

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 et Architecture	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	6
2.3 Level of Detail	7
2.4 3DTILES_bounding_volume_S2 extension	8
2.5 Création des fichiers glTF	11
2.5.1 Système de compression des géométries des bâtiments	11
2.5.2 Géométries des bâtiments	13
2.5.3 Hauteur des bâtiments	14
2.5.4 glTF par bâtiment ou par tuile	15
3 Architecture	17
3.1 Fonctionnement général	17
3.2 Database	18
3.2.1 Données OSM	18
3.2.2 Données générées	20
4 Subtrees	21
4.1 Introduction	21
4.2 Classes importantes	23
4.3 Morton Index	27
4.4 Création de l'arbre de Subtrees	31
4.4.1 Flowcharts des fonctions clé	31
4.4.2 Génération de Availability	35
4.4.3 Concaténations	36
4.5 Transmission au client Cesium	37
4.5.1 Écriture de JSON	38
5 Conclusion	41

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	Partition du monde en 6 <i>root cells</i> [S2G]	10
2.5	Vue à l'horizon	12
2.6	Level 0	12
2.7	Level 1	12
2.8	Level 2	12
2.9	Level 3	12
2.10	Hauteur minimum d'un bâtiment [Ope]	15
3.1	Calcul du SSE [Cesc]	17
4.1	Exemple de hiérarchie de Subtrees	22
4.2	Classes utilisées pour la création d'une hiérarchie de Subtrees	23
4.3	Représentation des niveaux de tuiles dans un BitSet	24
4.4	Classes utilisées pour la création d'une hiérarchie de Subtrees	25
4.5	Indexes X Y vers index de Morton [Cesd]	28
4.6	Index local et index global [Cesd]	30
4.7	Flowchart de la fonction <code>getSubtree</code>	31
4.8	Flowchart de la fonction <code>createMaxRankSubtree</code>	32
4.9	Flowchart de la fonction <code>createSubtree</code>	34
4.10	Génération de Availability : État initial	35
4.11	Génération de Availability : Procédure	35
4.12	Concaténation de 4 Availability	36
4.13	Header d'un fichier au <i>Subtree Binary Format</i> [Cese]	37

Chapitre 1

Introduction

Durant ce travail, j'ai écrit un programme sur la base du framework **Baremaps**¹ permettant à un client **Cesium**² d'obtenir les ressources nécessaires pour à afficher les bâtiments de le dataset d'**OpenStreetMap**³ en 3D.

Ces ressources utilisent la spécification **3D Tiles Next**⁴ de Cesium tout en respectant la version 1.1 de leur **standard de tuiles vectorielles**⁵. Cette nouvelle version de leur spécification introduit entre autre la fonctionnalité de *implicit tiling* [Cese] qui permet une utilisation facilitée de datasets volumineux. Les ressources qui seront fournies au client Cesium utiliseront donc cette fonctionnalité.

Le programme se divise en deux parties : la première est celle qui s'occupe de la génération des tuiles vectorielles contenant les bâtiments d'OpenStreetMap en 3D et la seconde est celle qui va construire les *Subtrees*, des objects indispensables au client Cesium pour optimiser l'affichage des tuiles. Ces deux parties ne sont pas équivalentes en termes de complexité. La première est plus simple et a été réalisée en premier. La seconde a nécessité d'avantages d'études ainsi que le développement de plusieurs algorithmes pour être implémentée.

-
1. <https://github.com/baremaps/baremaps>
 2. <https://cesium.com/>
 3. <https://www.openstreetmap.org/>
 4. <https://cesium.com/blog/2021/11/10/introducing-3d-tiles-next/>
 5. <https://docs.ogc.org/cs/22-025r4/22-025r4.html>

Fondements écrits par Antoine Drabble

Afin d'évaluer si le projet s'apprêtait 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 2

3D Tiles et Architecture

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 appellée *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.

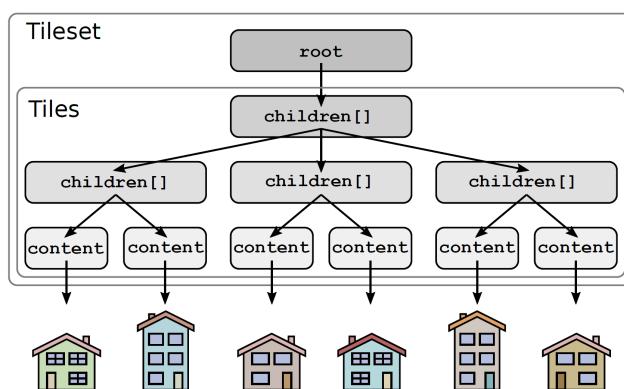


Figure 2.1 – Exemple de Tileset contenant un arbre de Tiles [Cesa]

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.

1. <https://cesium.com/blog/2021/11/10/introducing-3d-tiles-next/>
2. <https://cesium.com/>
3. <https://github.com/CesiumGS/3d-tiles/tree/main/reference-cards>

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.

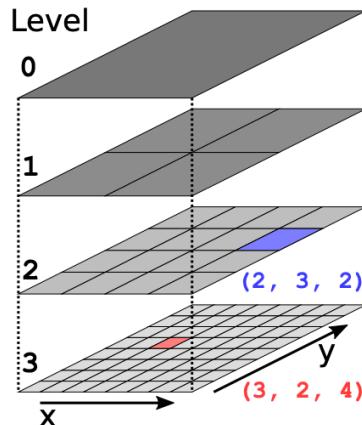


Figure 2.2 – Division en quadtree par implicit tiling [Cesc]

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 4.3.

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 dans leur chapitre dédié section 4.1. 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 :

2.2. 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 2.5, je ne parlerai pas des extensions dans ce rapport puisqu'elles n'ont pas été utilisées dans mon projet.

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

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. 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

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'oeil 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 :

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

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.

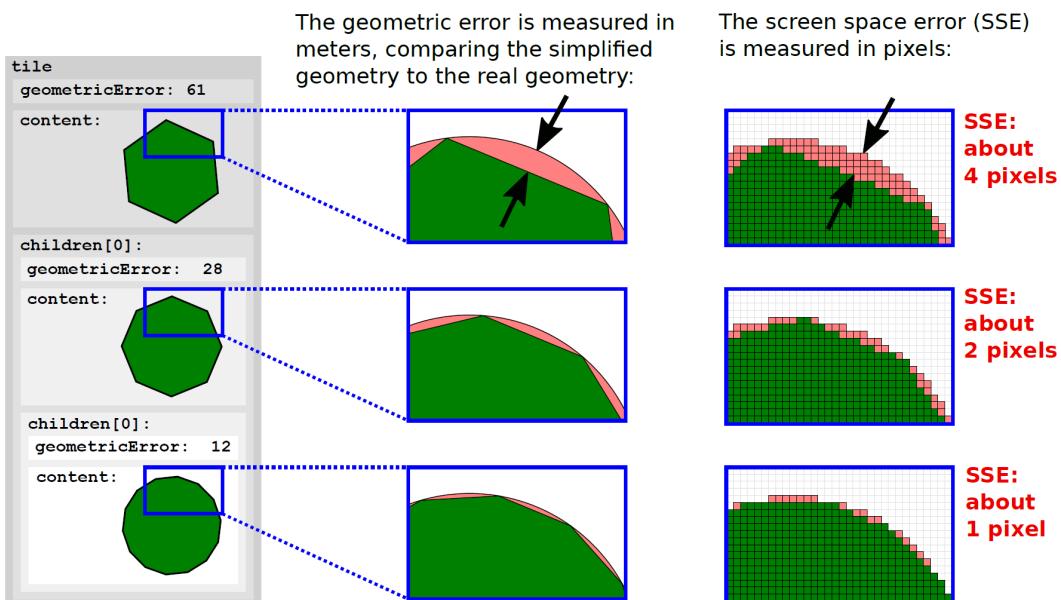


Figure 2.3 – Calcul du SSE [Cesc]

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 sur une partition en plusieurs niveaux de tuiles faite par la librairie **S2 Geometry**⁸. Ce sujet rentre dans la thématique des **Binary Space Partitioning**⁹. Tout comme la partition du monde en 2 dimensions et son indexation avec un index de Motron qui sera vu dans la section 4.3, le partitionnement du monde peut aussi se faire avec la librairie S2 Geometry et son indexation avec une courbe de Hilbert. Ces deux méthodes sont donc des *Binary Space Partitioning*. Ici, ces méthodes permettent de diviser une surface et permettent de donner rapidement et efficacement les informations spatiales des tuiles.

Pour utiliser cette extension, il suffit de rajouter les champs **extensionsUsed** et **extensionsRequired** dans le fichier JSON du Tileset. Puis, il faut rajouter une 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" (cf. Figure 2.4).

