Développement d'un prototype de bibliothèque d'analyse spatiale 3D côté client dans un contexte de jumeau numérique urbain

Auteur : Denis, Théo
Promoteur(s) : Kasprzyk, Jean-Paul
Faculté : Faculté des Sciences
Diplôme : Master en sciences géographiques, orientation géomatique, à finalité spécialisée en geodata-expert
Année académique : 2025-2026
URI/URL : http://hdl.handle.net/2268.2/25054

Faculté des sciences
Département de géographie

DÉVELOPPEMENT D'UN PROTOTYPE DE BIBLIOTHÈQUE D'ANALYSE SPATIALE 3D CÔTÉ CLIENT DANS UN CONTEXTE DE JUMEAU NUMÉRIQUE URBAIN

Mémoire présenté par : **Théo DENIS**

pour l'obtention du titre de

Master en sciences géographiques,
orientation géomatique,
à finalité spécialisée geodata-expert

Année académique
2025-2026

Président du jury :
Pr René WARNANT

Membres du jury :
Dr Jean-Paul KASPRZYK (Promoteur)
Pr Roland BILLEN
Dr Roberta RAVANELLI

Remerciements

Je tiens tout d'abord à exprimer ma vive reconnaissance à mon promoteur, Monsieur Kasprzyk, pour la qualité de son encadrement, ses nombreux conseils avisés et sa grande disponibilité tout au long de ce travail.

J'adresse également mes sincères remerciements à Monsieur Billen ainsi qu'à Imane Jebboub pour leur aide précieuse, leurs orientations et le temps qu'ils m'ont consacré.

Merci à ma famille et à mes amis pour leur soutien sans faille. Un merci tout particulier à mon papa pour les "casse-dalles" qu'il m'a préparés afin de me donner l'énergie nécessaire pour avancer.

Enfin, une pensée spéciale pour mon chat, Neptune, fidèle compagnon de rédaction qui a su m'épauler (le plus souvent depuis mon lit) durant l'écriture de ce mémoire.

Abstract (Français)

Les jumeaux numériques urbains et les standards de modélisation 3D, tels que CityJSON, occupent désormais une place centrale dans la gestion des villes intelligentes. Toutefois, un paradoxe technologique persiste : alors que la puissance de calcul côté client ne cesse de croître, l'analyse spatiale tridimensionnelle dans les navigateurs web accuse un retard significatif par rapport à son homologue bidimensionnelle. L'absence d'équivalent 3D à des bibliothèques comme Turf.js contraint les développeurs à déléguer systématiquement le géotraitement à des serveurs distants, une tâche rendue parfois complexe par l'émergence de bases de données stockant nativement le format JSON.

Ce mémoire vise à combler ce vide technologique en proposant un prototype de bibliothèque JavaScript capable d'effectuer des opérations de géotraitement 3D, à savoir les calculs de volumes, d'intersections et la génération de zones tampons, directement au sein du navigateur. Notre approche méthodologique repose sur une architecture orientée objet rigoureuse, conçue pour manipuler des données CityJSON indépendamment de leur niveau de détail. Cette structure intègre des outils tiers spécialisés (Three.js, earcut, three-csg-ts) pour implémenter deux algorithmes de calcul de volumes (la méthode exacte des tétraèdres et l'approche stochastique de Monte-Carlo) ainsi que des opérations booléennes sur les solides via la géométrie de construction de solides (CSG).

La validation de ces développements, réalisée sur le modèle 3D du quartier d'Outremeuse à Liège, a livré des résultats contrastés. D'une part, la méthode des tétraèdres a démontré une efficacité remarquable, offrant une précision identique aux solutions serveur de référence (PostGIS), avec des temps d'exécution de l'ordre de la milliseconde. D'autre part, les tests ont mis en lumière les limites actuelles du géotraitement en JavaScript pur, notamment pour le calcul d'intersections. La restriction à une précision de 32 bits (simple précision), inhérente à l'écosystème WebGL/Three.js, engendre une dégradation numérique significative lors de la manipulation de coordonnées projetées, compromettant la conformité des résultats à la norme ISO 19107. Ce travail confirme ainsi le potentiel du déport des calculs de géotraitement vers le client, tout en identifiant les verrous technologiques, tels que la précision des nombres à virgule flottante, qu'il reste à lever.

Abstract (Anglais)

Urban Digital Twins and 3D modeling standards, such as CityJSON, now play a central role in smart city management. However, a technological paradox persists : while client-side computing power continues to grow, three-dimensional spatial analysis in web browsers lags significantly behind its two-dimensional counterpart. The lack of a 3D equivalent to libraries like Turf.js compels developers to systematically offload geoprocessing to remote servers, a task sometimes complicated by the emergence of databases that natively store the JSON format.

This thesis aims to bridge this technological gap by proposing a prototype JavaScript library capable of performing 3D geoprocessing operations—namely, volume calculations, intersections, and buffer zone generation—directly within the browser. Our methodological approach relies on a rigorous object-oriented architecture, designed to handle CityJSON data regardless of its level of detail. This structure integrates specialized third-party tools (Three.js, earcut, three-csg-ts) to implement two volume calculation algorithms (the exact tetrahedron method and the stochastic Monte Carlo approach) as well as Boolean operations on solids via Constructive Solid Geometry (CSG).

The validation of these developments, carried out on the 3D model of the Outremeuse district in Liège, yielded mixed results. On the one hand, the tetrahedron method demonstrated remarkable efficiency, offering precision identical to reference server solutions (PostGIS) with execution times in the millisecond range. On the other hand, the tests highlighted the current limitations of geoprocessing in pure JavaScript, particularly for intersection calculations. The restriction to 32-bit precision (single precision), inherent to the WebGL/Three.js ecosystem, causes significant numerical degradation when manipulating projected coordinates, compromising the results' compliance with the ISO 19107 standard. Thus, this work confirms the potential of shifting geoprocessing calculations to the client while identifying the technological hurdles, such as floating-point precision, that remain to be overcome.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

JNU : Jumeau Numérique Urbain

API : Application Programming Interface

CRS : Coordinate Reference System

CSG : Constructive Solid Geometry

BSP : Binary Space Partitioning

LoD : Level of Detail

1 Introduction

Les jumeaux numériques urbains (JNU) s'imposent aujourd'hui comme des outils technologiques majeurs, permettant de modéliser et de gérer les villes dans un environnement virtuel. Parmi les principaux constituants de ces JNU, on trouve les modèles tridimensionnels des villes, souvent définis à partir du standard CityGML ou de son homologue léger, CityJSON. Ce dernier a permis l'émergence de systèmes de bases de données exploitant nativement le format JSON pour stocker les modèles urbains dans une base de données. Cependant, bien que ces systèmes offrent une plus grande flexibilité de stockage et des performances accrues, l'absence de support des fonctions spatiales constitue l'une de leurs plus grandes faiblesses. Or, si la visualisation des objets urbains peut être gérée efficacement côté client, l'analyse spatiale reste tributaire du côté serveur.

En effet, un écart conséquent subsiste entre les écosystèmes 2D et 3D en JavaScript. Alors que des bibliothèques comme Turf.js permettent de réaliser du géotraitement 2D directement dans le navigateur, le géotraitement 3D côté client représente un vide technologique. Actuellement, aucune bibliothèque JavaScript ne permet d'effectuer des opérations complexes, telles que des calculs de volumes ou d'intersections, sur des données CityJSON sans requérir l'intervention d'un serveur.

Néanmoins, l'augmentation constante de la puissance de calcul des ordinateurs personnels, couplée au développement des moteurs JavaScript modernes, offre l'opportunité de déléguer ces traitements à la machine de l'utilisateur. Dans ce mémoire, nous proposons de combler ce vide en développant un prototype de bibliothèque JavaScript de géotraitement 3D.

Concrètement, notre approche se structure autour d'une architecture orientée objet rigoureuse conçue pour instancier et manipuler des données CityJSON indépendamment de leur niveau de détail (LoD). Ce prototype implémente des algorithmes d'analyse spatiale avancés : le calcul des volumes est assuré par deux approches complémentaires, à savoir la méthode exacte des tétraèdres pour la rapidité et la méthode de Monte-Carlo pour l'estimation par voxels. En outre, ce prototype tire parti de la géométrie de construction de solides (CSG) pour déterminer l'intersection entre objets urbains et propose une génération de zones tampons basée sur la dilatation tridimensionnelle des boîtes englobantes.

Pour valider ces développements, nous utiliserons un véritable jeu de données : le modèle 3D du quartier d'Outremeuse à Liège. L'utilisation de ce dataset, issu d'un véritable jumeau numérique, nous permettra de confronter nos résultats à des solutions de référence côté serveur (PostGIS) et d'évaluer la performance de nos algorithmes dans un contexte opérationnel.

Ce mémoire s'articule autour de cinq chapitres principaux, succédant à cette introduction. Le deuxième chapitre dresse un état de l'art des concepts fondamentaux, abordant l'architecture web, les jumeaux numériques urbains, les standards de données 3D CityGML et CityJSON ainsi que leurs systèmes de bases de données associés. Le troisième chapitre définit formellement la problématique et les objectifs de notre recherche. Le quatrième chapitre, cœur de ce travail,

détaille la méthodologie adoptée. Il présente l'architecture orientée objet de notre bibliothèque et décrit l'implémentation technique des algorithmes de calcul des volumes, des intersections et des zones tampons. Enfin, le cinquième chapitre est consacré à la validation des fonctions de notre bibliothèque à partir du modèle 3D du quartier d'Outremeuse, avant de conclure ce travail dans un sixième et dernier chapitre.

2 État de l'art

2.1 Fondamentaux du développement web : l'architecture Client-Serveur

Cette section propose un bref rappel sur la distinction client/serveur.

Le développement web est constitué de deux composantes principales : le front-end et le back-end, chacun ayant des rôles bien distincts mais interdépendants. Le front-end, également appelé côté client, concerne tout ce que l'utilisateur voit et avec quoi il interagit directement. Le back-end, ou côté serveur, gère la logique, la base de données et le serveur. Ainsi, ces deux composantes permettent à l'utilisateur d'interagir avec le serveur au travers de son navigateur web. Grâce à ce dernier, l'utilisateur envoie des requêtes au serveur (le plus souvent, il s'agit de requêtes HTTP(S)) et le serveur retourne une réponse au client après avoir effectué le traitement des données (pouvant provenir de fichiers, de bases de données, etc.) (Kornienko et al., 2021) (Oluwatosin, 2014). À titre d'illustration, nous pouvons imaginer un utilisateur qui remplit un formulaire d'inscription au travers de son navigateur (front-end) et envoie ensuite les données au serveur (back-end). Ce dernier les traite et les enregistre dans une base de données si elles sont valides. Le serveur retourne alors une réponse à l'utilisateur qui confirme ou non l'inscription de ce dernier et un message de confirmation peut ainsi être affiché dans le navigateur.

Le back-end peut être développé par l'utilisation de divers langages de programmation, dont les plus connus sont PHP, Python, Ruby, Java, etc. Ces derniers sont généralement utilisés au travers de frameworks qui permettent un développement bien plus rapide et sécurisé (Laravel et Symfony pour PHP, Django et Flask pour Python, Ruby on Rails pour Ruby, etc.) (Kornienko et al., 2021).

Le front-end est quant à lui majoritairement développé au travers de HTML, CSS et JavaScript, ainsi que de ses frameworks associés (JQuery, AngularJS, React.JS) (Xing et al., 2019). Les opérations rattachées au front-end sont exécutées directement sur la machine de l'utilisateur, ce qui requiert donc l'allocation d'une partie de la puissance de la machine de l'utilisateur au navigateur. Initialement, JavaScript était principalement utilisé pour améliorer l'interactivité des pages web en interagissant avec les éléments HTML et CSS. Ce langage n'était nullement destiné à réaliser un nombre conséquent d'opérations en raison de sa relative lenteur, comparé aux autres langages de programmation. Néanmoins, l'apparition du moteur V8 [1] développé par Google a considérablement repoussé les limites de JavaScript en lui conférant une puissance de calcul comparable aux langages compilés comme Java et C++. Dès lors, l'intégration du moteur V8 aux navigateurs a permis d'effacer la frontière entre les logiciels de bureau classiques et les applications web, ceux-ci pouvant désormais avoir des fonctionnalités similaires (Xing et al., 2019). Ainsi, des applications web de plus en plus complexes ont pu voir le jour (traitement d'images, réalisation de figures 3D, etc.) (G.-A. Nys, 2023). En outre, JavaScript est également devenu un langage serveur grâce au développement de Node.JS, permettant ainsi d'utiliser un seul langage pour développer le front-end et le back-end d'une application (Xing et al., 2019).

1. Ce moteur permet d'interpréter le JavaScript pour ensuite pouvoir le compiler.

2.2 Jumeaux numériques urbains : définition et applications

Les jumeaux numériques urbains (JNU) s'imposent aujourd'hui comme des outils technologiques majeurs, offrant de puissantes capacités pour modéliser, analyser et gérer les villes au sein d'un environnement virtuel. Malgré l'intérêt croissant pour les JNU, leur définition demeure sujette à une certaine ambiguïté, marquée par une absence de consensus terminologique (Jeddoub et al., 2023). Néanmoins, une revue exhaustive de la littérature réalisée par Jeddoub et al. (2023) permet de définir un jumeau numérique urbain comme une représentation numérique dynamique de la ville fonctionnant comme un écosystème de systèmes interconnectés, qui dépasse la simple modélisation 3D statique par l'intégration de flux de données continus entre le monde physique et virtuel.

Plus précisément, un jumeau numérique urbain vise à (Jeddoub et al., 2023) :

— Connecter les mondes réels et virtuels : il repose idéalement sur un flux de données bidirectionnel et automatique (notamment par l'utilisation de capteurs/internet des objets) ;
— Intégrer des données hétérogènes : il combine des données sémantiques, géospatiales et temporelles, permettant ainsi de centraliser des données provenant de sources disparates ;
— Servir d'outil décisionnel : il permet de réaliser des simulations, des analyses prédictives et de visualiser les phénomènes urbains pour répondre aux défis de la ville (mobilité, énergie, durabilité, etc.).

Par conséquent, un jumeau numérique urbain est bien plus qu'un simple modèle tridimensionnel ; il s'agit véritablement d'une copie numérique d'un environnement urbain qui intègre des données en temps réel et évolue avec son homologue physique (Jeddoub et al., 2023).

Néanmoins, le premier objectif reste davantage théorique que pratique. Jeddoub et al. (2023) précisent que la plupart des initiatives actuelles s'apparentent plutôt à des ombres numériques (*digital shadows en anglais*). En effet, l'automatisation complète de l'action du virtuel vers le monde réel reste encore rare à l'échelle urbaine [2].

Les applications des jumeaux numériques urbains sont multiples, allant de la simulation climatique à l'analyse du trafic. Dans ce mémoire, nous ne détaillerons pas l'ensemble de ces applications, mais nous en présenterons quelques exemples à titre d'illustration.

À Munich, un jumeau numérique urbain a permis de réaliser des simulations de croissance et de densification urbaine. Ces simulations ont notamment permis de prédire les besoins en logements et infrastructures commerciales sur 20 ans (Hijazi et al., 2022). À Zurich, un JNU a également été employé pour visualiser des scénarios de densification et planifier le développement d'immeubles de grande hauteur. En outre, Zurich exploite ce même jumeau numérique pour simuler des flux d'air froid et identifier les îlots de chaleur urbains, afin d'augmenter la résilience de la ville face au changement climatique (Schrotter & Hürzeler, 2020).

2. Dans ce mémoire, nous utiliserons souvent le terme jumeau numérique, par abus de langage, même lorsqu'une application ne respecte pas strictement la définition d'un jumeau numérique urbain, en particulier sur l'automatisation des flux de données du virtuel vers le réel.

La ville d'Herrenberg[3] s'est servie d'un JNU pour analyser le trafic et la mobilité. Des simulations ont été réalisées pour visualiser les flux de trafic (piétons, cyclistes, voitures, etc.) et tester des scénarios de réduction des embouteillages. En outre, ce JNU a permis d'étudier la pollution en intégrant des données provenant de capteurs de particules fines couplées à des simulations de flux d'air. Ce couplage permet de visualiser la dispersion de la pollution dans la géométrie urbaine (Dembski et al., 2020).

Les JNU peuvent également être utilisés pour déterminer le potentiel solaire d'une ville, comme à Helsinki (Hämäläinen, 2021). Un jumeau numérique a ainsi été utilisé pour analyser l'ombre créée par le bâti et donc estimer le potentiel solaire. En outre, toujours dans le domaine de l'énergie, il est possible d'étudier les fuites thermiques des bâtiments afin d'éduquer les propriétaires et les citoyens à adopter une consommation d'énergie plus respectueuse de l'environnement (Hämäläinen, 2021).

Vienne a développé une approche dans laquelle le jumeau numérique sert de point d'accès central aux géodonnées. Au lieu de dériver des données 3D à partir de cartes 2D, la ville inverse le processus grâce à ce JNU : le modèle 3D sémantique devient la source primaire grâce à laquelle les autres produits (cartes, modèles de terrain) sont dérivés. Cela permet une parfaite cohérence temporelle. En effet, si le modèle central est actualisé, tous les produits dérivés le sont également, éliminant les divergences entre les jeux de données (Lehner & Dorffner, 2020).

Ces quelques exemples, très loin d'être exhaustifs, illustrent toute la puissance des jumeaux numériques urbains. Bien que ces derniers combinent plusieurs sources de données, aussi bien statiques que dynamiques, nous allons nous concentrer dans ce mémoire sur les modèles 3D statiques des villes. Ceux-ci constituent un élément essentiel des JNU et sont souvent structurés à partir du modèle de données standardisé CityGML ou de son homologue léger CityJSON (Jeddoub et al., 2023). Ces standards, ainsi que les systèmes de bases de données associés, feront l'objet de la section suivante.

2.3 CityGML, CityJSON et bases de données associées

Dans la section précédente, nous avons abordé brièvement la définition et les applications des jumeaux numériques urbains. Désormais, nous allons nous concentrer uniquement sur les modèles 3D, l'un des principaux composants de ces systèmes. Dans un premier temps, nous aborderons la norme CityGML, qui permet de définir le contenu de ces modèles de manière standardisée. Nous aborderons ensuite 3DCityDB, une solution permettant le stockage et la gestion de ces modèles standardisés. Par la suite, nous détaillerons l'équivalent léger de CityGML, CityJSON ainsi que deux systèmes de bases de données se basant sur cet équivalent JSON.

3. Ville située à proximité de Stuttgart dans le sud-ouest de l'Allemagne.

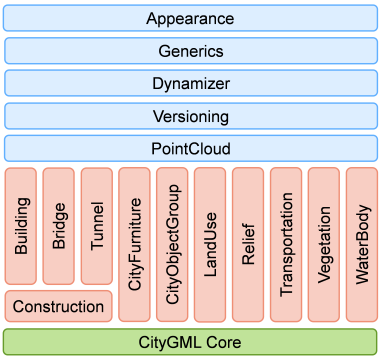


FIGURE 1 – Illustration du module central, des modules thématiques et des modules transversaux du standard CityGML (KOLBE et al., 2021).

2.3.1 CityGML

La norme internationale CityGML est une norme définie par l'Open Geospatial Consortium permettant la modélisation, le stockage et l'échange de modèles urbains 3D sémantiques (KUTZNER et al., 2020). CityGML définit des façons de modéliser la géométrie, la sémantique, la topologie et l'apparence visuelle d'objets urbains 3D que l'on rencontre couramment dans les villes (bâtiments, ponts, tunnels, routes, etc.) (OHORI et al., 2018). En outre, CityGML définit également plusieurs niveaux de détail. Ainsi, la norme CityGML permet de décrire aussi bien des objets avec très peu de détails que des objets particulièrement détaillés ressemblant à leurs homologues physiques (BILJECKI et al., 2016).

Afin de représenter efficacement l'ensemble des entités d'une ville en 3D, le standard CityGML adopte une architecture modulaire divisée en deux parties principales : un module central (CityGML Core, représenté en vert sur la figure 1) et plusieurs modules d'extension thématiques (représentés en rouge sur la figure 1). Le module central définit les concepts fondamentaux et sert de socle commun référencé par les extensions [4]. Quant aux modules thématiques, ceux-ci sont indépendants les uns des autres et couvrent des domaines d'application spécifiques. Cette modularisation a pour objectif d'optimiser le traitement des données en évitant de manipuler un modèle complet superflu et en se concentrant exclusivement sur les entités urbaines pertinentes (TAN et al., 2023). Les cinq modules transversaux représentés en bleu sur la figure 1 apportent des aspects de modélisation spécifiques (apparence, gestion des versions, etc.), utilisables conjointement avec l'ensemble des modules thématiques (KOLBE et al., 2021).

Au sein de cette architecture modulaire, CityGML a pour particularité de reposer sur une double structure interne qui distingue la géométrie de la sémantique. Concrètement, le modèle sépare l'information géométrique, définissant la représentation visuelle des objets (leur forme, taille et position), de l'information sémantique (leur fonction, leurs matériaux, etc.). Ces deux couches sont toujours interconnectées. Par exemple, si le niveau sémantique indique la présence de deux fenêtres et d'une porte sur un mur, la géométrie de ce mur doit impérativement inclure

4. En d'autres termes, les objets thématiques s'appuient sur le module central pour hériter de propriétés communes.

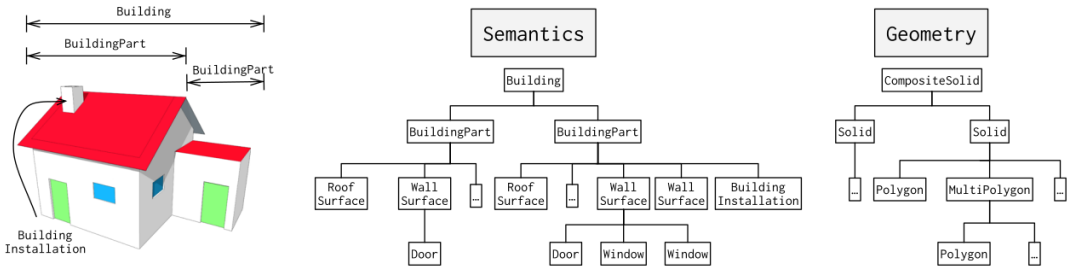


FIGURE 2 – Illustration de la double décomposition sémantique et géométrique d'un bâtiment dans le standard CityGML (LEDOUX et al., 2019).

la modélisation de ces mêmes ouvertures. Cette cohérence garantit que chaque élément modélisé correspond à une donnée précise, permettant ainsi de dépasser la simple visualisation pour effectuer des requêtes d'analyse spatiale sur la ville (TAN et al., 2023).

Cette architecture modulaire permet donc de décomposer de manière sémantique et géométrique une ville, selon une structure hiérarchique récursive, en un ensemble d'objets urbains interconnectés (LEDOUX et al., 2019). Par exemple, une ville peut être subdivisée en bâtiments qui peuvent eux-mêmes être décomposés en éléments tels que les murs, les fenêtres, les toitures, etc.

La figure 2 illustre précisément cette double décomposition pour un bâtiment. Sémantiquement, l'objet principal correspond à une entité *Building*, à laquelle est associée une géométrie de type *CompositeSolid*. Ce bâtiment peut être divisé en sous-éléments de type *BuildingPart*, représentant des volumes distincts. Ces sous-parties peuvent à leur tour contenir des surfaces (*RoofSurface*, *WallSurface*, etc.) ou des installations (*BuildingInstallation*) représentant des composants permanents du bâtiment, tels que des balcons ou des cheminées (LEDOUX et al., 2019). Des décompositions similaires sont possibles pour les deux autres classes du module *Construction* (les ponts et les tunnels).

Une des caractéristiques principales des données CityGML est leur niveau de détail (*Level of Detail* en anglais, abrégé LoD). Plus celui-ci est élevé, plus les entités numériques vont être proches de leur homologue physique (BILJECKI et al., 2016).

Initialement, il existait cinq niveaux de détail. Le niveau 0 (LoD0) correspondait simplement à l'empreinte 2D des objets. LoD1 représentait les objets par des volumes simples proches de parallélépipèdes avec des toits plats. Ces géométries pouvaient être obtenues en étirant les empreintes du modèle LoD0. Le niveau LoD2 ajoutait des détails et permettait la modélisation des objets à l'aide des différentes classes sémantiques décrites précédemment (toitures, murs...). Les toits restaient relativement simples, mais ils pouvaient être pentus. Le niveau LoD3 affinait la géométrie des bâtiments, permettant d'avoir des modèles détaillés, en incluant des éléments architecturaux tels que les balcons et les débords de toit, tandis que le niveau LoD4 permettait de représenter l'intérieur des bâtiments, y compris le mobilier (BILJECKI et al., 2016) (OHORI et al., 2018). La figure 3 illustre ces différents niveaux de détails.

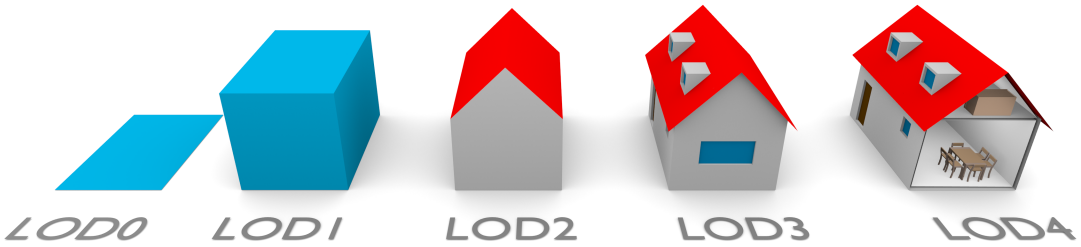


FIGURE 3 – Les cinq niveaux de détail initiaux du standard CityGML (version 2.0) (BILJECKI et al., 2016).

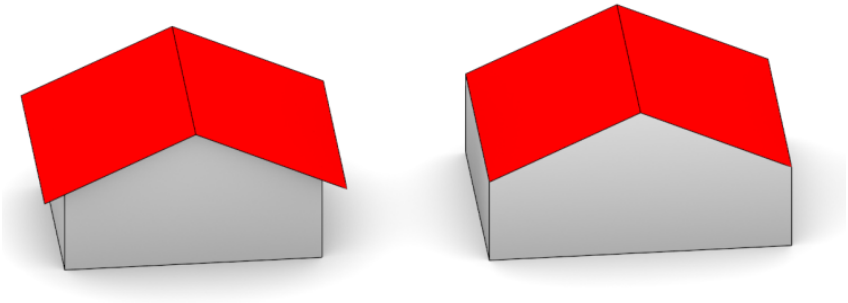


FIGURE 4 – Modèles d'un même bâtiment en LoD2 mais obtenus selon des méthodes d'acquisition différentes. Celui de gauche présente des débords de toit, tandis que celui de droite en est dépourvu (BILJECKI et al., 2016).

Cependant, cette classification en cinq niveaux de détail s'est avérée insuffisante. Il a été montré que cette classification présentait plusieurs inconvénients et lacunes, et qu'un système de classification plus précis était nécessaire. Par exemple, en fonction de la méthode d'acquisition, la figure 4 montre qu'un même bâtiment pouvait présenter ou non des débords de toit, bien qu'il soit modélisé en LoD2 dans les deux cas (BILJECKI et al., 2016). La présence ou l'absence de débords de toit peut influencer les méthodes de calcul des volumes des bâtiments ; cela sera détaillé ultérieurement dans ce mémoire. Pour pallier ces limitations, un nouveau système de classification des niveaux de détail a été proposé. BILJECKI et al. (2016) ont proposé une classification en 16 niveaux (quatre niveaux de détail affinés pour chaque niveau de LoD0 à LoD3). Ces 16 niveaux de détail sont illustrés par la figure 5. Enfin, la version 3 de CityGML a apporté une évolution supplémentaire en supprimant le LoD4 et en permettant d'intégrer la représentation intérieure des bâtiments directement dans les autres LoD (KUTZNER et al., 2020).

