les routes, les rivières, etc. Deuxièmement, il y a les tables qui sont générées par mes propres scripts qui stockeront les tuiles 3D ainsi que les Subtrees pour éviter de les re-générer à chaque fois car cela peut prendre plusieurs heures si le dataset est grand.

Pour accéder aux données de ces tables, j'utilise la librairie `javax.sql.DataSource` qui me permet de me connecter à la base de données.

2.5. DATABASE

2.5.1 Données OSM

Dans les données OSM, les parties qui nous intéressent sont les tables `osm_nodes` et `osm_relations`. Ces tables contiennent les données de tous 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 OpenStreetMap⁹ nous permet d'avoir la liste complète des tags existants et nous intéressants 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
FROM osm_ways
WHERE (tags ? building or tags ? building:part) and
      st_intersects(geom, st_makewindow(%1$s, %2$s, %3$s, %4$s, 4326))
UNION
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
FROM osm_relations
WHERE (tags ? building or tags ? building:part) and
      st_intersects(geom, st_makewindow(%1$s, %2$s, %3$s, %4$s, 4326));
```

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

Grâce à cette requête, je récupère toutes les informations nécessaires pour définir et générer les bâtiments en fichiers glTF. %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.

2.5.2 Données générées

Les données générées sont stockées dans les tables `td_subtrees` et `td_tile_gltf`. La première contient toutes les informations nécessaires à la définition d'un Subtree. La seconde contient les fichiers glTF des tuiles. Ces tables sont définies comme suit :

```
CREATE TABLE td_subtrees
(
    morton_index bigint,
    level         integer,
    binary_file   bytea,
    UNIQUE (morton_index, level)
);