6. https://github.com/CesiumGS/3d-tiles/tree/main/extensions/3DTILES_bounding_volume_S2
 7. https://en.wikipedia.org/wiki/Hilbert_curve
 8. <http://s2geometry.io/>
 9. https://en.wikipedia.org/wiki/Binary_space_partitioning

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"  
                }  
            }  
        }  
    ]  
}
```

CHAPITRE 2. 3D TILES ET ARCHITECTURE

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 indexées 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.

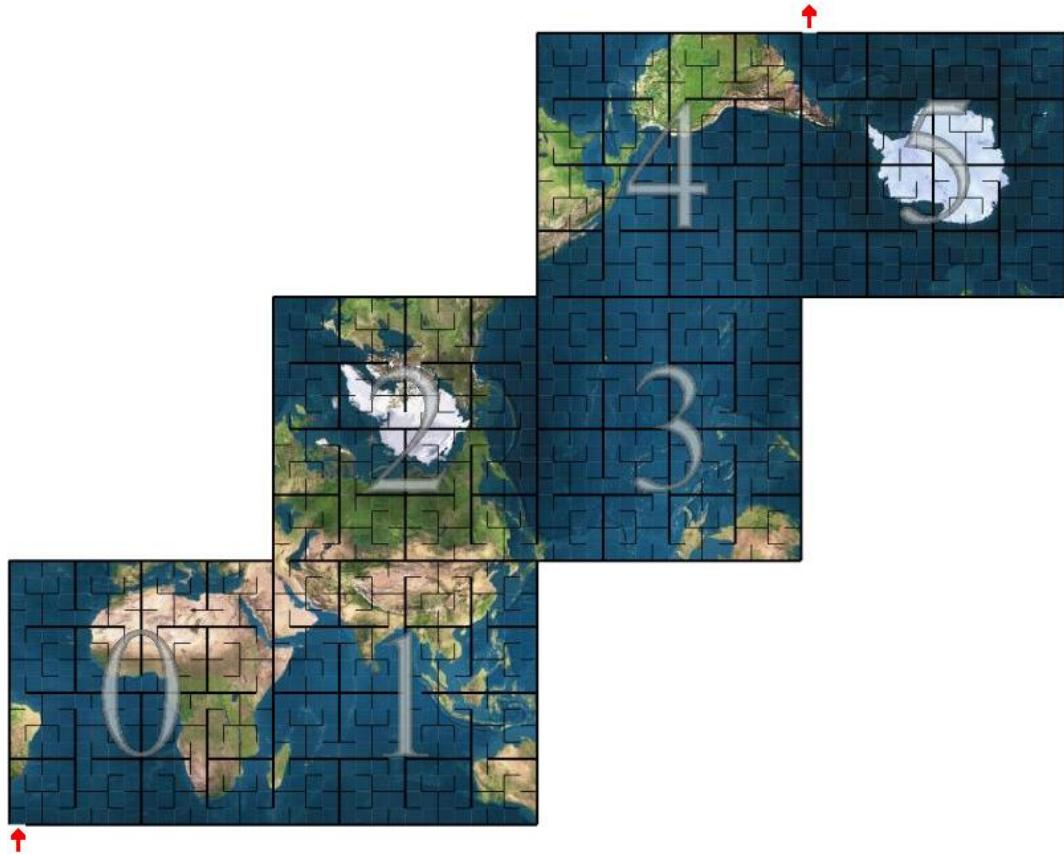


Figure 2.4 – Partition du monde en 6 root cells [S2G]

2.5. CRÉATION DES FICHIERS GLTF

2.5 *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.

2.5.1 *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 :

- `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 les figures ci-dessous où chaque niveau de détail est symbolisé par une couleur :

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

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

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

12. baremaps-core/src/main/java/org/apache/baremaps/tddiles/GltfBuilder.java

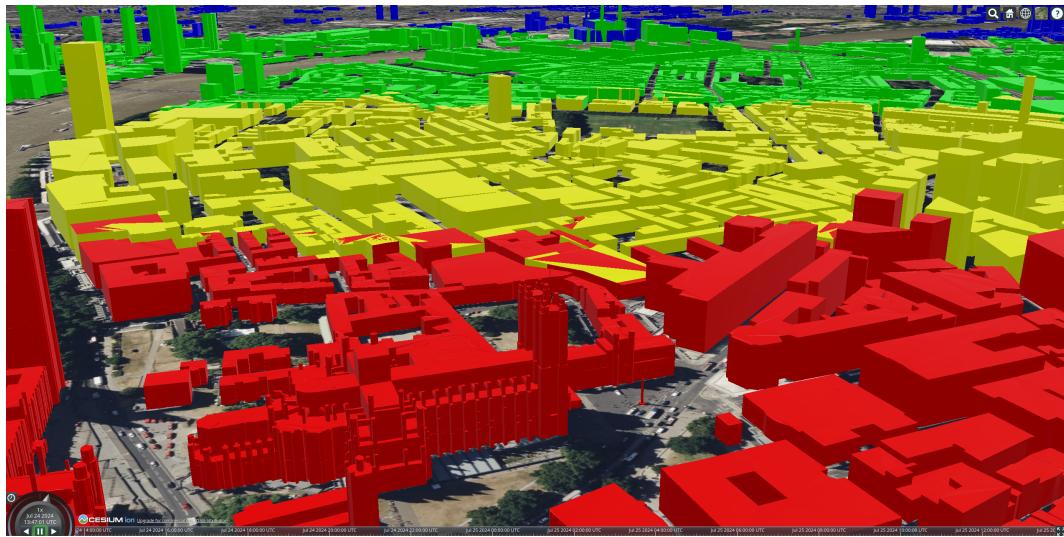


Figure 2.5 – Vue à l'horizon

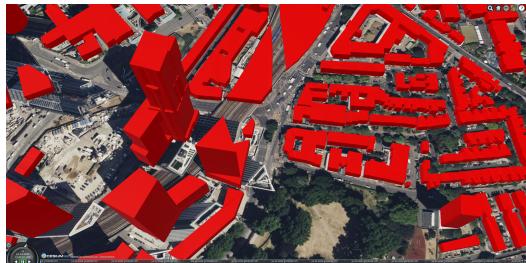


Figure 2.6 – Level 0



Figure 2.7 – Level 1



Figure 2.8 – Level 2



Figure 2.9 – Level 3

Faire ainsi permet de réduire drastiquement le temps passé à générer ces bâtiments ainsi que de garantir un affichage plus fluide chez le client tout en gardant une qualité d'affichage correcte.

Outre la compression des géométries, une autre autre qui pourrait être développée est un filtrage des tuiles en fonction de leur quantité de bâtiments et de leur niveau de compression. Par exemple, si une tuile ne contient qu'un ou deux petits bâtiments pourrait être omise à l'affichage si elle se trouve assez loin de la caméra pour être à un niveau de compression élevé.

2.5. CRÉATION DES FICHIERS GLTF

2.5.2 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, il faut une méthode qui fonctionne avec des polygones concaves et qui déforme le moins possible les géométrie des bâtiments. Une méthode faisant explicitement cela est la méthode `simplify` de la classe `TopologyPreservingSimplifier`¹⁴ de la librairie `org.locationtech.jts` qui fonctionne avec des polygones concaves et qui, comme son nom l'indique, préserve la topologie initiale de la géométrie. Cette librairie `org.locationtech.jts` 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 classe `TopologyPreservingSimplifier`. C'est donc cette classe que j'ai utilisée pour trianguler les bâtiments.

Grâce à cette 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 `TopologyPreservingSimplifier` comporte un paramètre `tolerance` qui permet de définir la distance minimale entre deux points afin d'optimiser le coût de la triangulation.

Si nous voulons pousser la génération des bâtiments plus loin, beaucoup de données sont disponibles concernant le toit des bâtiments. Ils peuvent être de différentes formes et comportent des défis très intéressants à relever. Un travail de Bachelor a été écrit par M. Philipp Shultz à propos de la visualisation de données OpenStreetMap en WebGL (disponible à l'adresse <https://ths.rwth-aachen.de/wp-content/uploads/sites/4/schulz20203d.pdf>). Au chapitre 3.2, il parle de la génération de toits de bâtiments en fonction de leur tag OSM. Il serait intéressant de reprendre ces idées pour améliorer la génération des bâtiments en 3D.

13. `org.locationtech.jts.geom.Geometry`

14. <https://locationtech.github.io/jts/javadoc/org/locationtech/jts/simplify/TopologyPreservingSimplifier.html>

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

2.5.3 Hauteur des bâtiments

Le calcul de la hauteur des bâtiments doit se faire en fonction de la [définition¹⁶](#) 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`, `roof:height` et `rooftop: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 `rooftop:height` est présent,
 - La hauteur du toit `rooftop: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 `rooftop:levels` est présent,
 - La hauteur du toit `rooftop: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.

Il se peut que des géométries ne soient pas en contact avec le sol et flottent dans les airs. Ces informations sont fournies par les tags `building:min_level`, `building:min_height` et `min_height`. Pour calculer la hauteur entre la géométrie et le sol, on peut la calculer de la même manière que 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.
- Si le tag `building:min_height` est présent,
 - La hauteur du vide en dessous du bâtiment est égale à la valeur du tag `building:min_height`.
- Si le tag `min_height` est présent,
 - La hauteur du vide en dessous du bâtiment est égale à la valeur du tag `min_height`.

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

2.5. CRÉATION DES FICHIERS GLTF

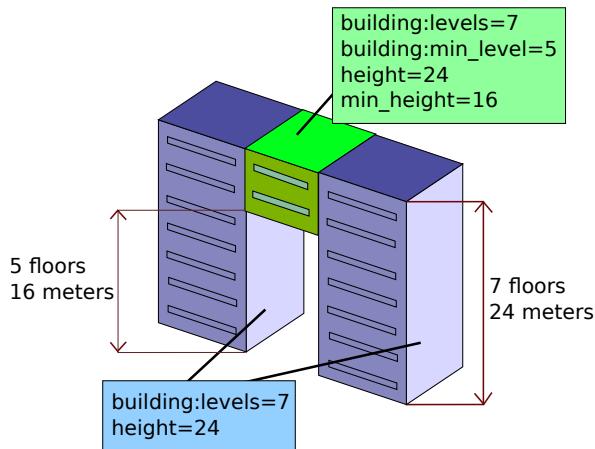


Figure 2.10 – Hauteur minimum d'un bâtiment [Ope]

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

2.5.4 glTF par bâtiment ou par tuile

Quand on pense aux manière 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. 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 4.2) **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

Architecture

3.1 Fonctionnement général

Voici le diagramme représentant l'architecture globale de l'application :

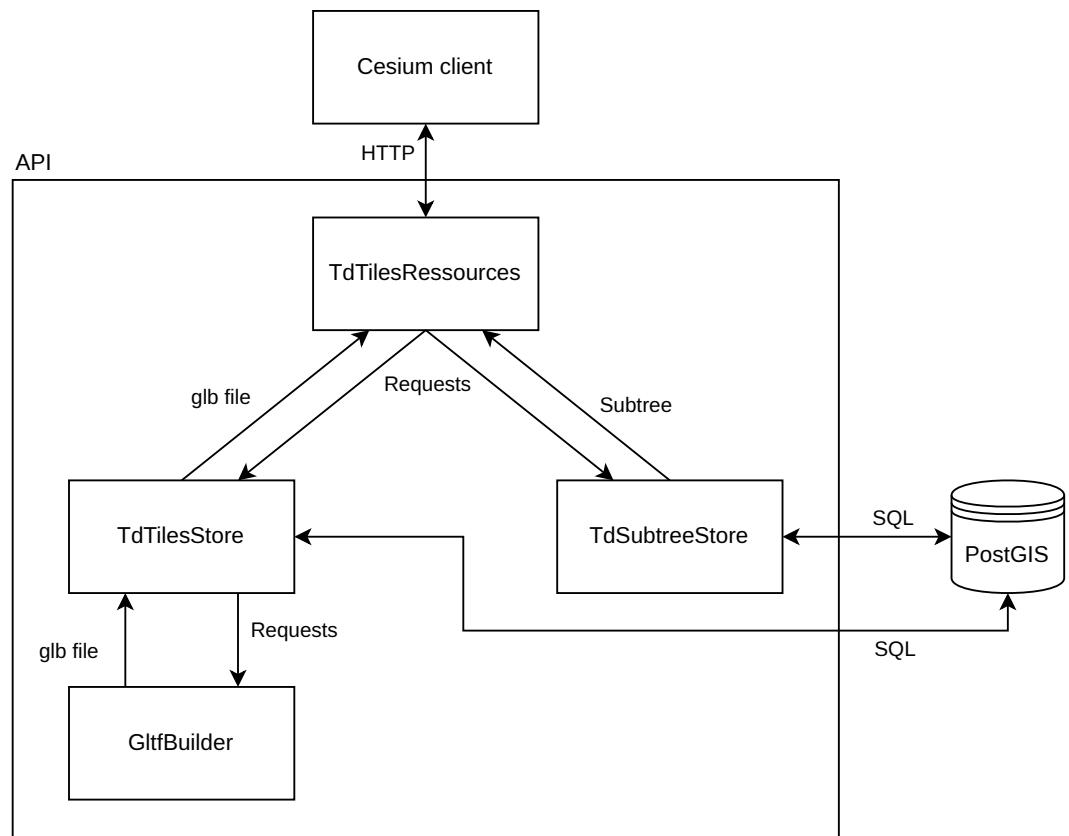


Figure 3.1 – Calcul du SSE [Cesc]

Le client Cesium va en premier lieu demander les Subtrees à l'API afin qu'il puisse savoir quelles tuiles afficher. `TdTilesRessources` va transmettre cette requête à `TdSubtreeStore` qui va se charger de récupérer le Subtree dans la base de donnée si il existe ou sinon d'en créer un, de mettre à jour la base de donnée et de le retourner. Une fois que le client Cesium a reçu le Subtree, il va ensuite demander les Subtrees si il y en a et les contenus des tuiles affichables. Lorsque `TdTilesRessources` reçoit une demande de contenu d'une tuile, elle va transmettre cette demande à `TdTileStore` qui va se charger de récupérer le contenu de la tuile dans la base de donnée si il existe ou sinon de passer la demande à `GltfBuilder`. Ce dernier va générer le contenu de la tuile et le stocker dans la base de donnée avant de le retourner.

3.2 Database

La base de donnée [PostGIS](#)¹ 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 java `javax.sql.DataSource` qui me permet de me connecter à la base de données.

3.2.1 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 [OpenStreetMap](#)² nous permet d'avoir la liste complète des tags existants et nous intéressent concernant les bâtiments.

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

1. <https://postgis.net/>

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

3.2. DATABASE