Initialement, CityGML ne pouvait être encodé qu'en GML (basé sur XML), ce qui rendait la manipulation et l'extraction des données particulièrement complexes (LEDOUX et al., 2019). Dans les sections suivantes, nous aborderons d'autres solutions pour stocker les données 3D respectant le standard CityGML, notamment trois systèmes de bases de données (3DCityDB, cjdb et les bases orientées documents NoSQL) et une alternative légère à CityGML : CityJSON.

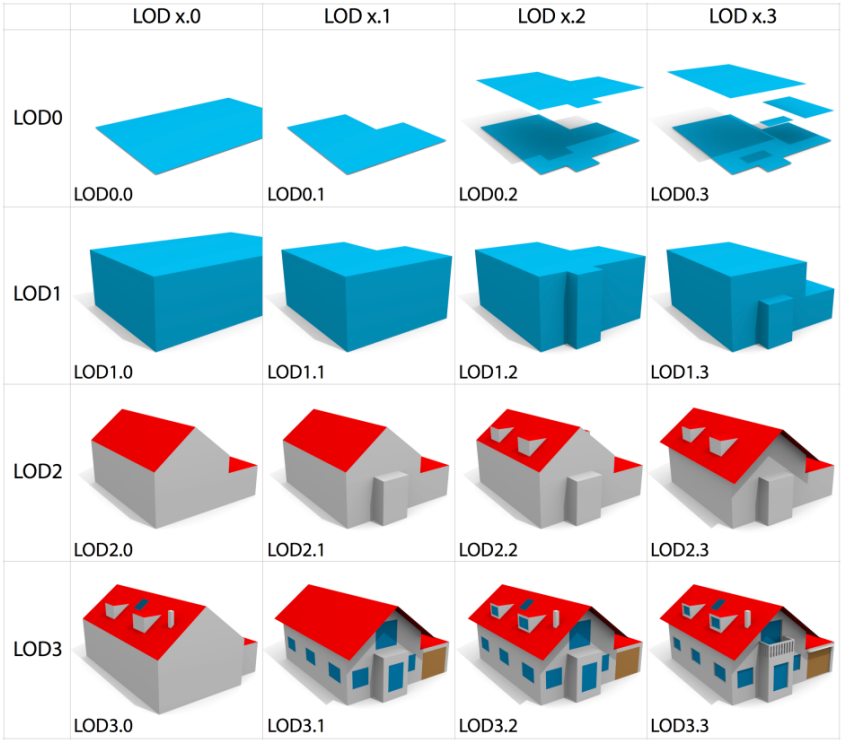


FIGURE 5 – Classification affinée des niveaux de détail en 16 grades (BILJECKI et al., 2016).

2.3.2 3DCityDB

Cette section traite d'une solution de stockage et de gestion en base de données de modèles urbains définis selon la norme CityGML.

3DCityDB est une solution de géodatabase open source conçue pour stocker, gérer, visualiser et interroger efficacement des données respectant le standard CityGML. 3DCityDB est une solution complète comprenant (YAO et al., 2018) :

— Un schéma de base de données optimisé pour PostgreSQL/PostGIS et Oracle Spatial qui permet de convertir la hiérarchie complexe de CityGML en un ensemble de tables SQL interconnectées, tout en garantissant la cohérence des données ;
— Une suite logicielle pour importer/exporter facilement des données CityGML depuis/vers une base de données ;
— Des services web pour permettre notamment la visualisation des modèles directement dans le navigateur.

Sa robustesse en fait un système de gestion de bases de données fréquemment utilisé pour le développement de jumeaux numériques urbains (KASPRZYK et al., 2024). Bien que ce système soit compatible avec Oracle Spatial, il est souvent utilisé avec l'extension spatiale PostGIS de PostgreSQL pour permettre des opérations géospatiales complexes (KASPRZYK et al., 2024).

Pour respecter fidèlement le standard CityGML tout en limitant le nombre de tables, le schéma de la base de données a été optimisé pour simplifier les structures hiérarchiques de

CityGML en regroupant certaines classes dans des tables communes. L'objectif est de réduire le nombre de jointures nécessaires lors d'une requête SQL (Yao et al., 2018). Toutefois, la complexité du modèle CityGML impose, malgré les efforts d'optimisation, un schéma relationnel dense (composé de plus de 60 tables). Ce schéma oblige à l'utilisation massive de jointures lors des requêtes SQL, même si ces dernières sont simples. (Kasprzyk et al., 2024) .

2.3.3 CityJSON

CityJSON est un encodage JSON[5] pour le modèle de données CityGML. Il a été conçu comme une alternative au format d'échange de CityGML, qui est basé sur le XML ; ce dernier pouvant être verbeux et complexe à analyser (Ledoux et al., 2019). L'utilisation du format JSON vise à rendre l'encodage compact et simple d'utilisation. En outre, le format JSON est largement utilisé dans des applications web, étant notamment le principal encodage utilisé lorsque deux applications doivent échanger des données via des API Web, loin devant le XML (Ledoux et al., 2019). Les données encodées en format JSON sont essentiellement composées de deux types de données : des objets qui sont des ensembles de paires clé/valeur[6] (appelées propriétés), représentés par des accolades, et des tableaux qui sont des ensembles ordonnés de valeurs, représentés par des crochets (Erickson, 2024).

Ainsi, des données CityJSON ne sont qu'une succession ordonnée d'objets et de tableaux qui s'imbriquent les uns dans les autres : des objets représentant des éléments urbains qui contiennent d'autres objets représentant les géométries de ces derniers, ces géométries qui englobent elles-mêmes des tableaux pour stocker les sommets des primitives géométriques, etc. Néanmoins, cette succession d'objets et de tableaux doit satisfaire certaines conventions pour respecter la structure CityJSON.

2.3.3.1 Structure des données CityJSON

Un fichier CityJSON est constitué d'un unique objet JSON racine qui encapsule l'ensemble des données. Cet objet doit impérativement contenir les propriétés suivantes (Ledoux et al., 2019) (Ledoux & Dukai, 2024) :

— *CityObjects* : objet qui contient tous les objets représentant les entités urbaines de notre modèle de ville, la clé étant l'identifiant unique de ces entités. Ces objets contiennent éventuellement la propriété *geometry* qui stocke la géométrie des objets ;
— *vertices* : tableau qui contient les coordonnées de l'ensemble des sommets, ces derniers étant indexés comme cela sera expliqué par la suite ;
— *transform* : objet qui contient 2 propriétés : *scale* (échelle) et *translate* (translation) pour appliquer une transformation linéaire sur les coordonnées des sommets afin de pouvoir les compresser pour diminuer la taille du fichier CityJSON ;
— *metadata* : contient l'ensemble des métadonnées du fichier. On y retrouve notamment la

5. *(JavaScript Object Notation)*. Le format JSON a été initialement créé spécifiquement pour JavaScript ; sa syntaxe étant presque identique à celle des objets de ce langage (Erickson, 2024).
6. Chaque clé devant être unique.

propriété *referenceSystem*, qui contient l'unique système de coordonnées du fichier ;

— *type et version* : type de données (obligatoirement CityJSON) et la version utilisée du standard, CityJSON étant un format qui évolue avec le temps.

La propriété CityObjects est le cœur des données CityJSON, car on y retrouve l'ensemble des entités urbaines, toutes stockées au premier niveau selon une hiérarchie aplatie. Contrairement au CityGML où les objets sont profondément imbriqués [7], l'ensemble des objets de la ville est stocké au même niveau. Il est ainsi possible d'accéder directement à un objet enfant grâce à son identifiant, sans devoir parcourir l'ensemble de la hiérarchie. Néanmoins, la structure hiérarchique originelle n'est pas perdue ; cette dernière est préservée via les propriétés *parents* et *children* des objets stockés dans la propriété CityObjects [8]. Ces propriétés stockent l'identifiant de chaque objet urbain parent et enfant, respectivement. De la sorte, en partant d'un quelconque CityObject, ces propriétés nous permettent de traverser l'ensemble de la hiérarchie (Ledoux et al., 2019) (Ledoux & Dukai, 2024).

2.3.3.2 Gestion des coordonnées des sommets

Une caractéristique importante du format CityJSON est l'indexation des sommets. Les coordonnées géométriques (x,y,z) de tous les objets sont regroupées dans la liste unique de la propriété *vertices* de l'objet racine. En conséquence, les objets contenant la géométrie des entités urbaines (propriété *geometry* d'un CityObject) ne stockent pas directement les coordonnées des sommets, mais plutôt un indice référençant les sommets par leur position dans cette liste globale. Cette approche présente deux avantages majeurs. Premièrement, cela favorise une compression significative des fichiers. Il est en effet courant que plusieurs faces d'un même objet se partagent un sommet commun. Cette convention permet donc d'éviter une redondance du stockage des coordonnées, celles-ci ne seront stockées qu'une seule fois. Deuxièmement, cela permet de conserver la topologie : si deux surfaces partagent un même sommet, alors elles font référence au même indice. Ce mécanisme permet d'éviter les problèmes de discontinuité [9] (Ledoux et al., 2019).

2.3.3.3 Géométrie des objets urbains

La géométrie des entités urbaines est contenue dans la propriété *geometry* des objets urbains. Cette propriété se présente sous la forme d'une liste contenant différents objets, chacun représentant une primitive géométrique [10]. Chaque élément de la liste intègre comme propriétés fondamentales le type de la primitive (MultiSurface, Solid ...), le niveau de détail noté *lod* qui fait directement référence aux niveaux de détail du standard CityGML et la propriété *bounda-*

7. Par exemple, pour accéder à un objet de type *BuildingPart*, il est d'abord nécessaire de traverser l'objet parent *Building*.

8. Ultérieurement dans ce mémoire, nous désignerons ces objets représentant les entités par les termes 'CityObject' ou 'objet urbain'.

9. Si les coordonnées des sommets étaient répétées à chaque fois au lieu d'utiliser un indice, des erreurs numériques pourraient créer de légères différences entre les occurrences des coordonnées d'un même sommet. Ces petits écarts, même s'ils sont d'un ordre de grandeur millimétrique, peuvent, par exemple, créer des trous dans un solide, un type de géométrie qui doit obligatoirement être fermé.

10. Ainsi, plusieurs géométries peuvent être associées à un CityObject.

ries. Cette propriété stocke l'implémentation concrète de la primitive. En effet, elle contient les indices des sommets selon une hiérarchie qui varie en fonction de la complexité de la primitive géométrique (Ledoux et al., 2019).

Les trois types de primitives géométriques les plus complexes (MultiSurface, Solid et MultiSolid) sont ici détaillés.

Le MultiSurface est une primitive qui représente un ensemble non ordonné de surfaces dans un espace 3D (Ledoux & Dukai, 2024). Un MultiSurface est défini à partir d'une liste d'anneaux linéaires. Le premier anneau correspond toujours à la limite extérieure de la surface, tandis que les anneaux suivants décrivent des limites intérieures pour permettre la modélisation de trous dans la surface (Ledoux et al., 2019). Un exemple d'encodage d'un MultiSurface est montré ci-dessous.

```
{
  "type": "MultiSurface",
  "lod": 2,
  "boundaries": [ // liste des surfaces
    [ [0, 1, 2, 3] ],              // Surface 1 : un quadrilatère simple
    [ [4, 5, 6, 7], [8, 9, 10] ]   // Surface 2 : un polygone avec un trou (le 2ème anneau)
  ]
}
```

Le Solide (*Solid*) représente un objet fermé dont on peut calculer le volume. Un solide doit obligatoirement être étanche[11]. Un solide est toujours défini à partir d'une coquille extérieure qui représente un solide plein. Éventuellement, on peut modéliser des cavités dans ce solide en définissant des coquilles intérieures. La modélisation des coquilles est identique à la modélisation des MultiSurfaces. Néanmoins, bien que la modélisation soit identique, l'ensemble des coquilles doit être hermétiquement fermé, ce qui n'est pas le cas des MultiSurfaces. Un exemple d'encodage de solide est montré ci-dessous (Ledoux et al., 2019).

```
{
  "type": "Solid",
  "lod": 2,
  "boundaries": [
    [ // Début de la coquille extérieure (Exterior Shell)
      [ [0, 3, 2, 1] ],   // Face inférieure
      [ [4, 5, 6, 7] ],   // Face supérieure
      [ [0, 1, 5, 4] ],   // Face latérale 1
      [ [1, 2, 6, 5] ],   // Face latérale 2
      // ... (autres faces nécessaires pour fermer le volume)
    ],
    [ // Début d'une coquille intérieure (Interior Shell)
      [ [8, 11, 10, 9] ],   // Face inférieure
      [ [12, 13, 14, 15] ],   // Face supérieure
      // ... (autres faces nécessaires pour fermer le volume)
    ]
  ]
}
```

11. On peut remplir ce solide d'eau ; il ne doit pas y avoir de "fuites".

Enfin, le MultiSolide (*MultiSolid*) permet l'agrégation de plusieurs primitives de type *Solid* au sein d'une même entité géométrique. Il s'agit donc d'une liste de solides, contenant eux-mêmes des listes de coquilles, composées de surfaces constituées d'anneaux (Ledoux et al., 2019).

```
{
  "type": "MultiSolid",
  "lod": 2,
  "boundaries": [
    [ // Solide 1 (ex: le bâtiment principal)
      [
        [ [0, 1, 2, 3] ],
        [ [4, 5, 6, 7] ]  // ...
      ]
    ],
    [ // Solide 2 (ex: une annexe au bâtiment)
      [
        [ [10, 11, 12, 13] ],
        [ [14, 15, 16, 17] ]  // ...
      ]
    ]
  ]
}
```

2.3.3.4 Conformité à la norme ISO 19107

La définition des primitives géométriques supportées par CityJSON doit respecter des contraintes strictes héritées de la norme ISO 19107 (Ledoux, 2018). La norme ISO 19107 établit le cadre conceptuel standardisé de la géométrie et de la topologie vectorielles, fournissant les règles mathématiques nécessaires pour définir, manipuler et orienter rigoureusement des objets spatiaux complexes jusqu'à trois dimensions (ISO, s. d.).

Si une primitive géométrique ne respecte pas cette norme, elle est considérée comme invalide. Cette invalidité peut conduire à d'importantes erreurs lors de l'utilisation de fonctions de géotraitement [12]. Parmi l'ensemble des contraintes définies, les contraintes ci-dessous auront une grande importance dans ce mémoire (Ledoux, 2018) :

- Les arêtes des primitives doivent être droites et les faces doivent être planes ;
- Les objets de type *Solid* doivent être topologiquement clos (*watertight* en anglais), les polygones des coquilles doivent être connectés de manière étanche et ne doivent pas s'intersecter ;
- L'orientation des polygones des solides doit être correctement définie. Pour savoir si un polygone est correctement orienté, il suffit d'appliquer la règle de la main droite [13]. Si le vecteur normal d'un polygone de la coquille extérieure pointe vers l'extérieur de l'objet, le polygone est correctement orienté. Pour les coquilles intérieures, en suivant cette même règle, le vecteur normal doit pointer vers l'intérieur des cavités.

12. Par exemple, le calcul du volume de l'intersection de deux solides peut être erroné si les deux solides ne respectent pas les contraintes définies par la norme ISO 19107.

13. La règle de la main droite consiste à suivre l'ordre des sommets avec les doigts de la main droite ; le pouce indique la direction du vecteur normal.

2.3.3.5 Gestion de la sémantique

La gestion de la sémantique, qui est un concept clé dans le standard CityGML, est définie différemment en CityJSON par rapport à CityGML. En CityGML, chaque surface thématique (Mur/toit/...) est instanciée comme un objet XML distinct qui contient sa propre géométrie. Dans CityJSON, la sémantique est directement intégrée comme une propriété des objets *geometry* représentant les primitives. Ainsi, la géométrie n'est pas fragmentée et reste définie en un bloc unique (Ledoux et al., 2019).

Plus concrètement, la sémantique est intégrée à l'objet *geometry* via la propriété *semantics*, qui est elle-même un objet. Cet objet contient deux propriétés : *surfaces*, qui liste les différentes classes sémantiques présentes (par exemple, 'WallSurface' ou 'RoofSurface'), et *values*, qui est un tableau d'entiers. Ce dernier possède une valeur pour chaque polygone de la géométrie et permet donc d'associer chaque surface à sa classe sémantique correspondante[14]. Ceci est réalisé par un mécanisme d'indexation semblable à l'indexation des sommets, chaque classe sémantique ayant un indice associé (Ledoux et al., 2019) (Ledoux & Dukai, 2024). Pour mieux visualiser la gestion de la sémantique dans CityJSON, nous présentons ci-dessous un exemple d'encodage de la sémantique appliqué à un cube.

```
{
  "type": "Solid",
  "lod": 2,
  "boundaries": [
    [
      [[0, 3, 2, 1]], // Toit
      [[4, 5, 6, 7]], // Sol
      [[0, 1, 5, 4]], // Premier Mur
      ... autres murs
    ]
  ],
  "semantics": {
    "surfaces": [
      { "type": "RoofSurface"},  // correspond à l'indice 0
      { "type": "GroundSurface" },   // correspond à l'indice 1
      { "type": "WallSurface" }       // correspond à l'indice 2
    ],
    "values": [
      [ 0, 1, 2, 2, 2, 2 ] // Correspondance directe avec l'ordre des faces dans boundaries
    ]
  }
}
```

2.3.3.6 Gestion des CRS

Le standard CityGML permet la cohabitation de plusieurs systèmes de coordonnées ; ceux-ci peuvent différer en fonction des objets. Cependant, le format CityJSON propose une autre approche en imposant un CRS unique pour l'ensemble du fichier, déclaré explicitement dans

14. Le format CityJSON permet d'associer une classe sémantique aussi bien aux polygones de la coquille extérieure qu'à ceux des coquilles intérieures.

les métadonnées. Ce choix permet de simplifier les traitements géospatiaux en garantissant une homogénéité spatiale au sein d'un même jeu de données CityJSON (Ledoux et al., 2019).

2.3.4 cjdb

La section 2.3.2 a mis en évidence la complexité structurelle de 3DCityDB, solution qui nécessite plus de 60 tables pour gérer des objets CityGML. CityJSON a ensuite été présenté, une alternative simplifiant l'encodage d'objets urbains. L'analyse se porte désormais sur cjdb, une solution de stockage s'inscrivant dans la continuité directe de la philosophie de CityJSON. En effet, cjdb tire profit de la structure de ce format léger pour simplifier le stockage de modèles urbains en base de données. Ainsi, tandis que CityJSON vise à simplifier l'encodage des données urbaines, cjdb applique cette même logique au stockage en base de données afin de réduire le nombre de tables et, ce faisant, de simplifier et d'optimiser les requêtes (Powałka et al., 2024).

Pour atteindre cet objectif, cjdb utilise les fonctionnalités de PostgreSQL [15], en particulier son type de données jsonb. Ce type de données stocke les données JSON dans un format binaire, le rendant plus efficace pour les requêtes et manipulations (PostgreSQL, 2025) (Logto, 2024). Ce format permet à cjdb de stocker les attributs et les géométries des objets urbains directement dans leur forme JSON, conservant ainsi la structure de CityJSON. Il en résulte un allègement significatif du schéma de la base de données ; seules trois tables sont nécessaires pour stocker les objets urbains (Powałka et al., 2024). Ce modèle s'aligne sur le paradigme Simple Features : chaque ligne de la table principale correspond à un objet urbain unique dont l'ensemble des géométries est stocké dans une seule colonne. Il en est de même pour les attributs qui sont stockés dans une unique colonne (Powałka et al., 2024) [16].

Les trois tables de cjdb sont les suivantes (Powałka et al., 2024) :

— city_object : Table centrale regroupant l'ensemble des objets urbains, dont les attributs et la géométrie sont chacun encapsulés dans une colonne unique de type 'jsonb' ;
— city_object_relationships : Table assurant le respect de la hiérarchie du modèle en stockant les liens de parenté entre les différents objets [17] ;
— cj_metedata : Table stockant les métadonnées globales du fichier importé (version, CRS, paramètres de transformation des coordonnées ...) qui se trouvent initialement à la racine d'un objet CityJSON.

Toutefois, bien que l'utilisation du type jsonb allège considérablement le schéma de la base de données, elle introduit une complexité supplémentaire lors de l'analyse spatiale. En effet, les géométries n'étant plus stockées dans leur format natif PostGIS, l'utilisation des fonctions PostGIS 3D nécessite des étapes de conversion intermédiaires au sein des requêtes SQL, alourdissant

15. Contrairement à 3DCityDB, cjdb n'est pas compatible avec Oracle Spatial.
16. 3DCityDB stocke chaque polygone de la géométrie dans une ligne différente. En outre, les attributs sont également stockés dans une autre table et sont répartis sur plusieurs lignes. L'ampleur de la simplification opérée par cjdb est ainsi évidente.
17. Cette table est composée de trois colonnes : l'identifiant de la relation, une clé étrangère pointant vers l'objet urbain parent et une autre pointant vers l'objet urbain enfant.

ainsi les processus de traitement (Kasprzyk et al., 2024).

Enfin, bien que cjdb soit défini à partir de l'encodage CityJSON, cjdb a choisi une autre approche pour traiter les sommets des géométries. En effet, ceux-ci ne sont pas référencés par un index global, cjdb s'en affranchissant lors du stockage des géométries (Powałka et al., 2024). Leurs références sont remplacées par les coordonnées réelles au sein de la propriété *boundaries*. Bien que cette absence d'index augmente l'espace requis pour stocker les données, cela permet de rendre chaque objet complètement indépendant [18].

2.3.5 Bases de données NoSQL

2.3.5.1 Introduction au paradigme NoSQL

Jusqu'à présent, les solutions évoquées (3DCityDB, cjdb) pour le stockage et la gestion d'entités urbaines reposent sur une architecture relationnelle (SQL). Ces systèmes organisent les données de manière structurée sous forme de tables reliées par des relations strictes. Cela permet de garantir l'intégrité des données et d'avoir un système robuste. Néanmoins, la reconstruction d'un objet urbain nécessite souvent de multiples jointures entre les différentes tables. Cjdb a cependant montré qu'il est possible de réduire considérablement le nombre de jointures par l'utilisation du type jsonb.

Le paradigme NoSQL (*Not Only SQL*) est une alternative au modèle relationnel qui stocke les données de manière plus flexible et naturelle (MongoDB, s. d.). NoSQL s'affranchit ainsi des contraintes tabulaires strictes.

Parmi les quatre grands types de bases de données NoSQL, les bases de données orientées documents s'avèrent être les plus pertinentes pour le stockage de données au format JSON. Ce modèle stocke les informations sous forme de documents dont la structure est comparable, voire identique, aux objets JSON. Contrairement aux tuples d'une table relationnelle, chaque document contient des paires de champ et de valeur pouvant accepter des types variés (chaînes, nombres, tableaux ou d'autres objets). Cette architecture offre une grande souplesse, rendant ce modèle particulièrement adapté au stockage de données semi-structurées ou non structurées (MongoDB, s. d.).

2.3.5.2 Application du modèle orienté document aux données urbaines

Les bases de données orientées documents (MongoDB) ont été utilisées par G. Nys et Billen (2021) pour proposer une architecture qui s'affranchit complètement de la complexité structurelle héritée des bases relationnelles, allant ainsi plus loin dans la simplification que cjdb. Les auteurs ont choisi d'utiliser nativement le format CityJSON pour structurer la base de données (G. Nys & Billen, 2021).

Concrètement, cette méthode décompose le modèle urbain en collections indépendantes (CityModel, AbstractCityObject, Geometry, etc.) et adopte le paradigme "what you store is what

18. En CityJSON, chaque objet est nécessairement dépendant de la racine de l'objet via l'indexation des sommets.

you access". Cette architecture permet d'éliminer les jointures inhérentes aux structures comme 3DCityDB, offrant ainsi des performances de lecture nettement supérieures et une plus grande flexibilité pour l'ajout dynamique d'attributs (G. Nys & Billen, 2021).

Cependant, si cette approche, à l'instar de cjdb, excelle dans le stockage flexible et la restitution rapide de données, elle se heurte au support restreint de fonctions spatiales natives au sein des bases NoSQL par rapport à des bases de données relationnelles utilisant PostGIS. En effet, les fonctions géospatiales restent limitées dans les bases orientées documents, MongoDB ne permettant de faire qu'un petit nombre de requêtes spatiales (G. Nys & Billen, 2021).

2.4 Géotraitement dans le navigateur

Dans la section précédente, nous avons abordé en détail des solutions pour stocker en bases de données (côté back-end) des données CityGML et CityJSON. Nous avons mis en évidence que l'utilisation directe du format JSON dans les bases de données complique grandement l'utilisation des fonctions spatiales, notamment celles de PostGIS. Dans cette courte section, nous allons passer en revue les outils existants permettant de réaliser des opérations de géotraitement côté client (front-end).

2.4.1 Géotraitement en JavaScript

Il existe aujourd'hui un écart technologique conséquent entre les écosystèmes géospatiaux 2D et 3D en JavaScript.

Dans le domaine du géotraitement 2D, l'écosystème est mature. Des bibliothèques comme OpenLayers ou Leaflet gèrent efficacement la visualisation (Kasprzyk & Devillet, 2021), tandis que Turf.js s'impose comme une référence pour l'analyse spatiale côté client (création de zones tampons, intersections, calculs de distance) (Turf.js, s. d.).

En revanche, le géotraitement 3D côté client représente un vide technologique. Si des outils comme Cesium ou Three.js sont performants dans le rendu et l'interaction avec les modèles urbains (Rafamatanantsoa et al., 2024), il n'existe actuellement aucune bibliothèque JavaScript dédiée à l'analyse spatiale 3D (calcul de volumes, intersections, etc.).

Ainsi, si les applications web de jumeau numérique urbain peuvent se reposer sur des outils JavaScript pour offrir une visualisation fluide via le navigateur client, les tâches de géotraitement doivent toujours être déléguées à un serveur. En conséquence, cela alourdit le travail du serveur et diminue l'interactivité. Par ailleurs, l'émergence de formats comme CityJSON, stockés tels quels dans des bases de données (comme dans cjdb), complexifie l'exploitation des fonctions spatiales du serveur, mettant en évidence le besoin d'une bibliothèque de géotraitement 3D côté client.

Dans la section suivante, nous aborderons une application web de jumeau numérique urbain qui utilise des technologies décrites dans les sections précédentes.

2.5 City2Twin

City2Twin est un framework open source de jumeau numérique urbain qui permet d'intégrer, de stocker et d'afficher à la fois des modèles 3D et des données dynamiques (comme la qualité de l'air). Il se distingue comme étant la première initiative de ce type à utiliser le format CityJSON, en profitant de sa simplicité intrinsèque pour faciliter la manipulation et l'intégration de modèles 3D dans une plateforme de visualisation (Rafamatanantsoa et al., 2024).

