المملكة المغربية

ROYAUME DU MAROC

INSTITUT AGRONOMIQUE
ET VETERINAIRE HASSAN II

معهد الحسن الثاني
للزراعة والبيطرة

LIÈGE université

Projet de Fin d'Études présenté pour l'obtention du diplôme d'Ingénieur en Topographie

Modélisation 3D et développement d'une approche pour la structuration, le stockage et l'exploitation des infrastructures routières en 3D selon le standard CityGML 3.0 et son encodage CityJSON

Présenté et soutenu publiquement par

YARROUDH Anass

JURY

Pr. MOHA EL AYACHI	**Président**	**IAV HASSAN II**
Pr. RAFIKA HAJJI	**Rapporteuse**	**IAV HASSAN II**
M. GILLES-ANTOINE NYS	**Rapporteur**	**UGEOM de L'ULiège**
M. ABDERRAZZAQ KHARROUBI	**Examinateur**	**UGEOM de L'ULiège**

Septembre 2022

Adresse : Madinat Al Irfane, B.P. 6202. Rabat – Maroc
Tél : (00 212) 0537 77 17 58/59
Fax : (00 212) 0537 77 58 45
Site web : http://www.iav.ac.ma

العنوان: ص. ب 6202 الرباط المعاهد الرباط – المغرب
الهاتف: 59 / 58 17 77 0537 (00 212)
الفاكس: 45 58 77 0537 (00 212)
موقع الأنترنت:http://www.iav.ac.ma

Dédicace

À mes très chers parents Hassan et Hafida, aucune dédicace ne pourrait exprimer ma reconnaissance et ma gratitude pour vos affections inépuisables et vos précieux conseils. Aucun mot ne serait assez expressif pour traduire l'amour profond et la considération que je vous dois. J'espère que vous trouverez dans ce travail l'expression de ma gratitude infinie pour votre patience et votre confiance.

À mes chères sœurs Manal et Wissal, et mon cher frère Mohamed-Amine, grâce à vous, j'ai réalisé que la famille est sacrée. Cette dédicace serait pour moi, la meilleure façon de vous honorer et vous montrer à quel point vous avez été magnifiques. Qu'Allah, le tout puissant, vous préserve et vous accorde santé, bonheur et prospérité.

À mon cher oncle Mohammed, je n'oublierai jamais vos encouragements et votre soutien. Que ce travail soit un témoignage de l'amour que je porte pour vous.

À mes chers amis, ceux que je connais depuis l'enfance et ceux avec qui j'ai partagé les dernières années. Je n'oublierai jamais ces instants magiques. Ils seront gravés à jamais dans mon esprit. Je vous aime.

À mon cher professeur Rafika, je vous remercie de vos conseils et de votre présence à mes côtés durant tous ces mois de travail, et durant toutes ces années à l'institut. J'ai un énorme respect pour vous pas seulement en tant que professeur mais aussi la personne que vous êtes. Veuillez trouvez ici, professeur, l'expression de mes sincères remerciements.

À mon cher encadrant Gilles-Antoine, je vous remercie de votre patience, de tous vos précieux conseils, de votre écoute active et de votre disponibilité. Je n'aurai pas tant réussi si je n'avais pas reçu vos conseils. Veuillez croire à l'expression de ma profonde reconnaissance et de mon grand respect.

À tous ceux qui me sont chers, À toute personne qui m'a fait du bien un jour, je dédie ce travail.

Remerciements

J'aimerais tout d'abord exprimer mes sincères remerciements à ma chère encadrante madame **Hajji Rafika**, professeur chercheur au sein du département de Géodésie et de Topographie à l'Institut Agronomique et Vétérinaire Hassan II, pour la confiance qu'elle m'a témoignée et de m'avoir fait bénéficier tout au long de ce travail de sa grande compétence, de sa rigueur intellectuelle, de son dynamisme et de son efficacité certaine.

Je tiens également à remercier mon encadrant monsieur **Gilles-Antoine Nys**, doctorant et assistant à l'unité de géomatique de l'Université de Liège, qui s'est toujours montré à l'écoute et très disponible tout au long de la réalisation de ce mémoire. Je le remercie aussi pour sa générosité et ses conseils distillés.

Mes remerciements vont aussi à toute l'équipe du cabinet ESSADIQI, tout particulièrement à son directeur général monsieur **Essadiqi Abdelhak** et à madame **Abdjallah Dénalo Evrard**, pour l'accueil et le soutien logistique qui m'ont été offerts, et de n'avoir épargné aucun effort, ni moyen pour me permettre de réussir ce travail.

Je tiens aussi à présenter un grand remerciement à monsieur **Bouali Zakaria**, ingénieur geomètre-topographe, monsieur **Kharroubi Abderrazzaq** et monsieur **Ballouch Zouhair**, chercheurs à l'unité de géomatique de l'Université de Liège, pour l'accompagnement, l'aide et le temps accordé pour réussir ce travail.

Je suis très honoré à remercier les membres de jury d'avoir accordé de leur temps afin d'examiner ce projet et me faire part de leurs remarques pertinentes.

Je souhaite également rendre hommage à mes chers professeurs du département de la filière des sciences géomatiques et ingénierie topographique qui m'ont apporté leur aide et qui m'ont accompagné le long de mon cursus.

Finalement, je présente mes respectueux remerciements à toute personne qui a participé de près ou de loin à la réalisation de ce projet.

Résumé

Les modèles 3D de villes sont utilisés comme base pour le développement de jumeaux numériques de villes. Jusqu'à présent, la plupart des travaux se sont surtout concentrés sur les modèles de bâtiments. Pour les réseaux de transport, cette situation semble avoir changé au cours des dernières années, vu la disponibilité croissante des données routières. Cependant, cette évolution est relativement récente et la plupart des normes existantes se concentrent surtout sur une représentation linéaire des routes.

Ce projet a pour objectif le développement d'une approche pour la production des plateformes routières cohérentes géométriquement en trois dimensions, selon le module Transport de CityGML 3.0 et son encodage CityJSON.

L'approche suivie sert premièrement à modéliser l'espace routier en 3D, à un niveau de détails élevé (LoD2), sur la base des caractéristiques linéaires de la route extraites à partir d'un nuage de points. Le modèle 3D créé est par la suite traduit en fichier CityJSON valide géométriquement et conforme aux spécifications de la dernière version publiée. Toutes les classes du module Transport de CityGML 3.0 sont implémentés.

Ensuite, le jeu de données CityJSON généré est stocké dans une base de données orientée document qui est MongoDB. Les schémas de données sont définis préalablement à partir des schémas CityJSON.

Finalement, les données sont récupérées à partir de la base de données pour les mettre à disposition via une application web. Les fonctionnalités de Measur3D sont exploitées et des modifications sont apportées à la couche client, pour permettre la visualisation et l'inspection des surfaces sémantiques.

Mots clés : modèle 3D de ville, plateformes routières, CityGML, module Transport, CityJSON, base de données orientée documents, application web, surfaces sémantiques

Encadrant
Pr. Hajji Rafika

Co-encadrant
M. Gilles-Antoine Nys

Candidat
Yarroudh Anass

Abstract

3D city models are used as a basis of developing digital twins for cities. Most studies have mostly focused on the building modeling. For transportation networks, this situation seems to have changed in recent years, given the increasing availability of road data. However, this evolution is relatively recent and most of the existing standards are mostly focused on a linear representation of roads.

This project aims to develop an approach of producing a 3D geometrically consistent road models, according to the CityGML 3.0 Transportation module and its CityJSON encoding.

The first task is to produce a 3D model of the road space, at a high level of detail (LoD2), from the linear features extracted from a point cloud. The model is converted into a geometrically valid CityJSON file that satisfies the latest released specifications. All classes of the CityGML 3.0 Transportation module are implemented.

Then, the resulting CityJSON dataset is stored in a document-oriented database which is MongoDB. The data schemas are previously defined out of CityJSON schemas.

Finally, all data are retrieved from the database and made available through a web application. Measur3D functionalities are used and changes are made to the client layer to allow semantic surface visualization and inspection.

Keywords : 3D city model, road platforms, CityGML, Transportation module, CityJSON, document-oriented database, web application, semantic surfaces

Table des matières

Liste des figures

Liste des tableaux

Acronymes

3DCityDB 3D City Database
ACID Atomicity, Consistency, Isolation, Durability
API Application Programming Interface
BASE Basic Availability, Soft state, Eventual consistency
BSON Binary JSON
CRUD Create, Read, Update, Delete
CSF Cloth Simulation Filter
ETL Extract, Transform, Load
FME Feature Management Engine
GDF Geographic Data Files
GML Geography Markup Language
GNSS Global Navigation Satellite Systems
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
IMU Inertial Measurement Unit
ISO International Organization for Standardization
JSON JavaScript Object Notation
LASER Light Amplification by Stimulated Emission of Radiation
LAS LASer
LandInfra Land and Infrastructure
LiDAR Light Detection And Ranging
LoD Level of Detail
MERN MongoDB, Express, React, Node
MMS Mobile Mapping System
MNT Modèle Numérique de Terrain
NoSQL Not only SQL
OGC Open Geospatial Consortium
RCP ReCAP Project
RCS ReCAP Scan
REST Representational State Transfer
SGBDR Système de Gestion de Base de Données Relationnelle
SOR Statistical Outlier Removal
SQL Structured Query Language
UML Unified Modeling Language
URL Uniform Resource Locator
UUID Universally Unique Identifier
WGS84 World Geodetic System 1984
XLink XML Linking
XML Extensible Markup Language
YAML Yet Another Markup Language

Introduction générale

Contexte et motivation

Les modèles 3D de villes consistent en des représentations numériques de zones urbaines et de paysages en trois dimensions, notamment les bâtiments, les routes, la végétation et l'eau.

Auparavant, ces modèles étaient surtout utilisés à des fins de visualisation. Cependant, ils permettent une représentation fine des zones urbaines et sont donc progressivement utilisés pour de nombreuses applications dans les domaines d'urbanisme, du transport, de l'énergie et de l'environnement.

Ces modèles sont utilisés comme base pour le développement de jumeaux numériques de villes, représentant avec une grande précision les paysages et les zones urbaines ainsi que la dynamique de la ville en termes de processus et d'événements.

CityGML est la norme internationale de l'Open Geospatial Consortium (OGC) pour la représentation et l'échange de modèles 3D de ville. Il définit la géométrie tridimensionnelle, la topologie, la sémantique et l'apparence des objets les plus pertinents dans les zones urbaines. Les propriétés tant spatiales que sémantiques sont structurées en niveaux de détail.

Récemment publié, CityJSON est un standard d'interopérabilité utilisé pour l'échange d'information relative au contexte urbain bâti. Il s'agit d'un encodage JSON (JavaScript Object Notation) du modèle de données CityGML. Ce formalisme est axé sur la légèreté et la facilité de mise en œuvre de maquettes 3D. Il est considéré comme le nouveau pilier technologique qui permettra de construire les jumeaux géo-numériques de villes.

Problématique

Jusqu'à présent, la plupart des travaux se sont surtout concentrés sur les modèles de bâtiments. D'une part, en raison de leur rôle prépondérant dans le tissu urbain et, d'autre part, en raison du manque d'informations et de sources de données pour d'autres domaines thématiques (Beil & Kolbe, 2017).

Pour les réseaux de transport, cette situation semble avoir changé au cours des dernières années. Les informations fournies sur les routes sont de plus en plus disponibles. Cependant, cette évolution est relativement récente et la plupart des normes existantes se concentrent surtout sur une représentation linéaire des routes. Ainsi pour répondre à ce besoin, nous nous posons la question suivante :

Comment produire des plateformes routières 3D modélisées selon le standard CityGML et structurées de manière optimisée pour une gestion efficace au sein des maquettes urbaines 3D ?

Objectifs

Le présent travail s'inscrit dans le cadre de recherches en cours à l'Unité de Géomatique qui travaille sur la modélisation 3D, la structuration et l'optimisation des données spatiales 3D. Il rentrent également dans les axes de recherche mixte avec la filière des sciences géomatiques de l'Institut Agronomique et Vétérinaire Hassan II.

L'objectif principal est : le développement d'une approche pour la modélisation 3D standardisée et l'optimisation du stockage des infrastructures routières. La finalité du projet vise à mettre en place :

1. Une méthodologie pour créer des modèles 3D de route, cohérents géométriquement, en CityJSON ;
2. Leur structuration et leur stockage optimisés ;
3. Leur mise à disposition via une application web.

Plan du mémoire

La présent document est structuré en deux parties et cinq chapitres.

La première partie correspond à l'État-de-l'art qui est une synthèse des travaux et publications scientifiques relatifs au sujet. Il est subdivisé en deux chapitres :

Le premier chapitre est consacré à la modélisation et à la structuration des infrastructures routières. Les différentes normes routières, leurs modèles de données et leurs limites sont discutées. Ensuite, le module conceptuel CityGML 3.0 et son encodage CityJSON sont présentés par en se focalisant sur le module Transport et ses nouvelles fonctionnalités.

Le deuxième chapitre est dédié aux technologies de stockage et de visualisation des modèles 3D de ville. Les différents modèles de base de données sont discutés. Ensuite, les solutions de stockage et de visualisation des jeux de données CityJSON sont présentées et comparées.

La deuxième partie correspond au travail pratique du projet. Elle est structurée en deux chapitres :

Le quatrième chapitre présente le cadre méthodologique du projet. Il porte sur les outils de travail utilisés, notamment la technologie d'acquisition des données routières, les stations de traitement et les solutions logicielles. Ensuite, la méthodologie suivie pour la réalisation du projet est détaillée et argumentée.

Le dernier chapitre est consacré à l'implémentation de la méthodologie présentée et les résultats du traitement. Ainsi, les perspectives de recherche pour la continuité de ce travail sont détaillées.

Première partie

État de l'art

Chapitre 1

Modélisation et structuration des infrastructures routières

Introduction

Ce chapitre est consacré à la présentation des approches de modélisation et de structuration des infrastructures routières. Les différentes normes routières, leurs modèles de données et leurs limites sont discutées. Ensuite, le module conceptuel CityGML 3.0 et son encodage CityJSON sont présentés en se focalisant sur le module Transport et ses nouvelles fonctionnalités.

1.1 Modèles détaillés des routes et leurs applications

Les réseaux routiers comptent parmi les éléments d'infrastructure les plus importants des villes modernes. Selon X. Zhang et al. (2019), leur répartition a des répercussions importantes sur la répartition spatiale des activités socio-économiques.

Avec l'augmentation de l'information géographique 3D disponible, le potentiel d'utilisation de ces données pour diverses applications augmente également (Boersma, 2019). En effet, les modèles détaillés des routes jouent un rôle crucial dans diverses applications, notamment la navigation, la conduite autonome et la planification/gestion urbaine (X. Zhang et al., 2019).