```
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));
```

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.

3.2.2 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)
);
```

Chapitre 4

Subtrees

4.1 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 4.2) 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. Une *Z-order curve* ou un *Morton order*¹ est utilisé pour définir les indexes (section 4.3).

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

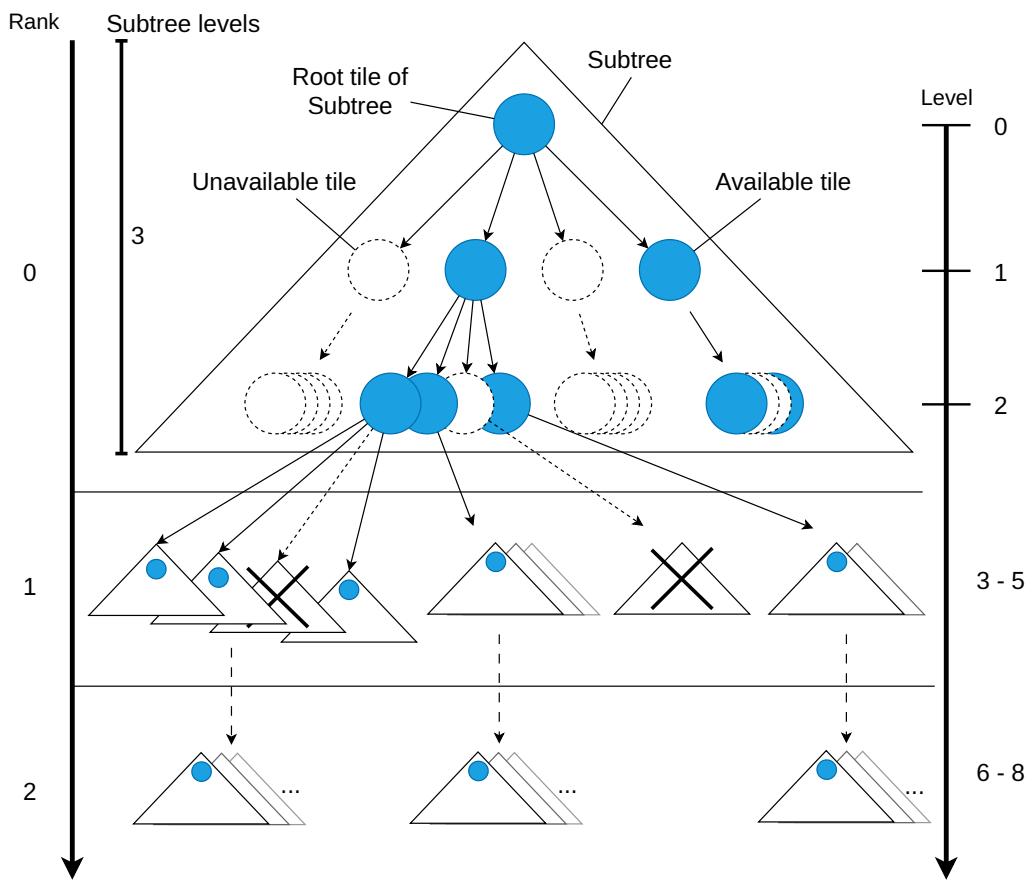


Figure 4.1 – Exemple de hiérarchie de Subtrees

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 de disponibilités citées plus haut. Les *Availables* seront traités plus en détails dans la section 4.2.

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. Par soucis de compréhension, j'appellerai le *rank*, ou *rank* en Anglais, d'un Subtree le niveau de profondeur de ce Subtree par rapport à la racine du tileset. Le Subtree de rang 0 est le Subtree de la racine du tileset. Les Subtree de rang maximum sont les Subtrees qui contiendront les contenus à afficher.

Finalement, 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.
- Un Subtree doit contenir au minimum un niveau.