Concrètement, City2Twin fait usage de cjdb (et par conséquent de CityJSON) pour le stockage des modèles 3D. Rafamatanantsoa et al. (2024) justifient leur choix par la légèreté et la simplicité de cjdb qui offre de meilleures performances lors des requêtes récupérant les géométries 3D. Côté données dynamiques, des données de capteurs de qualité de l'air sont récupérées par des API externes et sont gérées dans une base de données respectant le standard SensorThings API. Ces données sont ensuite couplées avec les objets 3D statiques par le concept "CityThings", qui permet de lier les bases de données statiques (cjdb) et les bases de données dynamiques (données des capteurs) [19] (Rafamatanantsoa et al., 2024).

L'architecture technique est entièrement open-source : elle utilise PostgreSQL pour les bases de données, un serveur Flask pour le back-end et le framework Giro3D (basé sur Three.js [20]) pour la visualisation 3D interactive côté client (Rafamatanantsoa et al., 2024).

La plateforme permet une visualisation web fluide sans plugin requis, l'interrogation des objets (en cliquant dessus), la gestion des couches [21], la gestion des attributs (ajouter/supprimer/modifier un attribut d'un objet) ainsi que le filtrage spatial et attributaire des données. La plateforme propose également des modules d'analyse urbaine tels que la visualisation de séries temporelles pour la qualité de l'air ou l'affichage de simulations énergétiques sur les bâtiments (Rafamatanantsoa et al., 2024).

Enfin, pour tester leur approche, Rafamatanantsoa et al. (2024) ont utilisé un jeu de données CityJSON obtenu par Ballouch et al. (2024). Ce jeu de données, représentant le quartier d'Outremeuse à Liège, a été généré à partir d'une segmentation sémantique de nuages de points LiDAR 3D. Il contient les bâtiments et la végétation de ce quartier en LoD2 ainsi que le réseau routier en LoD1 (Ballouch et al., 2024). Ce jeu de données sera également utilisé dans ce mémoire pour valider nos résultats.

19. Ainsi, un objet urbain (stocké dans cjdb) correspond à un objet sur lequel un capteur est placé.
20. Nous reviendrons sur la bibliothèque JavaScript Three.js ultérieurement dans ce mémoire.
21. On peut par exemple visualiser plusieurs fichiers CityJSON dans City2Twin.

3 Problématique et objectifs du mémoire

L'emploi de bases de données telles que cjdb ou de systèmes NoSQL (comme MongoDB) pour le stockage et la gestion des modèles urbains implique l'usage du type de données JSON (JSONB) pour enregistrer des données CityJSON. Si ce type d'architecture permet d'accélérer le traitement des requêtes de visualisation, il rend techniquement compliqué, voire impossible, l'utilisation des fonctions spatiales natives côté serveur.

En parallèle, l'augmentation constante de la puissance de calcul des ordinateurs personnels (RAM, processeur, carte graphique, etc.) couplée au développement du moteur JavaScript V8 offre l'opportunité de déléguer le géotraitement au côté client.

Néanmoins, au moment de la rédaction de ce mémoire (2025), il est impossible de déléguer le géotraitement 3D au côté client. Il n'existe en effet aucun équivalent 3D à Turf.js pour le géotraitement. Par conséquent, l'objectif de ce mémoire est de combler ce vide technologique en développant un prototype de bibliothèque JavaScript de géotraitement 3D. Ce prototype tire profit de la montée en puissance des machines clientes pour soulager les serveurs.

Plus concrètement, ce prototype devra permettre de réaliser des opérations de géotraitement côté client sur des volumes clos (solides) stockés en CityJSON (provenant directement de fichiers ou de cjdb), indépendamment de leur LoD. Ces opérations consistent à calculer le volume des objets selon deux méthodes différentes, ainsi que le volume de l'intersection de deux bâtiments en récupérant la géométrie commune. Enfin, un prototype de fonction permettant de calculer des zones tampons rudimentaires sera également implémenté.

Pour valider ces développements, nous utiliserons un jeu de données de test réel, à savoir le modèle 3D du quartier d'Outremeuse à Liège. L'utilisation de ce dataset provenant d'un véritable jumeau numérique urbain nous permettra de déterminer si cette approche est réaliste et d'identifier les éventuelles limites techniques. Enfin, nous évaluerons les performances des algorithmes.

4 Méthodologie

4.1 Choix technologiques et approche méthodologique

Afin de répondre aux objectifs fixés dans la section précédente, le développement de notre prototype repose exclusivement sur le langage JavaScript. Nous justifions ce choix par la volonté de proposer un équivalent tridimensionnel à la bibliothèque Turf.js. Ainsi, cela permet d'effectuer des opérations de géotraitement directement au sein du navigateur du client sans nécessiter d'infrastructures lourdes. En outre, le choix de JavaScript permet notamment aux développeurs de jumeaux numériques urbains d'intégrer facilement cette bibliothèque à leur projet.

Le JavaScript est un langage multi-paradigme, offrant donc au développeur plusieurs possibilités de programmer pour développer un projet : impérative, fonctionnelle et objet (Mozilla, s. d.-b). Dans ce mémoire, nous avons opté pour le paradigme objet [22], ce dernier s'imposant naturellement. Il comporte en effet plusieurs avantages. Tout d'abord, diviser notre code en différentes classes permet une meilleure lisibilité de ce dernier. De surcroît, la modularité inhérente au paradigme objet permet aux futurs développeurs de facilement poursuivre le développement de la bibliothèque.

En outre, le format CityJSON se prête particulièrement bien à ce choix de paradigme. En effet, la structure de ce format, qui part de l'objet racine représentant le modèle urbain global pour ensuite se décliner en *CityObjects* , pour enfin aboutir aux géométries et à leurs primitives, peut intuitivement être représentée par un ensemble de classes interconnectées.

Depuis sa première version sortie en 1995, JavaScript a subi plusieurs évolutions [23] qui ont transformé JavaScript en un langage de plus en plus complet. Pour pouvoir diviser notre code en classes, nous avons développé notre bibliothèque à partir de la norme ECMAScript 2015 (ES6) de JavaScript. Cette norme introduit la possibilité de définir des classes formellement. De plus, JavaScript ES6 permet de définir des modules, autorisant l'exportation et l'importation du code directement entre fichiers JavaScript [24]. Les modules permettent une structuration rigoureuse du projet, nous permettant, à l'instar de langages comme Java, de n'avoir qu'une seule classe par fichier. JavaScript ES6 permet également l'utilisation de structures de données telles que les dictionnaires (objets Map) optimisés pour la recherche [25]. Au cours du développement, nous avons privilégié l'utilisation des objets Map pour optimiser les temps de recherche. Cet aspect sera détaillé ultérieurement dans ce mémoire.

22. Plus précisément, il s'agit d'une approche fondée sur la programmation orientée objet (POO). Ce paradigme de programmation consiste à modéliser un système en définissant des classes qui servent de plans ou de modèles. À partir de ces classes, on instancie (crée) des objets qui regroupent des données (appelées attributs) et des fonctions (appelées méthodes) pour représenter des aspects précis du système (Mozilla, s. d.-f).

23. JavaScript est l'implémentation concrète du standard ECMAScript, une norme internationale qui définit officiellement la syntaxe et les fonctionnalités du langage à chaque nouvelle version. (Mozilla, s. d.-c)

24. Avant JavaScript ES6, la pratique la plus courante côté client consistait à assembler plusieurs fichiers JavaScript en les chargeant via le HTML.

25. Un objet Map est un dictionnaire particulier qui est optimisé pour la recherche. Il contient des paires clé/valeur. La connaissance de la clé permet d'obtenir rapidement la valeur associée, qui peut être un nombre, un objet, etc.

Enfin, nous avons également tiré profit des fonctionnalités plus récentes de la norme ECMAScript 2022. Cette norme introduit notamment les attributs et méthodes privées. Un attribut privé est un attribut qui peut être accédé et modifié uniquement depuis l'intérieur de la classe. Pour y accéder ou les modifier, des méthodes publiques (getters/setters) doivent être utilisées. Dans des langages orientés objet comme Java, il est courant de mettre tous les attributs en privé. Cette fonctionnalité assure une encapsulation stricte des données, garantissant dès lors l'intégrité des classes. En divisant notre code en classes et en encapsulant les données grâce aux attributs privés, notre bibliothèque s'utilise ainsi comme une API (*Application Programming Interfaces*) [26]. L'utilisateur n'a pas à se préoccuper du contenu des classes et devra juste invoquer quelques méthodes pour réaliser des opérations de géotraitement.

ECMAScript 2022 introduit en outre la méthode hasOwn() pour vérifier si un objet dispose d'un certain attribut ou non. Cette méthode sera abondamment utilisée dans ce mémoire. Elle permet par exemple de vérifier si des données CityJSON sont conformes [27] ou de vérifier si un objet urbain (CityObject) dispose d'une géométrie ou non, en vérifiant l'existence de la propriété *geometry*.

Néanmoins, l'utilisation de la méthode hasOwn() n'est pas suffisante pour traiter seule les problèmes liés aux données. Si une erreur est constatée, la bibliothèque doit prévenir l'utilisateur et éviter un arrêt brutal de l'exécution du script. En conséquence, nous avons décidé de créer plusieurs classes d'erreurs personnalisées (CityJSONError pour les problèmes généraux avec les données, IntersectionError en cas de problèmes avec le calcul des intersections, etc.). Ainsi, cela offre au développeur plusieurs choix de stratégies à adopter si une erreur est rencontrée grâce à l'utilisation des blocs try/catch/finally. De la sorte, le développeur peut choisir d'afficher un message d'erreur différent en fonction du type d'erreur rencontré ou de tenter une méthode alternative. En outre, cette gestion orientée objet des erreurs s'intègre parfaitement dans une architecture orientée objet comme celle de ce mémoire. Ce choix méthodologique de représenter toutes les erreurs par des classes s'inspire directement de Java, langage orienté objet de référence.

En développant cette bibliothèque, nous avons pensé à garantir une certaine flexibilité. En effet, la bibliothèque doit fonctionner aussi bien pour des données CityJSON provenant de la lecture de fichiers standards [28] ou encore issues de bases de données telles que cjdb [29]. Comme évoqué dans la section 2.3.4, les données provenant de cjdb sont légèrement plus fragmentées en comparaison avec les fichiers standards et l'indexation des sommets n'y est pas présente. Par conséquent, la bibliothèque a été développée pour assurer une compatibilité totale entre ces deux formats.

26. Les API sont des constructions mises à disposition dans différents langages de programmation pour permettre aux développeurs de créer plus facilement des fonctionnalités complètes (Mozilla, s. d.-a). Une API peut être comparée à une boite noire que l'utilisateur peut facilement utiliser par l'intermédiaire de quelques boutons.

27. Par exemple, cela permet de vérifier si des données CityJSON ont bien la propriété obligatoire *transform* à la racine, qui permet d'appliquer la transformation linéaire sur les coordonnées.

28. Par exemple, ces derniers peuvent être importés directement par l'utilisateur au travers d'un formulaire dans une interface client ou transmis de manière brute par le serveur sans prétraitement préalable.

29. Pour illustrer, nous pouvons imaginer que, sur demande de l'utilisateur, le serveur effectue des requêtes SQL vers une base de données cjdb, récupère ainsi les données et les transmet de manière brute au client.

4.2 Architecture et modélisation des classes

Après avoir détaillé le paradigme de programmation utilisé pour développer notre prototype et après avoir décrit les technologies utilisées, nous pouvons désormais présenter l'architecture globale de notre bibliothèque. Dans cette section, nous allons décrire l'objectif de chaque classe d'un point de vue statique. En d'autres termes, nous répondrons principalement aux questions suivantes : "Que représentent ces classes ? Quelles informations contiennent-elles ?". Dans la section suivante, nous montrerons concrètement comment toutes ces classes s'imbriquent pour importer des données CityJSON et comment le développeur peut s'en servir.

La figure 6 résume, sous la forme d'un diagramme de classes, cette architecture. Tous les attributs et méthodes des classes ne sont pas directement repris ; ceux-ci apparaîtront progressivement lorsque nous évoquerons les algorithmes de géotraitement.

4.2.1 CityModel

Nous commençons la description de notre architecture par la classe **CityModel** [30] , véritable pierre angulaire de notre bibliothèque. Elle représente la racine d'un objet CityJSON [31]. Cette classe orchestre l'ensemble des données urbaines en stockant les métadonnées globales qui se trouvent initialement à la racine des données CityJSON (système de coordonnées, version...), des références vers les différents objets urbains qui composent le modèle ainsi que le tableau contenant les coordonnées des sommets.

Pour être plus précis, cette classe a la responsabilité de stocker et de gérer les métadonnées essentielles définies à la racine d'un objet CityJSON (ou dans la table cj_metadata de cjdb). Ces métadonnées incluent les informations relatives au système de coordonnées de référence, les paramètres de transformation des coordonnées des sommets ainsi que la version de CityJSON utilisée. Cette classe ne se contente pas uniquement de les stocker, elle a aussi été définie pour vérifier si ces paramètres sont correctement définis. Par exemple, si des données CityJSON ne contiennent pas le paramètre obligatoire *transform* ou que celui-ci est mal défini, cette classe retourne une erreur de type CityJSONError. Ainsi, cette classe s'assure de l'intégrité générale des données CityJSON.

Outre la gestion des métadonnées, **CityModel** centralise l'ensemble des objets urbains [32], représentant ainsi un modèle urbain dans son ensemble. Cette centralisation s'effectue par l'utilisation, comme structure de données, d'un objet Map. Ce dernier permet d'associer chaque objet à une clé unique, optimisant ainsi l'accès à un objet urbain particulier grâce à son identifiant [33].

30. Dans cette section, toutes les classes de notre architecture seront représentées en gras. Cela permet d'éviter la confusion avec les propriétés des objets CityJSON, qui porteront souvent le même nom que nos classes.

31. Par léger abus de langage, on peut dire qu'elle représente un fichier CityJSON. Si les données proviennent de cjdb, elle contient, entre autres, les informations provenant de la table cj_metadata.

32. Pour rappel, ceux-ci correspondent soit à une ligne (tuple) de la table city_object de cjdb, soit à un objet stocké dans la propriété CityObjects de l'objet (fichier) CityJSON.

33. Pour le lecteur désireux d'approfondir les aspects algorithmiques, le choix d'un dictionnaire Map permet d'avoir une complexité algorithmique inférieure à $O(N)$ pour la recherche. Si nous avions utilisé naïvement un tableau pour stocker les différents objets urbains, la recherche d'une entité particulière impliquerait de devoir

En d'autres termes, **CityModel** est donc la porte d'accès vers l'ensemble des objets urbains.

Enfin, la classe **CityModel** conserve en mémoire l'ensemble des coordonnées des sommets après leur transformation en coordonnées réelles (via la propriété *transform*). Cette conservation des coordonnées permettra d'hydrater [34] les objets représentant les géométries en remplaçant les indices des sommets par leurs coordonnées. Si les données proviennent de cjdb, ce paramètre contient la valeur *null* [35], cjdb ne créant pas d'index sur les sommets.

4.2.2 CityObject

La classe **CityObject** représente véritablement les objets urbains, entités élémentaires d'un jumeau numérique. Cette classe modélise tout objet urbain, qu'il s'agisse d'un bâtiment, d'un arbre ou de tout autre mobilier urbain.

Conformément à la structure du CityJSON, cette classe stocke des attributs fondamentaux tels que l'identifiant unique de l'objet (qui est également utilisé pour le dictionnaire de la classe **CityModel** décrite précédemment) et le type de l'objet (Building, BuildingPart, Bridge...). En outre, cette classe permet de stocker l'ensemble des informations non géométriques associées à l'objet telles que son année de construction ou sa fonction [36] . Cela est réalisé par l'intermédiaire de l'attribut *attributes* de la classe **CityObject**, qui contient l'ensemble des données attributaires génériques.

Comme détaillé précédemment, un objet urbain peut avoir plusieurs autres objets urbains 'enfants' ainsi que plusieurs objets urbains 'parents'. Pour garantir l'intégrité des données CityJSON, il est essentiel de conserver la hiérarchie. En conséquence, la classe **CityObject** conserve des références vers les objets parents et enfants [37]. La conservation de ces références permet le parcours de l'ensemble de la hiérarchie dans les deux sens depuis n'importe quel objet, conformément au format CityJSON.

En outre, chaque instance de **CityObject** conserve une référence vers son **CityModel** associé. Cette liaison permet d'accéder aux métadonnées globales situées à la racine du modèle, telles que le système de coordonnées de référence (CRS). Comme cela sera expliqué plus tard, les fonctions d'intersection que nous avons développées permettent de calculer l'intersection entre deux objets urbains provenant de deux modèles différents [38]. La conservation d'une référence vers le **CityModel** parent permet de s'assurer que les deux objets urbains sont bien définis

parcourir le tableau, la complexité serait alors de $O(N)$. Un dictionnaire, selon l'implémentation (table de hachage/arbre de recherche), permet d'avoir une complexité en $O(1)$ ou $O(log(N))$ pour la recherche. Ce gain de temps peut devenir non négligeable si le nombre d'objets urbains devient important, en particulier si de nombreuses opérations de recherche sont effectuées.

34. En orienté objet, hydrater un objet signifie le définir en lui fournissant les valeurs requises pour permettre son utilisation.

35. En JavaScript, le mot clé *null* représente l'absence de valeur.

36. Dans cjdb, ces informations sont stockées dans la colonne *attributes* de la table city_object. Dans CityJSON, il s'agit de la propriété *attributes* des objets urbains.

37. En JavaScript, les références sont des pointeurs sécurisés. Les objets urbains ne stockent donc pas une copie des objets parents ou enfants, mais stockent véritablement un accès direct vers ces derniers.

38. Cela peut être utile si l'utilisateur utilise plus d'une base de données cjdb ou plusieurs fichiers CityJSON.

dans le même CRS.

Pour terminer, la classe **CityObject** est le point d'accès vers les géométries de l'objet, stockées à l'aide de la classe **Geometry** décrite ci-dessous. Elle contient une liste de références vers des instances de la classe **Geometry**. CityJSON permet à un objet urbain d'avoir plusieurs géométries ; l'utilisation d'une liste s'avère donc nécessaire. Ainsi, la classe **CityObject** délègue l'ensemble de la représentation spatiale à la classe **Geometry**.

4.2.3 Geometry

La classe **Geometry** est la porte d'entrée de la modélisation concrète des objets urbains. Cette classe fait le lien entre la classe **CityObject** et les classes propres aux primitives : **Solid**, **MultiSurface**.

Cette classe représente la propriété *geometry* des objets urbains (*CityObject*) dans le format CityJSON et la colonne *geometry* de la table city_object dans cjdb. Elle contient comme attributs le type de la primitive géométrique ainsi que le niveau de détail (LoD), tous deux paramètres obligatoires en CityJSON. Le troisième paramètre obligatoire de la propriété *geometry* est *boundaries*, qui contient véritablement l'implémentation de la géométrie selon une hiérarchie propre à chaque type de primitive (cf. section 2.3.3). Néanmoins, la gestion de ce dernier est déléguée aux classes **Solid** ou **MultiSurface**. Ainsi, la classe **Geometry** n'héberge pas les différents sommets de l'objet urbain, mais contient plutôt des références vers les classes spécialisées. Deux cas de figure peuvent être rencontrés :

— Si la primitive est de type MultiSurface ou CompositeSurface, **Geometry** contient une référence vers un objet **MultiSurface** stocké dans l'attribut *multiSurface* ;
— Si nous rencontrons une primitive de type Solid ou MultiSolid, nous utilisons alors l'attribut *solids*. Cet attribut est une liste contenant des références vers des objets **Solid**. Si la primitive est de type Solid, alors la liste ne contient qu'une seule référence [39].

Ce choix de déléguer la gestion de l'implémentation des géométries permet de traiter indifféremment des géométries simples ou complexes au travers d'une même interface commune. Nous aurions pu tenter de fusionner les classes représentant les primitives avec la classe **Geometry** et obtenir une grande classe regroupant toute la géométrie. Cependant, ce choix aurait complexifié la lecture du code et aurait diminué son caractère modulaire. Par exemple, si nous souhaitons calculer le volume d'un MultiSolid, il est préférable d'utiliser une méthode du type 'computeVolume()' sur chaque solide contenu dans l'attribut *solids* plutôt que de faire le calcul directement à l'intérieur de la classe **Geometry**. Si nous réfléchissons d'un point de vue orienté objet, nous n'effectuons pas de calculs de volume sur une géométrie, mais sur des solides. Cet exemple, parmi d'autres, justifie ainsi notre décision de déléguer la gestion des primitives à des classes spécialisées.

Dans CityJSON, les objets *geometry* peuvent également contenir comme quatrième propriété

39. Dans ce cas, l'attribut *multiSurface* de la classe **Geometry** contient la valeur null. Si nous sommes dans le cas d'un MultiSurface, la liste de l'attribut *solids* a une longueur nulle.

la dimension sémantique par l'intermédiaire de la propriété *semantics*. Comme expliqué dans la section 2.3.3, cette propriété, qui est un objet, contient deux propriétés distinctes : les définitions des types de surfaces et les liens entre les polygones de la géométrie et les types de surfaces. Dans le cas d'un MultiSolid, les différents types sont communs à l'ensemble des solides. En revanche, il est trivial que les solides constitutifs ne définissent pas le même tableau d'indices sémantiques. Ainsi, si la propriété *semantics* est définie, nous avons choisi de conserver la définition des types de surfaces dans la classe **Geometry**. En revanche, la gestion des indices sémantiques est déléguée aux classes représentant les différentes primitives géométriques. Nous utilisons la propriété *values* pour hydrater les objets **Solid**/**MultiSurface**[40].

4.2.4 Solid/MultiSurface

Les dernières classes principales de notre architecture sont les classes qui représentent l'implémentation concrète des géométries : **MultiSurface** et **Solid**. Ces classes constituent véritablement le cœur algorithmique de ce mémoire, les fonctions de géotraitement étant directement implémentées dans ces classes.

La classe **Solid** modélise la primitive du même nom définie dans le standard CityJSON pour représenter des volumes clos. Cette classe est abondamment utilisée, étant idéale pour représenter des objets urbains clos tels que des bâtiments. Dans CityJSON, un solide est constitué d'une coquille extérieure et de plusieurs coquilles intérieures optionnelles. Pour stocker ces coquilles, notre classe contient un attribut pour stocker la coquille extérieure et un autre attribut pour stocker les coquilles intérieures dans une liste. Si un solide n'est pas doté de coquille intérieure, la longueur de cette liste est nulle. Cette division nous permet de respecter la norme ISO 19107 et de faciliter le développement de fonctions de géotraitement. Ce point sera détaillé ultérieurement.

En outre, cette classe contient un attribut booléen qui certifie si l'ensemble des faces d'un solide est un triangle ou non. Bien qu'il semble accessoire, il s'agit d'un attribut essentiel. Dans les faits, la plupart des fonctions de géotraitement peuvent donner des résultats aberrants si les solides ne sont pas triangulés. Cet attribut agit donc comme un garde-fou et permet de signaler à la classe que le solide doit prioritairement être triangulé avant de pouvoir y appliquer une quelconque fonction de géotraitement.

Un dernier attribut essentiel est l'attribut *semanticsValues* qui permet d'associer chaque face

40. Nous avons donc appliqué la même méthodologie pour les indices sémantiques que pour la propriété *boundaries* des données CityJSON. Pour le lecteur intéressé, nous justifions ici notre choix méthodologique. Pour le comprendre, prenons comme exemple un mur d'un bâtiment urbain quelconque (un *Solid*) et associons au type "mur" la valeur 1. Maintenant, imaginons que l'utilisateur souhaite le trianguler. Ce mur, qui était initialement représenté par un seul polygone, va être divisé en un ensemble de triangles. Ainsi, le nombre de surfaces augmente et il est absolument nécessaire de dupliquer l'indice sémantique du mur. Par conséquent, l'algorithme doit ajouter autant de fois que nécessaire la valeur 1 dans le tableau des indices sémantiques. Du point de vue de l'objet **Geometry**, aucun changement n'est observé. En effet, l'objet représente toujours un *Solid* et surtout, la définition des surfaces est inchangée. Le changement n'est visible que si l'on se place du point de vue de l'objet **Solid**. Ce dernier doit juste dupliquer la valeur 1 et, de surcroît, n'a pas à se préoccuper de ce que représente cette valeur. Pour en donner du sens, nous devons placer notre point de vue depuis l'objet **Geometry**. Cet exemple nous permet de justifier pourquoi nous ne stockons pas les indices sémantiques et leurs définitions dans la même classe.

à son type de surface. Cet attribut est optionnel et contient la valeur *null* si la sémantique n'est pas définie.

La classe **MultiSurface** est en tous points semblable à la classe Solid. La seule différence notable est l'absence des attributs stockant les coquilles extérieures et intérieures. Ces derniers sont remplacés par un unique attribut stockant les surfaces.

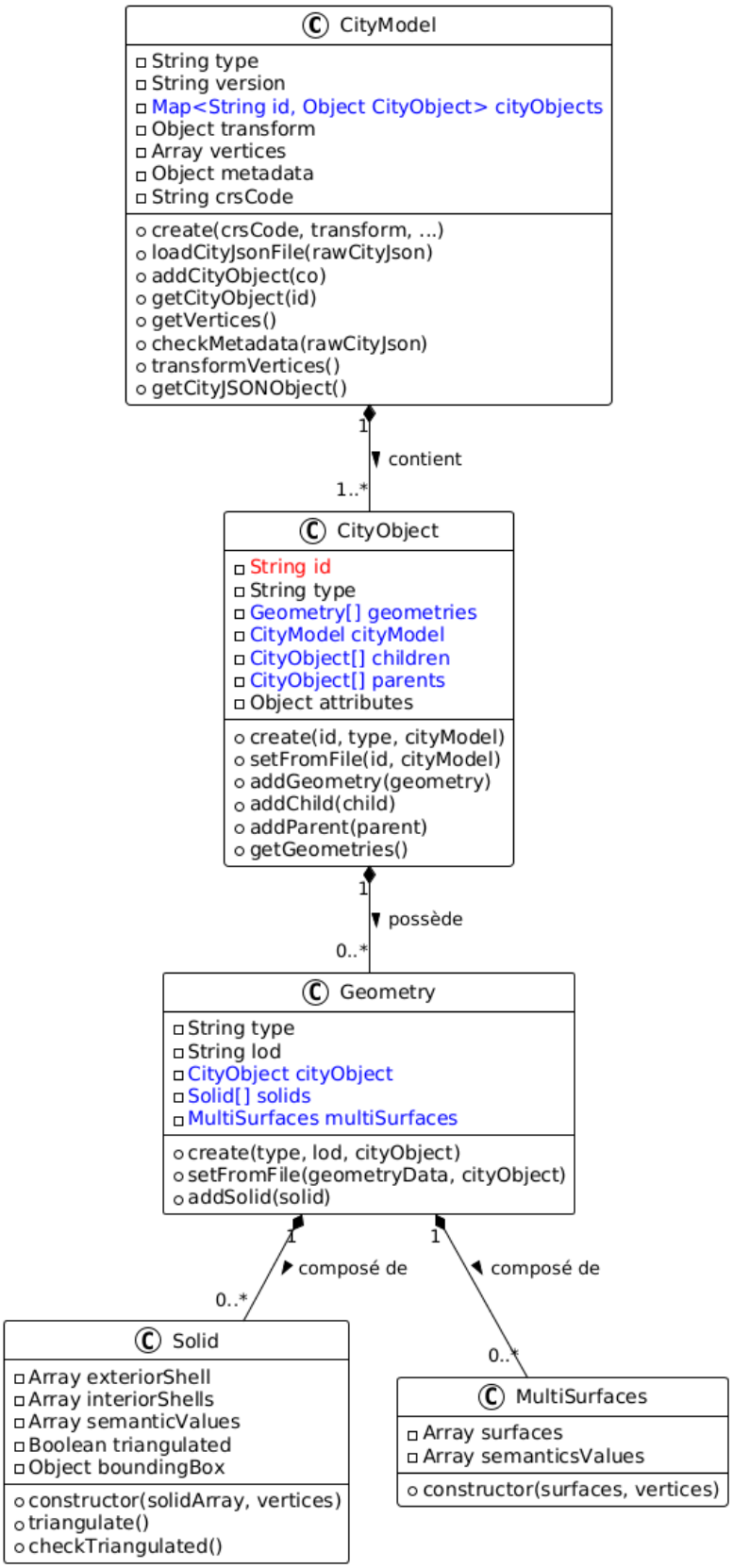


FIGURE 6 – Diagramme de classes de l'architecture de base de la bibliothèque. Ce diagramme illustre les relations entre les différentes classes. Seuls les attributs et méthodes importants pour cette section sont représentés. Les attributs contenant des références vers des instances d'autres classes sont représentés en bleu.