Selon Beil & Kolbe (2017), la planification et la gestion des grands projets d'infrastructure peut se faire aujourd'hui numériquement. Des simulations visuelles peuvent servir de base à l'estimation des besoins et des coûts. Cela implique également l'entretien et la maintenance des routes. En fait, la maintenance en particulier a été identifiée comme un cas d'utilisation nécessitant à la fois une représentation linéaire du réseau et une représentation surfacique détaillée (Boersma, 2019). Ainsi, la connaissance de l'état des chaussées, par une cartographie des dégradations, combinée avec des représentations surfaciques des routes peut être utilisée pour évaluer les coûts des réparations prévus (Beil & Kolbe, 2017).

Les modèles détaillés des routes peuvent servir également pour des applications relatives aux véhicules. La navigation automobile est l'une des applications les plus connues qui s'appuie fortement sur les modélisations géométrique et sémantique des données spatiales. Les systèmes de navigation aident les chauffeurs à planifier leurs déplacements, à créer des itinéraires et les suivre à l'aide des instructions (Boersma, 2019).

D'ailleurs, la connaissance de la forme des objets de l'espace routier peut être d'une grande utilité pour les applications de la conduite autonome (Beil & Kolbe, 2017). Les informations sur les bords des routes peuvent être utilisées afin d'accroître la sécurité de conduite. Les véhicules connectés, en combinaison avec les données sur la longueur et la largeur de certaines sections de la route, peuvent être utilisés pour aider les conducteurs à effectuer des manœuvres de dépassement.

1.2 Modélisation des routes

A grande échelle, les routes sont représentées généralement avec des polygones. Il s'agit donc d'une représentation surfacique des routes. Il est possible d'en extraire les axes routiers pour avoir une représentation linéaire. L'ensemble des axes routiers connectés constitue un réseau routier.

Les réseaux routiers peuvent être stockés sous forme de *graphes*. Selon Boersma (2019), un graphe est une structure topologique de données topologiques qui modélise la connectivité entre les points. Ces points sont appelés sommets ou nœuds et les liaisons entre ces points sont dites arêtes. Bien que la topologie soit la caractéristique principale d'un graphe, on peut donner aux arêtes la géométrie du segment de route correspondant.

Il existe plusieurs façons de stocker les graphes :

1. *Liste de raccords :* elle contient une paire d'indices de nœuds pour chaque paire de nœuds raccordés. C'est une méthode simple de stockage mais elle peut être inefficace si on cherche un raccord spécifique, étant donné qu'il faut parcourir toute la liste.
2. *Matrice d'adjacence :* une matrice A de dimensions $n \times n$, où n correspond aux nombres de nœuds du graphe. $A_{ij} = 1$ lorsque i est adjacent à j, et zéro dans le cas contraire. Cette méthode permet de vérifier facilement l'adjacence mais lorsqu'un graphe est peu dense, la matrice devient un moyen très verbeux de stocker peu d'informations.
3. *Liste d'adjacence :* où chaque indice de nœuds pointe vers les indices des nœuds adjacents. Elle permet également des recherches faciles.

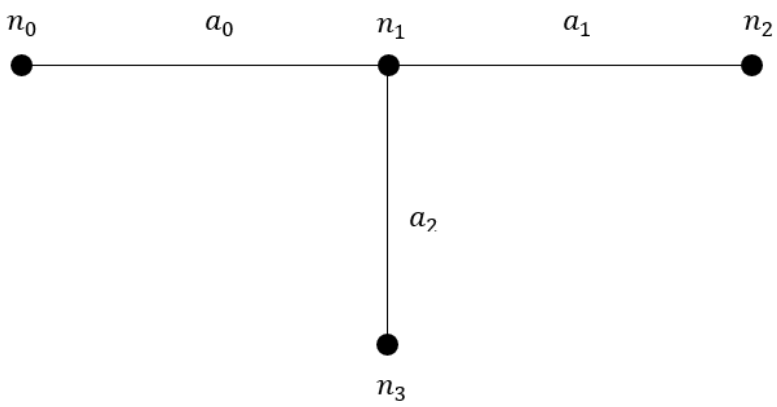


Figure 1.1 – Exemple de graphe avec quatre noeuds et trois arêtes

Le graphe représenté dans la Figure 1.1 contient quatre noeuds et trois arêtes. Il peut être stocké sous forme une liste d'arêtes, une matrice d'adjacence ou une liste d'adjacence :

— Liste d'arêtes : $E = \{(n_0, n_1), (n_1, n_2), (n_1, n_3)\}$

— Matrice d'adajcence : $A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

La première ligne de la matrice A correspond à l'adjacence du nœud n_0 par rapport à l'ensemble des nœuds du graphe. Par exemple, l'élément A_{01} représente l'adjacence entre n_0 et n_1, qui est dans ce cas vraie, et donc 1.

— Liste d'adjacence : $[(n_1), (n_0, n_2, n_3), (n_1), (n_1)]$

1.3 Complexité des intersections

Boersma (2019) définit les intersections comme des parties du réseau routier où plusieurs routes se rencontrent et où les usagers de la route peuvent choisir une direction parmi plusieurs.

Cette définition décrit les intersections comme des points critiques du réseau où les flux de circulation y convergent. Par conséquent, elles peuvent avoir des attributs que d'autres parties n'ont pas et sont considérées comme la partie la plus complexe des réseaux routiers.

Cette complexité est également due aux différentes classifications des intersections (Boersma, 2019). En fait, les intersections peuvent être classifiées en fonction du nombre de routes connectées et leur configuration : intersections en T avec deux branches, en X avec quatres branches, en Y avec trois branches, non circulaire avec plus que quatre branches et ronds-points (Figure 1.2).

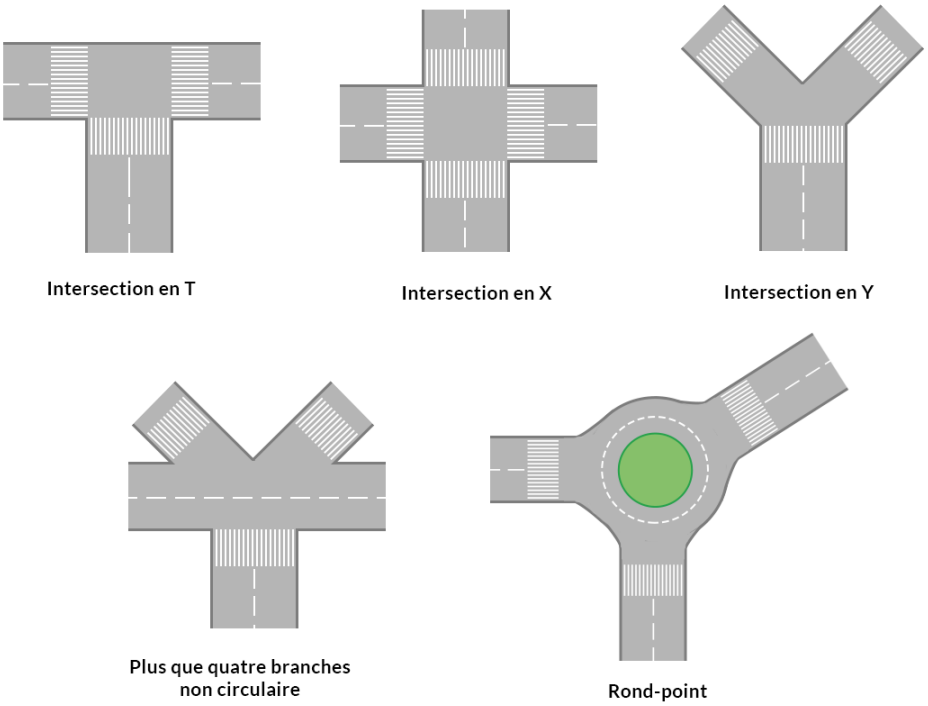


Figure 1.2 – Intersections classifiées en fonction de leur configuration

La classification peut se baser également sur la manière dont le flux du trafic est contrôlé : intersections planes sans feux de circulation, avec des feux de circulation et intersections non planes consistant en des passages supérieurs comme les jonctions d'autoroutes.

1.4 Acquisition des données routières

L'extraction de routes à partir de données recueillies par télédétection est un sujet de recherche qui a fait l'objet de nombreuses approches différentes (Clode et al., 2007). La technologie LiDAR (Light Detection And Ranging) fait partie de ces systèmes de télédétection utilisés pour la collecte des données routières. D'autres méthodes d'acquisition peuvent être utilisées, notamment la photogrammétrie aérienne. Cependant, nous nous intéressons aux systèmes LiDAR dans ce travail.

1.4.1 LiDAR

Selon Rahman (2021), l'utilisation de la technologie LiDAR s'est développée dans l'industrie du transport au cours des dernières décennies. Elle est devenue la principale technologie de collecte de données routières à grande échelle et à grande densité. Elle sert pour diverses applications de transport, avec plus de flexibilité, de rentabilité et de précision.

L'un des principaux avantages du LiDAR est que, contrairement aux méthodes traditionnelles de levé des routes, il peut produire un nuage de points 3D à haute résolution de la zone étudiée dans un temps très court, sans perturber la circulation (Rahman, 2021). Ceci permet également de minimiser les coûts d'acquisition (Clode et al., 2007).

Selon Klapper (2020), le LiDAR présente également comme avantage de collecter des données indépendamment des conditions météorologiques. En outre, les données collectées par un capteur LiDAR contiennent également des informations précises sur la hauteur (Clode et al., 2007).

Le fonctionnement de la technologie LiDAR est similaire au celui du RADAR (RAdio Detection And Ranging), en utilisant un Laser (Light Amplification by Stimulated Emission of Radiation) au lieu d'ondes radio. Néanmoins, ces deux domaines sont distincts puisque les ondes que ces deux dispositifs exploitent sont de nature différente.

Les principaux composants d'un système LiDAR sont un scanner laser, un système de positionnement par satellites et un système de navigation interne (Klapper, 2020).

Scanner laser

Selon Klapper (2020), le scanner laser est composé de deux éléments principaux, l'émetteur et le récepteur.

Il mesure la distance des objets cibles à partir du temps de réflexion d'une impulsion lumineuse (Rahman, 2021) ou de la différence de phase entre les formes d'ondes envoyée, par l'émetteur, et reçue par le récepteur.

GNSS et IMU

Pour un géoréférencement direct des nuages de points, les systèmes LiDAR mobiles et aéroportés intègrent généralement un capteur GNSS (Global Navigation Satellite Systems) qui fournit des informations sur la localisation (X,Y,Z) des points dans un système global (Rahman, 2021).

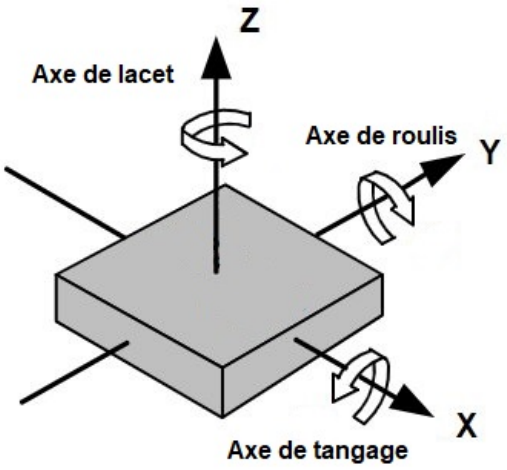


Figure 1.3 – Axes de rotation de la plateforme d'acquisition

La centrale inertielle IMU (Inertial Measurement Unit) fournit des informations sur l'orientation (roulis, tangage et lacet) de la plateforme d'acquisition (Figure 1.3). Par conséquent, les nuages de points sont géoréférencés et les coordonnées sont transformées du système de coordonnées scanner à un système de coordonnées global, particulièrement WGS84 (World Geodetic System 1984).

Selon Cheng et al. (2018), Il existe quatre principaux systèmes LiDAR : Scanner Laser Spatial (SLS), Scanner Laser Aéroporté (ALS), Scanner Laser Mobile (MLS) et Scanner Laser Terrestre (TLS), divisés en fonction de la plateforme (Tableau 1.1).

Malgré les avantages précités dont disposent les techniques d'acquisition de données LiDAR, elles comportent également certaines limites (Gneeniss, 2013) qui peuvent être résumées comme suit :

— La précision planimétrique des données LiDAR reste inférieure à celle fournie par les méthodes traditionnelles de levé ;
— Les données LiDAR manquent d'informations sémantiques ;
— Des caractéristiques géométriques telles que les angles des bâtiments, les arêtes, les crêtes et les lignes de rupture ne sont pas obtenues directement depuis les données LiDAR, mais après avoir effectué des processus de segmentation ou de classification.

1.4.2 Systèmes de cartographie mobile

Comme indiqué dans la section 1.4.1, il existe différents systèmes d'acquisition de données LiDAR. Leurs domaines d'application varient en fonction des caractéristiques de chaque système (Tableau 1.1).

La cartographie des routes utilise souvent un système LiDAR monté sur un véhicule. Il s'agit d'un système de cartographie mobile.

Selon Jaakkola (2015), un système de cartographie mobile, ou MMS (Mobile Mapping System), se compose, au minimum, d'une plateforme mobile, d'un système de positionnement (GNSS/IMU) et d'un scanner laser et il peut également inclure d'autres capteurs électro-optiques d'acquisition de données. Le véhicule en mouvement peut être équipé d'une caméra (Cho et al., 2017) de prise de vues.

Les environnements routiers sont l'une des cibles les plus naturelles des systèmes de cartographie mobile, vu la géométrie des mesures et les courtes distances impliquées qui sont favorables à la qualité des données (Jaakkola, 2015).

En fait, le MMS permet de mesurer la géométrie de la route, y compris la chaussée et les bordures de trottoir, ainsi que les marquages routiers peints sur la surface de la route. Ainsi, les objets situés le long de la route peuvent être levés simultanément.

Tableau 1.1 – Comparaison des systèmes LiDAR montés sur différentes plateformes (Cheng et al., 2018)

Plateforme	**Système**	**Perspective**	**Plage de scan**	**Densité**	**Applications**
Aéroportée	ALS	Vue d'en haut	Surfacique	Relativement faible	Cartographie du terrain, levés forestiers, zones urbaines en 3D
Véhicule	MLS	Vue de côté	Bande	Dense	Cartographie routière, zones urbaines en 3D
Trépied	TLS	Vue de côté	Ponctuelle	Dense	Surveillance des déformations
Satellite	SLS	Vue d'en haut	Surfacique	Faible densité	Levés forestiers, mesures atmosphériques, surveillance de la neige

1.5 Aperçu sur les normes routières

Il existe plusieurs normes et directives traitant des possibilités de représenter les routes dans différents types des modèles. Cependant, la plupart de ces normes se concentrent sur une représentation linéaire ou paramétrique des routes (Beil & Kolbe, 2017). Nous présentons dans les sections suivantes les différentes normes adoptées pour la structuration des routes :

1.5.1 Geographic Data Files

GDF 5.0 est une norme ISO (International Organization for Standardization) publiée en 2011 (ISO 14825, 2011). Il s'agit d'un modèle de données et d'un format d'échange pour les données structurées des réseaux routiers, qui sont utilisées dans la navigation automobile (Beil et al., 2020).

Dans GDF, les objets de transport sont modellisés en trois catégories, du niveau 0 au niveau 2. Ces niveaux ne correspondent pas aux LoDs de CityGML (Boersma, 2019).