4.2. CLASSES IMPORTANTES

4.2 Classes importantes

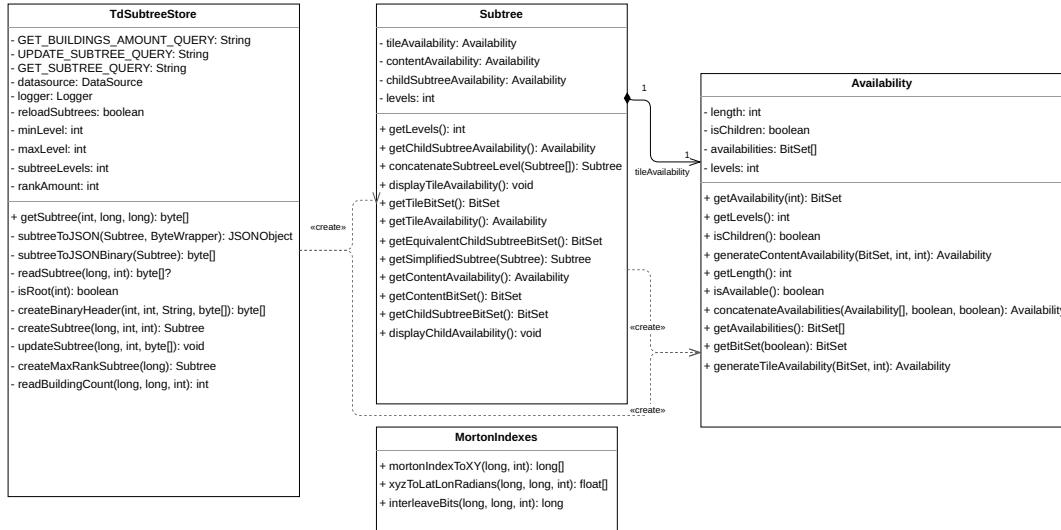


Figure 4.2 – Classes utilisées pour la création d'une hiérarchie de Subtrees

Classe Subtree³

La première classe écrite dans le but de gérer les Subtrees est la classe **Subtree**. Un objet de cette classe comporte tout les éléments nécessaires pour la gestion d'un Subtree. Il contient un objet **Availability** pour chaque type de disponibilité (tuile, contenu, enfants) ainsi que le nombre de levels contenu. La classe offre aussi tous les getters utiles pour accéder aux informations importantes. Elle fournit aussi une fonction statique de concaténation de 4 Subtrees en un d'ordre supérieur. Cette fonction sera utilisée pour la création de l'arbre de Subtrees. Finalement, elle offre deux fonctions de debug pour afficher les *Tile Availability* et le *Child Availability* dans la console.

Classe Availability⁴

La classe Availability est celle qui se chargera de stocker les listes de disponibilités. Pour cela, elle utilise un *BitSet* par soucis d'optimisation. Tout comme la classe Subtree, elle offre une fonction de concaténation de 4 *Availability* en une seule. De plus, elle propose deux fonctions pour générer un *Availability* complet à partir d'un *BitSet* représentant le dernier level du Subtree.

Trois types de disponibilités sont possibles : tuile, contenu et enfants. La disponibilité des tuiles indique si une tuile peut être accédée par le client Cesium. La disponibilité des contenus indique si une tuile contient des bâtiments ou non. Pour finir, la disponibilité des enfants indique si un Subtree enfant au Subtree actuel à une *root tile* (cf. Figure 4.1) qui est disponible. Cette dernière information servira au client Cesium à savoir si il doit demander au serveur un Subtree enfant ou non.

Une liste de disponibilités est donc, comme précisé auparavant, une liste de boolean représentant un des trois types de disponibilité pour chaque tuile du Subtree. Cela veut dire qu'une seule liste de disponibilité s'occupe de tout les niveaux du Subtree.

3. org/apache/baremaps/tdtiles/Subtree/Subtree.java

4. org/apache/baremaps/tdtiles/Subtree/Availability.java

Ces listes de disponibilités sont en théorie stockées dans un seul Bitset qui comprend tous ces niveau. Dans mon implémentation, j'ai choisi de stocker chaque niveau dans un BitSet différent. Cela ne change rien car à l'envoi du Subtree au client, les BitSets seront concaténés dans un seul buffer binaire.

Pour comprendre comment les différents niveaux de tuiles sont représentées dans un BitSet, je vous invite à regarder la figure ci-dessous :

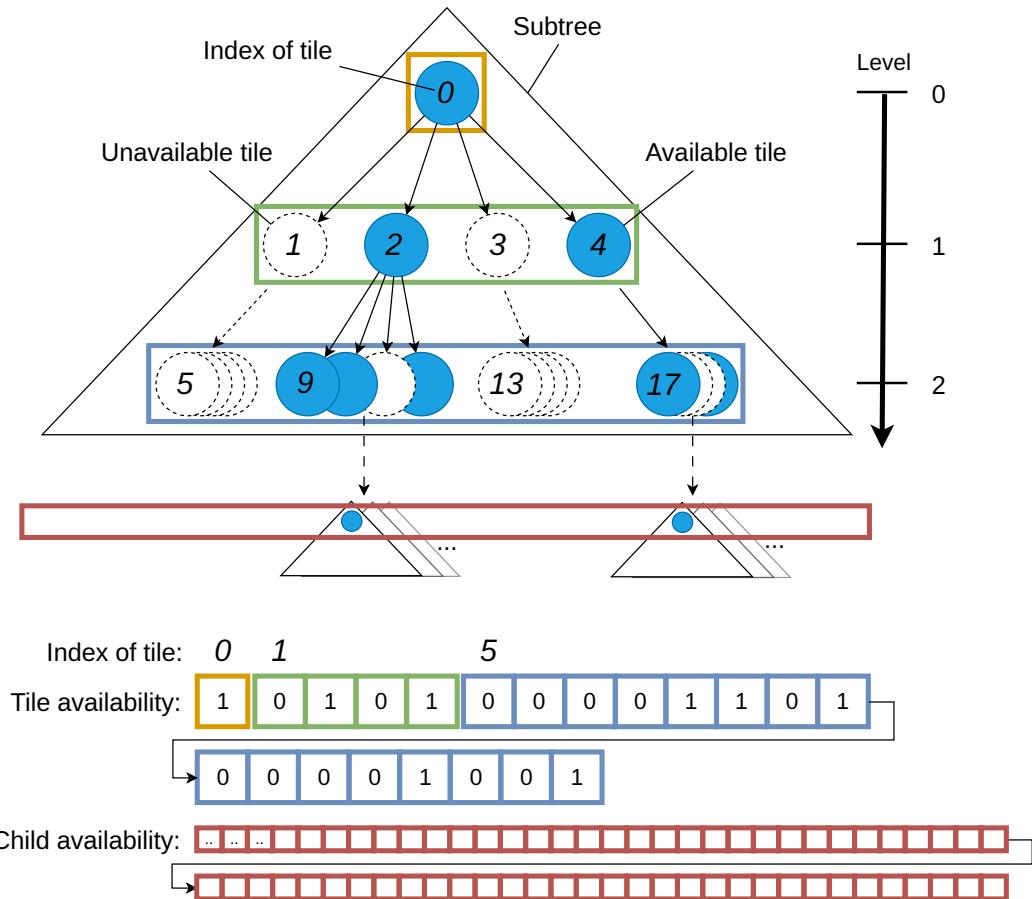


Figure 4.3 – Représentation des niveaux de tuiles dans un BitSet

Vous remarquerez que les listes de disponibilités n'ont pas toutes la même longueur. La longueur des listes de disponibilités des tuiles et des contenus suit la formule suivante :

$$length = \frac{(4^{subtreeLevels} - 1)}{3} \text{ avec } subtreeLevels \text{ le nombre de niveaux d'un Subtree}$$

La longueur des listes de disponibilités des enfants suit la formule suivante :

$$length = 4^{subtreeLevels}$$

Dans l'exemple de la Figure 4.3, les Subtrees sont composés de 3 niveaux (`subtreeLevels`). Les listes de disponibilités des tuiles et des contenus auront donc une longueur de 21 et la liste de disponibilités des enfants aura une longueur de 64.

4.2. CLASSES IMPORTANTES

Classe *TdSubtreeStore*⁵

Cette classe est celle qui orchestrera la création et la distribution des Subtrees. Ce processus suit le diagramme suivant :

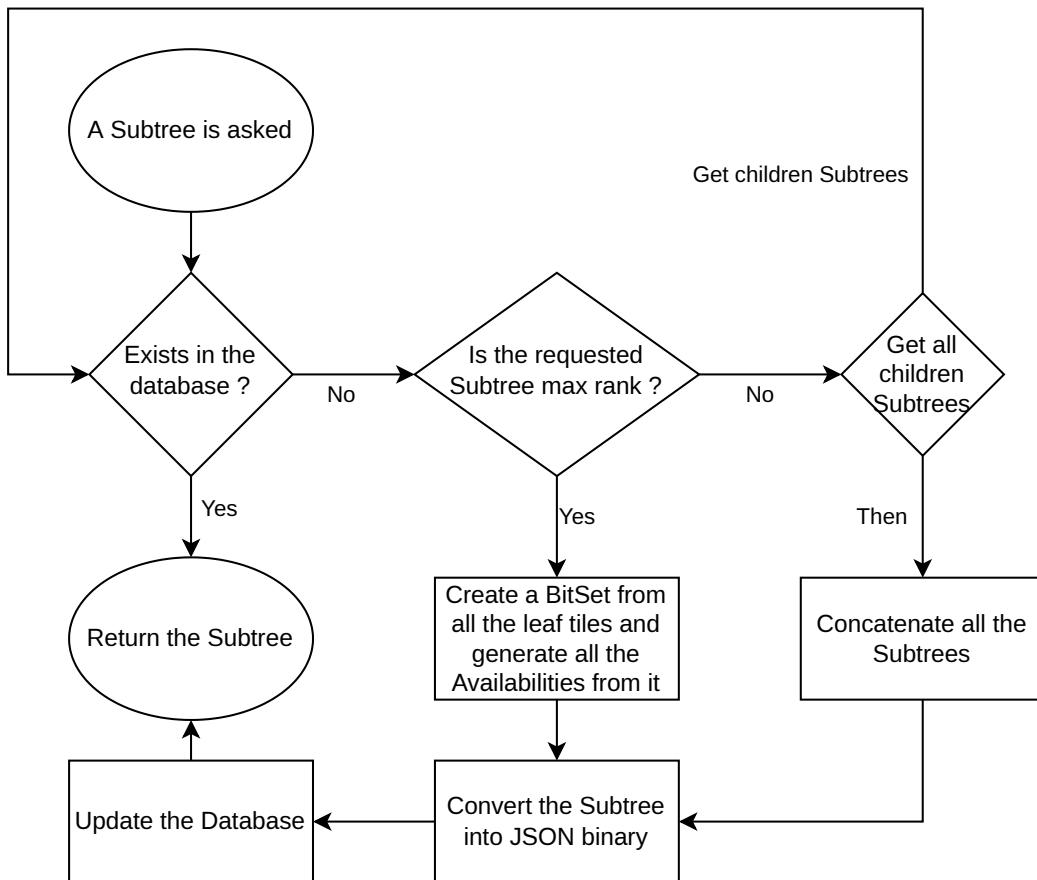


Figure 4.4 – Classes utilisées pour la création d'une hiérarchie de Subtrees

Vous observerez que la classe utilise un système récursif pour la création d'un Subtree. Puisqu'un Subtree parent doit être au courant des disponibilités des Subtrees enfants, il faudrait en théorie que ces derniers soient connus. Cependant, cela peut être optimisé en ne faisant que regarder si des bâtiments se trouvent dans la tuile enfant. Pour cela, une simple requête SQL simple permet de le déterminer :

```

SELECT EXISTS (
    SELECT 1
    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))
) OR EXISTS (
    SELECT 1
    FROM osm_relations
    WHERE (tags ? 'building' OR tags ? 'building:part')
        AND st_intersects(geom, st_makeenvelope(%1$s, %2$s, %3$s, %4$s, 4326))
) AS has_buildings
    
```