4.3 Processus d'importation et d'exportation de données CityJSON

4.3.1 Import de données CityJSON

Dans la section précédente, nous nous sommes intéressés à la définition statique des différentes classes de notre architecture en nous focalisant prioritairement sur leurs attributs et leur rôle. Désormais, nous allons détailler comment ces classes interagissent de manière dynamique pour charger et représenter un modèle urbain complet. L'ensemble de l'analyse réalisée ci-dessous est résumé dans le diagramme de séquence présenté à la figure 8. Cette figure permet de visualiser la chronologie des opérations, l'ordre de l'instanciation des différentes classes, etc. Les différentes étapes sont numérotées sur ce diagramme ; cela nous aidera à structurer notre analyse en y faisant référence par la suite. Ainsi, quand nous noterons *(étape X)*, cela correspond naturellement à l'étape X sur la figure 8.

Le point de départ du processus est un utilisateur qui doit manipuler un objet CityJSON provenant d'une source quelconque. Notre bibliothèque réduit considérablement le travail du développeur. Ce dernier doit juste instancier un objet **CityModel** (étape 1) et invoquer la méthode CityModel.loadCityJsonFile() [41] (étape 2). Cette fonction prend en argument les données CityJSON brutes sous la forme d'un objet [42]. Une fois cette méthode invoquée, le travail du développeur est terminé. Celle-ci va se charger d'orchestrer l'ensemble des instanciations des classes. Un exemple d'utilisation de notre bibliothèque est montré à la figure 7.

```
try {
    let cityModelImported = new CityModel().loadCityJsonFile(rawCityJSON); // on importe un objet CityJSON
    // actions qui modifient le modèle urbain ...
    let cityModelToExport = cityModelImported.getCityJSONObject(); // on récupère un objet CityJSON standard

    //On imagine une fonction qui permet de télécharger un fichier CityJSON dans le navigateur
    //JSON.stringify permet de convertir un objet en chaîne de caractères pour l'intégrer dans un fichier
    telechargerJSON(JSON.stringify(cityModelToExport));
} catch(error) {
    if(error instanceof CityJSONError) { // si une erreur s'est produite lors de l'import/export
        console.log("Une erreur s'est produite lors du traitement des données : " + error.message);
    }

    if(error instanceof IntersectionError) ...
}
```

FIGURE 7 – Exemple de code illustrant l'import et l'export de données CityJSON. La gestion des erreurs est également illustrée.

Une fois les données brutes réceptionnées, la méthode CityModel.loadCityJsonFile() initie une phase de validation générale. Pour ce faire, elle invoque la méthode interne CityModel.CheckMetadata() (étape 3). Celle-ci a la responsabilité de s'assurer que l'objet respecte bien

41. Pour éviter toute confusion, nous notons systématiquement les méthodes selon la convention suivante : classe.méthode(). Cela permet d'éviter de confondre des méthodes ayant le même nom mais provenant de classes différentes.

42. L'utilisateur doit donc s'assurer d'utiliser la méthode JSON.parse() si nécessaire. Cette méthode permet de convertir une chaine de caractères en objet exploitable par JavaScript. Il s'agit ici du seul prétraitement devant être réalisé par l'utilisateur.

la structure du standard CityJSON en vérifiant la présence des propriétés obligatoires à la racine. Cette vérification est effectuée grâce à la méthode hasOwn().

Bien qu'il ne soit officiellement pas obligatoire d'avoir un système de coordonnées de référence défini dans un objet CityJSON, nous avons décidé de rendre cette information obligatoire dans notre architecture. En effet, nous développons une bibliothèque de géotraitement ; l'absence de contexte spatial rendrait l'ensemble de nos fonctions inutilisables. Par conséquent, si le CRS n'est pas défini ou si une propriété obligatoire est absente, le processus est immédiatement interrompu et une exception de type CityJSONError est levée. Cette exception permet de signaler à l'utilisateur que les données fournies ne sont pas correctes.

Si aucun problème n'est constaté à l'étape précédente, l'étape consistant à décompresser les coordonnées peut débuter (étape 4). En effet, il est impératif de récupérer les coordonnées réelles avant d'effectuer toute opération de géotraitement. La méthode interne CityModel.transformVertices() est ainsi invoquée pour appliquer la transformation linéaire suivante sur l'ensemble des sommets du modèle :

$$\begin{aligned} x_{real} &= (x_{compressed} \times scale_x) + translate_x \\ y_{real} &= (y_{compressed} \times scale_y) + translate_y \\ z_{real} &= (z_{compressed} \times scale_z) + translate_z \end{aligned} \tag{1}$$

Une fois la décompression effectuée, les coordonnées des sommets sont stockées dans l'attribut *vertices* (cf. figure 6). Le flux d'exécution retourne ensuite à la méthode principale pour entamer la boucle qui instancie les objets urbains (étape 5). La boucle parcourt chaque entrée du dictionnaire *CityObjects* des données sources. A chaque itération, un nouvel objet de la classe **CityObject** est instancié puis hydraté via sa méthode CityObject.setFromFile() (étape 6).

Dans un premier temps, l'hydratation se concentre sur les attributs intrinsèques de l'objet : son identifiant, son type (par exemple, Building ou Tree), ainsi que ses attributs non géométriques et son extension géographique si ces derniers sont définis. En outre, nous devons nous préoccuper de la gestion de la hiérarchie. À ce stade de l'exécution, l'objet **CityObject** nouvellement instancié enregistre les identifiants des objets parents et enfants sous la forme de tableaux de chaînes de caractères contenant les identifiants de ces derniers [43]. En conséquence, l'objet ne stocke pas encore de références vers ces objets. La raison provient de la lecture séquentielle de l'objet CityJSON originel. En effet, au moment où un objet est créé, ses parents ou ses enfants peuvent ne pas encore avoir été instanciés. La définition formelle des liens hiérarchiques est donc volontairement différée à une étape ultérieure du processus.

Une fois les attributs généraux de l'objet **CityObject** définis, l'instanciation des classes représentant les géométries peut commencer (étape 7). Une boucle traite chaque géométrie définie. Pour chaque entrée, elle instancie un nouvel objet de la classe **Geometry** et invoque immédiatement la méthode Geometry.setFromFile() (étape 8). Naturellement, si l'objet urbain ne dispose

43. Cette étape est réalisée en utilisant les propriétés *children* et *parents* des objets urbains contenus dans l'objet CityJSON.

pas de géométries, aucune instanciation ne se produit.

Le flux d'exécution du code parcourt donc la méthode Geometry.setFromFile(). Dans un premier temps, cette méthode vérifie que les géométries sont correctement définies et retourne une erreur de type CityJSONError si l'une des données suivantes est manquante : le type, le LoD et la définition des primitives (propriété *boundaries*). Si aucune erreur n'est constatée, on hydrate l'objet avec le type et le LoD. La méthode a aussi pour responsabilité de vérifier si la propriété *semantics* est définie, auquel cas l'objet est hydraté avec la définition des types de surfaces.

Dans un deuxième temps, la méthode va être chargée de réaliser l'instanciation des classes représentant les primitives géométriques. La méthode va analyser le type de primitive et va instancier en conséquence la classe correspondante : **Solid** ou **MultiSurface** (étape 9 ou 12).

Lors de l'instanciation de ces classes, nous réglons le problème lié à la gestion des coordonnées des sommets. Comme évoqué précédemment, le standard CityJSON dissocie la structure topologique des données géométriques (stockées dans un tableau unique à la racine de l'objet). Or, pour réaliser un quelconque traitement géospatial, il est impératif de joindre ces deux informations[44]. Ainsi, lors de sa création, le constructeur de l'objet **Solid** (ou **MultiSurface**) reçoit en argument non seulement la structure des indices, mais également une référence vers le tableau global des coordonnées des sommets. Le constructeur remplace alors les indices par les coordonnées réelles (x,y,z). Cette approche nous permet d'encapsuler véritablement les différentes primitives, rendant ces dernières indépendantes du modèle global pour les opérations futures.

Après le remplacement des indices, le constructeur de la primitive appelle la méthode Solid.checkTriangulated(). Cette méthode parcourt l'ensemble des polygones de la primitive et s'assure qu'ils ne sont pas constitués de plus de trois sommets. Si ce n'est pas le cas, la primitive est naturellement considérée comme non triangulée. La dernière opération du constructeur consiste à hydrater l'objet avec les indices sémantiques des faces si ces derniers ont été fournis. Une fois cette étape réalisée, le flux d'exécution du code retourne à la méthode Geometry.setFromFile().

Cette méthode interroge ensuite l'objet instancié avec la méthode Solid.isTriangulated(). Si cette méthode retourne la valeur false, la méthode Solid.triangulate() est invoquée (étape 10). Nous avons choisi de trianguler immédiatement les primitives lorsqu'elles proviennent directement d'un objet CityJSON. Cela permet d'utiliser les méthodes de géotraitement par la suite.

Une fois la géométrie reconstruite dans notre architecture, le flux d'exécution remonte la chaîne d'appel : la méthode Geometry.setFromFile() se termine et redonne le contrôle à l'objet **CityObject**. Ce dernier peut également terminer son initialisation et rendre la main à la boucle

44. Bien que l'utilisation d'un index sur les sommets comporte de nombreux avantages, elle complique un peu notre architecture orientée objet. En effet, ce tableau des coordonnées lie tous les objets urbains à la racine. Ces derniers sont ainsi tous liés, et cela complique un peu leur encapsulation. Il n'aurait pas été cohérent de remplacer les indices par les coordonnées directement à partir de la classe **CityModel**. En effet, il n'y a aucun sens à effectuer une opération sur les primitives aussi haut dans l'architecture. Nous sommes donc obligés de maintenir un accès au tableau des coordonnées des sommets dans les plus basses classes de la hiérarchie.

principale du CityModel (étapes 14 et 15). L'objet urbain est alors ajouté au dictionnaire (Map) de l'attribut *cityObjects*. La boucle passe ensuite à l'itération suivante, répétant ce cycle pour chaque entrée de l'objet source.

Une fois l'ensemble des objets instanciés, la méthode CityModel.loadCityJsonFile() procède à une seconde itération à partir du dictionnaire *cityObjects*. Ce deuxième passage est nécessaire pour définir proprement les références des objets parents et enfants afin de respecter la hiérarchie.

Concrètement, pour chaque objet urbain, la méthode récupère la liste des identifiants de ses parents et enfants stockés précédemment (étapes 17 et 18). Grâce à l'attribut *cityObjects*, il est alors possible de retrouver rapidement les instances qui correspondent à ces identifiants. Les méthodes CityObject.addParent() et CityObject.addChild() sont ensuite invoquées pour hydrater les objets **CityObject** avec les références vers leurs parents/enfants.

Une fois ces étapes réalisées, l'objet CityJSON initial est finalement représenté en un ensemble d'objets. Il est désormais possible de leur appliquer des fonctions de géotraitement.

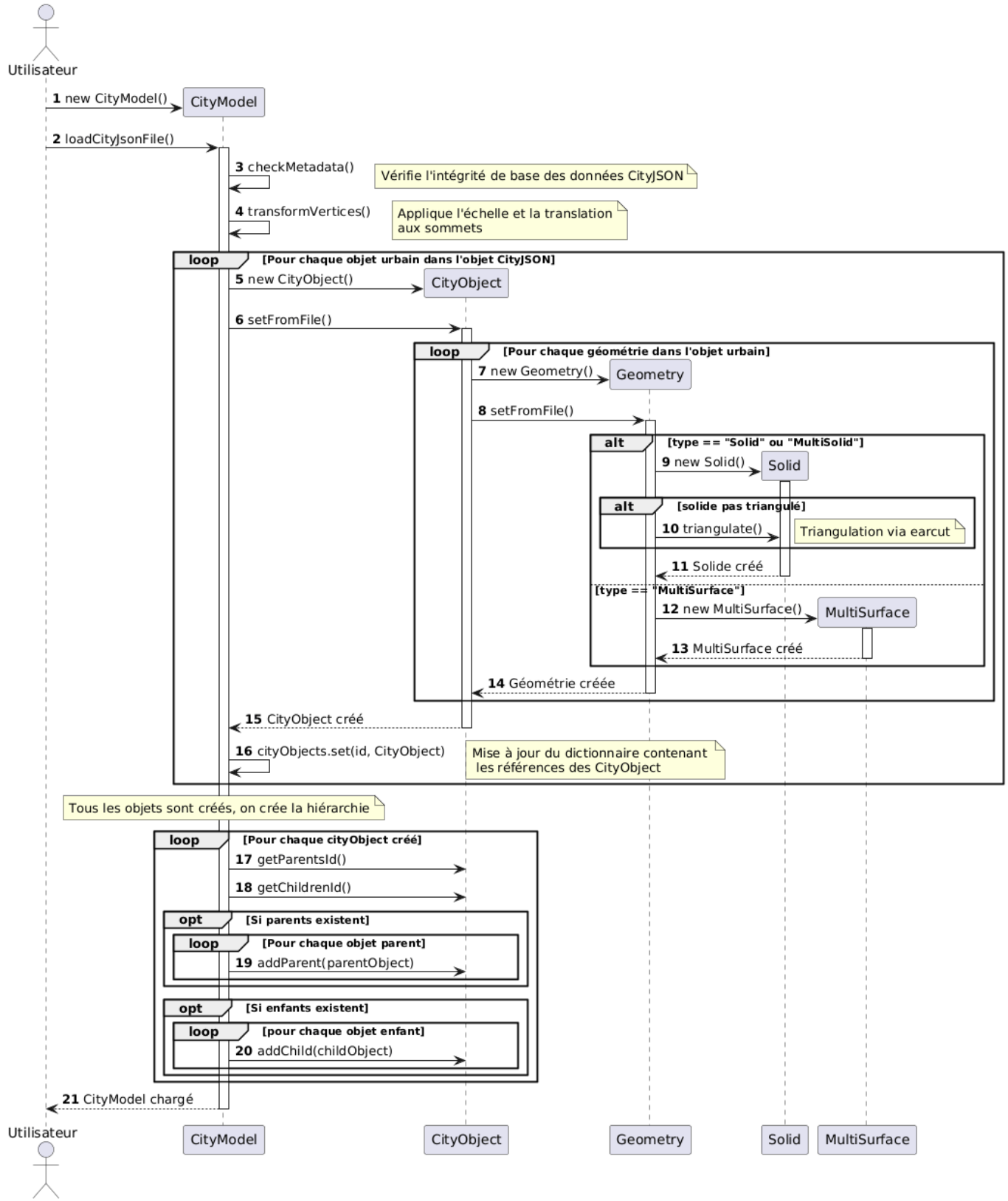


FIGURE 8 – Diagramme de séquence du processus d'importation de données CityJSON provenant directement d'un objet (fichier) CityJSON.

4.3.2 Export de données CityJSON

Dans la section précédente, nous avons détaillé comment nous traduisions un objet CityJSON en un ensemble d'objets. Néanmoins, il est essentiel de pouvoir effectuer la transformation inverse en récupérant un objet CityJSON standardisé à partir des différents objets instanciés. Cette étape est indispensable pour intégrer notre bibliothèque dans des jumeaux numériques urbains. Sans elle, il est impossible de visualiser les résultats dans un JNU tel que City2Twin ou de sauvegarder les résultats dans un fichier. Concrètement, cette étape consiste à extraire récursivement les informations contenues dans les différents objets afin de reconstruire une structure de données conforme à la syntaxe JSON.

Toutefois, cette opération n'est pas triviale et se heurte au problème de la gestion des indices des sommets. En effet, les objets représentant les primitives géométriques ne référencent pas les coordonnées des sommets à l'aide d'un index global. En outre, nous travaillons avec des modèles urbains dynamiques dont le nombre d'objets peut varier après des opérations de géotraitement. Il est aussi possible que certaines opérations modifient les objets urbains provenant des données CityJSON initiales. Dès lors, l'indexation provenant de l'objet d'origine devient obsolète et il est essentiel d'en créer une nouvelle.

L'ensemble de ce processus est réalisé par la méthode CityModel.getCityJSONObject() qui peut être invoquée par l'utilisateur. Cette fonction seule assure la conversion de l'ensemble des objets instanciés vers une structure conforme au standard CityJSON. Cette méthode commence par créer le squelette d'un objet CityJSON en définissant les propriétés fondamentales à la racine, telles que le type, la version ou les paramètres de transformation des sommets. Les métadonnées, dont le CRS, sont également traitées à cette étape. En outre, le dictionnaire principal contenant les futurs objets urbains (*CityObjects*) et la liste globale contenant les coordonnées des sommets (*vertices*) sont initialisés, ces derniers étant vides au départ.

Une fois la structure générale créée, la méthode commence à réaliser une boucle sur l'ensemble des objets **CityObject** stockés dans le modèle urbain. Pour chacun d'entre eux, on récupère, entre autres, l'identifiant, le type et les attributs non géométriques. Ces propriétés constituent la structure d'un objet urbain en CityJSON. Il convient désormais de traiter la composante géométrique. Pour récupérer l'ensemble des géométries associées à un objet **CityObject**, la méthode CityObject.getGeometries() est invoquée. Si cette méthode retourne au moins une instance de **Geometry**, la méthode Geometry.getAll() est ensuite utilisée pour récupérer les données géométriques brutes. De la sorte, nous obtenons une structure intermédiaire qui contient le type de primitive, les coordonnées explicites des sommets, la sémantique et le niveau de détail.

Dès lors, l'indexation des sommets peut commencer. Néanmoins, cette indexation soulève une question : sur quels critères pouvons-nous établir que deux sommets sont les mêmes ? Intuitivement, nous pouvons considérer que deux sommets sont identiques si la différence entre leurs trois coordonnées ne dépasse pas un certain seuil de tolérance, noté ϵ. La valeur d'epsilon doit répondre à un compromis. En effet, un ϵ trop élevé risquerait d'entraîner la fusion de sommets distincts. À l'inverse, un seuil trop strict pourrait conduire à dissocier des sommets pourtant

identiques mais jugés différents à cause du bruit numérique.

Dans notre bibliothèque, nous avons fixé $\epsilon = 0.001$. Cela correspond à une précision de l'ordre du millimètre. Cette valeur se justifie par le contexte urbain des données ; la précision millimétrique est suffisante et évite de donner du poids aux erreurs numériques. Les différents tests réalisés sur le jumeau numérique d'Outremeuse ont tous validé ce choix. Cette définition d'epsilon ne semble pas induire d'erreurs géométriques.

Sur la base de la définition du seuil de tolérance, une première méthode d'indexation naïve peut être développée. Elle repose sur la construction d'un tableau stockant les coordonnées (x,y,z) des sommets déjà traités. La position d'un élément dans le tableau détermine son index.

Concrètement, pour chaque nouveau sommet rencontré, le programme vérifie s'il correspond à un sommet déjà enregistré [45]. Cette vérification s'effectue en testant si les différences de coordonnées sont inférieures à ϵ. Soient un sommet en cours de traitement, noté *current*, et un sommet déjà indexé, noté i ; les deux sommets sont considérés comme identiques si les conditions suivantes sont réunies :

$$\begin{cases} |x_{current} - x_i| < \epsilon \\ |y_{current} - y_i| < \epsilon \\ |z_{current} - z_i| < \epsilon \end{cases} \tag{2}$$

Si le sommet i satisfait les conditions, nous récupérons directement son index. Dans le cas contraire, le sommet est considéré comme nouveau. Nous ajoutons alors son triplet de coordonnées au tableau et son index correspond alors à la taille de ce dernier.

Bien que cette approche fonctionne, un simple calcul montre que la complexité algorithmique de la méthode est de l'ordre de $O(N^2)$ [46]. Par conséquent, si le nombre de sommets tend à devenir important, cela va considérablement ralentir le processus de génération d'un objet CityJSON. Cette méthode est donc peu performante pour des modèles complexes.

Pour régler ces problèmes de performance, nous avons opté pour une autre approche algorithmique qui se fonde sur l'utilisation d'un dictionnaire via un objet Map. Nous avons déjà évoqué plusieurs fois ce type d'objet, en particulier pour le stockage des références des objets **CityObject** dans la classe **CityModel**. Pour rappel, cette structure utilise des clés pour retrouver rapidement l'objet associé à cette dernière.

Pour commencer, un objet Map commun à l'ensemble des objets **CityObject** devant être convertis en format CityJSON est créé. Ensuite, pour chaque sommet traité, une clé sous forme de chaîne de caractères est générée en concaténant ses coordonnées x, y et z. Ces coordonnées sont au préalable arrondies à un certain nombre de décimales qui correspond à la valeur ϵ choisie. Si nous choisissons $\epsilon = 0.001$, alors les coordonnées sont arrondies à trois décimales [47].

45. Ce sommet peut aussi bien venir de l'objet urbain en cours de traitement que d'un objet urbain précédemment traité.

46. Cette complexité s'explique par une boucle imbriquée dans une autre. Pour chaque sommet, on doit parcourir l'ensemble du tableau contenant les sommets déjà indexés.

47. A titre d'exemple, prenons un sommet dont les coordonnées sont (12,8749 ; 5,25042000 ; 73,87). La clé

Une fois la clé générée, on interroge l'objet Map sur l'existence de cette clé dans la structure. Dès lors, deux cas de figure se présentent :

— La clé existe déjà dans l'objet Map : cela implique que le sommet a déjà été rencontré et indexé. Nous récupérons alors directement l'index associé à ce sommet, stocké dans l'objet Map [48] ;

— La clé n'existe pas : il s'agit d'un nouveau sommet. Nous ajoutons alors la clé du sommet au Map. Nous associons à cette dernière l'index du nouveau sommet (qui correspond au nombre d'éléments dans l'objet Map). En outre, nous ajoutons ses coordonnées d'origine (non arrondies) au tableau global qui, en fin de processus, définira la propriété *vertices* de l'objet CityJSON.

Similairement à la méthode naïve, cet algorithme permet de reconstruire le tableau des sommets tout en remplaçant les coordonnées par des indices dans la définition des polygones des primitives. En évitant les boucles imbriquées, cet algorithme a une complexité algorithmique linéaire $O(N)$ où N est le nombre de sommets à indexer. Cette optimisation est particulièrement significative si le nombre d'objets urbains devient conséquent.

Une fois l'indexation des sommets terminée, l'algorithme est désormais capable d'associer une géométrie conforme au standard CityJSON aux objets urbains. Les attributs sémantiques sont aussi intégrés à la géométrie si ceux-ci sont définis.

Néanmoins, la fonction CityModel.getCityJSONObject() doit réaliser une dernière étape essentielle : la gestion de la hiérarchie. Il est en effet nécessaire de définir les propriétés *parents* et *children* des objets urbains. Pour ce faire, nous interrogeons l'objet CityObject en cours de traitement en invoquant les méthodes CityObject.getParents() et CityObject.getChildren(). Ces méthodes retournent des listes contenant les références directes vers les objets parents et enfants respectivement.

Si ces listes ne sont pas vides, la méthode parcourt chaque objet des listes et récupère son identifiant via la méthode CityObject.getId(). Ces identifiants sont ensuite stockés dans les tableaux *parents* et *children* de la structure CityJSON. La hiérarchie est ainsi bien définie.

L'ensemble de cette procédure est répété pour chaque objet **CityObject** présent dans le modèle urbain. Au terme de ce processus, nous obtenons un objet CityJSON complet et syntaxiquement valide.

4.3.3 Conclusion sur la gestion des données CityJSON

Dans cette longue section, nous avons abordé en détail comment un objet CityJSON est décomposé en différents objets. Cette section est conséquente mais est capitale pour la suite. En effet, au plus la structure de notre bibliothèque est robuste, au plus il sera simple d'y incorporer de nouvelles fonctionnalités. En outre, cette proposition d'architecture peut tout à fait être exportée

générée pour ce sommet sera "12,875|5,250|73,870"

48. Cet objet Map est constitué de paires clé$_{sommet}$ / $index_{sommet}$.

dans d'autres langages de programmation proposant une approche orientée objet dès qu'il est question de données CityJSON [49]. Comme nous le détaillerons par la suite, cette architecture nous aidera grandement à développer des fonctions de géotraitement.

Pour éviter de surcharger cette partie, nous n'avons pas évoqué le traitement des données provenant de cjdb. Leurs différences avec les objets CityJSON classiques rendent leur traitement légèrement différent. Cependant, notre architecture est suffisamment flexible pour les modéliser sans ajustements conséquents. Le lecteur intéressé par la gestion des données cjdb est invité à lire l'annexe correspondant à ce sujet (annexe 1).

49. Il est par exemple tout à fait possible de reproduire l'intégralité de notre architecture en python. Ce dernier est un langage couramment utilisé dans l'univers de la géoinformation, notamment pour développer des plugins QGIS/ArcGIS ou comme langage serveur.

4.4 Développement des fonctions de géotraitement

Dans la section précédente, nous avons détaillé l'architecture globale de notre bibliothèque en explicitant la modélisation orientée objet des données CityJSON ainsi que la répartition des rôles entre les différentes classes.

Désormais, il convient de s'intéresser à l'exploitation de cette architecture pour y développer des fonctions de géotraitement. Concrètement, cette section va répondre aux trois questions suivantes :

— Quels éléments devons-nous ajouter à cette structure pour permettre des opérations géométriques complexes ?
— Comment allons-nous implémenter les algorithmes requis pour le calcul des volumes, des intersections ou encore des zones tampons (buffer) ?
— Enfin, de quels outils techniques avons-nous besoin pour atteindre ces objectifs ?

Dans un premier temps, nous allons détailler les trois outils utilisés dans ce mémoire : three.js, earcut et three-csg-ts. Ces trois outils sont des bibliothèques JavaScript qui vont nous simplifier le développement de fonctions de géotraitement. En parallèle, nous évoquerons quelles modifications nous devons apporter à notre structure originelle pour intégrer ces outils. Plus particulièrement, nous détaillerons deux attributs et deux méthodes supplémentaires de la classe Solid qui sont directement liés aux outils utilisés. Dans un deuxième temps, nous détaillerons deux algorithmes pour calculer des volumes, un algorithme pour calculer des intersections entre des solides (donc uniquement entre des volumes clos) et un algorithme qui génère des prototypes de buffer.

Le diagramme de classes de notre bibliothèque, mis à jour en prenant en compte les attributs, méthodes et classes liés aux aspects spatiaux, est montré à la figure 9.