Une représentation de niveau 0 serait constituée d'éléments topologiques de base tels que des nœuds, des arêtes ou des faces, mais ces éléments ne sont pas disponibles pour les représentations de réseaux routiers (Beil & Kolbe, 2017).

Des entités simples telles que les éléments routiers et les carrefours constituent les objets du niveau 1. Un carrefour peut être défini comme un endroit où trois éléments routiers ou plus se rencontrent (Boersma, 2019).

Au niveau 2, les entités simples du niveau 1 sont regroupées en une entité de niveau supérieur (Beil & Kolbe, 2017). Le réseau routier est composé de routes et des intersections. Les routes sont constituées des éléments routiers du niveau 1 et peuvent également combiner plusieurs chaussées.

1.5.2 OpenDrive

Selon Beil et al. (2020), OpenDrive est un format de données libre, encodé en XML (eXtensible Markup Language), et qui permet de stocker les données du réseau routier. La dernière spécification OpenDrive 1.7.0 a été publiée comme une norme ASAM (Association for Standardisation of Automation and Measuring systems) en 2021 par VIRES Simulationstechnologie GmbH.

Un fichier OpenDrive est principalement utilisé pour des simulations de conduite, où les données du réseau routier interagissent avec d'autres objets visualisés dans la simulation (Boersma, 2019).

Les données stockées dans le fichier OpenDRIVE décrivent la géométrie de la route avec des lignes de référence. Celles-ci consistent en des séquences de primitives géométriques. Ces routes sont donc soit liées par des liens successeurs/prédécesseurs, soit par des jonctions. Dans le cas d'une liaison simple entre deux routes, une liaison standard peut être utilisée. Des jonctions sont nécessaires si une route entrante peut se connecter à plusieurs routes consécutives (Beil & Kolbe, 2017).

1.5.3 LandInfra

LandInfra, ou Land and Infrastructure, est une norme OGC (Open Geospatial Consortium) approuvée et publiée en 2016. Elle définit des concepts pour des applications d'infrastructures terrestres et de génie civil.

La classe de base de LandInfra est *Facility*, qui correspond à une installation d'infrastructure.

La norme s'appuie sur la série de normes d'information géographique ISO 19100. Elle couvre divers domaines définis par des classes d'exigences. Les plus pertinentes en termes de modélisation

des rues sont *Alignment* et *Road* (Beil et al., 2020).

L'alignement peut être défini comme un élément de positionnement qui fournit un système de référence linéaire pour le positionnement des éléments physiques. Pour les routes, il existe généralement un alignement horizontal pour l'axe routier. Dans le cas d'une route à double chaussées, il devrait y avoir des alignements séparés, mais ils peuvent également partager un alignement de référence au centre approximatif de toute la route (Beil & Kolbe, 2017).

Selon Gruler et al. (2016), une route peut être représentée de plusieurs façons dans la classe *Road*, notamment *RoadElements*, StringLines 3D, et les surfaces et couches 3D, ainsi que des collections de ces éléments.

LandInfra ne dispose pas d'un concept de niveau de détail, ni d'une classe distincte pour les intersections (Boersma, 2019).

1.5.4 RoadXML

RoadXML est une norme de données routières et un format de fichier ouvert, encodé en XML, utilisée généralement pour la simulation de conduite. La dernière version 3.0.0 a été publiée en avril 2020.

Selon Chaplier et al. (2010), RoadXML offre une description multi-couche de l'environnement pour un accès rapide aux données pour les applications en temps réel :

Couche topologique : l'emplacement de l'élément et ses liaisons avec le reste du réseau ;
Couche logique : la signification des éléments dans l'environnement de la route ;
Couche physique : les propriétés physiques d'un élément routier, par exemple la surface de la route ;
Couche visuelle : la géométrie de l'élément et sa représentation 3D.

Dans un fichier RoadXML, le réseau routier est construit à partir de différents sous-réseaux (en anglais, *Subnetworks*), dont chaque sous réseau est constitué de voies de circulation et des intersections utilisées pour connecter les éléments de la route. (Boersma, 2019)

Les sous-réseaux sont reliés entre eux par des jonctions dites *Subnetworksjunctions*. RoadXML ne dispose également pas de concept de niveaux de détail.

1.5.5 ASB et OKSTRA

ASB et OKSTRA ont été publiés par le ministère fédéral allemand des transports et de l'infrastructure numérique comme des normes routières, et sont utilisés par les administrations allemandes pour collecter et stocker des informations uniformes sur les routes et les infrastructures de transport (Beil et al., 2020).

Selon Demirel (2002), il s'agit d'un effort continu de normalisation de l'échange d'informations routières entre les administrations routières de l'Etat.

ASB décrit les structures des objets d'un point de vue technique, tandis que OKSTRA se concentre sur les descriptions formelles à l'aide de schémas de données et de représentations UML des objets de la route (Beil & Kolbe, 2017).

1.5.6 CityGML

Depuis sa première publication en 2008, CityGML est une norme de l'OGC qui définit un modèle conceptuel et un format d'échange pour la représentation, le stockage et l'échange de modèles de villes en 3D.

Elle définit la manière de représenter les objets 3D couramment trouvés dans une ville, entre autres les bâtiments, les routes, la végétation et l'eau, et les relations hiérarchiques entre eux. Il contient également différents niveaux de détail (LoD) pour les objets 3D.

Dans la nouvelle version CityGML 3.0, les infrastructures routières sont représentées par la sous-classe *Road* de la classe *AbstractTransportationSpace* du module *Transport*. Les

1.5.7 Discussion

La plupart des normes présentées ci-dessus se concentrent sur une représentation linéaire des routes, à l'exception de LandInfra et de CityGML dont la géométrie est surfacique. Cependant, contrairement à CityGML, LandInfra ne dispose pas d'un concept de niveau de détails, ni d'une classe distincte pour les intersections. Ainsi, il n'est pas possible de différencier les représentations multi-échelles des modèles sémantiques de villes en 3D.

Pour des applications telles que la navigation et les simulations de trafic, une représentation linéaire est généralement suffisante. Cependant, certaines applications peuvent exiger une représentation détaillée de l'espace routier. Dans ce cas, une représentation linéaire sera insuffisante et peut entraîner certains problèmes (Beil & Kolbe, 2017).

En fait, une représentation surfacique est souvent nécessaire pour montrer les détails géométriques tels que les arrêts de bus, les largeurs et les différents espaces de transport, ce qui peut être utile pour un grand nombre d'applications différentes (Figure 1.4).

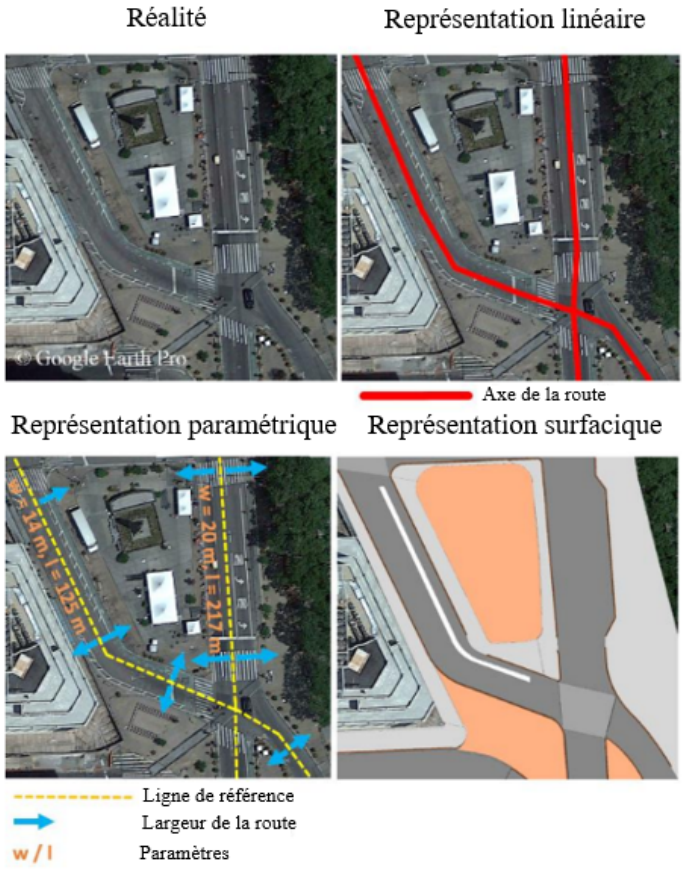


Figure 1.4 – Comparaison des différents types de représentation (Beil & Kolbe, 2017)

1.6 Modélisation de l'espace routier avec CityGML

La norme CityGML constitue le pilier technologique des modèles 3D de villes. La version 1.0 a été adoptée comme norme officielle de l'OGC en août 2008. En 2012, une deuxième version CityGML 2.0 a été approuvée. La version actuelle CityGML 3.0 a été publiée en 14 septembre 2021.

Alors que les versions précédentes standardisaient un format d'échange GML (Geography Markup Language), CityGML 3.0 standardise le modèle d'information sous-jacent et peut donc être mis en œuvre dans une variété de technologies autres que GML. Nous présentons dans les sections suivantes les spécifications de CityGML 2.0 et la nouvelle version CityGML 3.0 :

1.6.1 CityGML 2.0

La norme CityGML 2.0, publiée par l'OGC en 2012, définit de nombreuses classes et relations pour de nombreux objets urbains thématiques en ce qui concerne leurs propriétés spatiales, sémantiques et visuelles.

L'espace routier est représenté par le module *Transport* (en anglais, *Transportation*). Il consiste en une classe principale *TransportationComplex* et peut être thématiquement subdivisé en 4 sous-classes : Route (*Road*), Place (*Square*), Voie (*Track*) et Chemin de fer (*Railway*) (Beil & Kolbe, 2017). Les routes sont représentées comme *TransportationComplex*, qui est ensuite subdivisée en *TrafficAreas* et *AuxiliaryTrafficAreas* :

TrafficAreas correspondent aux éléments importants en termes d'utilisation du trafic, comme les voies de circulations, les zones piétonnes et les zones de cyclistes. En revanche, *AuxiliaryTrafficAreas* représentent les objets qui ont une importance mineure pour le transport, par exemple, les bordures de trottoir, les marquages routiers et les bandes de gazon (Figure 1.5).

Chaque *TransportationComplex* a les atrributs *class*, *function* et *usage*. L'attribut *class* décrit la classification de l'objet. *function* correspond au but de l'objet (autoroute, route de campagne, ou aéroport), tandis que *usage* peut être utilisé si l'usage de l'objet est différent de sa fonction.

En outre, *TrafficAreas* et *AuxiliaryTrafficAreas* peuvent avoir les attributs *class*, *function*, *usage* et *surfaceMaterial*. L'attribut *surfaceMaterial* spécifie le type de revêtement (asphalte, béton, gravier, terre, herbe, etc.) (Gröger et al., 2012).

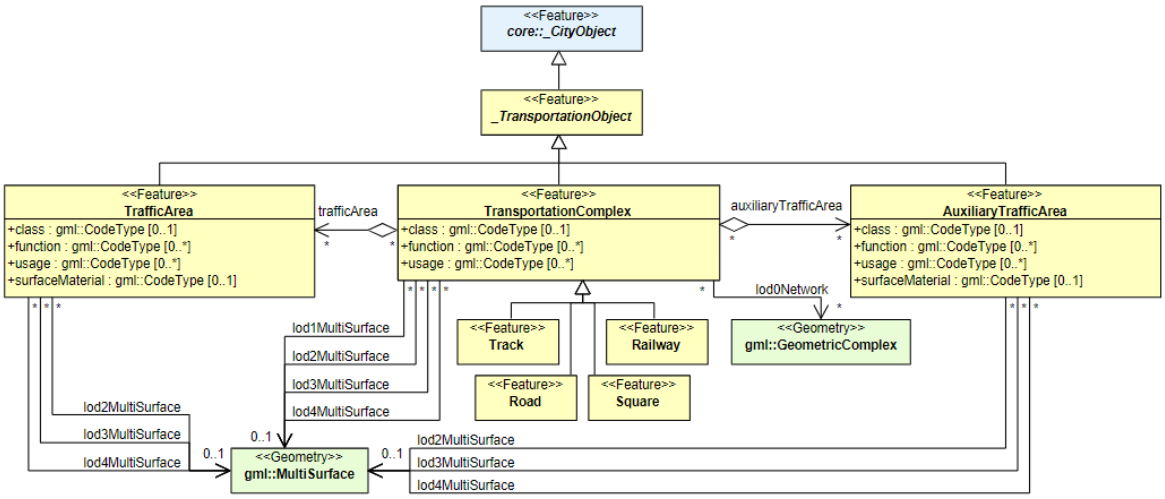


Figure 1.5 – Diagramme UML du modèle de Transport de CityGML 2.0

1.6.2 Niveaux de détail

Le concept du niveau de détail (LoD) de la norme CityGML 2.0 de l'OGC permet de différencier les représentations multi-échelles des modèles sémantiques de villes en 3D.

Comme illustré dans la Figure 1.6, les complexes de transport sont modélisés en cinq niveaux de détail :

LoD0. Les entités de transport sont représentées comme un réseau linéaire.

LoD1. Les entités de transport sont représentées comme des surfaces 3D, reflétant la forme réelle de l'objet, et pas seulement son axe.

LoD2-LoD4. Un complexe de transport est subdivisé thématiquement en *TrafficAreas* et en *AuxiliaryTrafficAreas*.

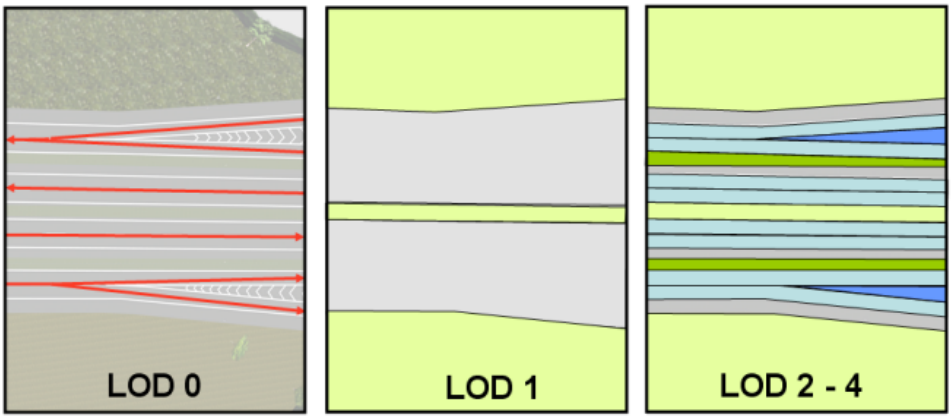


Figure 1.6 – Niveaux de détails des complexes de transport dans CityGML 2.0

Selon Beil & Kolbe (2017), le module Transport de CityGML 2.0 présente certaines limites à identifier :

1. Dans LoD0, l'axe utilisé dans la représentation linéaire du réseau routier n'est pas spécifié (axe de route, de voie de circulation, etc.) ;
2. La possibilité de représenter les intersections comme des nœuds dans LoD0 est impliquée par la classe géométrique définie GeometricComplex mais n'est pas spécifiée autrement ;
3. Les routes modélisées dans LoD1 doivent être représentées par des géométries Multi-Surface. Il n'est pas clair quels objets de l'espace routier doivent être inclus dans cette représentation ;
4. La segmentation du TransportationComplex en plusieurs objets est possible mais n'est pas spécifiée davantage. Jusqu'à présent, il est possible de représenter l'ensemble des rues d'une ville par un seul objet Route (Road) ;
5. Les intersections et les ronds-points ne sont pas représentés explicitement.