5. org/apache/baremaps/tdtiles/TdSubtreeStore.java

CHAPITRE 4. SUBTREES

Les deux méthodes ont des avantages et des inconvénients. La méthode calculant tous les Subtrees est énormément plus lente, mais les utilisateurs n'auront pas de temps d'attente supplémentaire lorsqu'ils se déplaceront dans la carte. La méthode calculant les Subtrees au fur et à mesure est quasiment instantanée, mais les utilisateurs devront attendre un peu plus longtemps lorsqu'ils se déplaceront dans la carte.

Pour le développement de l'application, je recommande donc de ne pas calculer tous les Subtrees en avance car il y en a rarement besoins. Par contre, pour un usage en production, il serait préférable de les calculer en avance.

Ce choix ainsi que le choix de re-créer les Subtrees ou les fichiers glTF des tuiles peuvent être modifiés dans la classe `TdTilesResources`⁶.

6. [org/apache/baremaps/server/TdTilesResources.java](https://github.com/apache/baremaps/blob/main/server/TdTilesResources.java)

4.3. MORTON INDEX

4.3 Morton Index

Le Morton index, également connu sous le nom de code Z, est une technique d'encodage spatiale utilisée dans le contexte de l'implicit tiling. Cette méthode permet de convertir des coordonnées multidimensionnelles en une valeur unidimensionnelle tout en préservant la proximité spatiale des points. Dans le cas de Cesium, le Morton index facilite le découpage et l'organisation hiérarchique des tuiles, rendant plus efficace la gestion et le rendu des grandes quantités de données géospatiales. En encodant les coordonnées des tuiles dans une structure arborescente, le Morton index permet une récupération rapide des tuiles nécessaires pour afficher une vue spécifique, optimisant ainsi les performances et la fluidité de la visualisation.

Il y a plusieurs opérations sur les indexes de Morton qui sont nécessaires pour la gestion des tuiles. Ces opérations sont les suivantes :

- Convertir des coordonnées (X, Y, Level) en un index de Morton.
- Convertir un index de Morton en des coordonnées (X, Y, Level).
- Obtenir les 4 indexes enfants d'un index de Morton.
- Obtenir le nombre de bits nécessaires pour encoder un index de Morton.

Conversion de coordonnées en index de Morton

Pour convertir des coordonnées (X, Y, Level) en un index de Morton, nous devons les *intercaler* pour former un seul entier. L'intercalation des bits est une technique qui consiste à mélanger les bits de deux entiers pour former un index de Morton. Supposons que nous ayons deux entiers X et Y , et que nous voulons les combiner en un seul entier Z en intercalant leurs bits.

1. Représentation binaire

Tout d'abord, convertissons X et Y en leurs représentations binaires :

$$X = x_n x_{n-1} \dots x_1 x_0$$

$$Y = y_n y_{n-1} \dots y_1 y_0$$

2. Intercalage des bits

Ensuite, nous intercalons les bits de X et Y pour obtenir Z . Le bit i de X sera placé dans la position $2i$ de Z , et le bit i de Y sera placé dans la position $2i + 1$ de Z . De plus, l'index de Morton dépend du niveau de profondeur de la tuile. Le nombre de bits nécessaires pour encoder un index de Morton est donc égal à $2 \times \text{level}$.

Formulons cela mathématiquement :

$$Z = \text{interleave}(x, y) = z_{2n+1} z_{2n} z_{2n-1} \dots z_1 z_0$$

où

$$z_{2i} = x_i \quad \text{et} \quad z_{2i+1} = y_i \quad \text{pour } i = 0, 1, \dots, n \quad \text{avec } n = 2 \times \text{level}$$

3. Exemple

Prenons un exemple simple avec $x = 5$ et $y = 3$. En binaire, nous avons :

$$x = 101_2 \quad \text{et} \quad y = 011_2$$

Intercalons les bits :

$$z = 011011_2$$

En décimal, cela donne :

$$z = 27_{10}$$

Ainsi, l'index de Morton pour $x = 5$ et $y = 3$ est 27.

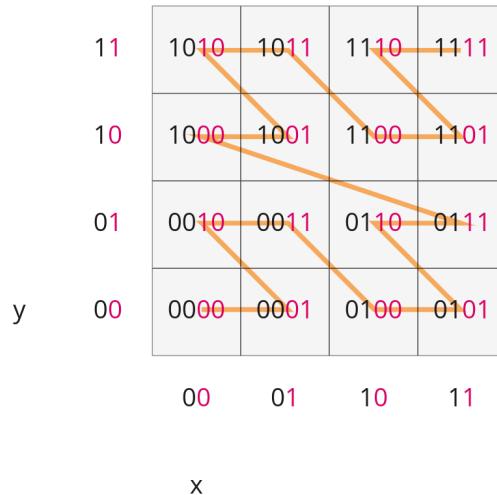


Figure 4.5 – Indexes X Y vers index de Morton [Cesd]

Conversion d'un index de Morton en coordonnées

Le processus inverse du *interleaving bits* consiste à séparer les bits d'un index de Morton Z pour récupérer les deux entiers X et Y .

1. Représentation binaire :

Supposons que nous avons un entier Z et que nous voulons en extraire les deux entiers X et Y . Représentons Z en binaire :

$$Z = z_{2n+1}z_{2n}z_{2n-1} \dots z_1z_0$$

4.3. MORTON INDEX

2. Séparation des bits :

Nous séparons les bits de Z pour reconstruire X et Y . Les bits pairs de Z formeront X , et les bits impairs de Z formeront Y .

Formulons cela mathématiquement :

$$X = x_n x_{n-1} \dots x_1 x_0$$

$$Y = y_n y_{n-1} \dots y_1 y_0$$

où

$$x_i = z_{2i} \quad \text{et} \quad y_i = z_{2i+1} \quad \text{pour} \quad i = 0, 1, \dots, n \quad \text{avec} \quad n = 2 \times \text{level}$$

3. Exemple :

Reprenons le dernier exemple avec $Z = 27$. En binaire, nous avons :

$$Z = 011011_2$$

Séparons les bits pour obtenir X et Y :

$$X = z_4 z_2 z_0 = 101_2 = 5_{10}$$

$$Y = z_5 z_3 z_1 = 011_2 = 3_{10}$$

Ainsi, les entiers X et Y extraits de l'index de Morton $Z = 27$ sont respectivement 5 et 3.

Obtention des indexes enfants

Pour trouver les indexés enfants d'un index de Morton, nous devons simplement effectuer un décalage de 2 bits vers la gauche de l'index de Morton puis ajouter l'index de l'enfant voulu (de 0 à 3).

Par exemple, pour obtenir l'index de Morton de l'enfant 2 de l'index de Morton 27, nous effectuons les opérations suivantes :

$$27 = 011011_2 \quad \Rightarrow \quad 01101100_2 = 108_{10}$$

$$108 + 2 = 110_{10} = 01101110_2$$

Ainsi, l'index de Morton de l'enfant 2 de l'index de Morton 27 est 110.

Une distinction qui se trouve parfois pratique est la distinction d'un index local versus un index global. Un index global est un index de Morton qui représente une tuile sur l'entièreté du Tileset, en comptant donc depuis l'index 0 du level 0. Tandis qu'un index local est un index de Morton qui représente une tuile avec pour référence l'index 0 du subtree.