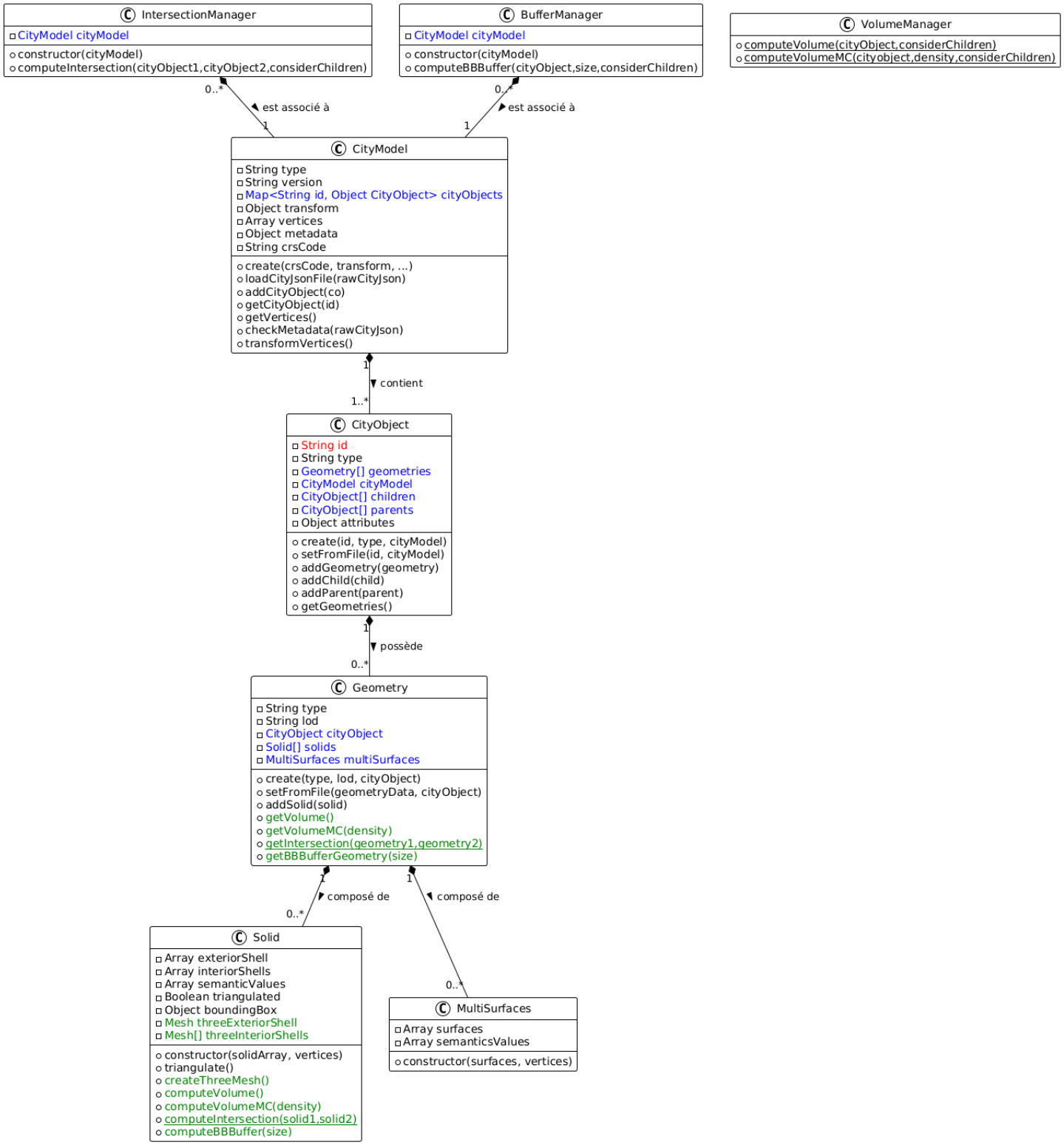


FIGURE 9 – Diagramme de classes de l'architecture complète de la bibliothèque. Les attributs et méthodes des classes principales relatifs aux fonctions de géotraitement sont repris en vert. Les attributs contenant des références vers des instances d'autres classes sont représentés en bleu. Les méthodes statiques sont soulignées. Tous les attributs et méthodes des différentes classes ne sont pas représentés.

4.4.1 Description des outils utilisés

Dans ce mémoire, nous avons utilisé trois bibliothèques différentes que nous avons intégrées à notre architecture. Ces trois bibliothèques sont :

— Three.js : bibliothèque développée pour le rendu visuel de géométries en 3D. Elle dispose d'un moteur de calcul géométrique et physique ;
— earcut : bibliothèque qui permet de trianguler des polygones avec un algorithme robuste et rapide permettant une triangulation en temps réel dans le navigateur ;
— three-csg-ts : bibliothèque qui permet d'effectuer des opérations booléennes (union, soustraction, intersection) sur des géométries 3D dans l'environnement Three.js.

4.4.1.1 Three.js

Avant de définir Three.js, il est nécessaire de s'intéresser à une technologie essentielle pour la gestion de graphismes 3D dans les navigateurs : WebGL. Le WebGL est une API JavaScript standardisée qui permet de générer des graphismes 2D et 3D interactifs directement dans les navigateurs (Mozilla, s. d.-e). Contrairement à d'anciennes technologies comme Flash, cette API ne nécessite l'installation d'aucun plugin en s'appuyant sur la balise HTML <canvas> (Mozilla, s. d.-e). Le WebGL permet de faire le pont entre la machine de l'utilisateur et le navigateur en permettant au JavaScript de manipuler directement la carte graphique pour dessiner des scènes 3D complexes (Mozilla, s. d.-e). En outre, cet accès à la carte graphique permet de générer des graphismes de haute performance. Néanmoins, programmer en WebGL peut devenir très complexe et demande aux développeurs d'avoir des connaissances pointues. Cette complexité d'utilisation a mené au développement de la bibliothèque Three.js.

Three.js est une bibliothèque JavaScript open source qui permet de créer bien plus simplement des scènes 3D dans le navigateur [50]. Cette bibliothèque agit comme une surcouche d'abstraction autour du standard WebGL. Ainsi, Three.js simplifie l'accès à la technologie WebGL et permet de développer des scènes et des applications 3D sans obliger le développeur à apprendre le fonctionnement interne de WebGL (Dirksen, 2014).

Dans notre bibliothèque, nous n'utilisons pas Three.js pour le rendu visuel, mais plutôt comme un moteur de calcul géométrique et physique. Concrètement, nous convertissons les solides (avec leurs coquilles internes et externes) issus de données CityJSON en objets compréhensibles par Three.js appelés "Mesh" (maillage). Dans l'architecture Three.js, un Mesh est l'objet qui incarne concrètement la forme d'un objet dans l'espace. [51]. Pour créer un objet Mesh, il est impératif que l'ensemble des surfaces soit triangulé, Three.js ne pouvant travailler qu'avec des triangles.

La création d'un tel objet nécessite la création au préalable d'un objet BufferGeometry qui enregistre les coordonnées (x,y,z) des différents sommets de la géométrie. Cet enregistrement se

50. Disponible à l'adresse : https://threejs.org/.
51. Three.js permet donc de visualiser dans le navigateur uniquement les différents objets Mesh définis par l'utilisateur. Cette étape de conversion en objet Mesh est obligatoire.

fait par l'utilisation de nombres à virgule flottante codés sur 32 bits (Float32), offrant dès lors une précision limitée (6 à 9 chiffres significatifs). Bien que cela semble anecdotique, l'utilisation du type Float32 par Three.js affectera considérablement les résultats de nos méthodes d'intersection. Ce point sera détaillé ultérieurement.

Ces objets Mesh sont capitaux pour la suite de notre mémoire. Nous avons donc doté notre classe Solid de deux attributs supplémentaires : threeExteriorShell et threeInteriorShells (cf. figure 9). Ces deux attributs stockent respectivement les objets Mesh pour la coquille extérieure et ceux des éventuelles coquilles intérieures. La définition de ces attributs se fait en invoquant la méthode Solid.createThreeMesh(), qui, à partir de la définition des géométries enregistrées dans les attributs exteriorShell et interiorShells, crée les maillages Three.js. La figure 9 montre le diagramme de classes actualisé en considérant l'utilisation des attributs et des méthodes liés à Three.js.

La création de ces objets nous permet d'utiliser l'ensemble des fonctions définies par Three.js sur les géométries provenant de données CityJSON. Cela va de la simple manipulation de vecteurs à des fonctionnalités plus poussées comme le "raycasting"[52].

4.4.1.2 earcut

Le second outil de notre architecture est la bibliothèque earcut[53]. Cette bibliothèque est spécifiquement dédiée à la triangulation de polygones directement dans le navigateur. Elle utilise un algorithme robuste et rapide pour permettre la triangulation en temps réel.

L'utilisation de cet outil répond à deux objectifs. Pour commencer, elle permet de trianguler des objets urbains provenant d'objets CityJSON afin que ces derniers puissent être convertis en objets Mesh. En effet, rien ne garantit que l'utilisateur va fournir des données déjà triangulées. Ensuite, earcut peut traiter des surfaces complexes comportant des trous. Cette fonctionnalité est cruciale, le standard CityJSON autorisant la définition de surfaces avec des anneaux intérieurs.

Néanmoins, l'utilisation d'earcut n'est pas triviale quand il s'agit de données définies en trois dimensions. Cette bibliothèque accepte en entrée uniquement des coordonnées en deux dimensions, alors que nos objets urbains sont définis dans un espace à trois dimensions. Pour pallier ce problème, on pourrait naïvement projeter les points constitutifs d'une surface sur un plan arbitraire tel que, par exemple, le plan XY. Néanmoins, en choisissant un tel plan, les surfaces verticales comme les murs seraient écrasées après la projection, rendant leur triangulation impossible[54]. Une projection plus rigoureuse est donc nécessaire.

Pour réaliser une triangulation rigoureuse des polygones, nous avons implémenté la méthode Solid.triangulate() au sein de la classe Solid. L'algorithme développé repose sur les étapes suivantes :

52. Le raycasting consiste à projeter une ligne infinie (qui peut être assimilée à un laser) depuis un point d'origine vers une direction précise pour voir ce qu'elle traverse. Cette technique sera utilisée lors du calcul de volumes.
53. Disponible à l'adresse : https://github.com/mapbox/earcut.
54. À l'inverse, si l'on projette sur XZ ou YZ, ce sont les surfaces horizontales qui seront écrasées.

1. Définition d'un repère local : Dans un premier temps, nous calculons le vecteur normal de la face que nous devons trianguler. Nous choisissons ensuite un vecteur $\vec{u}$ quelconque appartenant au plan de cette surface [55]. À partir de ce dernier et du vecteur normal, nous pouvons calculer un autre vecteur $\vec{v}$ contenu dans le plan de la surface, mais linéairement indépendant de $\vec{u}$. En normalisant ces deux vecteurs, nous obtenons ainsi une base orthonormée locale (u,v) alignée sur la surface du polygone ;
2. Projection : Chaque sommet 3D de la face est projeté sur ce plan local. Nous obtenons de la sorte un ensemble de coordonnées 2D ;
3. Triangulation : Nous transmettons à earcut cette liste de coordonnées 2D. En retour, earcut nous fournit une liste d'indices décrivant les triangles formés ;
4. Reconstruction 3D : Nous avons conservé au préalable la relation entre les sommets projetés (2D) et les sommets originaux (3D). Grâce aux indices retournés par earcut, nous pouvons reconstruire les triangles directement avec les coordonnées 3D initiales.

Cet algorithme nous permet d'obtenir rapidement une triangulation robuste de l'ensemble du modèle urbain en préservant l'intégrité spatiale des données originelles [56].

4.4.1.3 three-csg-ts

Le dernier outil utilisé dans ce mémoire est la bibliothèque "three-csg-ts" [57]. Cette dernière permet d'utiliser des procédés issus de la géométrie de construction de solides (Constructive Solid Geometry en anglais, abrégé CSG) directement à partir des objets Mesh de Three.js.

L'objectif des opérations de CSG est de permettre de créer des formes complexes à partir de volumes élémentaires (cubes,sphères ...) via des opérations booléennes (union, intersection, différence) (Segura et al., 2013). La figure 10 illustre la construction d'un objet 3D complexe à partir d'un cube, d'une sphère et de trois cylindres. D'un point de vue algorithmique, cette bibliothèque repose sur l'utilisation d'arbres de partition binaire de l'espace (Binary Space Partitioning en anglais, abrégé BSP), une structure de données spatiale particulièrement efficace pour des traitements rapides (Segura et al., 2013).

Concrètement, cette structure divise récursivement l'espace en deux sous-régions à l'aide de plans de séparation (hyperplans). Ces hyperplans sont définis à partir des faces des solides. Chaque hyperplan crée un nœud dans l'arbre. Ces nœuds définissent une position relative : "devant" ou "derrière" le plan de coupe [58] (Segura et al., 2013). En descendant dans la hiérarchie de l'arbre jusqu'aux feuilles, l'algorithme est capable de déterminer si une portion de l'espace se situe à l'intérieur ou à l'extérieur d'un volume (Segura et al., 2013). Ainsi, pour réaliser une opération booléenne, on fusionne les arbres des deux objets considérés. Cette fusion permet notamment de déterminer le volume commun à deux objets et ainsi d'obtenir l'intersection (Segura et al., 2013).

55. Cela se fait en choisissant deux sommets de la surface assez éloignés l'un de l'autre.
56. Par exemple, un volume clos restera bien clos après triangulation.
57. Disponible à l'adresse : https://github.com/samalexander/three-csg-ts.
58. La position devant/derrière est déterminée à partir de l'orientation d'une surface, donc à partir de l'orientation de son vecteur normal.

Bien que la bibliothèque mette à disposition les trois opérateurs fondamentaux (intersection, union et soustraction), nous utiliserons surtout la fonction d'intersection pour calculer les intersections entre objets CityJSON. Par ailleurs, l'opération de soustraction sera aussi utilisée si un solide contient des coquilles intérieures ; nous y reviendrons ultérieurement.

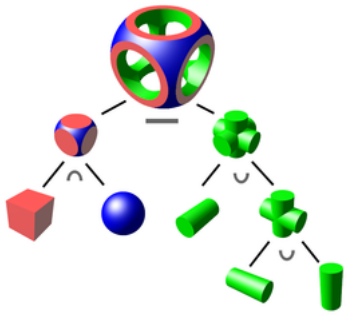


FIGURE 10 – Exemple d'opérations de CSG pour construire une forme complexe à partir d'un cube, d'une sphère et trois cylindres. Figure tirée de SEGURA et al. (2013).

4.4.2 Introduction au calcul de volumes

Après avoir défini l'architecture orientée objet de notre bibliothèque et intégré les outils tiers indispensables, nous pouvons désormais aborder le cœur de ce mémoire : le développement de fonctions de géotraitement. La première fonction que nous avons implémentée permet de calculer le volume d'objets urbains. Pour parvenir à cet objectif, nous avons développé deux méthodes :

- La méthode des tétraèdres : méthode donnant le volume avec une complexité algorithmique $O(N)$ où N est le nombre de triangles constituant le solide. Cette méthode peut s'appliquer sur des objets dont le niveau de détail (LoD) n'excède pas le LoD2.2 ;
- La méthode de Monte-Carlo : méthode se fondant sur la méthode de Monte-Carlo pour calculer des intégrales. Cette méthode utilise implicitement des voxels [59]. Elle permet de calculer le volume d'un solide indépendamment de son LoD. Cependant, cette méthode est beaucoup plus lente que la méthode des tétraèdres et elle ne permet d'obtenir qu'une approximation du volume.

4.4.3 Méthode des tétraèdres

4.4.3.1 Présentation de l'algorithme

La méthode des tétraèdres est une méthode développée par ZHANG et CHEN (2001) pour calculer le volume de solides tridimensionnels. Cette approche travaille directement sur le maillage de l'objet et ne nécessite pas de 'voxeliser' l'espace.

Le point de départ de cet algorithme consiste à décomposer un objet urbain en ses différentes surfaces. À partir de celles-ci, l'algorithme va diviser l'objet en une multitude de tétraèdres. Ces

59. Les voxels sont les équivalents 3D des pixels.

tétraèdres sont des solides élémentaires dont il est aisé de calculer le volume.

Concrètement, l'algorithme parcourt chaque triangle de l'objet et crée pour chacun d'entre eux un tétraèdre en reliant ses trois sommets à un point de référence unique. Ce point est par convention l'origine du repère (0,0,0) [60]. Une fois le tétraèdre créé, l'algorithme peut directement calculer son volume. Soit un tétraèdre défini par l'origine O et par le triangle défini par les sommets A(x_1,y_1,z_1), B(x_2,y_2,z_2) et C(x_3,y_3,z_3), le volume se calcule selon la formule suivante :

$$V_{OABC} = \frac{1}{6}\left(-x_3y_2z_1 + x_2y_3z_1 + x_3y_1z_2 - x_1y_3z_2 - x_2y_1z_3 + x_1y_2z_3\right) \tag{3}$$

Néanmoins, l'origine peut naturellement se trouver à l'extérieur de l'objet. Si l'algorithme se contente de sommer le volume des différents tétraèdres, il risque de considérer une partie de l'espace qui se trouve à l'extérieur de l'objet. Pour réellement isoler le volume de l'objet, il convient de présenter le concept de **volumes signés**.

Pour chaque tétraèdre, la méthode va attribuer un signe (+ ou -) à son volume en fonction de l'orientation de sa face par rapport à l'origine :

— Volume positif : Si le vecteur normal $\vec{N}$ de la face pointe dans la même direction que le vecteur reliant l'origine à la face $\vec{r}$[61]. Cela signifie que le produit scalaire $\vec{r} \cdot \vec{N}$ est supérieur à 0 ;

— Volume négatif : Si la normale pointe dans la direction opposée. Le produit scalaire $\vec{r} \cdot \vec{N}$ est négatif.

En sommant ces volumes signés, les volumes des parties de l'espace comprises dans les tétraèdres mais se trouvant hors de l'objet vont mathématiquement être annulés. Cette compensation permet d'isoler parfaitement le volume net contenu à l'intérieur du solide.

4.4.3.2 Exemple en deux dimensions

Pour comprendre intuitivement ce mécanisme de compensation des volumes signés, nous proposons une analogie en deux dimensions. Supposons que nous souhaitions calculer l'aire du pentagone ABCDE représenté sur la figure 11a. Nous définissons une origine O située à l'extérieur de la forme. L'équivalent 2D du tétraèdre est ici un triangle formé par l'origine et un segment du polygone.

Dans un premier temps, considérons le segment [BC]. Le vecteur normal $\vec{u}$ de ce segment pointe globalement dans la même direction que le vecteur position $\vec{OB}$. Le produit scalaire est donc positif, générant une aire signée positive représentée en jaune. Toutefois, bien que ce triangle englobe une partie du pentagone, il recouvre également une zone située à l'extérieur de l'objet. Pour isoler l'aire du pentagone, toute l'aire du quadrilatère OPEQ représenté en rouge sur la figure 11b doit être compensée.

60. Par conséquent, si les coordonnées de l'objet sont par exemple définies en Belgian Lambert 2008, alors l'origine correspond à l'origine de ce système de coordonnées de référence.

61. Plus précisément, ce vecteur peut être $\vec{OA}$, $\vec{OB}$ ou $\vec{OC}$. Le choix du vecteur n'a aucun impact sur le signe du produit scalaire.

Cette aire excédentaire sera compensée par des aires signées négatives issues d'autres segments du pentagone. Prenons l'exemple du segment [DE]. Contrairement au segment [BC], son vecteur normal $\vec{w}$ pointe ici en direction de l'origine O. Le produit scalaire est donc négatif, et l'aire signée du triangle ODE, représentée en jaune sur la figure 11c, sera négative.

Dès lors, en sommant l'aire signée des triangles construits à partir de l'ensemble des segments du pentagone, les volumes des zones situées hors du pentagone s'annulent mathématiquement. On isole ainsi bien l'aire comprise dans le pentagone.

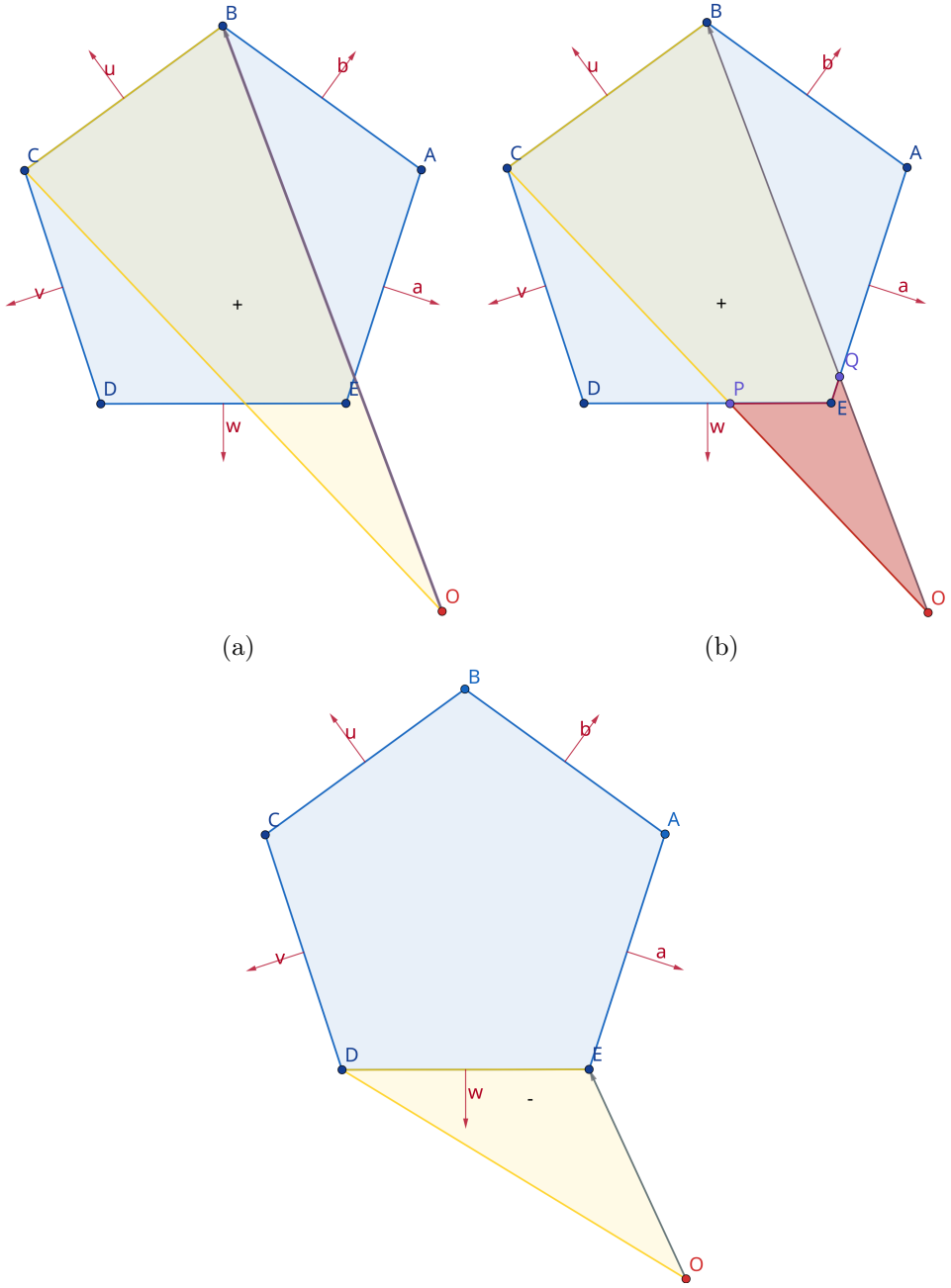


FIGURE 11 – Exemple en deux dimensions illustrant la méthode des tétraèdres.

4.4.3.3 Gestion des cavités

Comme expliqué précédemment, le format CityJSON autorise la définition de coquilles intérieures pour représenter des cavités. La méthode des tétraèdres peut facilement être adaptée pour considérer les cavités. Il suffit de modifier l'algorithme de la manière suivante :

1. Nous calculons le volume total délimité par l'enveloppe extérieure, nous obtenons ainsi le volume du solide plein ;
2. Nous calculons le volume de chaque coquille intérieure [62] ;
3. Nous soustrayons le volume des cavités du volume du solide plein pour obtenir ainsi le volume réel de l'objet urbain.

4.4.3.4 Robustesse et optimisation numérique

Bien que l'algorithme des tétraèdres soit mathématiquement exact, une mauvaise définition des objets urbains peut conduire à des aberrations lors du calcul des volumes. En effet, il est impératif que la géométrie d'un objet urbain respecte scrupuleusement la norme ISO 19107.

Si la norme n'est pas respectée et qu'une face est par conséquent mal orientée, le volume signé du tétraèdre associé sera inversé. Ce risque d'erreur est considérablement amplifié par la nature même des systèmes de coordonnées utilisés en géoinformation. Lorsque nous travaillons avec des CRS tels que le Belgian Lambert 2008, les coordonnées (X,Y) des sommets atteignent fréquemment des valeurs de l'ordre de 10^5 à 10^6 mètres. Ainsi, un tétraèdre formé par un triangle, aussi petit qu'il soit, et l'origine du repère (0,0,0) possède un volume immense dû à l'éloignement de l'origine par rapport à l'objet.

En conséquence, il suffit qu'une seule surface soit mal orientée pour engendrer un résultat complètement erroné. Pour pallier ce problème, nous avons décidé de recentrer systématiquement l'objet autour de l'origine avant d'en calculer son volume [63]. Cette opération réduit radicalement la distance entre les sommets et l'origine, minimisant ainsi le volume de chaque tétraèdre individuel. Nous limitons ainsi l'impact d'une mauvaise orientation sur le volume de l'objet.

En termes de performance, cette méthode présente une complexité algorithmique linéaire en $O(N)$ où N est le nombre de triangles. Elle est donc particulièrement adaptée au traitement de grands modèles urbains en temps réel.

4.4.3.5 Limite de cette méthode

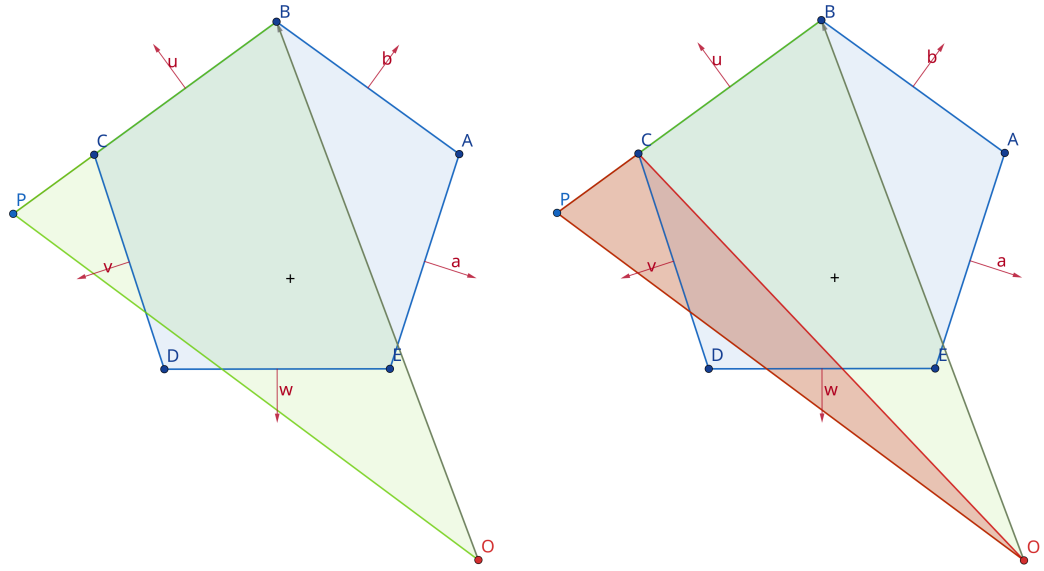
Cette méthode, bien que très efficace, ne se limite qu'aux objets géométriques strictement fermés. Si un bâtiment présente des débords de toit, le résultat de la méthode sera incorrect. En effet, CityJSON ne fait aucune distinction entre la toiture réelle et la partie qui déborde. Ces deux éléments sont représentés par le même type de surface.