1.6.3 Nouveautés du CityGML 3.0

Depuis la publication de CityGML 2.0, plusieurs demandes de modification et d'amélioration du module conceptuel CityGML ont été soumises à l'OGC. La nouvelle version CityGML 3.0 a été publiée en septembre 2021.

Ces modifications concernent le module de base de CityGML, ainsi que d'autres modules qui ont été mis à jour pour fonctionner avec le nouveau module de base (Figure 1.7).

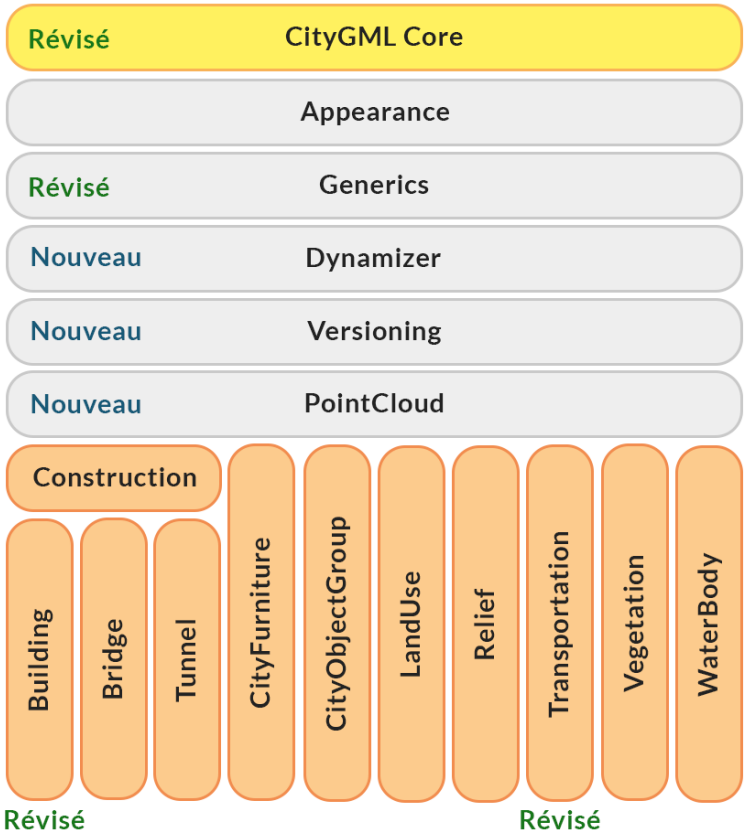


Figure 1.7 – Aperçu sur le module de CityGML 3.0

Dans les sections suivantes, les nouveautés du module de base et du module Transport de CityGML 3.0 seront présentés :

1.6.3.1 Concept d'Espace

Dans CityGML 3.0, une distinction sémantique claire des caractéristiques spatiales est introduite en faisant correspondre tous les objets de la ville aux concepts sémantiques d'espaces et de limites d'espace.

Selon Kutzner et al. (2020), un *Espace* est une entité d'étendue volumétrique dans le monde réel. A titre d'exemple, les bâtiments, les cours d'eau, les arbres et les espaces de circulation ont une étendue volumétrique et donc seront modélisés comme des espaces, ou plus précisément, comme des sous classes spécifiques de la classe *Espace*.

Par contre, une *Limite d'espace* est une entité ayant une étendue surfacique dans le monde réel. Ces limites délimitent et relient les espaces. Par exemple, les surfaces des murs et des toits qui délimitent les bâtiments. (Kutzner et al., 2020)

Il faut également distinguer entre un espace *Physique* et *Logique* :

— Un espace physique est dit physique s'il est entièrement ou partiellement délimités par des objets physiques. A titre d'exemple, les bâtiments sont des espaces physiques étant donné qu'ils sont délimités par des murs et des dalles ;
— Un espace logique n'est pas nécessairement délimité par des objets physiques mais définis en fonction de considérations thématiques. Les quartiers des villes qui sont délimités

par des frontières administratives virtuelles extraduées verticalement sont des espaces logiques.

Les espaces physiques, à leur tour, sont classés en espaces *occupés* et *non occupés* :

— Les espaces occupés représentent les objets physiques qui occupent l'espace dans l'environnement urbain. L'espace bloqué par le bâtiment, à titre d'exemple, ne peut plus être utilisé pour circuler en voiture ou pour placer un arbre ;
— Dans un espace non occupé, les entités volumétriques physiques n'occupent pas d'espace dans l'environnement urbain, c'est-à-dire qu'aucun espace n'est bloqué parces objets physiques. C'est le cas des chambres d'un bâtiment.

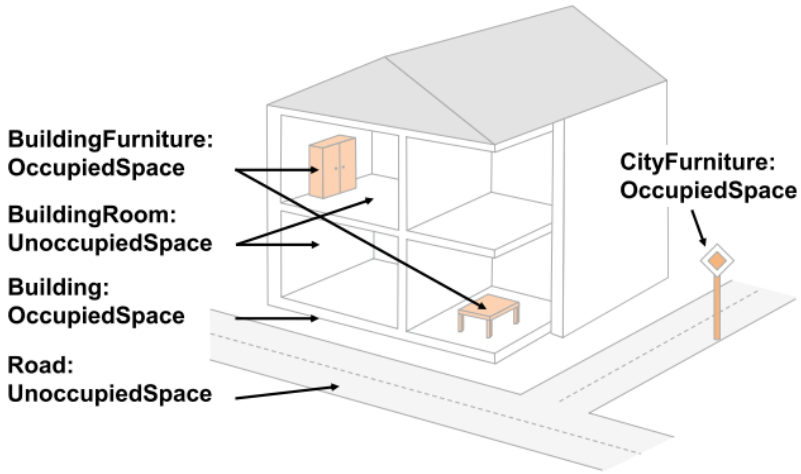


Figure 1.8 – Espaces occupés et non occupés (Kutzner et al., 2020)

1.6.3.2 Module Transport

Le module Transport est parmi les modules thématiques qui ont été révisés dans CityGML 3.0 pour répondre aux lacunes et insuffisances de CityGML 2.0.

Les objets de Transport sont maintenant définis comme des sous-classes concrètes de la classe abstraite *TransportationSpace*. Ces objets peuvent être subdivisés en *sections*, qui peuvent être des segments réguliers de route, des zones d'intersection ou des ronds-points (Figure 1.9).

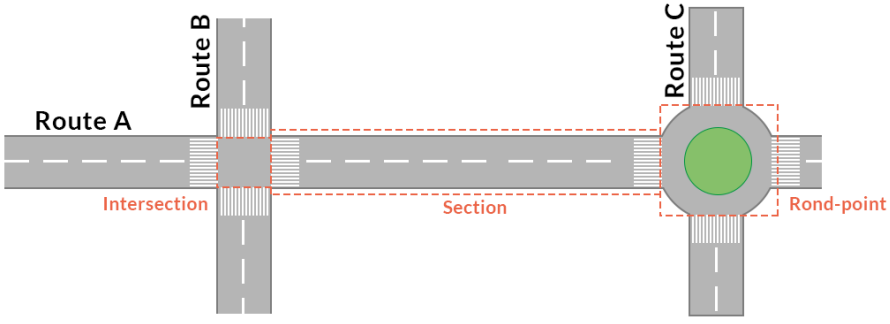


Figure 1.9 – Illustration des sections d'une route

Comme dans CityGML 2.0, les objets de transport peuvent également être subdivisés en *Traf-*

ficAreas et *AuxiliaryTrafficAreas*. Selon Kutzner et al. (2020), pour adapter la sémantique du module Transport au concept d'espace de CityGML 3.0, les classes *TrafficSpace* et *Auxiliary-TrafficSpace* ont été introduites, les deux zones représentant désormais les limites inférieures des deux espaces (Figure 1.10).

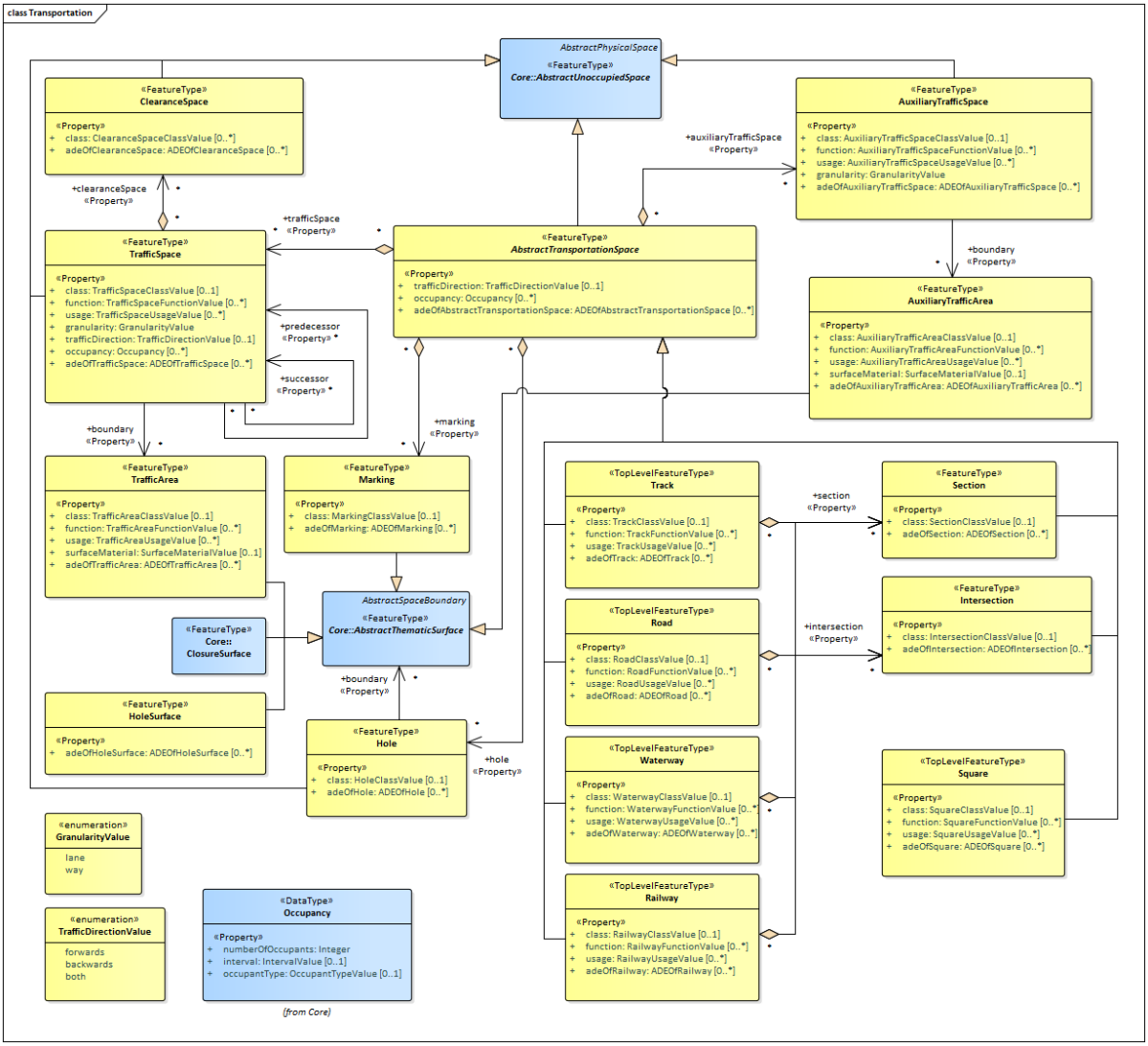


Figure 1.10 – Diagramme UML du modèle de transport dans CityGML 3.0

Chaque espace de trafic (*TrafficSpace*) peut avoir un espace de dégagement (*ClearanceSpace*) facultatif (Figure 1.11).

Le concept de niveaux de détails (LoDs) a été redéfini dans la version 3.0. Ainsi, à l'encontre de CityGML 2.0, les objets de l'espace routier du module Transport de CityGML 3.0 sont modélisés en 4 niveaux de détails (LoD0-LoD3). (Beil & Kolbe, 2017)

Dans LoD0, les routes sont modélisées par des lignes uniquement. À partir de LoD1, l'espace routier peut être modélisé par des objets linéaires et/ou surfaciques respectivement. En plus de l'axe routier, des représentations linéaires pour les zones piétonnes et les pistes cyclables deviennent possibles dans LoD2.

Les représentations LoD3 contiennent enfin un objet *TrafficSpace* pour chaque voie de circulation individuelle (Figure 1.12).

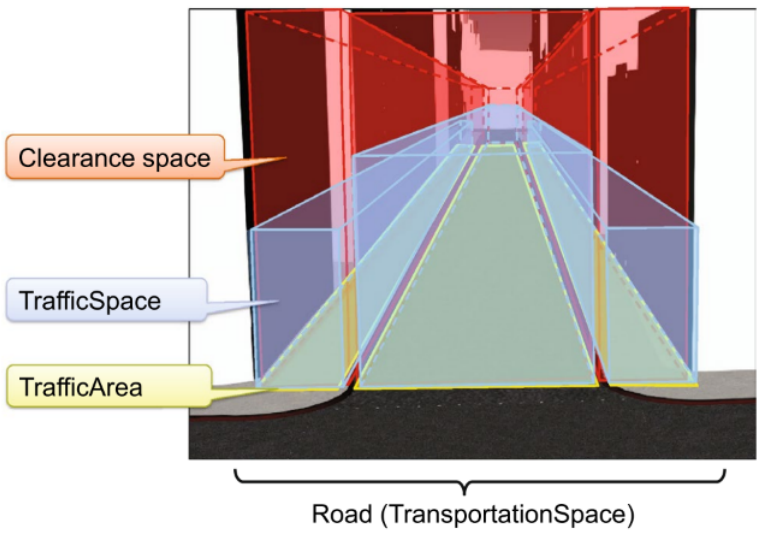


Figure 1.11 – Représentation d'une route dans CityGML 3.0 (Kutzner et al., 2020)

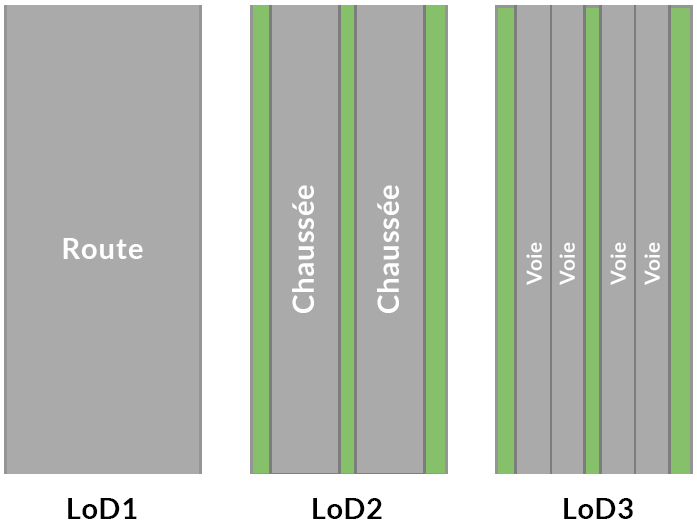


Figure 1.12 – Représentation surfacique des routes dans LoD1-3

Selon Kutzner et al. (2020), une classe *Waterway* a également été introduite comme une nouvelle sous-classe de la classe *TransportationSpace*. En outre, la nouvelle classe *Marking* permet d'ajouter des marquages routiers à la surface de la route et les classes *Hole* et *HoleSurface* peuvent être utilisées pour représenter, par exemple, des dégradations à la chaussée ou des trous, y compris leurs surfaces (Figure 1.10).