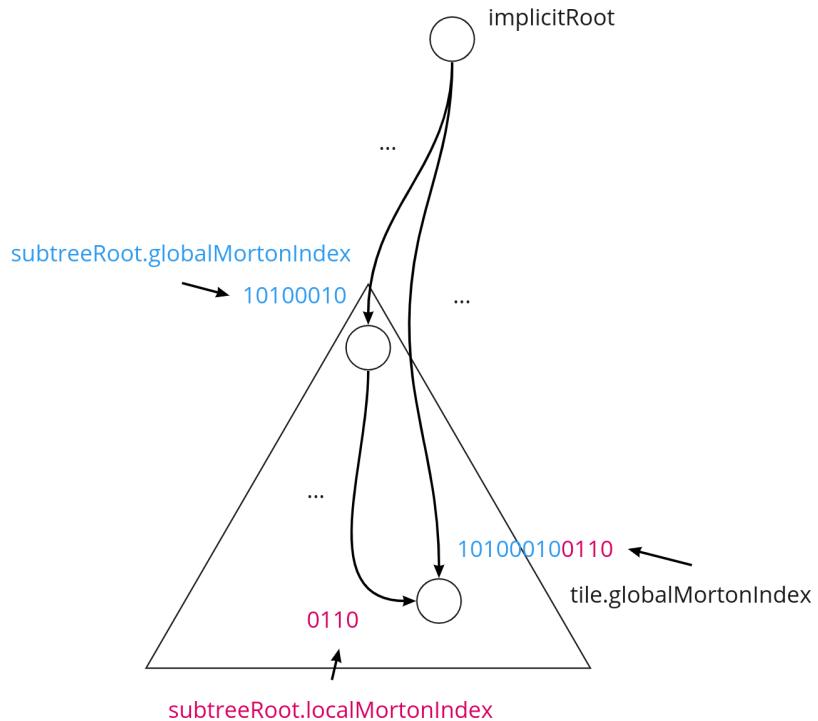


Figure 4.6 – Index local et index global [Cesd]

4.4. CRÉATION DE L'ARBRE DE SUBTREES

4.4.1 Création de l'arbre de Subtrees

getSubtree

Comme déjà légèrement présenté dans la section 4.2 Figure 4.4, la fonction `getSubtree` est une fonction qui retourne un Subtree demandé. La fonction n'est pas statique car des paramètres spécifiques à la tailles des Subtrees comme `subtreeLevels`, le nombre de niveau dans un Subtree (cf. Figure 4.1), et `maxLevels`, le nombre total de niveaux, doivent être connus au préalable.

Cette fonction de la classe `TdSubtreeStore`⁷ est la seule fonction appelable par l'utilisateur. Les deux fonction ci-dessous, `createMaxRankSubtree` et `createSubtree`, sont des fonctions internes à la création d'un Subtree qui ne peuvent pas être appelées dans un autre contexte.

Pour être demandé, l'index du Subtree doit correspondre un index *root* (cf. Figure 4.1, *Root tile of Subtree*) uniquement. Si le Subtree à l'index demandé existe dans la database existe, alors la fonction le retourne directement. Sinon, en fonction de si le Subtree demandé est un Subtree de rang maximal ou non, la fonction `createMaxRankSubtree` ou `createSubtree` sera appelée. Finalement, la fonction transforme le Subtree en un objet JSON binaire, met à jour la database et le retourne. La transformation en JSON binaire sera décrite dans la section 4.5.

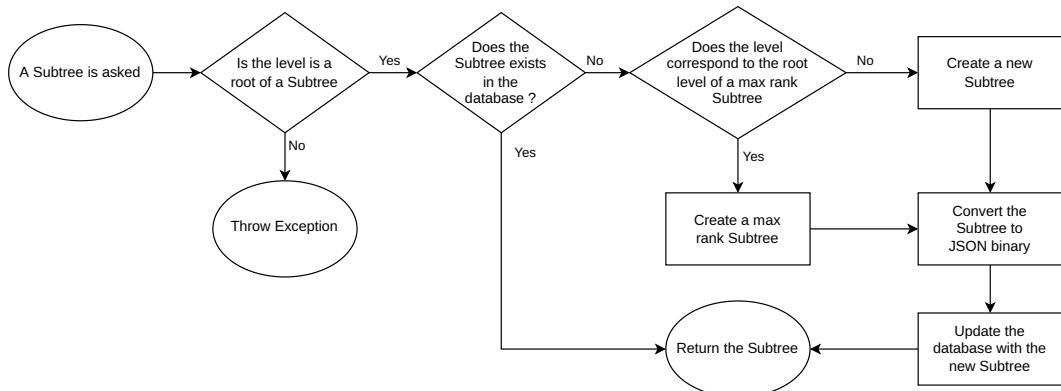


Figure 4.7 – Flowchart de la fonction `getSubtree`

7. org.apache/baremaps/tddiles/TdSubtreeStore.java

createMaxRankSubtree

Cette fonction se charge uniquement de la création d'un Subtree de rang maximal. Un rang maximal est un Subtree qui se trouve à la fin de la hiérarchie de Subtrees, une feuille de l'arbre des Subtrees (cf. l'échelle de gauche sur la Figure 4.1).

Pour commencer, si aucun bâtiments ne se trouve dans la zone de la tuile racine du Subtree, alors la fonction retourne un Subtree vide mais qui aura tout de même des objets Availability de bonne longueur et un nombre de levels correct. Sinon, la fonction va commencer par créer un BitSet qu'il remplira en parcourant toutes les tuiles de son niveau maximal (les plus petites tuiles qu'il contient). Si la tuile feuille contient des bâtiments, alors le BitSet sera mis à 1 à l'index de Motron local qui correspond à la tuile. Une fois que toutes les tuiles ont été parcourues, la fonction va demander à la classe Availability de générer deux objets différents Availability à partir de ce BitSet, un pour la disponibilité des tuiles et un pour la disponibilité du contenu. En ce qui consiste la liste de disponibilité des enfants du Subtree, elle doit être vide. La fonction crée donc un Availability vide de la bonne longueur pour cette liste. Finalement, la fonction retourne un Subtree avec les trois Availability générés.

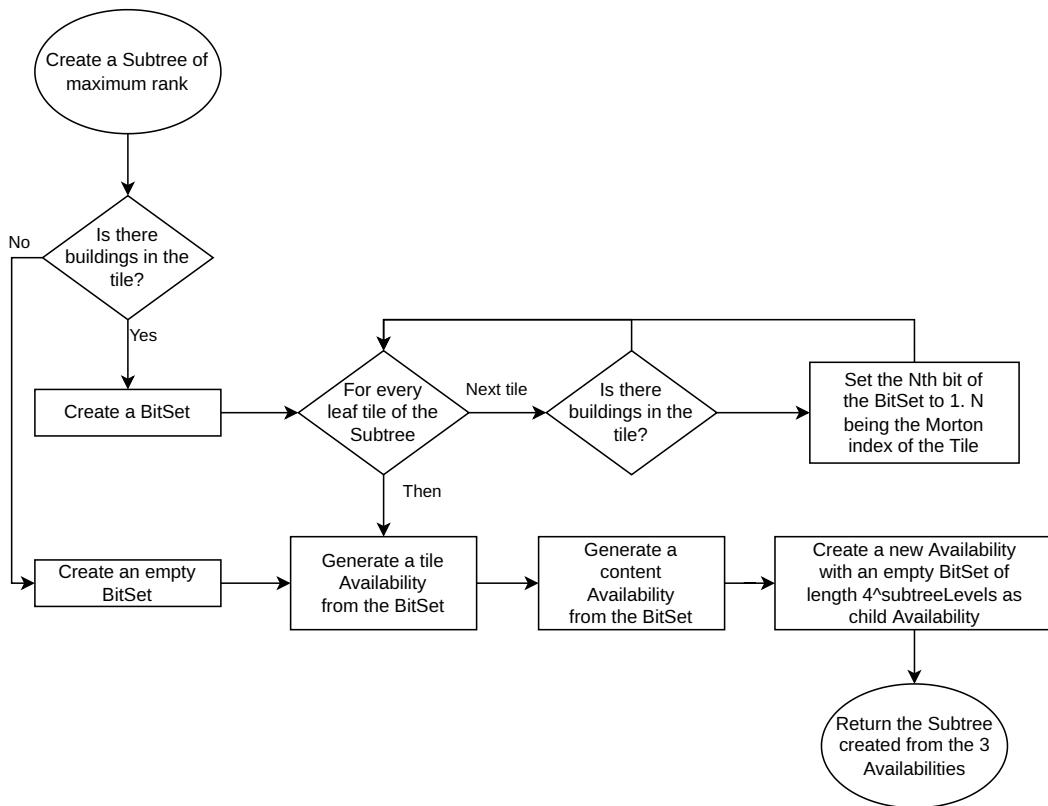


Figure 4.8 – Flowchart de la fonction *createMaxRankSubtree*

4.4. CRÉATION DE L'ARBRE DE SUBTREES

createSubtree

La fonction `createSubtree` est légèrement similaire à `createMaxRankSubtree` dans le sens qu'elle retourne aussi un Subtree sauf qu'elle ne s'occupe que des Subtree n'étant pas de rang maximal. Une autre similarité est que si aucun bâtiments ne se trouve dans la zone de la tuile racine du Subtree, alors la fonction retourne un Subtree vide.