62. D'après la norme ISO 19107, les surfaces des coquilles intérieures doivent être orientées vers l'intérieur. La méthode des tétraèdres requiert des surfaces orientées vers l'extérieur, nous devons donc inverser l'orientation de ces surfaces avant de procéder au calcul du volume.

63. Pour ce faire, nous calculons le centre de la boîte englobante (bounding box) C_{BB} de l'objet et nous effectuons la même translation sur chaque sommet, dont le vecteur de déplacement est déterminé à partir C_{BB}. De la sorte, nous ramenons le centre de la bounding box à l'origine (0,0,0).

Pour comprendre le problème engendré par les débords de toiture, nous pouvons reprendre notre exemple en deux dimensions. Imaginons que le segment de la toiture initial [BC] soit étendu jusqu'au point P pour créer un débord, formant ainsi le segment [BP]. Comme montré sur la figure 12a, l'algorithme va calculer l'aire signée du triangle OBP. Ce volume signé sera positif. Néanmoins, l'aire signée du triangle OPC, représentée en rouge sur la figure 12b, ne sera pas compensée. Cette aire sera donc toujours considérée à la fin de l'algorithme, entraînant une surestimation du volume calculé [64].

Si des objets urbains présentent des débords de toit, il incombe à l'utilisateur de décider d'appliquer ou non cette méthode [65]. Si l'utilisateur fait le choix de l'utiliser, il aura alors une approximation du volume de l'objet urbain. La différence avec le volume réel sera plus ou moins importante en fonction de la taille des débords de toit. Néanmoins, nous proposons une autre méthode insensible à la présence de débords de toit. Cette méthode est présentée dans la section suivante.



(a) (b)

FIGURE 12 – Illustration en deux dimensions du problème généré par la présence de débords de toit lors du calcul de volumes par de la méthode des tétraèdres.

4.4.4 Méthode de Monte-Carlo

La deuxième méthode implémentée dans notre bibliothèque pour calculer des volumes est une méthode fondée sur le calcul d'intégrales par la méthode de Monte-Carlo. En référence à son origine, nous appellerons donc cette méthode "Méthode de Monte-Carlo". Implicitement, cette méthode va faire l'usage de voxels pour discrétiser l'espace contenant un objet urbain.

64. Si l'on avait choisi une origine se situant au-dessus de la toiture, cela aurait conduit à une sous-estimation de l'aire du pentagone.

65. Cela est possible si le niveau de détail d'un objet urbain est le LoD2.3 ou supérieur au LoD3.0 (cf. section 2.3.1).

4.4.4.1 Description de l'algorithme

L'algorithme développé se décompose en quatre étapes majeures successives :

1. **Définition de la boîte englobante** : La première étape consiste à définir la boîte englobante (bounding box) de l'objet urbain à partir de ses coordonnées extrêmes (x_{min}, x_{max}, y_{min}, ...). Cette enveloppe a naturellement la forme d'un parallélépipède rectangle dont le volume V_{box} est trivial à calculer ;
2. **Discrétisation de l'espace** : L'algorithme génère ensuite une grille régulière de points à l'intérieur de cette boîte englobante. En d'autres termes, l'algorithme remplit cette boîte par une grille régulière de points de tests. Par défaut, cette grille a une résolution métrique. Cela signifie qu'un point est placé tous les mètres le long des trois axes (x,y,z). Par exemple, si la bounding box définit un parallélépipède rectangle de dimension 20m x 30m x 10m, la boite contiendra alors 20 x 30 x10 = 6000 points au total. En pratique, la résolution est légèrement supérieure à un mètre car les dimensions de la bounding box sont arrondies à l'unité supérieure pour définir le nombre de points [66]. Une grille contient au minimum 125 points, le minimum de points par coordonnées valant 5. En outre, l'utilisateur peut augmenter ou diminuer la résolution via un paramètre de densité. Il est important de noter que ce paramètre influence le temps de calcul de manière cubique. En effet, si l'utilisateur souhaite doubler la résolution (densité fixée sur 2), le nombre de points est doublé sur la longueur, mais également sur la largeur et la hauteur [67]. Le nombre de points à traiter est donc multiplié par 2^3 soit 8. En conséquence, un paramètre de densité trop élevé peut mener à avoir une grille contenant des millions de points.
3. **Test d'appartenance** : Pour chaque point de la grille, l'algorithme doit déterminer s'il se situe à l'intérieur ou à l'extérieur de l'objet urbain. Pour ce faire, les attributs de classe Solid contenant les objets Mesh de Three.js sont utilisés ainsi que les méthodes de raycasting fournies par cette même bibliothèque. Concrètement, un rayon infini est lancé depuis le point testé dans une direction arbitraire à la manière d'un laser [68]. En comptabilisant le nombre de fois que le rayon intersecte les faces de l'objet, deux conclusions sont possibles :
   — Si le nombre d'intersections est pair, le point est à l'extérieur ;
   — Si le nombre d'intersections est impair, le point est à l'intérieur.
4. **Estimation du volume** : Une fois l'ensemble de la grille parcourue, le volume de l'objet urbain V_{objet} est estimé via la formule suivante :

$$V_{objet} = \frac{N_{inside}}{N_{total}} \times V_{box} \tag{4}$$

66. Par exemple, si les dimensions de la boîte englobante sont 42,5 m x 37,2 m x 12 m, alors le nombre de points dans la boîte sera égal à 43x38x12 = 19608 points.
67. Avec une densité égale à 2, les points seront espacés d'environ 50 centimètres le long des trois axes.
68. La méthode est théoriquement indépendante de la direction du rayon. Néanmoins, il est préférable de projeter le rayon vers des surfaces idéalement plates pour contourner les éventuelles erreurs, comme des trous, comprises dans la géométrie. On obtient de meilleurs résultats en projetant les rayons vers les surfaces représentant le sol, souvent bien plates, que vers la toiture où les surfaces sont inclinées.

où N_{inside} est le nombre de points se trouvant à l'intérieur de la grille et N_{total} le nombre total de points de la grille.

4.4.4.2 Exemple en deux dimensions

Similairement à la méthode des tétraèdres, un exemple en deux dimensions permet de mieux comprendre le fonctionnement de cet algorithme. Nous reprenons l'exemple du pentagone ABCDE. Dans un premier temps, nous définissons la boîte englobante de l'objet, qui est le rectangle FGHI représenté en orange sur la figure 13a. Nous remplissons ensuite ce rectangle de points répartis uniformément. Visuellement, l'objectif de l'algorithme est de parvenir à discriminer les points situés à l'intérieur de la forme (en vert) de ceux situés à l'extérieur (en rouge).

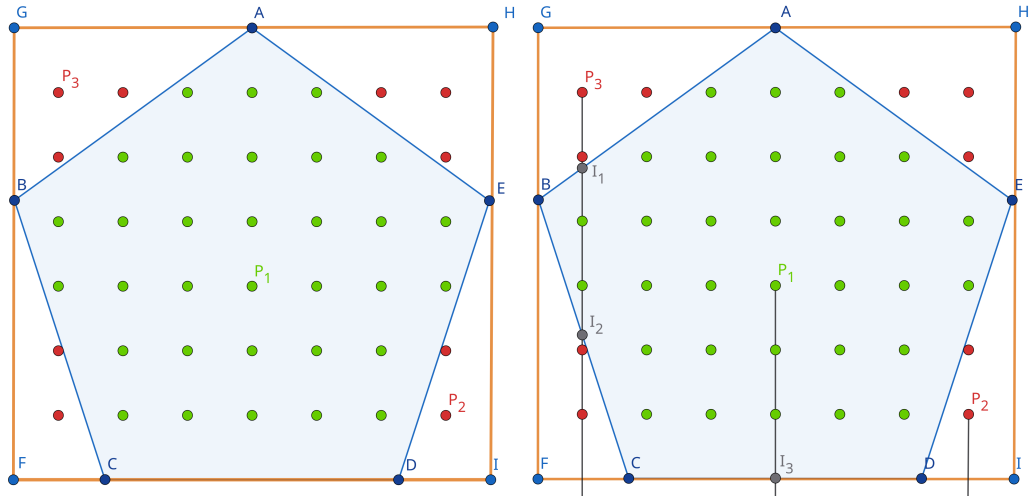
Nous illustrons sur la figure 13b la technique du raycasting. Le principe repose sur la projection d'un rayon (une demi-droite) depuis le point testé vers une direction arbitraire (dans notre exemple, c'est vers le bas) et de compter les intersections avec les segments du pentagone. Considérons trois points P_1, P_2 et P_3 représentés sur les figures 13a et 13b :

— Point P_1 : le rayon projeté intersecte le périmètre du pentagone en un unique point noté I_3 et représenté sur la figure 13b. Le nombre d'intersections étant impair, l'algorithme en déduit que le point se situe à l'intérieur de la forme ;
— Point P_2 : le rayon projeté ne rencontre aucun segment. Le nombre d'intersections est nul, le point est considéré comme à l'extérieur du polygone ;
— Point P_3 : le rayon intersecte la frontière du polygone à deux reprises aux points I_1 et I_2. Le nombre d'intersections étant pair (2), le point est logiquement exclu.

L'estimation de l'aire du pentagone est finalement obtenue en multipliant l'aire connue du rectangle englobant par la proportion de points verts sur l'ensemble des points de la grille.

4.4.4.3 Gestion des cavités

Cette méthode gère les solides contenant des coquilles intérieures. L'algorithme reste le même, mais le calcul sera beaucoup plus lourd. En effet, pour chaque point de la grille, il est nécessaire de vérifier son appartenance à la coquille extérieure et à chacune des coquilles intérieures afin de déterminer le volume de ces dernières. Leur volume sera ensuite soustrait au volume du solide plein.



(a) (b)

FIGURE 13 – Exemple en deux dimensions illustrant la méthode de Monte-Carlo.

4.4.4.4 Performance

La principale contrainte de cette approche réside dans sa performance. L'opération de raycasting pouvant être répétée des dizaines de milliers de fois, le temps d'exécution peut s'étendre de quelques secondes à plusieurs minutes pour les bâtiments imposants. En outre, plus le nombre de surfaces est important, plus les opérations de raycasting individuelles seront lentes. L'utilisateur doit judicieusement choisir la valeur de la densité pour parvenir à un compromis entre le temps de calcul et la précision du volume obtenu. Enfin, si l'on fait tendre la densité vers l'infini, on s'attend à ce que le volume obtenu par la méthode de Monte-Carlo converge vers le volume obtenu par la méthode des tétraèdres.

4.4.5 Implémentation de la méthode des tétraèdres et de la méthode de Monte-Carlo

Désormais, nous allons détailler comment nous avons implémenté ces algorithmes dans notre architecture. Dans cette section, nous parlerons uniquement de l'implémentation de la méthode des tétraèdres. L'implémentation de la méthode de Monte-Carlo est en tous points identique à la première méthode ; il convient juste d'ajouter le suffixe "MC" aux méthodes évoquées ci-dessous.

L'ensemble du processus est illustré dans un diagramme de séquence repris à la figure 15.

Pour orchestrer le calcul des volumes, nous avons ajouté une classe supplémentaire à notre architecture : VolumeManager. Cette classe n'a pas vocation à être instanciée et ne contient donc que des méthodes statiques. L'accès à l'algorithme des tétraèdres s'effectue via la méthode VolumeManager.computeVolume(). Cette méthode prend deux arguments : l'instance du CityObject représentant l'objet urbain dont on souhaite obtenir le volume et un paramètre booléen optionnel nommé considerChildren.

Ce second paramètre permet de prendre en compte les enfants de l'objet CityObject fourni à

la méthode. En effet, un objet urbain principal (comme un Building) est fréquemment composé de sous-éléments (des BuildingPart par exemple) définis comme ses enfants.

Si le paramètre considerChildren est activé (valeur true), la méthode agrège récursivement l'ensemble des descendants de l'objet dans la hiérarchie. Le volume retourné correspondra alors à la somme des volumes de l'objet parent et de tous ses enfants contenant au moins une géométrie. Dans le cas contraire (false), le calcul se limite strictement aux géométries de l'objet passé en argument. Par défaut, cet argument est réglé sur true [69].

Une fois la liste des objets à traiter définie à partir du paramètre considerChildren, la méthode VolumeManager.computeVolume() va déléguer tout l'aspect algorithmique aux autres classes. En effet, cette dernière va invoquer la méthode Geometry.getVolume().

Le flux d'exécution passe donc dans la méthode Geometry.getVolume(), qui vérifie dans un premier temps que l'algorithme est applicable à la géométrie de l'objet. Si la primitive n'est pas un Solid, MultiSolid ou CompositeSolid, la méthode retourne 0. Si le type de primitive est valide, la méthode itère sur l'ensemble des solides constitutifs de la géométrie et invoque pour chacun d'eux la méthode Solid.computeVolume(). C'est véritablement dans celle-ci qu'est implémenté l'algorithme des tétraèdres.

Une fois le volume des différents solides calculé, la classe Geometry additionne leur volume respectif. Une fois cette étape réalisée, les résultats remontent la chaîne d'appel : la classe VolumeManager totalise les volumes de toutes les géométries de l'objet et de ses éventuels enfants pour retourner à l'utilisateur une valeur scalaire unique représentant le volume total [70].

Pour illustrer, un exemple d'utilisation de notre bibliothèque pour calculer des volumes est présenté à la figure 14.

```
let cityModelImported = new CityModel().loadCityJsonFile(rawCityJSON); // on importe un objet CityJSON
let specificCityObject = cityModelImported.getCityObject("2041"); // on récupère un objet urbain spécifique
                                                                  // à partir de son identifiant

let volume = VolumeManager.computeVolume(specificCityObject);// on calcule avec la méthode des tétraèdres le volume
console.log(volume); // on affiche le volume dans la console

let allIds = cityModelImported.getAllCityObjectsId(); // on récupère tous les identifiants des objets CityObject
                                                      // référencés dans le modèle urbain
for(let idCityObject of allIds) {
    let currentCityObject = cityModelImported.getCityObject(idCityObject)
    // on utilise la méthode de Monte-Carlo sur tous les objets urbains
    console.log(VolumeManager.computeVolumeMC(currentCityObject,2,false)); //densité = 2 et considerChildren = false
    console.log(VolumeManager.computeVolumeMC(currentCityObject,1)); //densité = 1 et considerChildren = true (par défaut)
}
```

FIGURE 14 – Exemple de code illustrant l'utilisation des deux méthodes pour calculer des volumes. Les blocs try/catch/finally ne sont pas repris pour ne pas surcharger l'exemple.

69. Notons que ce deuxième argument est juste un raccourci pour l'utilisateur. En effet, ce dernier pourrait de lui-même implémenter cette récupération des objets 'enfants'. Néanmoins, nous avons fait le choix de la développer pour faciliter l'utilisation de notre bibliothèque.

70. Le calcul des volumes implique donc jusqu'à trois sommes implicites : une somme sur les solides (si l'on a un MultiSolid ou CompositeSolid), une somme sur les géométries si l'objet urbain est constitué de plusieurs géométries, et une somme sur les différents objets urbains considérés si le paramètre considerChildren est activé.

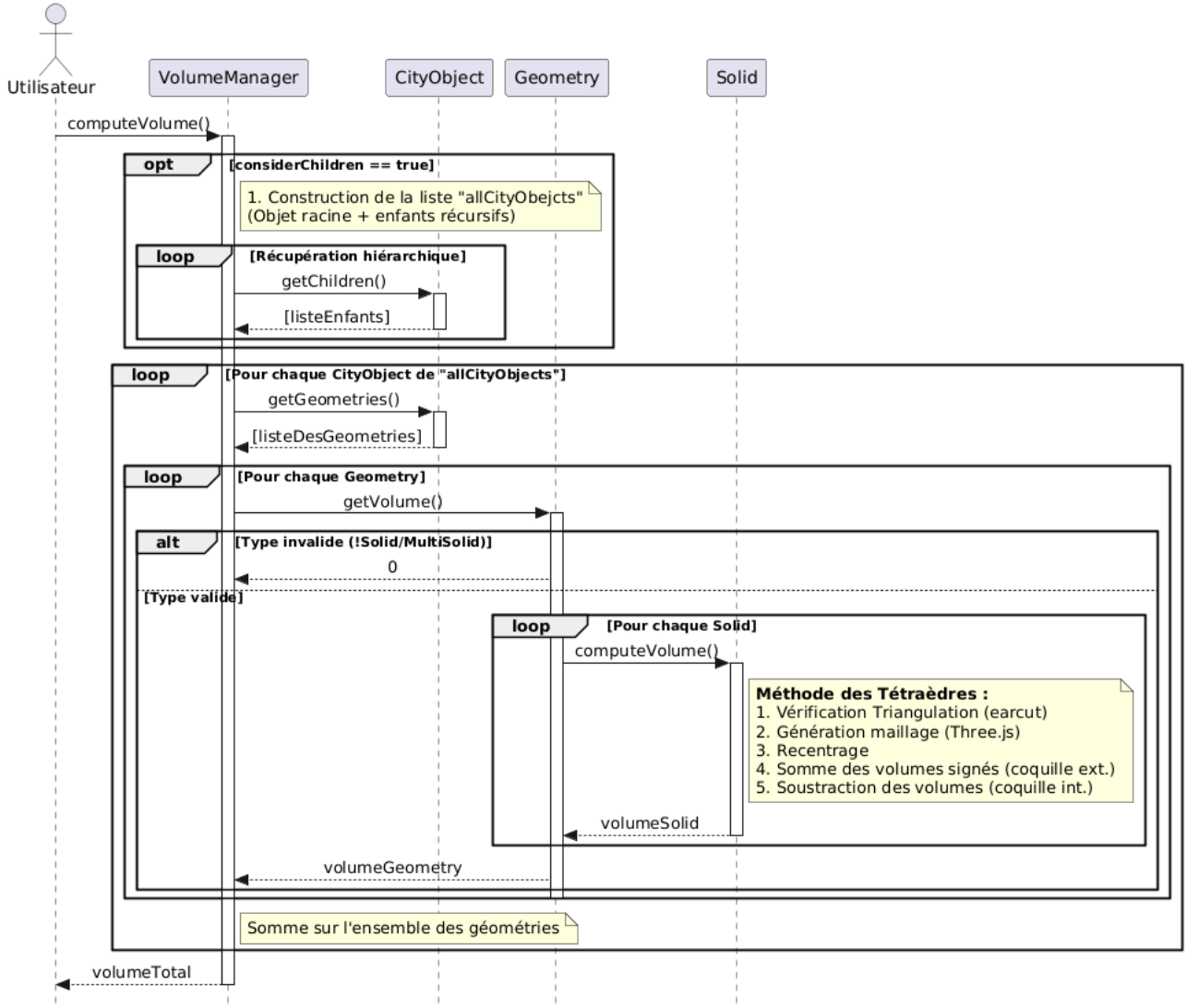


FIGURE 15 – Diagramme de séquence de l'algorithme permettant de calculer des volumes selon la méthode des tétraèdres.

4.4.6 Calcul d'intersections géométriques

4.4.6.1 Spécificités et complexité du calcul d'intersections

Le deuxième principal objectif de ce mémoire est de proposer un algorithme permettant de récupérer, dans le format CityJSON, le volume commun à deux géométries. En d'autres termes, nous devons permettre à l'utilisateur de calculer l'intersection entre deux objets urbains. Il s'agit d'une tâche bien plus ardue que le calcul de volumes, les deux raisons principales étant :

— **Le type de données attendu en retour** : les méthodes développées pour les volumes renvoient une simple valeur scalaire à l'utilisateur. En revanche, dans le contexte des intersections, il est nécessaire que l'utilisateur puisse visualiser le volume commun dans une interface de jumeau numérique urbain. Dès lors, il est essentiel que l'algorithme fournisse en retour des données CityJSON ;

— **La complexité des algorithmes** : Contrairement aux algorithmes relatifs aux volumes, ceux nécessaires au calcul des intersections sont bien plus lourds à implémenter. En effet, il est nécessaire de discriminer l'espace englobé par respectivement chaque solide du volume se trouvant à l'extérieur et de déterminer quelle partie est commune aux deux objets (solides). Il est donc nécessaire d'utiliser une structure de données qui, pour chaque point de l'espace, nous renseigne sur son appartenance ou non au volume d'un solide. Le volume commun étant ainsi l'ensemble des points qui remplissent la condition d'appartenance aux deux objets.

Comme expliqué précédemment, les bibliothèques développées pour réaliser des opérations de CSG sont performantes pour cette tâche. En effet, ces dernières doivent permettre de calculer l'intersection entre deux géométries pour créer des formes complexes à partir de formes simples. Pour y parvenir, ces bibliothèques utilisent couramment des arbres de partition binaire de l'espace (BSP), structure de données évoquée à la section 4.4.1.3. Nous y avons indiqué que cette structure divisait récursivement l'espace en deux sous-sections à partir d'hyperplans définis par les polygones de l'objet. La bibliothèque *three-csg-ts* utilisée dans ce mémoire ne fait pas exception et utilise cette structure de données.

Dans cette section, nous n'allons pas entrer dans les détails algorithmiques comme nous l'avions fait pour les algorithmes relatifs aux volumes. En effet, les arbres BSP sont des structures complexes. Réaliser une description détaillée de ces dernières dépasse donc le cadre de ce mémoire. En conséquence, nous allons utiliser les méthodes fournies par *three-csg-ts* comme une boite noire sans nous soucier de l'implémentation de ces dernières.

4.4.6.2 Implémentation du calcul d'intersections

Similairement à l'implémentation des méthodes pour les volumes, nous avons créé une classe supplémentaire pour orchestrer les opérations d'intersections : IntersectionManager. Toutefois, bien qu'analogue à VolumeManager, une différence majeure les distingue. Alors que cette dernière ne fonctionnait qu'avec des méthodes statiques, la classe IntersectionManager doit impérativement être instanciée par l'utilisateur.

Ce choix architectural se justifie par la nécessité de lier le gestionnaire d'intersection à un modèle urbain pour stocker les objets résultant des méthodes d'intersection. En effet, IntersectionManager possède en attribut une référence vers une instance de CityModel, attribut qui doit être défini lors de l'instanciation du gestionnaire. L'utilisateur est totalement libre quant au choix du modèle urbain à associer au gestionnaire. En effet, il peut également choisir un objet CityModel vierge, instancié spécifiquement pour isoler les résultats des intersections. Une autre possibilité consiste à utiliser un modèle préexistant contenant déjà des données (par exemple, le modèle des données sources) auquel les résultats seront ajoutés.

Concernant la nature des objets générés et ajoutés au modèle urbain, les résultats des intersections auront systématiquement comme type "GenericCityObject". En effet, l'intersection spatiale entre deux bâtiments (entre autres) ne constitue pas intrinsèquement un bâtiment. En outre, cette opération entraine une perte inévitable de la sémantique des surfaces. Après un calcul d'intersection, il n'est plus pertinent de distinguer les murs des toitures. Par conséquent, tous les objets créés appartiennent à la classe "GenericCityObject". Cette classe regroupe toutes les entités ne correspondant à aucune autre classe spécifique du standard.

Après avoir instancié le gestionnaire, l'utilisateur doit invoquer la méthode IntersectionManager.computeIntersection(CityObject1,CityObject2,considerChildren). Outre les deux objets urbains dont on souhaite obtenir l'intersection, cette méthode accepte le même paramètre booléen que la méthode VolumeManager.computeVolume(). Le fonctionnement de ce paramètre est strictement identique dans les deux méthodes. En effet, si le paramètre considerChildren est activé, tous les descendants des deux objets passés en argument seront agrégés. Ainsi, si l'utilisateur calcule l'intersection entre deux bâtiments disposant chacun d'un "BuildingPart" comme enfant, la méthode traitera les quatre entités. Dans cet exemple, l'algorithme générera alors jusqu'à quatre objets "GenericCityObject", correspondant à toutes les combinaisons possibles entre les entités provenant du premier et du second objet. Naturellement, si le paramètre est désactivé, seuls les deux objets passés en argument seront considérés.

Une fois la méthode IntersectionManager.computeIntersection() invoquée, cette méthode s'assure que les deux objets CityObject passés en argument sont définis dans le même système de coordonnées de référence (CRS). Cette opération est réalisée en interrogeant les instances de CityModel associées à ces objets. Si les CRS ne sont pas identiques, le processus est immédiatement interrompu et une exception de type IntersectionError est levée, garantissant ainsi l'intégrité des calculs spatiaux.

Si aucune erreur n'est levée, la méthode instancie un nouveau CityObject de type "GenericCityObject" et son niveau de détail (LoD) est fixé à 2 par défaut [71]. A ce stade du processus, aucune géométrie n'est encore associée à ce nouvel objet urbain.

Ensuite, la méthode itère sur l'ensemble des géométries définies pour les objets considérés. Pour chaque combinaison de géométries, la responsabilité du calcul algorithmique est déléguée à

71. L'identifiant de ce CityObject est créé de la manière suivante : "Identifiant du premier objet" + "-" + "Identifiant du second objet" + "-Intersection"

la classe Geometry en invoquant sa méthode statique Geometry.getIntersection(geom1, geom2). Le choix d'une méthode statique se justifie par le fait que l'opération ne dépend pas de l'état d'une instance spécifique de Geometry, mais prend en entrée deux géométries distinctes pour en produire une troisième.

Le flux d'exécution passe alors dans la méthode Geometry.getIntersection(). Dans un premier temps, cette dernière vérifie que les primitives géométriques traitées sont exclusivement de type "Solid", "CompositeSolid" ou "MultiSolid". En effet, cette contrainte résulte des algorithmes de CSG utilisés pour le calcul des intersections. Comme expliqué précédemment, ces algorithmes reposent sur des arbres BSP pour déterminer le résultat des opérations booléennes. Or, ces arbres ne peuvent être définis qu'à partir de volumes strictement clos.

Après avoir vérifié les types des primitives, la méthode instancie un nouvel objet Geometry destiné à devenir le réceptacle de l'ensemble des solides générés par l'intersection. Par la suite, la méthode procède à une double itération sur tous les solides constitutifs de la première géométrie et ceux de la seconde. Ces boucles imbriquées permettent de définir toutes les paires de solides (Solid1 de Geom1 avec Solid1 de Geom2, etc.) qui seront ensuite soumises au calcul d'intersection.

Ensuite, pour chaque paire créée, le calcul est délégué à la méthode statique Solid.compu teIntersection(solid1,solid2). C'est au sein de cette méthode que les intersections vont formellement être calculées. En s'appuyant sur la bibliothèque three-csg-ts, les objets Mesh [72] de chaque solide vont être convertis en arbres de partition binaire de l'espace. Si le solide dispose de coquilles intérieures, nous devons les soustraire de la coquille extérieure. Pour ce faire, nous utilisons d'abord l'opération booléenne de soustraction. Une fois cette étape réalisée, on peut fusionner les arbres des deux solides et réaliser l'opération d'intersection proprement dite.