CREATE TABLE td_tile_gltf
(
    x            bigint,
    y            bigint,
    level        integer,
    gltf_binary bytea,
    UNIQUE (x, y, level)
);
```

2.6 Création des fichiers glTF

Grâce à l'introduction du support des fichiers glTF ou plus précisément des fichiers glb dans la spécification 3D Tiles 1.1, il est devenu beaucoup plus facile de générer des fichiers 3D comparé à l'ancien système. Ici, j'utilise la librairie `jglTF`¹⁰ et la librairie `JTS`¹¹ pour créer les nodes, les meshes, les materials et les textures des bâtiments. Toute la logique concernant la création des fichiers glTF se trouve dans la classe `GltfBuilder`¹².

2.6.1 Système de compression des géométries des bâtiments

Comme vu à la section 2.3, il est nécessaire de pouvoir fournir plusieurs niveaux de détails en fonction du niveau de la tuile. Pour cela, j'ai implémenté un système de compression des géométries des bâtiments. Ce système se base sur les `levels` des tuiles discutés à la section 2.2. Plus le `level` est élevé, plus les bâtiments sont proches de la caméra et donc plus les bâtiments doivent être détaillés. Pour effectuer la compression, j'utilise la class `DouglasPeuckerSimplifier` qui permet de simplifier une géométrie en fusionnant les points les plus proches d'une géométrie.

En fonction du `level` de leur tuile, les bâtiments subissent jusqu'à 3 niveau de compression :

10. <https://github.com/javagl/JglTF>

11. <https://locationtech.github.io/jts/>

12. [baremaps-core/src/main/java/org/apache/baremaps/tdtiles/GltfBuilder.java](https://gitlab.com/baremaps/baremaps/-/blob/main/src/main/java/org/apache/baremaps/tdtiles/GltfBuilder.java)

2.6. CRÉATION DES FICHIERS GLTF

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

Pour visualiser les différences entre les niveaux de compression, je vous invite à regarder la figure 2.5 où chaque niveau de détail est symbolisé par une couleur :

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



Figure 2.5 – Calcul du SSE [Cesc]

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

2.6.2 Géométries des bâtiments

Avec le niveau de compression ainsi que la `Geometry`¹³, qui nous sert à déterminer le polygone au sol du bâtiment, 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`¹⁴ était utilisée. Le problème était que la `Delaunay triangulation`¹⁵ ne fonctionne pas avec des polygones concaves. Une version complémentaire de cette méthode est la `Constrained Delaunay triangulation`¹⁶ 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 mais elle propose aussi la classe `PolygonTriangulator`¹⁷ qui permet aussi de faire une triangulation de polygones concaves de manière moins optimisée mais plus rapide

13. org.locationtech.jts.geom.Geometry

14. org.locationtech.jts.triangulate.DelaunayTriangulationBuilder

15. https://en.wikipedia.org/wiki/Delaunay_triangulation

16. https://en.wikipedia.org/wiki/Constrained_Delaunay_triangulation

17. <https://locationtech.github.io/jts/javadoc/org/locationtech/jts/triangulate/polygon/PolygonTriangulator.html>

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.

2.6.3 Hauteur des bâtiments

Le calcul de la hauteur des bâtiments doit se faire en fonction de la [définition¹⁸](#) des bâtiments par OSM. Si aucun tag 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` pour donner la hauteur au bâtiment lors de sa modélisation.

2.6.4 glTF par bâtiment ou par tuile

Quand on pense aux manières de générer les fichiers 3D, il est facile de concevoir que deux méthodes sont possibles : générer un fichier glTF par bâtiment ou générer un fichier glTF par tuile. La première méthode nous permettrait de générer des features par bâtiment glTF, ce qui rendrait l'utilisation de metadata très intéressante.

Plus intéressant encore, générer ces fichiers 3D par bâtiment nous pousserai fortement à tenir une liste de bâtiments se trouvant dans chaque tuile. En faisant ainsi, il serait possible de prévenir le placement d'un même bâtiment dans plusieurs tuiles.

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

2.6. CRÉATION DES FICHIERS GLTF

Souvent, un bâtiment se trouve à cheval sur plusieurs tuiles et il est donc nécessaire de le placer dans plusieurs fichiers glTF quand on génère un fichier par tuile.

Pour pouvoir générer un fichier glTF par bâtiment, il faut utiliser le concept de *Multiple Content* introduit dans la version 1.1 de la spécification 3D Tiles [Cesb]. Pour utiliser cela, il faut remplacer le champ `content` de chaque tuile par un champ `contents` qui sera une liste de plusieurs `content`. Chaque `content` contiendra un `uri` qui pointera vers le fichier glTF du bâtiment.