Là où la fonction diffère est qu'elle se base sur le mécanisme de récursivité et de concaténation pour créer les Subtrees. Pour un Subtree demandé, la fonction va commencer par créer des Subtrees feuilles, du dernier niveau du Subtree demandé, puis les concaténer pour créer les Subtrees de niveau supérieur. On distingue donc deux cas de subtrees demandés :

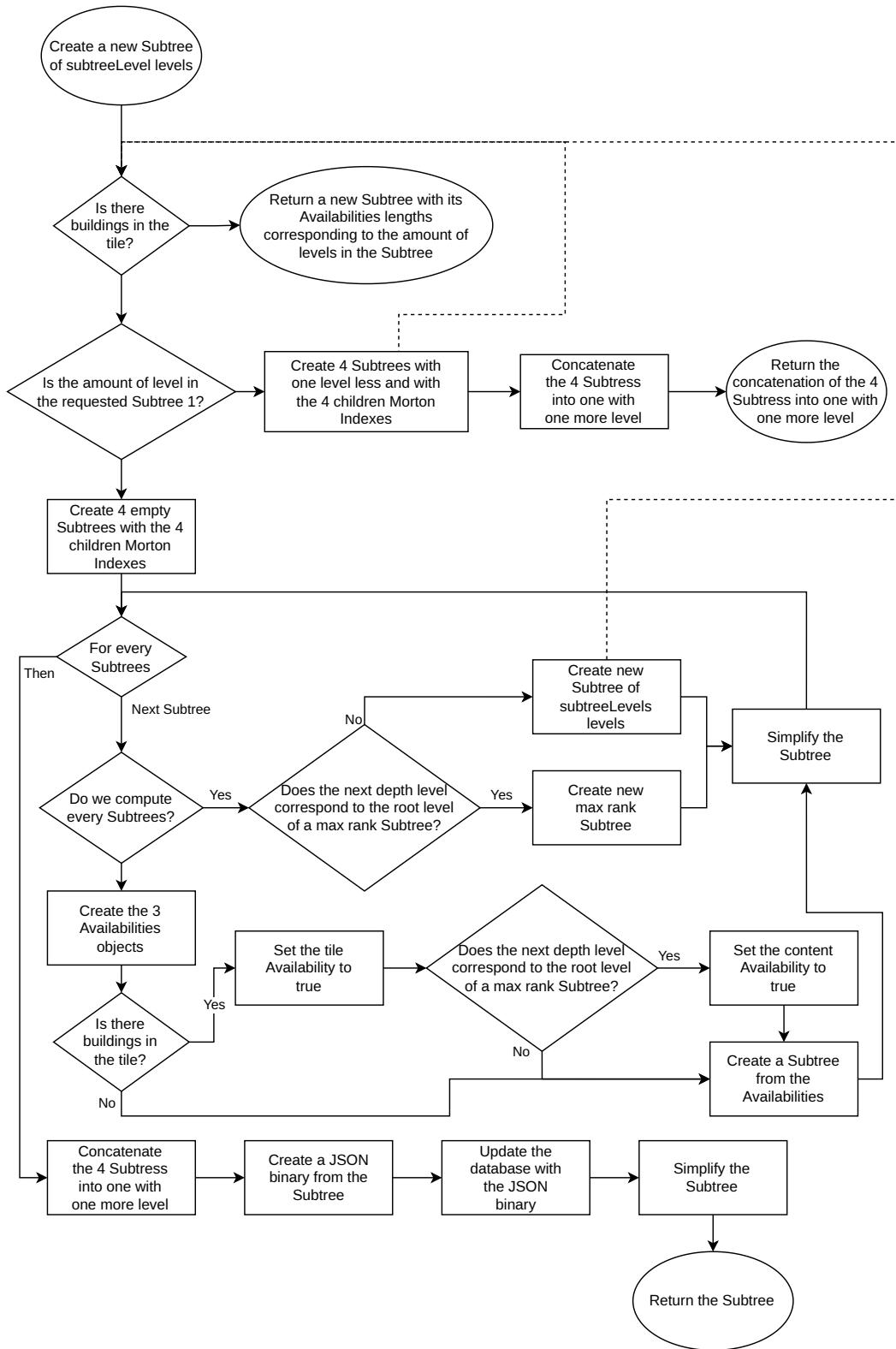
Subtree ayant plusieurs niveaux : Ce sont les Subtrees qui ne sont pas des feuilles du Subtree demandé initialement. Pour le créer, la fonction va donc s'appeler elle-même pour créer les Subtrees de niveau inférieur, puis les concaténer pour créer le Subtree demandé.

Subtree ayant un seul niveau : Ce sont les Subtrees qui sont des feuilles du Subtree demandé initialement. La fonction va tout d'abord créer une liste de 4 Subtrees. Ensuite, pour chacun de ces 4 Subtree, il y a de nouveau deux cas :

On veut générer tous les Subtrees du Tileset : En fonction de si ces 4 prochains Subtrees en dessous de cette feuille sont de rang maximal ou non, la fonction va appeler `createMaxRankSubtree` ou à nouveau `createSubtree` en demandant un Subtree complet pour les créer.

On ne veut générer que les Subtrees nécessaires : La fonction commence simplement par regarder si il y a des bâtiments dans les tuiles de ce niveau. Si ce n'est pas le cas, elle créera 4 Subtrees vides. Sinon, elle va créer un Availability pour la disponibilité des tuiles de longueur 1 ayant une valeur de 1. Pour l'Availability de contenu, si les Subtrees inférieurs sont de rang maximal, alors elle créera aussi un Availability de longueur 1 ayant une valeur de 1, sinon de valeur 0. En ce qui concerne la liste de disponibilité des enfants du Subtree, elle peut être ignorée puisqu'elle disparaîtra lors de la concaténation.

Finalement, la fonction va concaténer les 4 Subtrees créés, update la database avec le résultat de la concaténation et retourner le Subtree demandé simplifié.


 Figure 4.9 – Flowchart de la fonction *createSubtree*

4.4. CRÉATION DE L'ARBRE DE SUBTREES

4.4.2 Génération de Availability

Dans la fonction `createMaxRankSubtree`, deux objets `Availability` sont générés à partir d'un `BitSet`. Il faut donc trouver les autres `BitSets` de niveau supérieur à partir du `BitSet` donné.

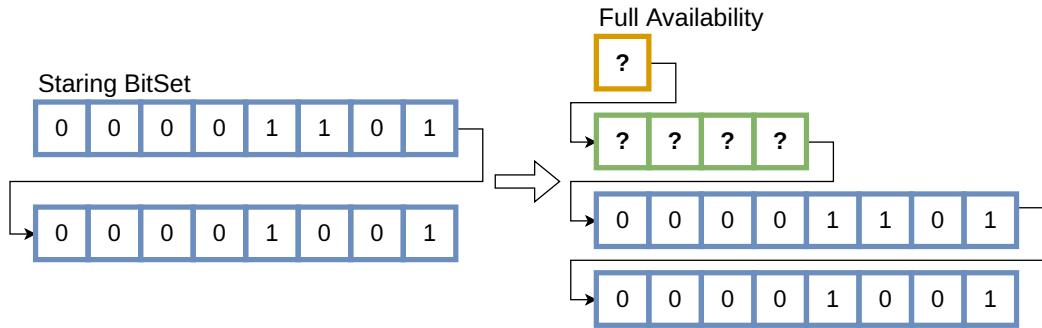


Figure 4.10 – Génération de Availability : État initial

Cela peut être fait en effectuant une opération `OR` sur chaque groupe de 4 bits du `BitSet`. Les bits du `BitSet` de niveau supérieur sont les résultats de ces opérations. On répète ensuite cette opération pour chaque niveau supérieur jusqu'à atteindre le niveau maximal.

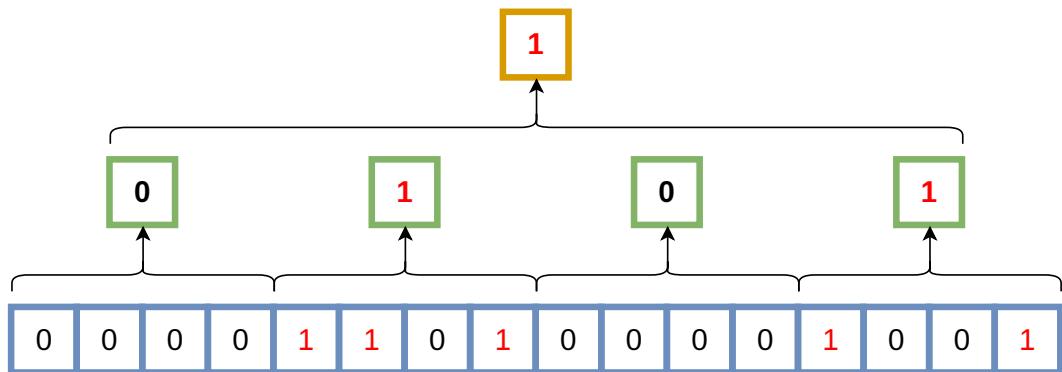


Figure 4.11 – Génération de Availability : Procédure

4.4.3 Concaténations

Les concaténations de Subtrees sont effectuées fréquemment dans la fonction `createSubtree`. Elles consistent à diviser 4 objets Availability en BitSets par niveau. Ces BitSets sont ensuite concaténés pour former un seul BitSet par niveau. On se retrouve alors avec un seul objet Availability ayant un niveau de plus que les objets initiaux, mais il lui manque encore le BitSet de taille 1 au niveau 0. Pour le créer, on peut simplement effectuer une opération OR sur les 4 BitSets de niveau 1.