Le résultat de cette opération est obtenu sous la forme d'une liste de triplets de coordonnées définissant les sommets du nouveau volume. À partir de cette liste, nous instancions un nouvel objet Solid pour représenter l'intersection calculée. Ce nouvel objet est alors retourné à la méthode Geometry.getIntersection(). Cette dernière va agréger l'ensemble des solides résultants au sein de l'objet Geometry créé en amont. Ce processus est répété pour l'intégralité des paires de solides identifiées. Une fois le traitement de l'ensemble des paires terminé, l'objet Geometry, désormais rempli des différents solides, est renvoyé au gestionnaire d'intersection.

Le flux d'exécution revient ainsi dans la méthode intersectionManager.compute Intersection(). La dernière étape de cette méthode consiste à associer cet objet Geometry au CityObject créé précédemment. Cette association est réalisée via la méthode CityObject.add Geometry(). Ce processus continue pour l'ensemble des paires de géométries (cf. figure 16).

Une fois les boucles sur les géométries terminées, le nouvel objet urbain créé préalablement, dorénavant complètement défini, est ajouté au CityModel référencé par le gestionnaire d'intersection. L'ensemble de ce processus est répété pour toutes les combinaisons de paires d'objets

72. Pour rappel, ces objets sont stockés dans les attributs threeExteriorShell et threeInteriorShells de la classe Solid.

urbains possibles.

Enfin, si l'utilisateur souhaite sauvegarder dans un fichier le modèle urbain actualisé avec les nouveaux objets, il lui suffit d'utiliser la méthode CityModel.getCityJSONObject() sur l'objet CityModel associé au gestionnaire. La figure 17 l'illustre en montrant un exemple complet d'utilisation de notre bibliothèque en combinant différentes méthodes décrites tout au long de ce chapitre.

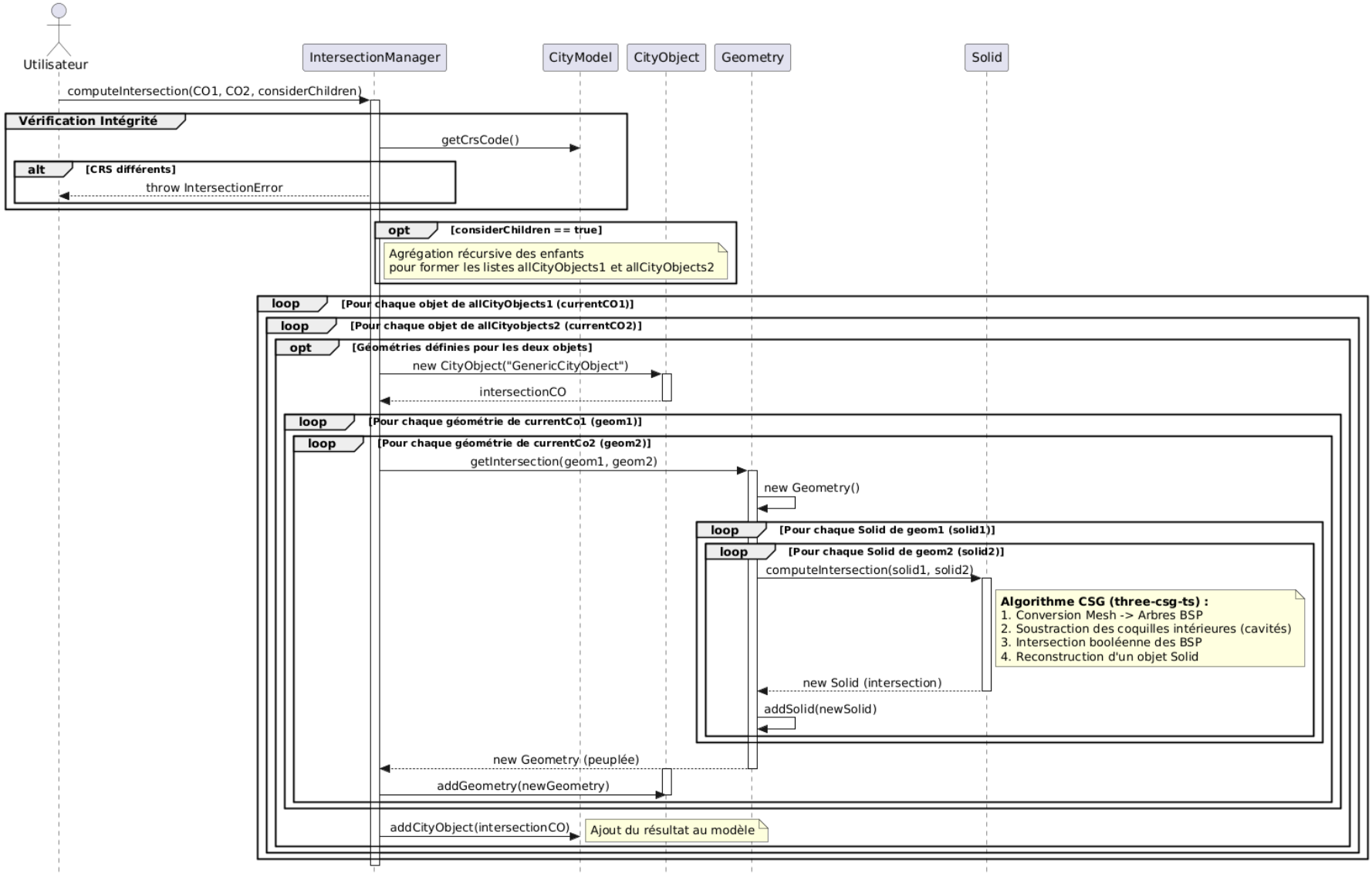


FIGURE 16 – Diagramme de séquence de l'algorithme permettant de calculer l'intersection entre deux objets urbains à l'aide de la bibliothèque *three-csg-ts*. Dans cette figure, 'CO' est souvent utilisé comme diminutif de CityObject.

4.4.7 Implémentation du calcul de zones tampons (buffers)

La dernière méthode de géotraitement implémentée dans notre bibliothèque permet de générer des prototypes de zones tampons (buffers). Contrairement à une approche classique épousant la forme des objets, notre méthode repose sur une dilatation tridimensionnelle de la boite englobante. Concrètement, cette opération consiste à étendre la boite englobante d'un objet urbain en projetant ses sommets vers l'extérieur. Cette projection est réalisée selon le vecteur normal propre à chaque point et selon une distance fixée par l'utilisateur. Nous nommerons cette distance rayon par la suite. Par exemple, si l'utilisateur fixe un rayon de 10 mètres, l'algorithme translate tous les sommets de la boite englobante de 10 mètres vers l'extérieur. Cette méthode, bien que rudimentaire, permet d'estimer rapidement la zone d'influence d'un objet urbain avec un algorithme simple à concevoir et rapide. Il est ensuite possible de calculer le volume de cette zone tampon ou de calculer les intersections entre celle-ci et d'autres objets urbains.

L'implémentation de cette méthode est semblable aux implémentations des calculs d'intersections et de volumes. En effet, cette opération est également orchestrée par un gestionnaire : la classe BufferManager. À l'instar de la classe IntersectionManager, celle-ci doit être instanciée en lui associant une référence vers un CityModel pour stocker les zones tampons créées par l'algorithme.

Le calcul peut être déclenché par l'utilisateur via l'invocation de la méthode BufferManager.computeBBBuffer(). Celle-ci nécessite trois arguments : l'objet urbain cible (CityObject), un rayon d'expansion positif et le booléen considerChildren pour la gestion récursive de la hiérarchie [73]. Après avoir vérifié si le CRS de l'objet CityObject est identique au CRS du CityModel associé au gestionnaire, la méthode instancie pour chaque objet traité un nouveau CityObject de type "GenericCityObject" [74].

La construction géométrique est déléguée via l'invocation de la méthode Geometry.getBB BufferGeometry() et ensuite, via la méthode Solid.computeBBBuffer() [75]. Cette dernière construit le buffer en définissant les huit sommets de la boite englobante et en les projetant vers l'extérieur selon leurs vecteurs normaux respectifs, à une distance définie par le rayon.

Ces sommets projetés permettent ensuite d'instancier un nouvel objet Solid qui est ici un simple parallélépipède rectangle (LoD 1). Ce solide est retourné à la méthode d'appel Geometry.getBBBufferGeometry() qui agrège les zones tampons de tous les solides dans un nouvel objet Geometry. Cette nouvelle géométrie est ensuite retournée au gestionnaire qui l'assigne au "GenericCityObject" nouvellement créé [76] pour ensuite le stocker dans le modèle urbain associé au gestionnaire. On obtient ainsi des zones tampons rudimentaires.

73. Si ce paramètre est activé, une zone tampon sera calculée sur chaque enfant de l'objet considéré.
74. Un buffer ne peut pas être associé à un type d'objet urbain particulier, il représente juste une zone d'influence.
75. L'algorithme est en tous points semblable à celui des intersections.
76. L'identifiant de ce nouvel objet urbain est défini de la sorte : "Identifiant de l'objet urbain source" + "-BBBuffer".

```
try {
    let cityModelImported = new CityModel().loadCityJsonFile(rawCityJSON); // on importe un objet CityJSON

    //on définit des paramètres de transformation linéaire des coordonnées
    let transformParams = {"scale":[1,1,1], "translate":[0,0,0]}
    // on définit un nouveau CityModel à partir du code EPSG du CRS, des paramètres de transformation,
    // du type de données et de la version utilisée
    // seul le code ESPG est obligatoire, des valeurs par défaut sont définies pour les autres paramètres
    let newCityModel = new CityModel().create("3812", transformParams, "CityJSON", "2");

    let bufferM = new BufferManager(newCityModel); // on associe le nouveau CityModel au bufferManager
    // on crée un buffer rudimentaire
    bufferM.computeBBBuffer(cityModelImported.getCityObject("1126"),10); // rayon = 10

    let bufferId = newCityModel.getAllCityObjectsId()[0]; // on récupère l'identifiant du buffer créé
    let bufferObject = newCityModel.getCityObject(bufferId);
    console.log(VolumeManager.computeVolume(bufferObject)); // on calcule le volume du buffer

    let intersectionM = new IntersectionManager(newCityModel);
    // on calcule l'intersection entre deux objets provenant d'un objet CityJSON importé et le buffer créé
    intersectionM.computeIntersection(cityModelImported.getCityObject("1129"),bufferObject);
    intersectionM.computeIntersection(cityModelImported.getCityObject("1120"),bufferObject);

    // on récupère un objet CityJSON pour sauvegarder les résultats de nos traitements
    let cityModelToExport = newCityModel.getCityJsonObject();
    telechargerJSON(JSON.stringify(cityModelToExport));

} catch (error) {
    if(error instanceof IntersectionError || error instanceof BufferError) {
        // on affiche l'erreur à l'utilisateur via l'élément HTML div
        document.getElementById('baliseDiv').textContent = error.message;
    } else if (error instanceof CityJSONError) {
        // on fait une requête ajax pour interroger le serveur (par exemple)
    }
    ...
}
```

FIGURE 17 – Exemple complet de code illustrant des calculs d'intersections, de buffers et de volumes.

5 Validation

5.1 Introduction et données utilisées

Le code complet de notre bibliothèque intégrant la méthodologie décrite dans le chapitre précédent est disponible à l'adresse suivante : https://github.com/Wilder25/CityJSONLib.

Pour réaliser la validation de notre bibliothèque, nous nous appuyons sur le modèle 3D du quartier d'Outremeuse à Liège évoqué à la section 2.5.

Les algorithmes développés dans ce mémoire fonctionnent uniquement sur des primitives solides ; nous restreignons donc notre validation aux bâtiments de ce jeu de données. Nous écartons ainsi le réseau routier et la végétation, qui ne sont pas des solides, contrairement aux bâtiments. Notre jeu de données pour la validation contient 3815 primitives géométriques solides.

Sur la base de ce jeu de données, la validation de notre bibliothèque se décompose en cinq points :

1. Vérifier l'intégrité du processus complet d'importation et d'exportation, en s'assurant que la bibliothèque produit un objet CityJSON valide en sortie ;
2. Évaluer la qualité de la triangulation effectuée par la bibliothèque earcut ;
3. Valider l'exactitude et la performance des différentes méthodes implémentées pour le calcul des volumes ;
4. Tester la robustesse de l'algorithme développé pour le calcul d'intersections ;
5. Valider la génération de prototypes de zones tampons.

5.2 Validation du schéma CityJSON

La première étape de notre validation consiste à s'assurer que notre architecture en classes préserve l'intégrité de la structure des données lors des phases d'importation et d'exportation. Concrètement, nous avons vérifié si les objets générés par la méthode CityModel.getCityJSONObject() respectent scrupuleusement le schéma du standard CityJSON, tant au niveau de la syntaxe que de la hiérarchie des objets.

Pour effectuer cette vérification, nous avons utilisé l'outil cjval [77], un validateur de référence pour ce format. L'ensemble des tests réalisés sur les fichiers [78] exportés s'est avéré concluant. En effet, aucune erreur n'a été détectée par cjval, confirmant ainsi que notre bibliothèque produit des objets CityJSON syntaxiquement valides.

77. Disponible à l'adresse suivante : https://github.com/cityjson/cjval.
78. qui contiennent les objets

5.3 Validation de la triangulation

La deuxième étape de notre validation consiste à évaluer la qualité de la triangulation réalisée par notre bibliothèque via l'outil earcut. Pour ce faire, nous avons comparé nos résultats à ceux obtenus par cjio [79], un outil Python de référence permettant d'effectuer diverses opérations sur des fichiers CityJSON, dont la triangulation.

Nous avons réalisé cette comparaison sur l'ensemble des 3815 primitives solides de notre jeu de données. Deux critères ont été utilisés : le temps d'exécution et la validité géométrique des primitives triangulées. La conformité de ces dernières avec la norme ISO 19107 a été vérifiée grâce à l'outil val3dity [80].

Les résultats de cette comparaison sont repris dans le tableau 1. L'analyse des données obtenues met en évidence des différences significatives entre les deux outils. Tandis que cjio parvient à valider 96,6% des primitives, earcut affiche un taux de validation inférieur : 88,3%. Dans la plupart des cas, val3dity indique des orientations de surfaces incorrectes et des coquilles non fermées pour les 11,7% de primitives incorrectes.

Néanmoins, cette moindre robustesse est compensée par une vitesse de calcul nettement supérieure. En effet, earcut s'avère environ dix fois plus rapide que cjio. Cette vitesse est un atout majeur dans notre contexte. Elle montre qu'earcut est capable de réaliser une triangulation en quasi temps réel directement au sein du navigateur client.

En conclusion, bien que notre bibliothèque offre une solution de triangulation performante, sa robustesse géométrique reste perfectible par rapport à des outils fonctionnant côté serveur. Par conséquent, afin de garantir des résultats optimaux lors de l'utilisation de nos fonctions de géotraitement, il reste préférable que l'utilisateur fournisse des données préalablement triangulées. Néanmoins, nos résultats démontrent que notre solution demeure correcte pour la grande majorité des cas, même en l'absence de prétraitement.

Outils	Validation par val3dity (%)	Temps d'exécution (s)
earcut	88.3	$\approx$ 1,9
cjio	96.6	$\approx$ 25

TABLE 1 – Comparaison des taux de validation (%) (via val3dity) des primitives des bâtiments d'Outremeuse après triangulation réalisée par les outils earcut et cjio. Le temps d'exécution est également repris.

79. Disponible à l'adresse suivante : https://github.com/cityjson/cjio.
80. Disponible à l'adresse suivante : https://github.com/tudelft3d/val3dity.

5.4 Validation des volumes

5.4.1 Méthode des tétraèdres

Afin de valider l'exactitude de notre implémentation de la méthode des tétraèdres, nous avons confronté ses résultats à ceux obtenus par une méthode de référence exécutée côté serveur. Dans cette section, nous n'allons pas utiliser les 3815 primitives. Nous utiliserons plutôt un échantillon de test constitué de 39 bâtiments issus du jeu de données d'Outremeuse. Cet échantillon a été construit pour être représentatif de la diversité du quartier, en y incluant aussi bien des structures simples (habitations) que des structures complexes (comme une église).

Ces bâtiments ont été importés dans une base de données 3DCityDB, nous permettant ainsi d'exploiter les fonctions spatiales avancées de PostGIS. Plus spécifiquement, nous utilisons la fonction CG_Volume de l'extension SFCGAL [81]. La requête SQL utilisée pour extraire ces volumes de référence est présentée à la figure 18.

```
select CG_Volume(CG_MakeSolid(surface_geometry.solid_geometry)),cityobject.gmlid as face from cityobject
join surface_geometry on cityobject.id = surface_geometry.cityobject_id
where cityobject.gmlid=:IDENTIFIANT_DU_BATIMENT
and surface_geometry.solid_geometry is not null;
```

FIGURE 18 – Requête SQL permettant de calculer le volume de bâtiments (solides) stockés dans une base de données 3DCityDB.

Les résultats de cette comparaison sont compilés dans le tableau 2 [82]. L'analyse des données obtenues montre une adéquation parfaite entre les volumes calculés par notre bibliothèque et ceux fournis par PostGIS. Les résultats sont strictement identiques, l'égalité se vérifiant jusqu'à la quatrième décimale (bien que le tableau soit tronqué à deux décimales par souci de lisibilité). Cette égalité prouve le caractère exact de la méthode des tétraèdres implémentée.

En termes de performance, notre méthode s'avère extrêmement efficace. Les temps de calcul restent systématiquement de l'ordre de la milliseconde, quel que soit le volume du bâtiment. À titre de comparaison, le calcul côté serveur peut nécessiter jusqu'à dix minutes pour les géométries les plus complexes.

Enfin, il convient de noter une différence majeure dans la gestion des erreurs entre les deux méthodes. PostGIS impose une conformité stricte à la norme ISO 19107 et refuse de calculer le volume de primitives invalides (ce qui explique l'absence de valeur pour certains bâtiments dans la colonne PostGIS). En revanche, notre bibliothèque est plus permissive et fournit un résultat même pour des géométries imparfaites.

81. SFCGAL est une bibliothèque développée en C++ autour de CGAL qui fournit des fonctions spatiales avancées en 2D et 3D (PostGIS, s. d.). CGAL est également une bibliothèque C++ open source qui est une référence pour la géométrie computationnelle en fournissant de nombreux algorithmes et structures de données pour manipuler des géométries 3D (CGAL, s. d.). Elle constitue la référence côté serveur de notre bibliothèque.

82. Par souci de lisibilité, ce tableau contient uniquement les résultats pour 15 primitives. L'ensemble des résultats pour les 39 primitives est disponible en annexe (annexe 2).

	Volume méthode des tétraèdres (m^3)	Volume PostGIS (m^3)	Temps méthode des tétraèdres (ms)
1	171,21	171,21	4
2	92,42	92,42	1
3	261,34	261,34	1
4	28705,67	28705,67	8
5	300,10	300,10	2
6	840,18	840,18	1
7	234,78	234,78	0
8	12040,21	/	5
9	986,26	986,26	1
10	153,43	153,43	1
11	86,57	/	0
12	5546,63	5546,63	2
13	589,45	589,45	0
14	712,68	712,68	2
15	222,99	222,99	0

TABLE 2 – Volume (m^3) de bâtiments extraits du jumeau numérique urbain du quartier d'Outremeuse à Liège. Les volumes ont été calculés par la méthode des tétraèdres (deuxième colonne) et par l'utilisation de la fonction CG_Volume() de PostGIS (troisième colonne). La quatrième colonne reprend le temps d'exécution de la méthode des tétraèdres pour chaque bâtiment en millisecondes (ms).

5.4.2 Méthode de Monte-Carlo

La méthode des tétraèdres ayant été validée comme exacte, nous l'avons utilisée comme référence pour évaluer la précision de la méthode de Monte-Carlo. Cette validation a été menée sur le même échantillon de 39 primitives.

Nous avons effectué les calculs avec deux niveaux de densité de grille distincts : une densité fixée à 1 et une densité fixée à 2. Les résultats sont compilés dans le tableau 3 [83]. Comme nous nous y attendions, nous observons une convergence des résultats vers la valeur exacte à mesure que la densité augmente.

L'analyse statistique des écarts confirme cette tendance :

— Avec une densité de 1, l'écart relatif absolu moyen est de 8,56% (écart-type de 8,04%) ;

— Avec une densité de 2, la précision s'améliore significativement, l'écart relatif absolu moyen tombant à 2,93% (écart-type de 2,45%).

83. Similairement à la validation de la méthode des tétraèdres, l'ensemble des résultats se trouve en annexe (annexe 3).

	Densité = 1			**Densité = 2**		
	Volume (m^3)	Temps (ms)	Écart avec la méthode des tétraèdres (%)	Volume (m^3)	Temps (ms)	Écart avec la méthode des tétraèdres (%)
1	137,49	7	-19,69	160,01	19	-6,54
2	87,22	2	-5,63	95,52	14	3,35
3	203,23	3	-22,24	245,36	21	-6,12
4	29 076,00	9196	1,29	28 909,27	75 214	0,71
5	230,68	7	-23,13	275,86	50	-8,08
6	756,51	15	-9,96	832,5	120	-0,91
7	221,1	3	-5,83	237,71	22	1,25
8	12 503,20	3516	3,85	12 279,02	29 004	1,98
9	1052,57	64	6,72	1018,23	524	3,24
10	135,87	3	-11,45	154,97	18	1,01
11	60,22	3	-30,44	80,05	21	-7,53
12	5652,18	176	1,9	5605,77	1435	1,07
13	576,29	30	-2,23	602,72	238	2,25
14	760,79	112	6,75	735,55	910	3,21
15	242,5	7	8,75	231,42	49	3,78

TABLE 3 – Volume (m^3) de bâtiments extraits du jumeau numérique urbain du quartier d'Outremeuse à Liège. Les volumes ont été calculés par la méthode de Monte-Carlo avec deux valeurs pour le paramètre de densité : 1 (à gauche) et 2 (à droite). Pour chaque bâtiment et densité, le temps d'exécution est repris en millisecondes (ms). L'écart relatif (en %) entre la méthode des tétraèdres et celle de Monte-Carlo est également indiqué.

Cependant, ce gain de précision s'accompagne d'une augmentation significative du temps de calcul. Le passage d'une densité de 1 à 2 entraine une augmentation cubique du nombre de points à traiter. Cette explosion du temps d'exécution est particulièrement visible dans le tableau 3 pour les bâtiments imposants, où on constate effectivement une augmentation du temps d'exécution par un facteur d'environ huit. À titre d'exemple, le calcul du volume d'une église (bâtiment n°4) nécessite 9 secondes avec une densité de 1 et grimpe à 75 secondes (!) avec une densité de 2.

En conclusion, bien que la méthode de Monte-Carlo soit pertinente pour obtenir un ordre de grandeur ou traiter des formes spécifiques (comme la présence de débords de toit, par exemple), sa lenteur constitue un frein majeur à son utilisation dans un contexte interactif.

5.5 Validation des intersections

La dernière étape principale de notre validation est de tester la robustesse de notre algorithme d'intersections. Pour ce faire, nous avons développé un protocole de test élémentaire qui consiste à calculer l'intersection d'un bâtiment avec lui-même. Le bâtiment utilisé est illustré à la figure 19a. Théoriquement, cette opération devrait restituer la géométrie originelle du bâtiment. Néanmoins, la dimension sémantique sera absente conformément aux choix d'implémentation décrits dans la méthodologie.

Cependant, le résultat de cette opération, illustré à la figure 19b, présente des anomalies visuelles significatives : la géométrie apparait dégradée, comportant des trous et des faces brisées. L'origine de ces artefacts réside principalement dans la gestion des nombres à virgule flottante par la bibliothèque Three.js, utilisée par la bibliothèque three-csg-ts.

Comme détaillé précédemment, Three.js stocke les coordonnées des sommets sur 32 bits (simple précision) [84], alors que les données sources sont encodées sur 64 bits (double précision). Dans un système de coordonnées projetées tel que le Belgian Lambert 2008 (CRS de notre dataset test), la partie entière des coordonnées atteint des ordres de grandeur de 10^5 à 10^6. La conversion en 32 bits, limitant le nombre de chiffres significatifs à environ 7, provoque une perte de précision sur la partie décimale, ne laissant qu'une précision de l'ordre de quelques centimètres. Le tableau 4 illustre cette dégradation numérique.

Ce phénomène d'imprécision est amplifié par les méthodes de CSG. L'utilisation d'arbres de partition binaire de l'espace (BSP) fragmente les polygones initiaux et augmente drastiquement la densité du maillage. Dans notre exemple, le bâtiment passe de 45 sommets initiaux à 161 après intersection [85]. Cette densification crée des sommets très proches les uns des autres qui, sous l'effet de l'arrondi 32 bits, peuvent être fusionnés ou inversés, brisant ainsi la topologie des faces.

Pour pallier ce problème, nous pouvons effectuer un recentrage systématique des coordonnées en les rapprochant de l'origine (0,0,0) avant tout calcul d'intersections. Cette translation permet de réduire la partie entière des coordonnées, maximisant ainsi la précision disponible pour la partie décimale.

Le résultat obtenu après recentrage est illustré à la figure 19c. On y constate que le bâtiment résultant est visuellement fidèle à la géométrie source. Toutefois, cette méthode présente deux limites. Premièrement, elle implique une perte du contexte spatial (géoréférencement) [86] durant le calcul [87]. Deuxièmement, bien que l'aspect visuel soit corrigé, l'analyse par val3dity indique

84. Convention obligatoire héritée du WebGL.

85. En outre, nous pensons que les méthodes de CSG fonctionnent mieux sur des maillages réguliers. Initialement, ces algorithmes sont conçus pour prendre en entrée des objets simples. Or, les bâtiments de notre dataset peuvent être assez complexes et présenter un maillage irrégulier (présence de triangles très petits ou d'autres très allongés). Nous faisons l'hypothèse que cela complexifie la création des arbres BSP et dégrade donc la définition de la frontière entre l'intérieur de l'objet et l'extérieur.

86. Il n'est dès lors plus possible de représenter le bâtiment sur un fond OpenStreetMap par exemple.

87. En outre, il est inutile d'utiliser les paramètres *transform* des objets CityJSON pour compresser les coordonnées. En effet, cela ne fait que reporter le problème car les infrastructures de jumeaux numériques urbains ont

que les géométries résultantes ne sont pas strictement conformes à la norme ISO 19107.