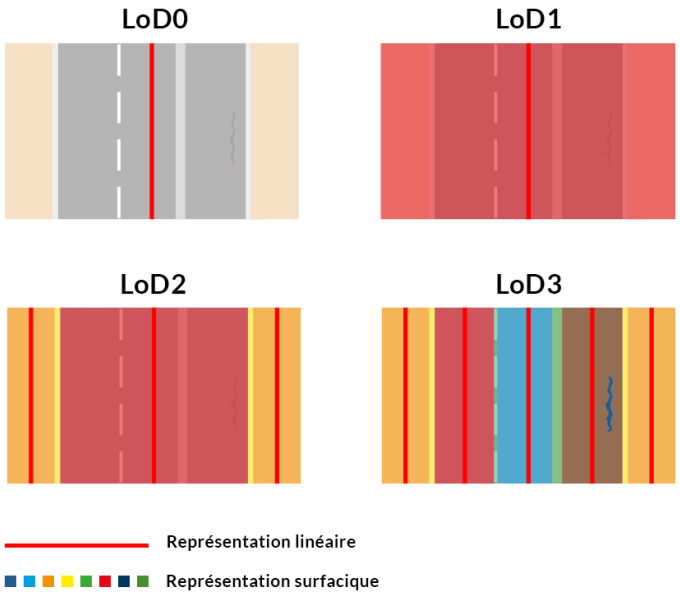


Figure 1.13 – Représentations linéaires et surfaciques dans chaque LoD (Beil & Kolbe, 2017)

1.6.4 Principales critiques à l'égard de CityGML

Les fichiers CityGML sont connus pour être très difficiles à examiner et à en extraire des informations. Selon Ledoux et al. (2019), ils présentent des problèmes à trois niveaux :

XML. Une critique reconnue du XML concerne le volume considérable de balises qu'il peut contenir, entraînant une perte de lisibilité pour les utilisateurs et une augmentation inutile du volume de transfert de données.

GML. Le principal problème avec GML est qu'il existe de nombreuses façons différentes de stocker la même géométrie. Le grand nombre de variations possibles signifie qu'un développeur doit déterminer toutes les variations possibles pour chaque type de géométrie et écrire du code pour les gérer de manière appropriée.

CityGML. Il s'appuie sur XML et GML, et il hérite donc de la plupart des avantages et des inconvénients de ces formats. D'autres aspects de CityGML sont également problématiques :

1. Des fichiers volumineux ;
2. La hiérarchie d'un seul bâtiment simple peut devenir assez profonde, ce qui se traduit par de nombreuses classes imbriquées hiérarchiquement dans le XML ;
3. Les surfaces peuvent être stockées de nombreuses manières différentes (similairement au stockage d'une géométrie en GML) ;
4. L'utilisation de GML signifie que toutes les géométries 3D sont des entités simples, ce qui signifie qu'aucune topologie n'est stockée.
5. Les objets d'un même fichier peuvent en théorie être d'un Système de Coordonnées de Référence différent. Cela signifie qu'un logiciel CityGML standard doit contenir des bibliothèques de projection.

De ce fait, il manque toujours de parsers JavaScript complets pour CityGML (nécessaires pour échanger et traiter des fichiers sur le web), ce qui rend très difficile, l'échange et le traitement efficaces de modèles CityGML sur le web.

1.6.5 CityJSON

CityJSON est devenu un standard communautaire de l'OGC depuis sa publication au 13 août 2021. Il s'agit d'un modèle conceptuel et un format d'échange des données 3D qui implémente la quasi-totalité du modèle de données CityGML. La version actuelle est CityJSON 1.1.2.

JSON est généralement préféré par les développeurs. Cette préférence est principalement due au fait que JSON est beaucoup plus simple que XML, étant donné que JSON est un format de données alors que XML est un langage de balisage, et qu'il est donc beaucoup plus facile de développer un logiciel pour JSON que pour XML. (Ledoux et al., 2019)

Selon Ledoux & Dukai (2022), un objet CityJSON minimal valide doit contenir les propriétés suivantes :

```
{
    "type": "CityJSON",
    "version": "1.0",
    "CityObjects": {},
    "vertices": [],
    "appearance": {}
}
```

Un objet CityJSON complet se présentera comme suit (Ledoux et al., 2019) :

```
{
    "type": "CityJSON",
    "version": "1.1",
    "extensions": {},
    "transform": {
        "scale": [1.0, 1.0, 1.0],
        "translate": [0.0, 0.0, 0.0]
    },
    "metadata": {},
    "CityObjects": {},
    "vertices": [],
    "appearance": {},
    "geometry-templates": {}
}
```

Dans son article, Ledoux et al. (2019) décrit les spécifications d'un fichier CityJSON :

Les objets de la ville sont stockés dans la propriété *CityObjects*. Il s'agit d'un dictionnaire dont les propriétés (clés) correspondent aux identifiants des objets de la ville.

```
"CityObjects": {
    "id-1": {
        "type": "Building",
        "geographicalExtent": [ 84710.1, 446846.0, -5.3, 84757.1,
           446944.0, 40.9 ],
        "attributes": {
          "measuredHeight": 22.3,
          "roofType": "gable",
          "owner": "Elvis Presley"
        },
        "children": ["id-2"],
        "geometry": [{...}]
    },
    "id-2": {
```

```
        "type": "BuildingPart",
        "parents": ["id-1"],
        "children": ["id-3"],
        ...
    }
```

Le schéma de données CityGML a été aplati et toutes les hiérarchies ont été supprimées. L'accès aux objets de la ville peut se faire directement par leurs identifiants. Ces objets de même structure contiennent au minimum une propriété *geometry* et une autre *attributes* où sont stockés les attributs.

Selon Ledoux & Dukai (2022), un objet *geometry* peut avoir comme *type* l'une des valeurs suivantes : MultiPoint, MultiLineString, MultiSurface, CompositeSurface, Solid, MultiSolid, CompositeSolid ou GeometryInstance.

Les données sémantiques sont également stockées dans la propriété *geometry*. Les surfaces sémantiques sont déclarées dans une liste *surface*, puis une liste *value* relie chaque surface à sa sémantique correspondante.

```
{
    "type": "MultiSurface",
    "lod": "2",
    "boundaries": [
        [[0, 3, 2, 1]],
        [[4, 5, 6, 7]],
        [[0, 1, 5, 4]],
        [[0, 2, 3, 8]],
        [[10, 12, 23, 48]]
    ],
    "semantics": {
        "surfaces" : [
            {
                "type": "WallSurface",
                "slope": 33.4,
                "children": [2]
            },
            {
                "type": "RoofSurface",
                "slope": 66.6
            },
            {
                "type": "+PatioDoor",
                "parent": 0,
                "colour": "blue"
            }
        ],
        "values": [0, 0, null, 1, 2]
    }
}
```

La propriété *boundaries* contient les primitives géométriques définies par leurs sommets. Contrairement au CityGML, les coordonnées des sommets d'une primitive géométrique sont stockées dans une liste distincte dans la propriété *vertices* et les primitives géométriques font référence à la position d'un sommet dans cette liste. C'est le même mécanisme d'indexation utilisé dans le format Wavefront OBJ.

La propriété *type* contient le type d'un objet de la ville (Figure 1.14). Selon Ledoux & Dukai (2022), ces objets peuvent appartenir au :

— 1er niveau : des objets de la ville qui peuvent exister par eux-mêmes
— 2ème niveau : des objets de la ville qui doivent avoir des *parents* pour exister

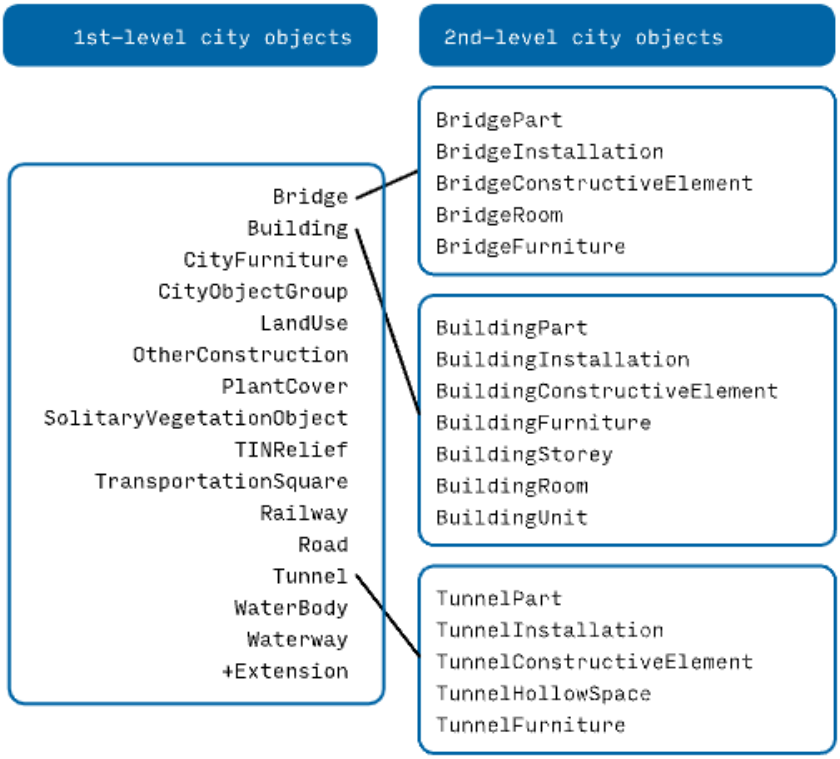


Figure 1.14 – Différents objets de la ville (Ledoux & Dukai, 2022)

Ainsi, les propriétés *parents* et *children* peuvent exister lorsque les objets de la ville sont associés les unes aux autres.

CityJSON stocke également les métadonnées dans la propriété *metadata*. Ces métadonnées sont développées conformément à la norme ISO 19115 et contiennent les informations suivantes :

```
"metadata": {
      "geographicalExtent": [],
      "identifier": "",
      "pointOfContact": {
            "contactName": "",
            "contactType": "",
            "role": "",
            "phone": "",
            "emailAddress": "",
            "website": "",
            "address": ""
      },
      "referenceDate": "",
      "referenceSystem": "",
      "title": ""
}
```

Le stockage d'attributs de métadonnées supplémentaires conformes à la norme ISO 19115 peut se faire avec une extension dite *MetadataExtended*. (Ledoux & Dukai, 2022)

La compression des fichiers CityJSON est également prise en charge. Ceci peut se faire de plusieurs manières. L'indexation des sommets permet de réduire la taille des fichiers. Les sommets sont généralement partagés entre plusieurs surfaces et leur répétition peut être coûteuse en termes d'espace de stockage.

Il est également possible de compresser les fichiers CityJSON si les coordonnées des sommets sont transformées en valeurs entières. En fait, la propriété *transform* permet de stocker la matrice de transformation pour obtenir les valeurs des coordonnées originales. Elle contient un facteur échelle (*scale*) et une translation (*translate*). Par exemple :

```
"transform": {
    "scale": [0.001, 0.001, 0.001],
    "translate": [442464.879, 5482614.692, 310.19]
}
```

L'utilisation des modèles de géométrie (*geometry-template*) permet de compresser les fichiers CityJSON davantage étant donné que certains objets ne doivent être déclarés qu'une seule fois. Chaque modèle peut être réutilisé : un objet de la ville peut avoir une géométrie de type *GeometryInstance* qui définit la position de l'objet (X,Y,Z), un lien vers le modèle de géométrie et une matrice de transformation.

1.6.6 Module Transport dans CityJSON

Le module Transport définit quatre types d'objets : *Road*, *Railway*, *Waterway* et *TransportSquare*.

Les classes *Section*, *Intersection* et *Track* de CityGML sont absentes car elles peuvent être traitées avec des attributs spécifiques. Cependant, les classes *TrafficArea*, *AuxiliaryTrafficArea*, *Marking* et *Hole* sont implémentées comme des surfaces sémantiques : la surface représentant une route doit être divisée en sous-surfaces, et chacune des sous-surfaces a une sémantique. Ledoux & Dukai (2022)

```
"rue": {
    "type": "Road",
    "geometry": [{
        "type": "MultiSurface",
        "lod": "2",
        "boundaries": [
            [[0, 3, 2, 1, 4]], [[4, 5, 6, 666, 12]], [[0, 1, 5]], [[20,
                21, 75]]
        ]
    }],
    "semantics": {
        "surfaces": [
            {
                "type": "TrafficArea",
                "surfaceMaterial": ["asphalt"],
                "function": "road"
            },
            {
                "type": "AuxiliaryTrafficArea",
                "function": "green areas"
            },
```

```
            {
                "type": "TrafficArea",
                "surfaceMaterial": ["dirt"],
                "function": "road"
            }
        ],
        "values": [0, 1, null, 2]
    }
    "children": ["sect1", "sect2"],
},
"sect1": {
    "type": "Road",
    "attributes": {
        "class": "section"
    },
    "parents": ["rue"],
    "geometry": [...],
},
"sect2": {
    "type": "Road",
    "attributes": {
        "class": "intersection"
    },
    "parents": ["rue"],
    "geometry": [...],
}
```

1.6.7 Topologie dans CityJSON

Pour de nombreuses applications de modèles 3D de ville, il est important de disposer d'un modèle topologiquement correct. La topologie correspond aux relations spatiales entre les différents objets du modèle et assure donc la cohérence et la logique spatiale des entités géométriques.

Dans CityGML, les relations topologiques sont construites à l'aide des XLinks (XML Linking) qui permettent de créer des liens entre les objets de ville. Ils sont également utilisés pour éviter la redondance dans la représentation d'un objet partagé (Beil & Kolbe, 2017).

En effet, bien que les XLinks soient en théorie puissants, dans la pratique, les liens doivent être résolus, ce qui est problématique, surtout pour les fichiers volumineux ou lorsqu'ils pointent vers des objets externes (Ledoux et al., 2019).