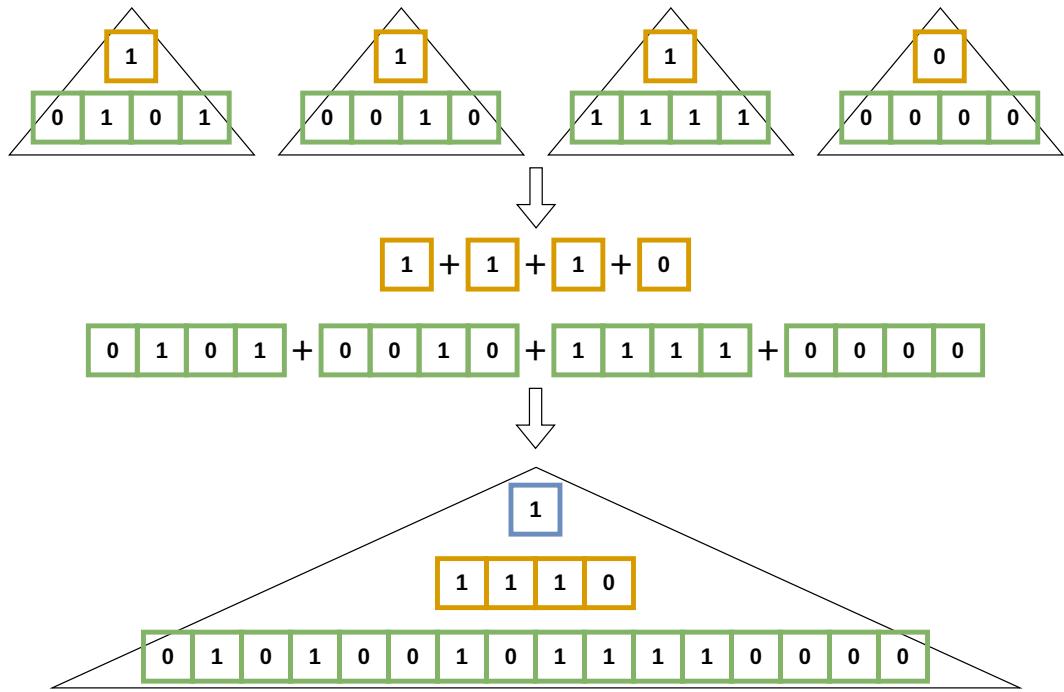


Figure 4.12 – Concaténation de 4 Availability

4.5. TRANSMISSION AU CLIENT CESIUM

4.5 Transmission au client Cesium

Maintenant que nous savons comment les Subtrees sont créés, il ne reste plus qu'à savoir les transmettre au client Cesium. Ce client accepte sous deux formats différents. Nous pouvons soit lui envoyer la description du Subtree sous forme de JSON et lui envoyer les listes de disponibilités dans des fichiers binaires séparément en précisant les URI de ces fichiers dans le JSON. Comme seconde option, nous pouvons tout lui envoyer en un seul fichier binaire qui respecte le *Subtree Binary Format*⁸. Sous ce format, les fichiers binaires sont inclus sous forme de buffer binaire. Les différentes listes de disponibilités pourront ensuite être retrouvées grâce à des offsets indiqués dans la partie JSON à la place des URI. Sauf raison externe, cette deuxième option est généralement plus simple à utiliser. C'est pourquoi je l'ai choisie pour mon implémentation.

Le *Subtree Binary Format* est un format où les informations sont stockées en little-endian et dont les fichiers comportent un header de 24 Bytes. Ce header contient les informations suivantes :

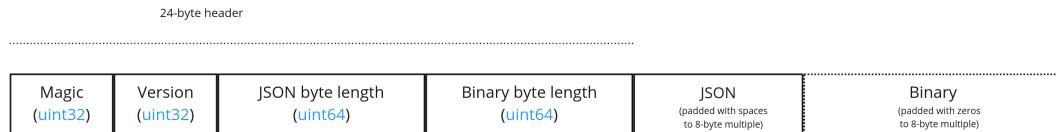


Figure 4.13 – Header d'un fichier au Subtree Binary Format [Cese]

Les informations contenues dans le header sont les suivantes :

- **Magic** : 4 Bytes, *magic number* identifiant ce fichier comme un Subtree. A toujours une valeur de 0x74627573.
- **Version** : 4 Bytes, version du format binaire. Actuellement, le 24.07.2024, il s'agit de 1.
- **JSON Byte Length** : 8 Bytes, longueur de la partie JSON en Bytes.
- **Binary Byte Length** : 8 Bytes, longueur de la partie binaire en Bytes.

Concernant le padding⁹, les deux derniers éléments, *JSON Byte Length* et *Binary Byte Length*, doivent compter les Bytes de padding. Le padding est nécessaire car chaque chunk doit être aligné sur une frontière de 8 Bytes. La partie doit être remplie avec le caractère espace (0x20) sur la fin, tandis que la partie binaire doit être remplie avec des zéros (0x00) sur la fin.

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

9. <https://www.lalanguefrancaise.com/dictionnaire/definition/padding>

4.5.1 Écriture de JSON

Le JSON doit contenir les objets suivants :

- **buffers** : une liste d'objet contenant le nom, les URI si nécessaire et la longueur des buffers binaires.
- **bufferViews** : une liste d'objet contenant les offsets des différentes parties des buffers binaires.
- **tileAvailability** : un objet décrivant liste de disponibilités des tuiles.
- **contentAvailability** : un objet décrivant liste de disponibilités des contenus.
- **childAvailability** : un objet décrivant liste de disponibilités des enfants.

Les objets **buffer** doivent contenir les informations suivantes :

- **name** : nom du buffer.
- **uri** : URI du fichier binaire, doit être omis si on utilise le *Subtree Binary Format*.
- **byteLength** : longueur du fichier binaire.

Les objets **bufferViews** doivent contenir les informations suivantes :

- **buffer** : index du buffer auquel le bufferView fait référence dans la liste de buffers.
- **byteOffset** : offset de la vue dans le buffer.
- **byteLength** : longueur de la vue.

Les objets **tileAvailability**, **contentAvailability** et **childAvailability** peuvent soit contenir un objet **constant** ayant une valeur de 0 ou 1 qui signifie que toutes les tuiles ont la même disponibilité. Soit contenir un objet **bufferView** qui fait référence à un bufferView contenant les listes de disponibilités avec un objet **availableCount** qui indique le nombre de tuiles disponibles. Notez que si le Subtree ne contient pas de liste de contenu, **contentAvailability** doit être omis.

- **constant** : objet contenant une valeur de 0 ou 1.
- **bufferView** : index du bufferView contenant les listes de disponibilités.
- **availableCount** : nombre de tuiles disponibles.

D'autres informations peuvent notamment être ajoutées au JSON tel que des metadata.

4.5. TRANSMISSION AU CLIENT CESIUM

Voici un exemple de JSON d'un des Subtrees créés par mon implémentation :

```
{  
    "buffers": [  
        {  
            "name": "Internal Buffer",  
            "byteLength": 24  
        }  
    ],  
    "contentAvailability": [  
        {  
            "availableCount": 54,  
            "bitstream": 1  
        }  
    ],  
    "childSubtreeAvailability": {  
        "availableCount": 0,  
        "constant": 0  
    },  
    "bufferViews": [  
        {  
            "byteOffset": 0,  
            "byteLength": 11,  
            "buffer": 0  
        },  
        {  
            "byteOffset": 11,  
            "byteLength": 11,  
            "buffer": 0  
        }  
    ],  
    "tileAvailability": {  
        "availableCount": 54,  
        "bitstream": 0  
    }  
}
```


Chapitre 5

Conclusion

Pour conclure, ce travail a permis de mettre en place un prototype d'une utilisation du client Cesium pour afficher des bâtiments venant d'OpenStreetMap en 3D. Ce prototype a démontré qu'il était possible d'utiliser la fonctionnalité d'implicit tiling de la spécification 3D Tiles Next tout en gardant un résultat satisfaisant et qu'il était possible de le faire en traitant les données de manière uniforme pour l'entièreté du dataset d'OpenStreetMap.

La génération des Subtrees est maintenant totalement fonctionnelle. Cette génération est régie par des paramètres pouvant changer intégralement la forme de la hiérarchie des Subtrees. Cela permet d'adapter à souhait quels niveaux de la division par implicit tiling contiendra du contenu 3D ainsi que de préciser leur niveaux de détails.

Actuellement, ce prototype est parfaitement utilisable à condition de passer un peu de temps à trouver les bons niveaux de compression des géométries des bâtiments 3D et les bons niveaux auxquels appliquer ces compression. Pour cela, de nombreuses mesures doivent être prises sur différentes plateformes pour trouver le bon compromis entre qualité et performance. Autant cruciale qu'elle l'est, cette étape sort du cadre de ce travail. Elle reste donc à être effectuée.

Un autre point qui semble être la prochaine étape logique du développement de ce projet est l'amélioration de la génération des objets glTF représentant les bâtiments. Beaucoup d'informations sont mises à disposition par le dataset proposé par OpenStreetMap. Comme expliqué dans le rapport du travail de Bachelor mentionné dans la section 2.5, il est possible de pousser la génération des bâtiments plus loin en intégrant leur toit ou en ajoutant des textures. Mais ce rapport montre aussi qu'il est possible de générer bien plus que des bâtiments, comme les routes ou la végétation.

Ian 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>. (accessed : 18.07.2024) (cf. p. 4, 7, 17).
- [Cesd] CESIUMGS. *Availability Indexing*. URL : <https://github.com/CesiumGS/3d-tiles/blob/main/specification/ImplicitTiling/AVAILABILITY.adoc>. (accessed : 22.07.2024) (cf. p. 28, 30).
- [Cese] CESIUMGS. *Implicit Tiling*. URL : <https://github.com/CesiumGS/3d-tiles/tree/main/specification/ImplicitTiling>. (accessed : 19.07.2024) (cf. p. 1, 37).
- [Ope] OPENSTREETMAP. *Minimum level of buildings*. URL : <https://wiki.openstreetmap.org/wiki/File:Minlevel.svg>. (accessed : 24.07.2024) (cf. p. 15).
- [S2G] S2GEOMETRY. *Bounding Volume S2 extension website*. URL : http://s2geometry.io/devguide/s2cell_hierarchy#s2cellid-numbering. (accessed : 19.07.2024) (cf. p. 10).