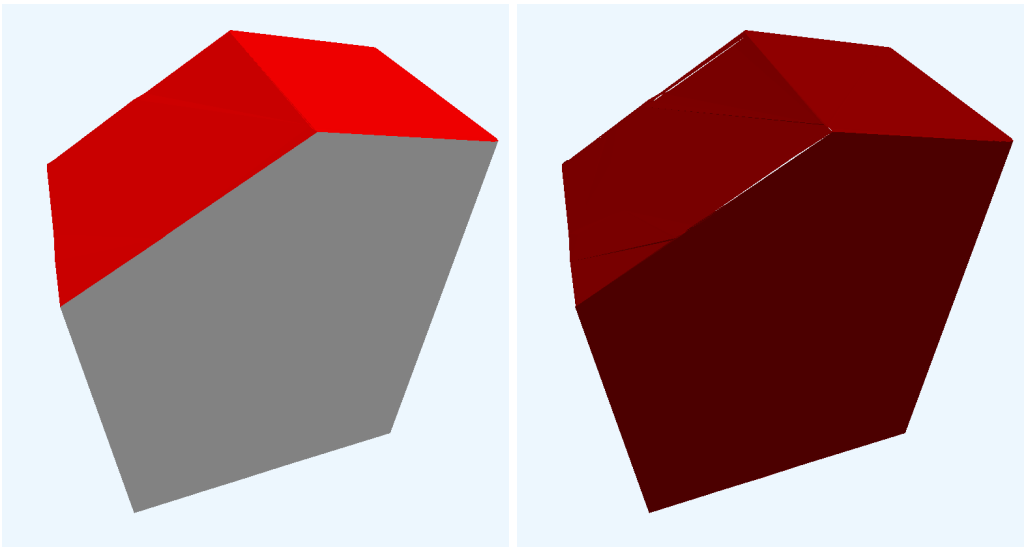
Enfin, des tests complémentaires réalisés en calculant l'intersection entre des bâtiments et des zones tampons parallélépipédiques ont confirmé ces observations. L'algorithme parvient à isoler correctement les volumes communs, mais des artefacts visuels apparaissent systématiquement si les coordonnées ne sont pas recentrées.

Pour conclure, notre validation des calculs d'intersections donne des résultats mitigés. L'algorithme est certes capable d'extraire correctement le volume commun à deux géométries, mais les limitations de précision induites par Three.js dégradent fortement l'apparence visuelle des intersections en produisant des géométries non conformes à la norme ISO 19107.

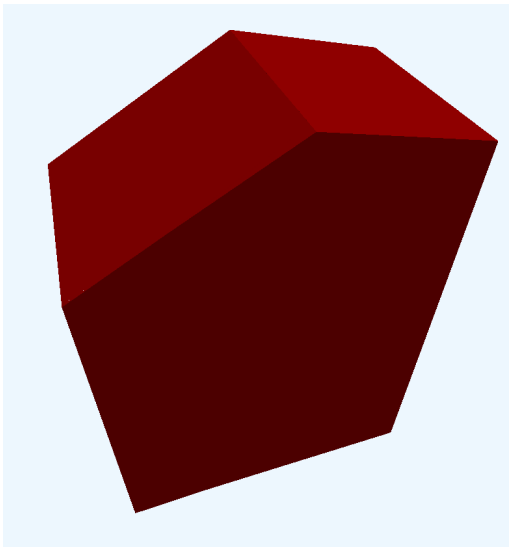
Coordonnées sur 32 bits (m)	Coordonnées sur 64 bits (m)	Différence absolue (cm)
736 164,125	736 164,1516	2,66
648 778,875	648 778,8443	3,07
736 163,5625	736 163,5916	2,91
648 780,625	648 780,6103	1,47
736 166,8125	736 166,7826	2,99
648 779,75	648 779,7573	0,73

TABLE 4 – Impact de la réduction de précision des nombres à virgule flottante (passage de 64 à 32 bits) sur des coordonnées Belgian Lambert 2008. La troisième colonne met en évidence l'erreur de positionnement engendrée (cm).

besoin de décompresser les coordonnées pour afficher les bâtiments. Le problème apparaîtra alors lors de la visualisation. La seule solution consiste à utiliser des référentiels locaux qui diminuent, même lors de la visualisation, la partie entière des coordonnées.



(a) (b)



(c)

FIGURE 19 – Exemple de calcul de l'intersection entre un bâtiment et lui-même. (a) Bâtiment source. (b) Résultat de l'intersection sans recentrage préalable des coordonnées. (c) Résultat de l'intersection avec recentrage préalable des coordonnées.

5.6 Validation des zones tampons

Pour valider les zones tampons, nous avons calculé le buffer d'un bâtiment et nous l'avons affiché dans City2Twin. Le résultat est illustré à la figure 20. Cette fonction fonctionne correctement en générant un parallélépipède rectangle qui constitue un prototype de buffer.

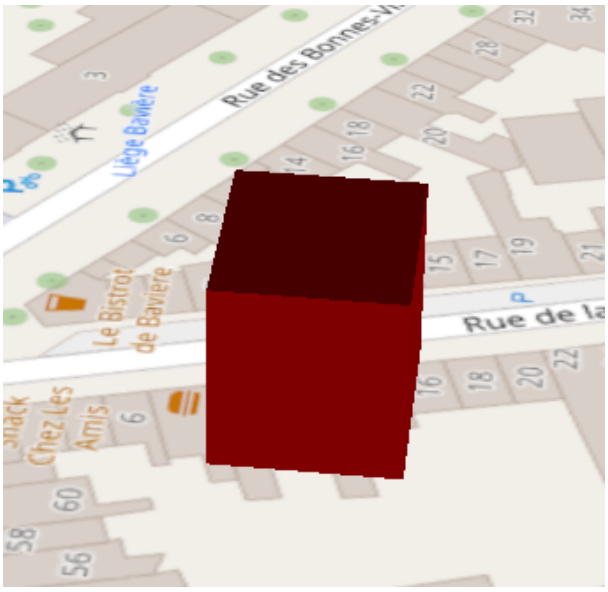


FIGURE 20 – Exemple de zone tampon parallélépipédique générée par notre bibliothèque et affichée dans l'interface City2Twin.

6 Conclusion

La problématique de ce mémoire repose sur un constat technologique paradoxal. Alors que la puissance de calcul des machines clientes et les capacités des moteurs JavaScript, tels que V8, n'ont cessé de croître, l'analyse spatiale tridimensionnelle dans les navigateurs web accuse un retard significatif par rapport à son homologue bidimensionnelle. Si des bibliothèques comme Turf.js ont développé le géotraitement 2D côté client, un vide technologique persiste pour la 3D, obligeant les jumeaux numériques urbains à déléguer systématiquement ces calculs à des serveurs distants. Or, le développement de systèmes de bases de données utilisant directement le format CityJSON pour le stockage complique grandement l'utilisation de fonctions d'analyse spatiale côté serveur. L'objectif de ce travail était donc de combler ce vide en proposant un prototype de bibliothèque JavaScript capable d'effectuer des opérations de géotraitement 3D directement dans le navigateur.

Pour répondre à ce défi, nous avons proposé une architecture orientée objet rigoureuse, fondée principalement sur la norme ECMAScript 2015 (ES6). Cette structure, qui s'articule autour de classes centrales telles que CityModel, CityObject, Geometry et Solid, constitue la pierre angulaire de notre bibliothèque. Elle permet non seulement de modéliser fidèlement la hiérarchie et la sémantique du format CityJSON, mais elle assure également l'importation et l'exportation des données, qu'elles proviennent de fichiers standards ou de bases de données cjdb. Ce choix architectural garantit la modularité du code et facilite son intégration future dans des infrastructures de jumeaux numériques existantes.

Afin de ne pas réinventer la roue et de maximiser l'efficacité du développement, cette architecture a intégré trois outils tiers : Three.js pour la gestion des maillages et le calcul vectoriel, earcut pour la triangulation rapide des polygones, et three-csg-ts pour les opérations booléennes sur les solides. L'intégration de ces bibliothèques au sein de nos classes, notamment via l'ajout d'attributs et de méthodes spécifiques dans la classe Solid, a permis de développer concrètement des fonctions de géotraitement.

Sur cette base, nous avons implémenté plusieurs algorithmes d'analyse spatiale. Pour le calcul des volumes, deux approches ont été confrontées : la méthode des tétraèdres, mathématiquement rigoureuse pour les volumes clos, et la méthode de Monte-Carlo, fondée sur une approche stochastique et l'utilisation implicite de voxels. En parallèle, nous nous sommes appuyés sur les arbres de partition binaire de l'espace (BSP) pour extraire le volume commun de deux objets urbains, et nous avons développé une méthode permettant de générer des prototypes de zones tampons.

La phase de validation, réalisée sur un jeu de données réel du quartier d'Outremeuse à Liège, a mis en évidence des résultats contrastés. D'une part, la méthode des tétraèdres a démontré une efficacité remarquable, offrant des résultats strictement identiques à ceux obtenus par PostGIS (SFCGAL) tout en maintenant des temps d'exécution de l'ordre de la milliseconde. De même, la triangulation via earcut, bien que légèrement moins robuste géométriquement que des outils

serveur comme cjio, s'est avérée dix fois plus rapide, validant la pertinence de son usage pour des traitements en temps réel côté client.

D'autre part, les tests ont révélé les limites techniques actuelles du géotraitement complexe en JavaScript pur. La méthode de Monte-Carlo, bien que fonctionnelle, souffre d'une lenteur excessive dès que l'on exige une précision élevée, la rendant inadaptée à une utilisation interactive. Plus critique encore, le calcul d'intersections s'est heurté à la gestion des nombres à virgule flottante. La restriction de Three.js à une précision de 32 bits (simple précision), inhérente à WebGL, engendre une dégradation numérique significative lors de la manipulation de coordonnées projetées (telles que le Belgian Lambert 2008). Si le recentrage des coordonnées permet de pallier visuellement ces artefacts, il ne résout pas le problème de fond : les géométries résultantes peinent à respecter la stricte conformité de la norme ISO 19107.

Ces limitations, inhérentes à l'écosystème JavaScript actuel, ouvrent la voie à de nouvelles perspectives de recherche. Pour dépasser les contraintes de précision et de performance observées, l'avenir du géotraitement 3D côté client semble résider dans l'utilisation du WebAssembly (Wasm). Cette technologie, qui agit comme un complément à JavaScript, permet d'exécuter dans le navigateur du code compilé à partir de langages performants comme C++ ou Rust (Mozilla, s. d.-d). Concrètement, cela offrirait l'opportunité de porter directement des routines de bibliothèques géométriques de référence, telles que CGAL, dans l'environnement client. Une telle approche permettrait de conserver la flexibilité de notre architecture orientée objet tout en bénéficiant de la robustesse arithmétique et de la puissance de calcul nécessaires pour faire du navigateur un véritable outil d'analyse spatiale 3D. Par exemple, il serait possible d'utiliser les algorithmes développés par Zhao et al. (2013) pour réparer les géométries provenant des calculs d'intersections, afin de rendre ces dernières conformes à la norme ISO 19107. Zhao et al. (2013) propose une méthode automatique pour corriger les objets urbains au format CityGML en LoD 2, mais cela requiert l'utilisation de bibliothèques C++. Dès lors, l'utilisation du WebAssembly permettrait l'intégration de cet algorithme dans le navigateur du client.

Références

Ballouch, Z., Jeddoub, I., Hajji, R., Kasprzyk, J.-P., & Billen, R. (2024). Towards a Digital Twin of Liege : The Core 3D Model based on Semantic Segmentation and Automated Modeling of LiDAR Point Clouds. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *X-4/W4-2024*, 13-20. https://doi.org/10.5194/isprs-annals-X-4-W4-2024-13-2024

Biljecki, F., Ledoux, H., & Stoter, J. (2016). An improved LOD specification for 3D building models. *Computers Environment and Urban Systems*, *59*, 25-37. https://doi.org/10.1016/j.compenvurbsys.2016.04.005

CGAL. (s. d.). *The Computational Geometry Algorithms Library*. Récupérée décembre 16, 2025, à partir de https://www.cgal.org/April24/index.html

Dembski, F., Wössner, U., Letzgus, M., Ruddat, M., & van der Laag-Yamu, C. (2020). Urban Digital Twins for Smart Cities and Citizens : The Case Study of Herrenberg, Germany. *Sustainability*, *12*, 17p. https://doi.org/10.3390/su12062307

Dirksen, J. (2014, juillet). *Three.js Essentials : Create and animate beautiful 3D graphics with this fast-paced tutorial* (1re éd.). Packt Publishing.

Erickson, J. (2024, avril 4). *What is JSON ?* Récupérée novembre 21, 2025, à partir de https://www.oracle.com/fr/database/what-is-json/

Hämäläinen, M. (2021). Urban development with dynamic digital twins in Helsinki city. *IET Smart Cities*, *3*(4), 201-210. https://doi.org/10.1049/smc2.12015

Hijazi, I., Donaubauer, A., Hamm, A., Falkenstein, A., & Kolbe, T. H. (2022). URBAN GROWTH SIMULATION USING URBAN DYNAMICS AND CITYGML : A USE CASE FROM THE CITY OF MUNICH. *ISPRS annals of the photogrammetry, remote sensing and spatial information sciences*, *X-4/W2-2022*, 97-104. https://doi.org/10.5194/isprs-annals-x-4-w2-2022-97-2022

ISO. (s. d.). *ISO 19107 :2019*. Récupérée novembre 22, 2025, à partir de https://www.iso.org/fr/standard/66175.html

Jeddoub, I., Nys, G.-A., Hajji, R., & Billen, R. (2023). Digital Twins for cities : Analyzing the gap between concepts and current implementations with a specific focus on data integration. *International Journal of Applied Earth Observation and Geoinformation*, *122*, 103440. https://doi.org/https://doi.org/10.1016/j.jag.2023.103440

Kasprzyk, J., & Devillet, G. (2021). A Data Cube Metamodel for Geographic Analysis Involving Heterogeneous Dimensions. *ISPRS International Journal of Geo-Information*, *10*(2), 87. https://doi.org/10.3390/ijgi10020087

Kasprzyk, J.-P., Nys, G.-A., & Billen, R. (2024). Towards a multi-database CityGML environment adapted to big geodata issues of urban digital twins. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLVIII-4/W10-2024*, 101-106. https://doi.org/10.5194/isprs-archives-XLVIII-4-W10-2024-101-2024

Kolbe, T., Kutzner, T., Smyth, C., Nagel, C., Roensdorf, C., & Heazel, C. (2021, septembre 13). *OGC City Geography Markup Language (CityGML) Part 1 : Conceptual Model Standard*. Récupérée juillet 25, 2025, à partir de https://docs.ogc.org/is/20-010/20-010.html

Kornienko, D. V., Mishina, S. V., Shcherbatykh, S. V., & Melnikov, M. O. (2021). Principles of securing RESTful API web services developed with python frameworks. *Journal of Physics Conference Series*, *2094*(3), 032016. https://doi.org/10.1088/1742-6596/2094/3/032016

Kutzner, T., Chaturvedi, K., & Kolbe, T. (2020). CityGML 3.0 : New Functions Open Up New Applications. *PFG – Journal of Photogrammetry Remote Sensing and Geoinformation Science*, *88*, 43-61. https://doi.org/10.1007/s41064-020-00095-z

Ledoux, H. (2018). val3dity : validation of 3D GIS primitives according to the international standards. *Open Geospatial Data, Software and Standards*, *3*. https://doi.org/10.1186/s40965-018-0043-x

Ledoux, H., & Dukai, B. (2024, avril 11). *CityJSON Specifications 2.0.1*. Récupérée novembre 21, 2025, à partir de https://www.cityjson.org/specs/2.0.1/

Ledoux, H., Ohori, K., Kumar, K., Dukai, B., Labetski, A., & Vitalis, S. (2019). CityJSON : a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data Software and Standards*, *4*, 4. https://doi.org/10.1186/s40965-019-0064-0

Lehner, H., & Dorffner, L. (2020). Digital geoTwin Vienna : Towards a Digital Twin City as Geodata Hub. *PFG – Journal of Photogrammetry Remote Sensing and Geoinformation Science*, *88*(1), 63-75. https://doi.org/10.1007/s41064-020-00101-4

Logto. (2024, juin 6). *Maîtriser le type JSONB de PostgreSQL en un article · Blog Logto*. Récupérée novembre 28, 2025, à partir de https://blog.logto.io/fr/maitriser-postgresql-jsonb

MongoDB. (s. d.). *What is NoSQL ? NoSQL databases explained*. Récupérée novembre 24, 2025, à partir de https://www.mongodb.com/resources/basics/databases/nosql-explained

Mozilla. (s. d.-a). *Introduction to web APIs - Learn web development | MDN*. Récupérée décembre 10, 2025, à partir de https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Client-side_APIs/Introduction

Mozilla. (s. d.-b). *JavaScript | MDN*. Récupérée décembre 3, 2025, à partir de https://developer.mozilla.org/fr/docs/Web/JavaScript

Mozilla. (s. d.-c). *JavaScript technologies overview - JavaScript | MDN*. Récupérée décembre 3, 2025, à partir de https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/JavaScript_technologies_overview

Mozilla. (s. d.-d). *WebAssembly | MDN*. Récupérée décembre 30, 2025, à partir de https://developer.mozilla.org/en-US/docs/WebAssembly

Mozilla. (s. d.-e). *WebGL : graphismes 2D et 3D pour le Web - Les API Web | MDN*. Récupérée décembre 8, 2025, à partir de https://developer.mozilla.org/fr/docs/Web/API/WebGL_API

Mozilla. (s. d.-f). *Working with objects - JavaScript | MDN*. Récupérée décembre 3, 2025, à partir de https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_objects

Nys, G.-A. (2023). Consistency guarantee. In *From consistency to flexibility : shifting the structure. Towards a new generation of geographical information systems* (p. 75-104).

Nys, G., & Billen, R. (2021). From consistency to flexibility : A simplified database schema for the management of CityJSON 3D city models. *Transactions in GIS*, *25*(6), 3048-3066. https://doi.org/10.1111/tgis.12807

Ohori, K., Biljecki, F., Kumar, K., Ledoux, H., & Stoter, J. (2018, septembre). Modeling Cities and Landscapes in 3D with CityGML. Springer. https://doi.org/10.1007/978-3-319-92862-3_11

Oluwatosin, H. S. (2014). Client-Server model. *IOSR Journal of Computer Engineering*, *16*(1), 57-71. https://doi.org/10.9790/0661-16195771

PostGIS. (s. d.). *Chapter 8. Référence des fonctions SFCGAL*. Récupérée décembre 16, 2025, à partir de https://postgis.net/docs/manual-3.5/fr/reference_sfcgal.html

PostgreSQL. (2025, novembre 13). *8.14. JSON Types*. Récupérée novembre 27, 2025, à partir de https://www.postgresql.org/docs/current/datatype-json.html

Powałka, L., Poon, C., Xia, Y., Meines, S., Yan, L., Cai, Y., Stavropoulou, G., Dukai, B., & Ledoux, H. (2024). cjdb : A Simple, Fast, and Lean Database Solution for the CityGML

Data Model. In T. H. Kolbe, A. Donaubauer & C. Beil (Éd.), *Recent Advances in 3D Geoinformation Science : Proceedings of the 18th 3D GeoInfo Conference* (p. 781-796). Springer. https://doi.org/10.1007/978-3-031-43699-4_47

Rafamatanantsoa, B. P., Jeddoub, I., Yarroudh, A., Hajji, R., & Billen, R. (2024). City2Twin : an open urban digital twin from data integration to visualization and analysis. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLVIII-2/W8-2024*, 387-394. https://doi.org/10.5194/isprs-archives-XLVIII-2-W8-2024-387-2024

Schrotter, G., & Hürzeler, C. (2020). The Digital Twin of the City of Zurich for Urban Planning. *PFG – Journal of Photogrammetry Remote Sensing and Geoinformation Science*, *88*(1), 99-112. https://doi.org/10.1007/s41064-020-00092-2

Segura, C., Stine, T., & Yang, J. (2013). Constructive Solid Geometry Using BSP Tree. https://api.semanticscholar.org/CorpusID:16447792

Tan, Y., Liang, Y., & Zhu, J. (2023). CityGML in the Integration of BIM and the GIS : Challenges and Opportunities. *Buildings*, *13*(7), 1758. https://doi.org/10.3390/buildings13071758

Turf.js. (s. d.). *Introduction | Turf.js*. Récupérée juillet 26, 2025, à partir de https://turfjs.org/docs/intro

Xing, Y., Huang, J., & Lai, Y. (2019). Research and Analysis of the Front-end Frameworks and Libraries in E-Business Development. *Proceedings of the 2019 11th International Conference on Computer and Automation Engineering*, 68-72. https://doi.org/10.1145/3313991.3314021

Yao, Z., Nagel, C., Kunde, F., Hudra, G., Willkomm, P., Donaubauer, A., Adolphi, T., & Kolbe, T. (2018). 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, *3*. https://doi.org/10.1186/s40965-018-0046-7

Zhang, C., & Chen, T. (2001). Efficient Feature Extraction For 2D/3D Objects In Mesh Representation. *IEEE International Conference on Image Processing*, *3*, 935-938 vol.3. https://doi.org/10.1109/ICIP.2001.958278

Zhao, Z., Ledoux, H., & Stoter, J. (2013). AUTOMATIC REPAIR OF CITYGML LOD2 BUILDINGS USING SHRINK-WRAPPING. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *II-2/W1*, 309-317. https://doi.org/10.5194/isprsannals-II-2-W1-309-2013

Annexe 1 : Chargement de données CityJSON depuis cjdb

Contrairement à l'importation d'un fichier CityJSON standard où les données sont centralisées dans un objet unique, l'utilisation de cjdb implique une fragmentation des données répartie sur trois tables distinctes (cj_metadata, city_object, city_object_relationships). L'utilisateur doit par conséquent extraire ces données via des requêtes SQL préalables avant de les injecter dans la bibliothèque.

Le code illustré à la figure 21 illustre ce processus d'importation :

— Initialisation du modèle urbain : La première étape consiste à instancier la classe CityModel. Les métadonnées globales (CRS, paramètres de transformation,version), qui proviennent théoriquement de la table cj_metadata, sont passées manuellement au constructeur pour hydrater le conteneur principal ;

— Instanciation et hydratation des objets urbains : Une boucle itère ensuite sur le tableau cityObjectTuples, contenant les résultats de la requête sur la table city_object. Pour chaque tuple, un CityObject est instancié. A la différence de l'import fichier, l'utilisateur doit utiliser la méthode spécifique CityObject.setFromCjdb(). Cette méthode est conçue pour hydrater l'objet en traitant directement les colonnes jsonb contenant les attributs et la géométrie. Chaque objet est ensuite ajouté au dictionnaire du modèle via CityModel.addCityObject().

— Reconstruction de la hiérarchie : Enfin, la structure hiérarchique est restaurée. L'utilisateur doit fournir un objet prétraité *hierarchy* dérivé de la table city_object_relationships, associant chaque objet à ses parents et enfants. La méthode CityModel.createHierarchy() parcourt alors l'ensemble des objets instanciés pour établir les références, finalisant ainsi la reconstruction de la hiérarchie complète en mémoire.

```
// création d'un cityModel à partir des données provenant de la table cjdb_metadata
// les données provenant de cjdb étant plus fragmentées, l'utilisateur doit réaliser un minimum de pré-traitement
let cjdbCityModel = new CityModel(crsCode, transformParameters, type, version);

// l'utilisateur doit fournir un tableau contenant les données pour les différents tuples de la table cj_cityobject
for (let cityObject of cityObjectTuples) {
    // cjdbObject doit être un dictionnaire dont chaque entrée correspond à un champ de cj_cityobject
    cjdbCityModel.addCityObject(new CityObject().setFromCjdb(cityObject,cjdbCityModel));
}

// une fois tous les objets créés, il faut recréer la hiérarchie
// à partir de la table city_object_relationships, l'utilisateur doit prétraiter les données en amont
// pour avoir un tableau hierarchy de la forme :
// {GMLIDCityObject1 : [enfants,parents], "GMLIDCityObject2": [enfants, parents]}
// où enfants et parents sont des tableaux contenant les gmlid des enfants/parents
// si pas d'enfants ou de parents, les tableaux doivent être vides
cjdbCityModel.createHierarchy(hierarchy);


// configuration finie
```

FIGURE 21 – Exemple de code illustrant l'importation de données CityJSON provenant de cjdb.

Annexe 2 : Validation de la méthode des tétraèdres

	Volume (m^3)	Volume PostGIS	temps (ms)
1	171,21	171,21	4
2	92,42	92,42	1
3	261,34	261,34	1
4	28705,67	28705,67	8
5	300,1	300,10	2
6	840,18	840,18	1
7	234,78	234,78	0
8	12040,21	/	5
9	986,26	986,26	1
10	153,43	153,43	1
11	86,57	/	0
12	5546,63	5546,63	2
13	589,45	589,45	0
14	712,68	712,68	2
15	222,99	222,99	0
16	226,39	226,39	1
17	209,57	209,57	1
18	120,16	120,16	0
19	616,54	616,54	2
20	457,22	457,22	0
21	360,87	360,87	1
22	646,77	646,77	0
23	1971,01	1971,01	1
24	125,76	312,59	1
25	1062,34	1062,34	0
26	1218,91	1218,91	2
27	1159,95	1159,95	1
28	644,39	644,39	0
29	933,15	933,15	9
30	98,8	98,80	0
31	160,81	160,81	0
32	124,13	124,13	1
33	164,76	164,76	0
34	582,51	582,51	0
35	566,98	566,98	0
36	222,57	222,57	0
37	614,08	/	2
38	103,35	103,35	0
39	100,67	100,67	0

Annexe 3 : Validation de la méthode de Monte-Carlo

	Densité = 1			**Densité = 2**		
	Volume (m^3)	Temps (ms)	Différence avec la méthode des tétraèdres (%)	Volume (m^3)	Temps (ms)	Différence avec la méthode des tétraèdres (%)
1	137,49	7	-19,69	160,01	19	-6,54
2	87,22	2	-5,63	95,52	14	3,35
3	203,23	3	-22,24	245,36	21	-6,12
4	29076,00	9196	1,29	28909,27	75214	0,71
5	230,68	7	-23,13	275,86	50	-8,08
6	756,51	15	-9,96	832,5	120	-0,91
7	221,1	3	-5,83	237,71	22	1,25
8	12503,20	3516	3,85	12279,02	29004	1,98
9	1052,57	64	6,72	1018,23	524	3,24
10	135,87	3	-11,45	154,97	18	1,01
11	60,22	3	-30,44	80,05	21	-7,53
12	5652,18	176	1,9	5605,77	1435	1,07
13	576,29	30	-2,23	602,72	238	2,25
14	760,79	112	6,75	735,55	910	3,21
15	242,5	7	8,75	231,42	49	3,78
16	240,28	6	6,13	232,72	52	2,8
17	225,83	8	7,76	216,39	61	3,25
18	138,95	7	15,64	127,91	56	6,45
19	630,42	31	2,25	626,86	259	1,67
20	466,82	10	2,1	463,2	73	1,31
21	361,59	5	0,2	363,35	50	0,69
22	618,94	12	-4,3	638,88	86	-1,22
23	1963,15	52	-0,4	1987,83	422	0,85
24	435,25	15	-0,71	447,99	121	2,2
25	1002,86	16	-5,6	1041,78	130	-1,93
26	1209,63	38	-0,76	1231,52	303	1,03
27	1073,32	32	-7,47	1127,01	243	-2,84
28	644,21	14	-0,03	646,83	112	0,38
29	992,48	164	6,36	966,57	1331	3,58
30	106,3	3	7,59	99,71	24	0,92
31	171,01	3	6,34	163,89	31	1,91
32	130,6	4	5,22	125,1	30	0,78
33	178,89	4	8,58	171,4	22	4,03
34	546,32	8	-6,21	578,97	64	-0,61
35	495,36	10	-12,63	556,06	73	-1,93
36	226,61	8	1,82	228,11	76	2,49
37	536,19	15	-12,68	595,25	111	-3,07
38	74,73	2	-27,7	91,95	9	-11,03
39	74,86	2	-25,64	94,23	10	-6,4