En revanche, CityJSON facilite la construction de modèles 3D de ville topologiquement corrects (Boersma, 2019). En fait, même avec l'absence d'un moyen normalisé de référencer d'autres objets, il est possible de relier les objets de différents niveaux (Figure 1.14) par des relations *parent-children*.

En outre, dans CityGML, chaque géométrie possède sa propre liste de coordonnées. Cela peut entraîner des erreurs topologiques. Dans CityJSON, les sommets ne doivent être stockés qu'une seule fois et peuvent être réutilisés par différentes géométries (Boersma, 2019).

Néanmoins, CityJSON ne propose pas une méthode pour la représentation d'un graphe à partir des lignes et des nœuds, même s'il existe des classes CityObjects qui permettent ces primitives géométriques, mais une méthode pour les relier n'a pas encore été développée (Boersma, 2019).

Conclusion

La plupart des normes de structuration de données routières se concentrent sur une représentation linéaire des routes. La norme CityGML, grâce au concept de niveaux de détails qui sert à différencier les représentations multi-échelles des modèles sémantiques de villes, permet une représentation surfacique des complexes de transport. Son encodage CityJSON offre plus de légèreté et de flexibilité de mise en œuvre des maquette 3D.

Dans le chapitre suivant, les différentes technologies de stockage et de visualisation des jeux de données CityJSON seront présentées et discutées.

Chapitre 2

Stockage et visualisation des modèles 3D de ville

Introduction

Ce chapitre est dédié aux technologies de stockage et de visualisation des modèles 3D de ville. Les différents modèles de base de données sont discutés. Ensuite, les solutions de stockage et de visualisation des jeux de données CityJSON sont présentées et comparées.

2.1 Modèles de bases de données

Il existe plusieurs façons de stocker les données de manière persistante et de les récupérer efficacement par la suite. Le modèle de base de données peut être défini comme une structure logique de la base de données qui détermine comment les données seront stockées, récupérées et mises à jour et quelles relations existent entre les éléments de ces données (Rai & Singh, 2015).

Les premiers modèles développés pour les bases de données sont le modèle *hiérarchique* et le modèle *en réseau*. Le principal inconvénient de ces modèles est la redondance de données. Il est possible que la même information soit stockée plusieurs fois en raison de la structure arborescente du modèle.

Nous présentons dans les sections suivantes les principaux modèles de base de données utilisés aujourd'hui :

2.1.1 Modèle relationnel

Dans les bases de données relationnelles, les données sont structurées en tables, dites *relations*, composées de *lignes* et de *colonnes* (Figure 2.1). Chaque ligne contient une entité ayant des *attributs* organisés en colonnes contenant des éléments du même type de données.

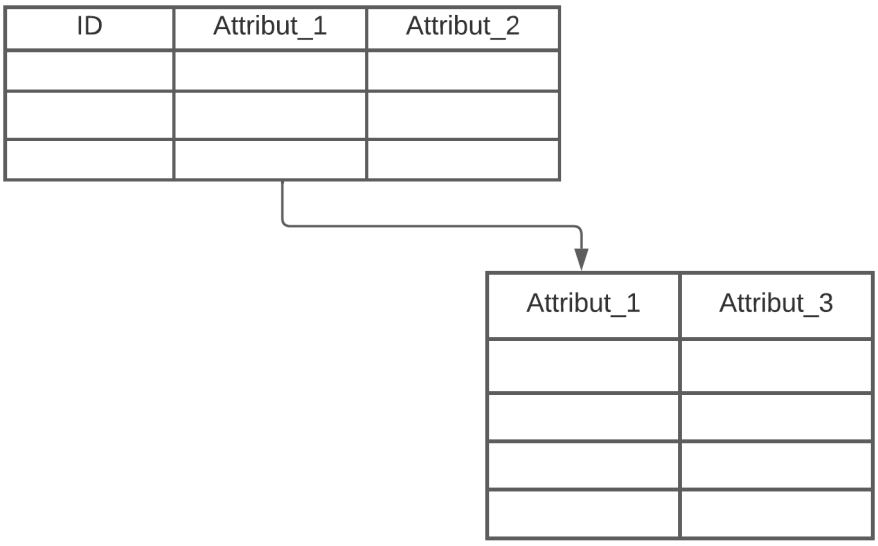


Figure 2.1 – Modèle relationnel des bases de données

Un SGBDR (Système de gestion de bases de données relationnelles) fournit un moyen très

simple de construire, d'accéder et de mettre à jour une base de données (Rai & Singh, 2015). Le langage standard utilisé pour le système de gestion des bases de données relationnelles est SQL (Structured Query Language) (Xu & Aljarallah, 2014).

Tous les principaux SGBDR utilisent des implémentations de SQL (MySQL, Microsoft SQL server, PostgreSQL, Oracle, etc.). Selon Henricsson & Gustafsson (2011), cela garantit la portabilité et les données peuvent facilement être migrées d'un SGBDR à un autre.

Les propriétés ACID (Atomicité, Cohérence, Isolation et Durabilité) définissent les propriétés clés de la base de données SQL afin de garantir une évolution cohérente, sécurisée et robuste de la base de données lors d'une transaction (Tableau 2.1)

Le principal inconvénient de ce modèle est qu'il ne supporte que les valeurs textuelles et numériques, et ne prend pas en charge les types de données abstraites telles que les données audio, vidéo et géographiques (Rai & Singh, 2015).

Tableau 2.1 – Propriétés ACID des bases de données SQL

Propriété	**Définition**
Atomicité	Chaque transaction est atomique, c'est-à-dire que si une partie du système échoue, l'ensemble du système échoue.
Cohérence	Chaque transaction est soumise à des règles précises.
Isolation	Aucune transaction n'interfère avec une autre transaction.
Durabilité	Si un utilisateur a effectué la transaction, les autres utilisateurs reçoivent les mêmes données qui ont été transmises.

2.1.2 Modèle orienté-objet

Dans la programmation orientée objet, tout est un *objet*, et de nombreux objets sont assez complexes, ayant différentes *propriétés* et *méthodes*.

Selon Rai & Singh (2015), le modèle de base de données orienté objet combine le concept de la programmation orientée objet avec le modèle de base de données relationnelle pour fournir un système intégré de développement d'applications.

Dans ce modèle, les données et leurs relations sont contenues dans une structure unique appelée *objet* avec différents *attributs*, appelés également *propriétés*.

Ce modèle prend en charge les types de données abstraites telles que les données audio, vidéo et géographiques.

La Figure 2.2 illustre un exemple du schéma d'une base de données orientée-objet :

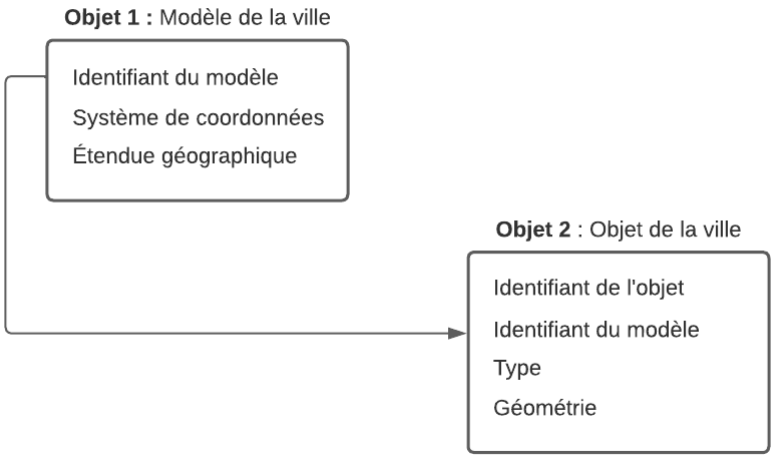


Figure 2.2 – Exemple du schéma d'une base de données orientée-objet

2.2 Bases de données NoSQL

Avec l'explosion du volume de données, les requêtes SQL sont devenues un véritable défi pour la gestion de grandes bases de données (Ali et al., 2019).

Aujourd'hui, plusieurs applications produisent une grande quantité de données qui peuvent être semi-structurées ou non-structurées.

Les données structurées sont généralement des données tabulaires représentées par des colonnes et des lignes dans une base de données. Les données non structurées sont des informations qui ne sont pas organisées d'une manière prédéfinie ou qui n'ont pas de modèle de données prédéfini.

Les données semi-structurées sont des informations ne correspondent pas à des représentations tabulaires mais qui présentent néanmoins une certaine structure.

Selon Ghotiya et al. (2017), cette grande quantité de données ne peut être gérée par des bases de données relationnelles qui ne fonctionnent que sur des données structurées. D'où le besoin d'une base de données adaptée aux applications actuelles.

Les bases de données NoSQL (Not only SQL), ou Bases de données non-relationnelles, ont émergé comme un modèle alternatif pour la gestion des bases de données. Elles sont associées au Big Data, qui peut être considéré, selon Staring (2020), comme une grande quantité de données avec des structures de données flexibles et un traitement en temps réel.

Selon Henricsson & Gustafsson (2011), l'utilisation du modèle relationnel, et donc des tables, pour stocker et récupérer les données n'est pas toujours la solution optimale. En effet, les données qui ne sont pas normalisées et qui ne sont pas conformes au modèle relationnel doivent être restructurées.

Les structures d'information utilisées dans les bases de données NoSQL sont légèrement différentes de celles utilisées dans les bases de données SQL et sont parfois considérées comme plus flexibles que les tables (Ali et al., 2019).

Les bases de données relationnelles souffrent également d'un problème d'extensibilité horizontale. En fait, l'extensibilité d'une base de données SQL se résume à la mise à niveau du matériel

informatique et du serveur sur lequel la base de données fonctionne (Henricsson & Gustafsson, 2011).

Il s'agit d'une extensibilité verticale sur un seul serveur. Il est possible qu'à un moment donné, le matériel le plus performant soit en place, mais que la base de données nécessite une mise à l'échelle encore plus importante. Il faut alors répartir la base de données sur plusieurs serveurs : extensibilité horizontale.

En revanche, les bases de données NoSQL sont considérées comme des bases de données distribuées. Selon Staring (2020), une base de données distribuée est un *cluster* de nœuds. Cela signifie que les bases de données NoSQL s'échelonnent horizontalement en ajoutant plus de nœuds lorsque le volume des données augmente (Figure 2.3).

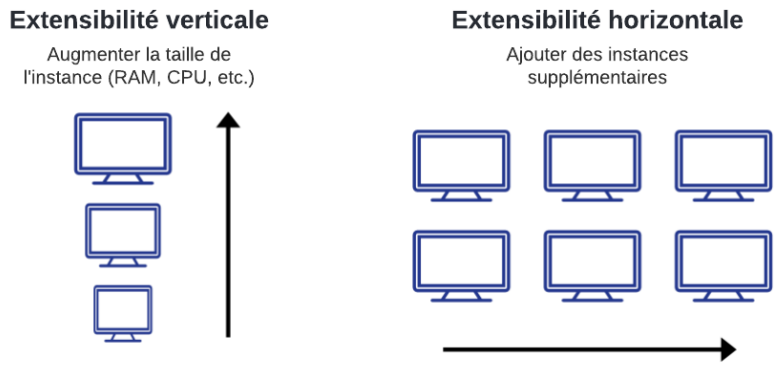


Figure 2.3 – Extensibilité verticale et horizontale des bases de données

En outre, les propriétés ACID des bases de données SQL peuvent représenter une surcharge pour certaines applications qui ont besoin de partager de grandes quantités d'informations (Ali et al., 2019).

Les solutions NoSQL ne suivent pas les propriétés ACID mais les propriétés BASE (Basically Available, Soft state and Eventual consistency) (Tableau 2.2) qui offre une haute disponibilité, mais au coût de la cohérence. Il en résulte un système dans lequel la dé-normalisation est favorisée (Nys & Billen, 2021).

Tableau 2.2 – Propriétés BASE des bases de données NoSQL

Propriété	**Définition**
Basically Available	Les bases de données NoSQL garantissent la disponibilité des données en les répartissant et en les répliquant sur les nœuds du cluster de base de données.
Soft state	L'état des données pourrait changer sans interactions avec l'application en raison de la cohérence éventuelle.
Eventual consistency	Le système sera finalement cohérent après la saisie des données.

2.2.1 Types des bases de données NoSQL

Il existe de nombreuses bases de données NoSQL différentes, mais les principales catégories sont les bases de données clés-valeurs, à colonnes larges, à graphes et à documents :

— **Base de données clés-valeurs** : Il s'agit d'une collection de paires *clé-valeur* dont les clés correspondent aux identifiants (Staring, 2020). Le modèle de données est donc simple : *map* et *dictionary* qui permettent à l'utilisateur de demander des valeurs en fonction de clés spécifiées, ce qui permet une recherche rapide et des options de stockage massif. Selon Ali et al. (2019), le principal inconvénient de ce modèle est l'absence d'un schéma permettant de créer une vue personnalisée des données.
— **Base de données orientée colonnes** : Il s'agit des bases de données clés-valeurs étendues, dont la valeur contient une hiérarchie de paires clé-valeur (Staring, 2020). Les hiérarchies sont des familles de colonnes qui peuvent être utilisées, avec les clés, pour l'indexation.
— **Base de données orientée graphes** : Selon Henricsson & Gustafsson (2011), les bases de données à base de graphes sont considérées comme des graphes plus ou moins complexes où chaque nœud, et chaque arête entre eux, peut avoir des attributs.
— **Base de données orientée documents** : Il s'agit également des bases de données clés-valeurs étendues dans lesquelles la valeur est représentée par un document (Staring, 2020). La structure de ces documents dépend de leur implémentation, mais les principaux formats utilisés sont XML, JSON et YAML (Yet Another Markup Language) (Henricsson & Gustafsson, 2011). Ces documents ne sont pas nécessairement contraints à des schémas ou des tables statiques. Au contraire, différents documents peuvent contenir des clés et des valeurs différentes.

2.2.2 MongoDB

MongoDB est un système de gestion de bases de données NoSQL, ouvert et multiplateforme, écrit en C++ et orienté documents.

Selon Henricsson & Gustafsson (2011), MongoDB utilise une forme binaire de JSON appelée *BSON* (Binary JSON) pour stocker les données. Les documents MongoDB ont une syntaxe structurée en JSON, mais ils sont stockés sous forme de documents BSON. Lorsque les données sont récupérées, les documents sont à nouveau convertis en JSON (Celesti et al., 2019).