Cependant, un problème survient si l'on utilise en même temps l'implicit tiling lors de la génération des *Subtrees*. Ces Subtrees, définis plus en profondeur dans leur chapitre dédié, nécessitent une liste de disponibilité (section 3.4) **par content**. En d'autres termes, si une tuile contient X bâtiments, il faudra X listes de disponibilité sur la tuile. Sur chacune de ces listes, une seule valeur sera à `true`, la valeur correspondant à l'index de la tuile sur laquelle le bâtiment se trouve. Cela rend donc la génération des Subtrees beaucoup plus complexe et énormément plus volumineuse. Pour plus de précisions sur le pourquoi de cette complexité, je vous invite à lire le chapitre dédié aux Subtrees.

Chapitre 3

Subtrees

3.1 Introduction

Lorsqu'on utilise l'implicit tiling, le client Cesium à besoin de recevoir en premier lieu une liste de tuiles disponibles. Cette liste doit lui être donnée sous forme de Subtree. Les *Subtrees* dans Cesium constituent une composante essentielle pour la gestion et l'optimisation de la visualisation des données géospatiales volumineuses. Plutôt que de charger l'intégralité des données géospatiales, les Subtrees ont pour but d'indiquer au client Cesium quelles tuiles sont disponibles, quelles tuilles ont un contenu à afficher quelles sont ses enfants (`children`). Pour chacune de ces trois informations, un Subtree contient un objet *Availability* (section 3.4) stockant une liste de boolean représentant chaque tuile. Pour savoir quel index de cette liste correspond à quelle tuile, un autre style de *Binary Space Partitioning* est utilisé. Des *Z-order curve* ou *Morton order*¹ sont utilisés pour définir les indexes (section 3.3).

Une fois ces Subtrees générés, il faudra les envoyer au client. Cela se fera en les transformant en JSON puis en un fichier respectant le *Subtree Binary Format*²

1. https://en.wikipedia.org/wiki/Z-order_curve

2. <https://github.com/CesiumGS/3d-tiles/tree/main/specification/ImplicitTiling#subtree-binary-format>

Hiérarchie de Subtrees

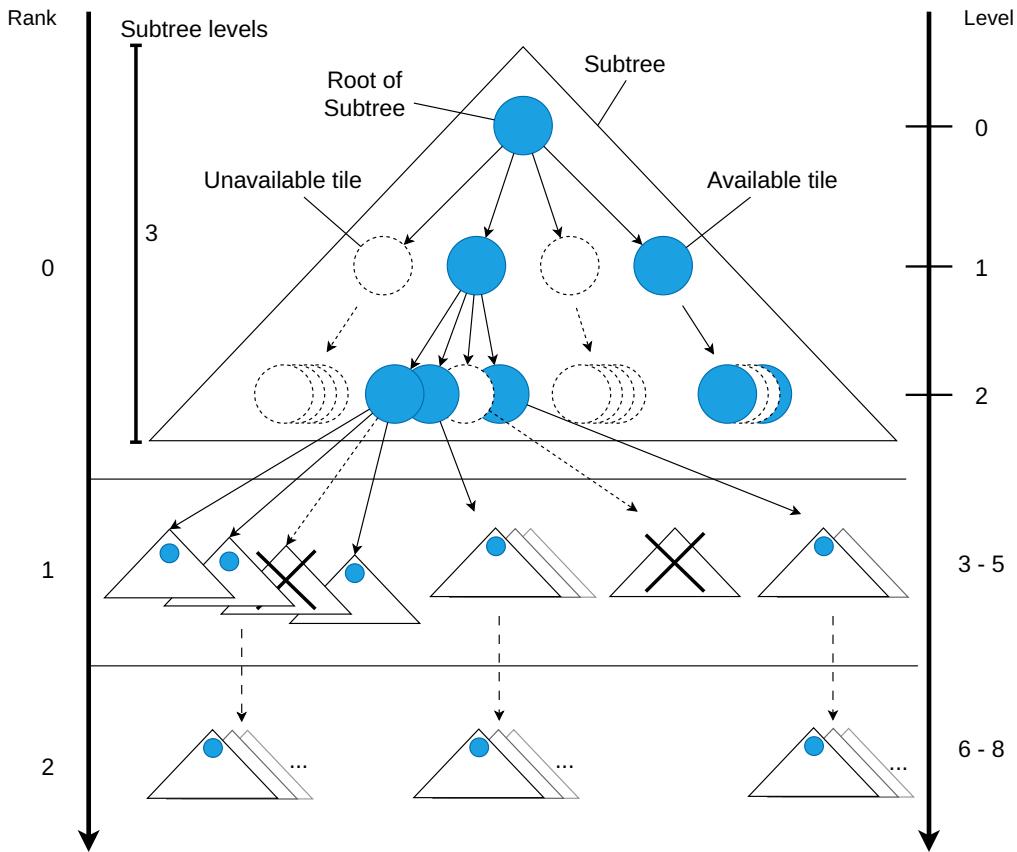


Figure 3.1 – Exemple de hiérarchie de Subtrees

Le tileset est partitionné en Subtrees de taille `subtreeLevels` fixe. Leur schéma de subdivision suit la même logique que les tuiles d'un implicit tiling. Chaque *level* ou niveau représente une subdivision de la tuile de niveau supérieur. Un Subtree est généralement de plusieurs niveau de profondeur. Chaque niveau est composé d'une liste de noeuds représentant la disponibilité d'une tuile. En fonction du shema de subdivision, quadtree ou octree, chaque noeud peut avoir 4 ou 8 enfants respectivement. On peut retrouver ces noeuds dans les listes citées plus haut à une exception près : les listes de disponibilités des tuiles et des disponibilités des contenus représentent à l'identique chacun de ces noeuds mais la liste de disponibilité des enfants est une liste représentant la disponibilité de chaque Subtree enfant du Subtree courant. Les *Availability*s seront traités plus en détails dans la section 3.4.

Plusieurs Subtrees peuvent être liés entre eux pour former un arbre de Subtrees. Leur liaison se fait entre le noeud *root* d'un subtree et un noeud au niveau maximum d'un autre Subtree.

Quelques règles doivent être respectées lors d'une construction d'une hiérarchie de Subtrees :

- Les Subtrees doivent être de même taille.
- Les Subtrees doivent avoir le même schéma de subdivision.
-

3.2. SUBTREE CLASS

3.2 Subtree Class

La première classe écrite dans le but de gérer les Subtrees est la classe **Subtree**. Un objet de cette classe comporte tout

3.3 Morton Index

3.4 Availabilities

3.5 Transmission

Chapitre 4

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

A handwritten signature in black ink, appearing to read "I. Escher".

Bibliographie

- [Cesa] CESIUMGS. *3D Tiles reference cards V1.0*. URL : <https://github.com/CesiumGS/3d-tiles/blob/main/3d-tiles-reference-card.pdf>. (accessed : 18.07.2024) (cf. p. 3).
- [Cesb] CESIUMGS. *3D Tiles reference cards V1.1*. URL : <https://github.com/CesiumGS/3d-tiles/blob/main/3d-tiles-reference-card-1.1.pdf>. (accessed : 21.07.2024) (cf. p. 15).
- [Cesc] CESIUMGS. *3D Tiles specification*. URL : <https://github.com/CesiumGS/3d-tiles/tree/main/specification#core-implicit-tiling>. (accessed : 18.07.2024) (cf. p. 4, 7, 13).
- [S2G] S2GEOMETRY. *Bounding Volume S2 extension website*. URL : http://s2geometry.io/devguide/s2cell_hierarchy#s2cellid-numbering. (accessed : 19.07.2024) (cf. p. 10).

Annexes

Glossaire

géométrie Une structure de donnée contenant plusieurs points et formant un polygone. 11

tags Objets contenus par une entité de la base de donnée OpenStreetMap servant à stocker des informations supplémentaires. Non obligatoires.. 11