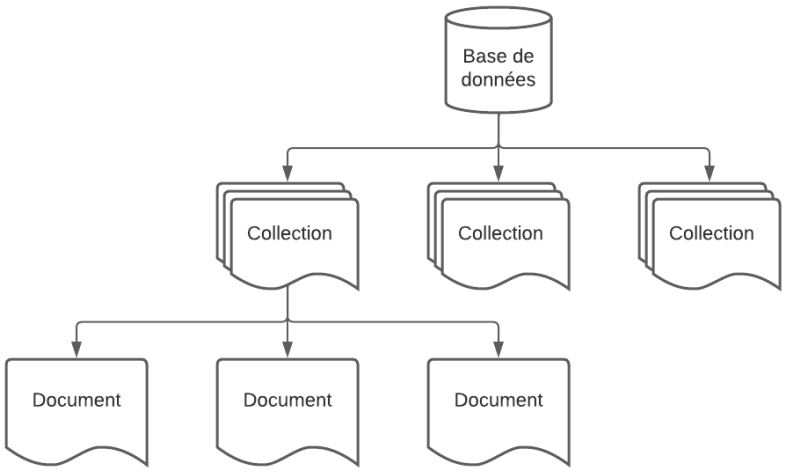


Figure 2.4 – Modèle orienté-documents de la base de données MongoDB

Un document JSON est constitué de plusieurs paires clé-valeur (Henricsson & Gustafsson, 2011). Selon Staring (2020), ces documents n'ont pas de schéma, ce qui signifie que les champs de chaque document peuvent être différents et que le type de données d'un champ peut également varier d'un document à l'autre.

L'organisation libre des données encodées en JSON offre l'avantage d'une structure de données hautement dynamique et flexible (Celesti et al., 2019). Les documents sont regroupés dans des *collections*. Une base de données, à son tour, peut être considérée comme une collection de collections (Figure 2.4).

2.3 Stockage de CityJSON

Les systèmes de fichiers peuvent être considérés comme le moyen le plus simple de stocker des données. Cependant, ils présentent des lacunes dans certaines applications qui exigent de l'extensibilité, de l'efficacité et de la flexibilité dans l'accès aux données (Li, 2021). Deux approches de stockage des fichiers CityJSON peuvent être considérées :

2.3.1 3DCityDB

Les modèles 3D de ville en CityGML sont généralement stockés dans des bases de données relationnelles telles que PostgreSQL avec l'extension PostGIS et qui sont étendues avec des outils dédiés aux modèles 3D de ville tels que 3DCityDB (Mao et al., 2014).

3DCityDB n'est pas une base de données, mais ajouté en tant que schéma de base de données à un système de gestion de base de données relationnelle spatialement augmenté, en correspondance avec la norme CityGML (Bendiksen, 2021).

Toutefois, l'utilisation de 3DCityDB pour le stockage de CityJSON présente quelques inconvénients.

L'inconvénient majeur de 3DCityDB est le manque de flexibilité de la solution relationnelle qui pourrait limiter son utilisation en imposant un grand nombre de jointures récursives pour représenter les hiérarchies du modèle de données orienté-objet (Nys & Billen, 2021). De plus, il peut être nécessaire d'ajouter des tables pour supporter de nouvelles fonctionnalités, et donc une demande supplémentaire de ressources dans une base de données à extensibilité verticale (voir 2.2).

En outre, 3DCityDB est développé pour stocker les fichiers CityGML encodés en XML. C'est une raison essentielle pour laquelle le schéma de la base de données dans 3DCityDB est complexe et verbeux (Li, 2021). Le stockage des données JSON n'est pas pris en charge dans 3DCityDB. Ceci peut être considéré comme un inconvénient pour les jeux de données CityJSON qui doivent être traduits en CityGML avant de pouvoir les stocker.

2.3.2 Stockage dans MongoDB

Selon Staring (2020), l'utilisation de JSON permet de cartographier les attributs avec une flexibilité accrue par rapport à 3DCityDB, ce qui permet de réduire le nombre de tables ou de collections et donc moins de jointures ou de requêtes.

A cet égard, les bases de données NoSQL, comme MongoDB, apparaissent plus adéquates au stockage des modèles 3D de villes encodés en CityJSON (voir 2.2). Ceci revient à leur extensibilité horizontale, et également à leur flexibilité étant donné que les documents ne sont pas nécessairement contraints à des schémas statiques.

Cependant, dans certaines situations, il est nécessaire que les documents à stocker respectent une structure particulière ou répondent à certaines contraintes. C'est le cas de CityJSON qui

utilise des schémas JSON pour documenter son modèle de données et pour valider si un fichier CityJSON respecte la structure et la syntaxe autorisées (Ledoux et al., 2019).

Par conséquent, le stockage d'un fichier CityJSON est contraint à respecter des schémas prédéfinis pour l'ensemble des objets de la ville, leurs attributs, leurs géométries et d'autres contraintes supplémentaires.

Selon Staring (2020), la validation du schéma de données est possible dans MongoDB et permet d'imposer un schéma spécifique aux opérations d'insertion et de mise à jour.

Le schéma suivant est un exemple de schéma JSON défini pour les modèles de géométrie ou *geometry-template* :

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://www.cityjson.org/schemas/1.1.0/geomtemplates.schema.
     json",
  "title": "CityJSON geometry templates schema v1.1.0",
  "GeometryInstance": {
    "type": "object",
    "properties": {
      "type": {
        "enum": [
          "GeometryInstance"
        ]
      },
      "template": {
        "type": "integer"
      },
      "boundaries": {
        "type": "array",
        "items": {
          "type": "integer"
        },
        "minItems": 1,
        "maxItems": 1
      },
      "transformationMatrix": {
        "type": "array",
        "items": {
          "type": "number"
        },
        "minItems": 16,
        "maxItems": 16
      }
    },
    "required": [
      "type",
      "template",
      "boundaries",
      "transformationMatrix"
    ],
    "additionalProperties": false
  }
}
```

L'ensemble des schémas de données CityJSON sont publiées sur `https://3d.bk.tudelft.nl/schemas/cityjson/`.

A l'encontre des bases de données relationnelles, dans MongoDB, il est possible de stocker un modèle de ville CityJSON, si sa taille ne dépasse pas 16 Mo, dans un seul document sans le décomposer en différentes collections.

Cependant, selon Nys & Billen (2021), cela réduit considérablement les possibilités ultérieures :

— Les requêtes et l'indexation doivent être complexes pour parcourir la structure des objets embarqués ;
— Une indexation composée de plusieurs attributs n'est pas recommandée pour des requêtes efficaces ;
— La mise à jour d'un sous-objet devient plus complexe car cela impose de parcourir profondément des objets embarqués, de récupérer l'objet à modifier puis de réinsérer la nouvelle version dans le modèle ;

Par conséquent, la solution proposée est de créer différentes collections pour faciliter l'accès et la manipulation des différents éléments (Nys & Billen, 2021).

Un modèle CityJSON, en tant que fichier récupéré, est donc constitué par la combinaison de toutes ces collections, c'est-à-dire la jointure des documents correspondants. Cela est possible avec la fonction *lookup* qui crée une jointure gauche avec une autre collection et permet de filtrer les données à partir des données fusionnées.

2.4 Visualisation des modèles 3D de routes

CityJSON a été conçu en tenant compte des besoins en programmation, afin que les outils et les API (Application Programming Interface) qui le prennent en charge puissent être rapidement construits (Stoter et al., 2020).

Selon Stoter et al. (2020), CityJSON prend en charge l'utilisation de données 3D au-delà de leur échange, et il est donc convivial pour le développement web et mobile.

L'accès et la visualisation des modèles 3D de ville à travers un service web doit tenir compte de deux défis principaux. Premièrement, une architecture de serveur permettant le stockage et la récupération rapide d'une grande quantité de données. Deuxièmement, un client web capable de visualiser de manière efficace les données fournies par le serveur (Prandi et al., 2015).

2.4.1 Architecture à trois couches

L'architecture à trois couches est un modèle qui structure les applications en trois niveaux informatiques logiques et physiques : couche de présentation, couche d'application et couche de données (Figure 2.5).

Ce module complet d'architecture à trois niveaux constitue la structure de base pour la majorité des applications web (Hussain & Sharma, 2019).

La couche de présentation, ou côté client (en anglais, *frontend*), est l'interface utilisateur et la couche de communication de l'application, où l'utilisateur final interagit avec l'application. Son objectif principal est d'afficher des informations à l'utilisateur et de recueillir des données fournies par celui-ci.

La deuxième couche est la couche d'application. Selon Hussain & Sharma (2019), la couche intermédiaire comprend la partie la plus importante de la logique application. Il s'agit du serveur web qui traite les requêtes HTTP (Hypertext Transfer Protocol) et envoie une réponse au côté

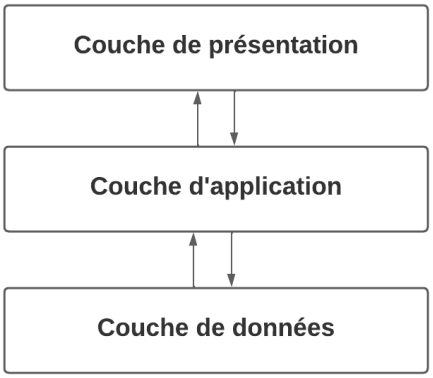


Figure 2.5 – Architecture à trois couches

client via d'appels API (voir la section 2.4.2). Il peut également ajouter, supprimer ou modifier des données dans la couche de données.

La troisième couche, appelée couche de données ou niveau d'accès aux données (en anglais, *backend*), correspond à la base de données où les données sont stockées et gérées. Il peut s'agir d'un système de gestion de base de données relationnelles ou d'une base de données NoSQL.

2.4.2 REST API

API est l'acronyme d'Application Programming Interface (interface de programmation d'applications). Il fournit un moyen normalisé d'échange de données entre deux applications.

REST est l'acronyme de Representational State Transfer. Il s'agit d'une architecture logicielle qui utilise un sous-ensemble de protocoles HTTP pour effectuer des opérations CRUD (Create, Read, Update and Delete) (Li, 2021).

Selon Henricsson & Gustafsson (2011), les méthodes HTTP sont mises en correspondance avec les opérations CRUD de la manière suivante :

— **POST** : créer une ressource
— **GET** : lire une ressource
— **PUT** : mettre à jour une ressource
— **DELETE** ; supprimer une ressource

Dans l'architecture fonctionnelle REST, les ressources, telles que les bases de données, reçoivent des identifiants uniques sous la forme d'URI (Henricsson & Gustafsson, 2011).

2.4.3 Ninja Viewer

Ninja a été développée par Vitalis et al. (2020) comme une implémentation d'une application web pour les données CityJSON, et qui peut être utilisée comme référence pour d'autres applications. Il permet à l'utilisateur de charger et d'examiner facilement un modèle CityJSON via un navigateur web.

Il est écrit en JavaScript à l'aide du framework Vue.js. Vitalis et al. (2020) justifie ce choix par la possibilité de créer des composantes réutilisables permettant de réduire la redondance du code, et également le recours à la liaison de données, en anglais Data Binding, qui permet de synchroniser les données du fournisseur et de l'utilisateur, et donc réduire la quantité de code nécessaire à la manipulation des données dans une application web.

Le Tableau 2.3 résume les principales composantes Vue.js de l'application Ninja Viewer :

Tableau 2.3 – Composantes Vue.js de Ninja Viewer

Composante	Description
ThreeJsViewer	Visualisateur 3D implémenté en Three.js qui prend en entrée un fichier CityJSON et est responsable du rendu de ses géométries en 3D
CityObjectsTree	Présentation arborescente de la structure hiérarchique des objets de la ville
CityObjectCard	Carte affichant les détails d'un objet de la ville et permettant l'édition des données JSON

Le visualisateur prend en charge tous les types géométriques surfaciques et volumétriques de la norme ISO 19107, notamment MultiSurface, CompositeSurface, Solid, MultiSolid et CompositeSolid. Il permet également d'interpréter l'aspect sémantique du modèle 3D de la ville.

La visualisation d'un modèle 3D de ville est réalisée grâce à la librairie Three.js, qui permet de créer des scènes 3D dans un navigateur web.

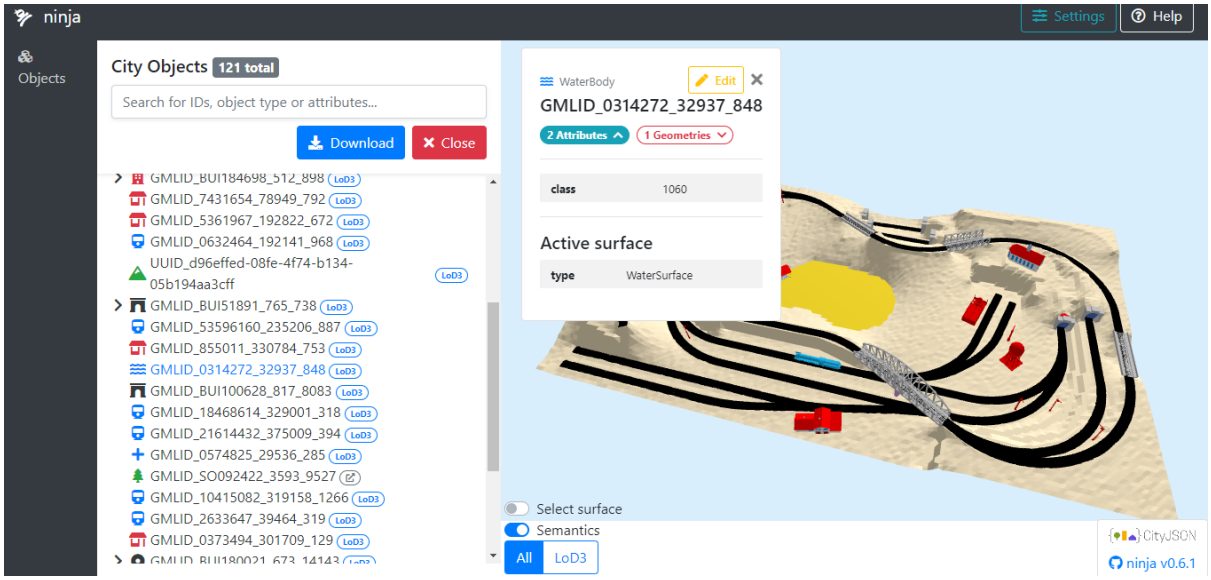


Figure 2.6 – Interface utilisateur de l'application Ninja

Ninja fonctionne comme un visualisateur web des fichiers CityJSON mais ne propose pas une architecture de serveur permettant le stockage et la récupération des données depuis une base de données.

2.4.4 Measur3D

A l'encontre de Ninja, Measur3D est construit comme une application MERN d'une architecture à trois couches.

MERN est l'abréviation de MongoDB, Express.js, React.js et Node.js, du nom des quatre technologies clés qui composent l'application (Figure 2.7) :

— **MongoDB** : Base de données NoSQL orientée-document (voir 2.2.2)

— **Express.js** : Framework pour construire des applications web basées sur Node.js
— **React.js** : Framework JavaScript libre pour faciliter la création d'application web mono-page
— **Node.js** : environnement de serveur permettant d'exécuter JavaScript sur le côté serveur

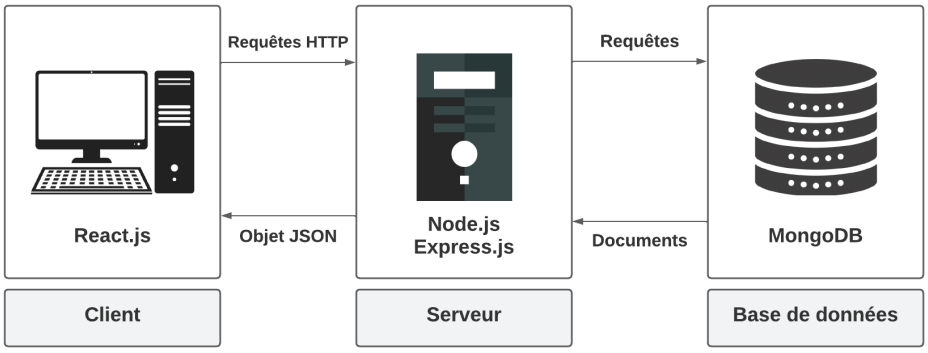


Figure 2.7 – Schéma d'architecture d'une application full-stack MERN

Selon Nys & Billen (2021), la création d'une application full-stack MERN facilite un déploiement intelligent, lorsque la compacité et la légèreté sont exigées.

En fait, l'architecture MERN permet de construire facilement une architecture à trois couches entièrement en utilisant JavaScript et JSON.

2.4.4.1 Côté client

La couche supérieure de Measur3D est développée avec React.js qui permet de construire des interfaces complexes à l'aide de composantes réutilisables, de les connecter à des données serveur et de les rendre en HTML.

Selon Nys & Billen (2021), la visualisation des modèles 3D de ville sur Measur3D est basée sur la bibliothèque Three.js.

La couche client permet également toutes les opérations CRUD courantes sur les modèles et les objets de la ville (Figure 2.8).

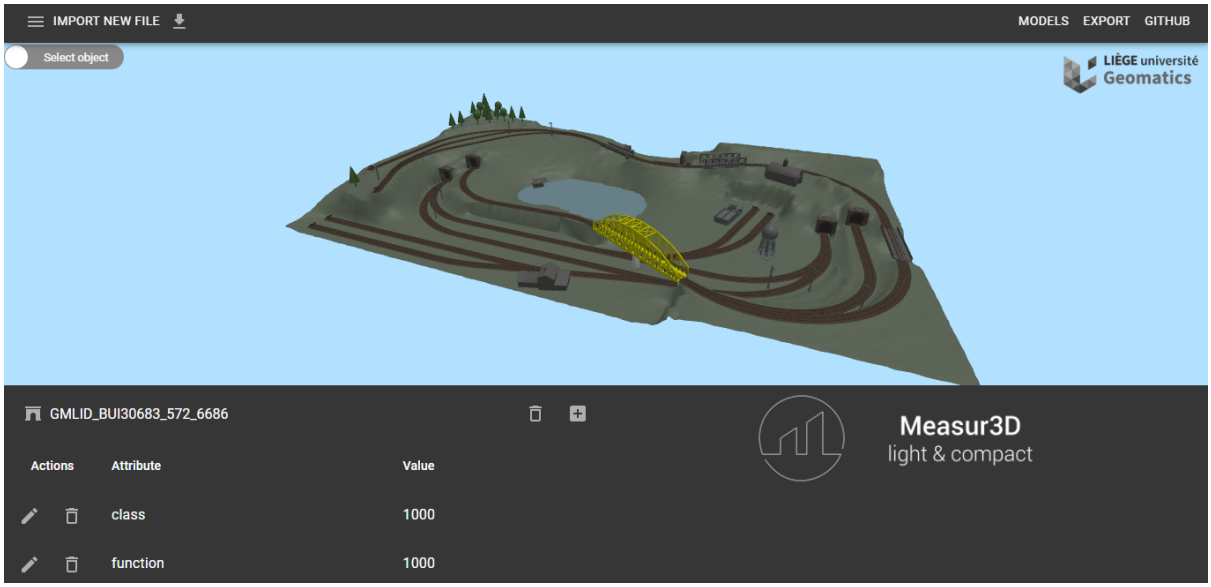


Figure 2.8 – Interface utilisateur de l'application Measur3D

2.4.4.2 Côté serveur

Le serveur est basé sur Node.js qui est un environnement d'exécution de JavaScript dans le côté serveur.

Node.js suit un modèle d'entrée/sortie asynchrone, piloté par les événements et non bloquant. Ces deux dernières propriétés en font un serveur web très rapide et résilient (Nys & Billen, 2021).

En outre, Express.js est une bibliothèque JavaScript qui simplifie la tâche d'écriture du code du serveur web pour Node.js.

En effectuant des requêtes HTTP depuis l'interface utilisateur, il est possible d'exécuter des fonctions de base pour les modèles CityJSON et la gestion de leurs fonctionnalités. Cette API permet de manipuler des modèles de ville compacts sur une base de données orientée document.

2.4.4.3 Couche d'accès aux données

La couche d'accès aux données est une base de données NoSQL orientée documents qui est MongoDB.

Nys & Billen (2021) justifient le choix d'une solution orientée document par la flexibilité du schéma et de la prise en charge des données JSON. Les modèles 3D de villes sont stockés dans différentes collections pour faciliter l'accès aux différents objets et leur mise à jour.

Les schémas de données sont définis à l'aide de Mongoose, une bibliothèque libre qui crée une connexion entre MongoDB et l'environnement d'exécution JavaScript Node.js. Chaque schéma correspond à une collection MongoDB et définit la forme des documents de cette collection.

Conclusion

Les bases de données NoSQL apparaissent plus adéquates au stockage des jeux de données CityJSON. Cela revient d'une part à leur flexibilité et leur extensibilité horizontale, et d'autre part à l'utilisation de JSON qui permet de réduire le nombre de tables ou de collections et donc moins de jointures ou de requêtes.

Il existe des solutions de visualisation des fichiers CityJSON, entre autres Ninja Viewer et Measur3D. Bien que Ninja permette l'inspection des informations sémantiques, Measur3D offre plus de fonctionnalités quant au stockage et à la gestion des modèles 3D de villes.

Dans la partie suivante, nous présenterons le travail pratique du projet. La méthodologie suivie pour la réalisation du projet, ainsi que les outils de travail sont présentés. Ensuite, les résultats obtenus sont discutés.

Deuxième partie

Partie pratique

Chapitre 3

Cadre méthodologique

Introduction

Dans ce chapitre, la méthodologie générale adoptée pour produire des plateformes routières 3D en CityJSON est présentée. Nous dresserons premièrement la démarche générale détaillée, ensuite nous présenterons la zone d'étude. Les outils de travail et les logiciels utilisés sont également présentés. Finalement, nous présentons les différentes étapes suivies d'une manière détaillée.

3.1 Méthodologie générale

L'objectif principal du projet est de produire des plateformes routières 3D cohérentes géométriquement en CityJSON. L'approche proposée consiste en une modélisation géométrique et sémantique de l'infrastructure routière selon le modèle de données CityGML 3.0.

Elle repose principalement sur l'extraction semi-automatique des caractéristiques linéaires de la route, entre autres les bordures de trottoirs et les limites de la chaussée, à partir d'un nuage de points. Ces éléments linéaires sont utilisés par la suite pour construire les différentes surfaces sémantiques. Cette approche permet d'assurer une topologie valide entre les différentes composantes de la route.

Le modèle 3D de route généré sera traduit en fichier CityJSON 1.1 valide. La validation se fera à deux niveaux : Premièrement, une validation du schéma de données, c'est-à-dire vérifier si la syntaxe du fichier est correcte et conforme aux spécifications CityJSON. Ensuite, une validation des primitives géométriques en conformité avec la norme ISO 19107, qui spécifie les schémas conceptuels de description des caractéristiques spatiales des entités géographiques.

Un fichier CityJSON de routes valide est par la suite stocké dans une base de données orientée documents, puis récupéré à l'aide d'appels API et envoyé vers le client d'une application web à trois couches, pour visualisation et inspection des informations géométriques et sémantiques de l'espace routier et ses objets. La Figure 3.1 résume la méthodologie générale suivie :

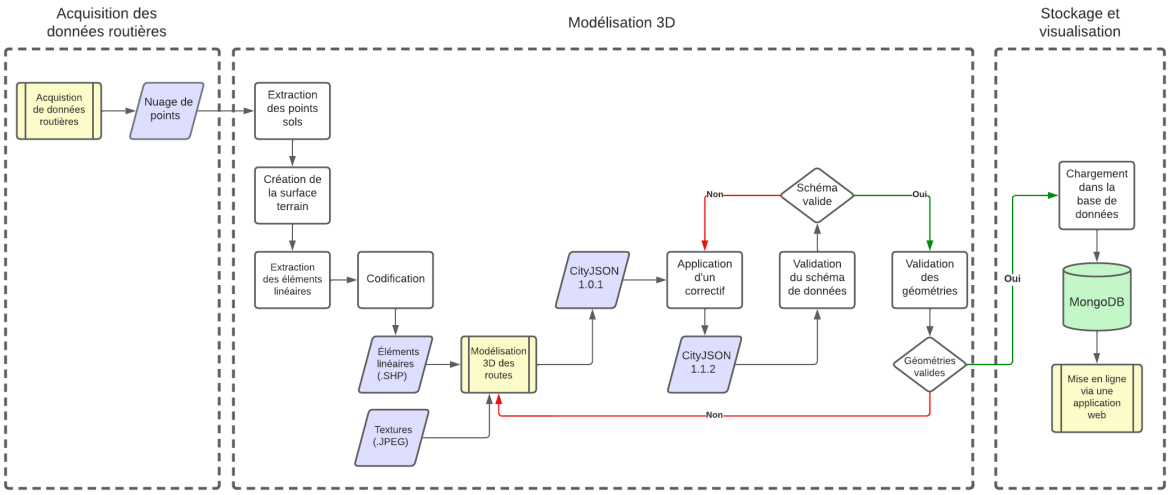


Figure 3.1 – Méthodologie générale suivie dans ce travail

3.2 Zone d'étude

Le travail pratique du projet consiste, dans un premier temps, en une modélisation 3D de la route. Le modèle 3D est généré à partir d'un nuage de points acquis par lasergrammétrie mobile.

Le levé a été effectué par un système de cartographie mobile d'un tronçon d'une longueur de 1.3 km de l'Avenue Tarik Ibn Ziad à Temara (Figure 3.2).

Figure 3.2 – Tronçon de l'Avenue Tarik Ibn Ziad levé en Juin 2022

3.3 Outils de travail

3.3.1 Système d'acquisition

L'acquisition des données routières a été réalisée par le système de cartographie mobile Trimble MX2. Il s'agit d'un système LiDAR mobile, à un ou deux scanners laser, déployé sur un véhicule en mouvement, en combinaison avec un système de positionnement composé d'un GNSS, d'une centrale inertielle IMU et d'un odomètre (Figure 3.3).

Le système de positionnement GNSS permet de localiser le système de cartographie mobile dans le système de coordonnées global WGS84. En plus des informations d'orientation angulaire, l'unité de mesure inertielle IMU, permet le positionnement en cas de perte de signaux GNSS pour certains satellites, ce qui se produit souvent dans les environnements urbains denses avec de hauts bâtiments et des rues étroites.

L'odomètre mesure la distance de déplacement du véhicule dans le but de rectifier les erreurs de localisation.

Le système à deux scanners laser utilise une configuration LiDAR en papillon pour minimiser les ombres.

Figure 3.3 – Trimble MX2 à deux scanners laser

Le système est également équipé d'une caméra panoramique d'un champ de vision à 360°, constituée de 6 caméras de prise de vues dont 5 caméras placées dans le plan horizontal et une caméra installée au-dessus (Figure 3.4).

Figure 3.4 – Caméra Ladybug5 de prise de vues panoramiques 360°

Le Tableau 3.1 résume les spécifications techniques de Trimble MX2 :

Tableau 3.1 – Spécifications techniques de Trimble MX2

Portée maximale	250 m
Exactitude/Précision	+/- 1 cm
Champ de vision	360°
Fréquence d'acquisition	72000 pts/s
Centrale inertielle (IMU)	Applanix AP20
Précision trajectoire (GNSS)	0.02 à 0.05 m
Système de prise de vues	Ladybag5 (Optionnel) : 6 capteurs, total 30Mpx, 90% d'une sphère
Poids	25 kg

Le système comporte également une suite logicielle intégrant l'ensemble des flux de travail, incluant l'acquisition de données, le traitement de la trajectoire du véhicule et le traitement du nuage de points et des photographies panoramiques (voir la section 4.1).

3.3.2 Stations de traitement

Le traitement des données LiDAR massives est coûteux en termes de temps et de performances en raison de la taille des nuages de points et de la nature itérative des algorithmes de traitement. Par conséquent, des stations relativement performantes sont nécessaires pour le traitement des données LiDAR. Les spécifications techniques des stations de travail utilisées sont reportées dans le Tableau 3.2 :

Tableau 3.2 – Spécifications techniques des stations de traitement

Système d'exploitation	Microsoft Windows 10 Professionnel	Microsoft Windows 10 Professionnel
Processeur CPU	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz	Intel(R) Core(TM) i7-9700F CPU @ 3.00GHz 3.00 GHz
Mémoire RAM	8.00 Go	64.00 Go
Type du disque dur	HDD	SSD

3.3.3 Solutions logicielles

En plus de la suite logicielle utilisée pour le traitement des données brutes acquises par le système de cartographie mobile (voir la section 4.1), d'autres solutions logicielles ont été exploitées pour réaliser le travail pratique du projet.

Autodesk ReCap Pro

Il s'agit d'un logiciel utilisé principalement pour la création des modèles 3D à partir de photo-

graphies ou de balayages lasers. Il permet de préparer les nuages de points pour une utilisation dans la majorité des logiciels d'Autodesk.

Autodesk ReCap Pro est un logiciel payant. Cependant, à des fins éducatives et académiques, les étudiants et les enseignants peuvent accéder gratuitement pendant un an aux produits et services Autodesk, incluant ReCap Pro.

Autodesk InfraWorks

InfraWorks est un autre outil Autodesk destiné à la création, l'évaluation et la diffusion de projets d'infrastructure dans un environnement BIM (Building Information Modelling). Il offre des options de travail collaboratif dans le cloud, l'automatisation des processus et la collaboration avec d'autres plateformes.

Il permet de concevoir des routes et des ouvrages d'art complexes pour les projets d'infrastructure publique, ainsi que des modèles 3D de sites simples. Le logiciel a été également utilisé sous une licence étudiant valable pour une année.

CloudCompare

Un logiciel open source de traitement de nuages de points 3D et de maillages triangulaires. Il a été conçu à l'origine pour réaliser des comparaisons entre deux nuages de points 3D denses ou entre un nuage de points et un maillage. Il s'appuie sur une structure de données Octree spécifique dédiée à cette tâche.

CloudCompare a été étendu par la suite à un logiciel de traitement de nuages de points plus générique, incluant de nombreux algorithmes avancés (recalage, rééchantillonnage, traitement des champs scalaires, calcul de statistiques, segmentation interactive ou automatique, amélioration de l'affichage, etc.)

Feature Management Engine Workbench

Il s'agit d'un outil ETL (Extract, Transform and Load) qui rationalise la conversion des données spatiales entre différents formats géométriques et numériques. Elle est spécialement conçue pour être utilisée avec des systèmes d'information géographique (SIG), des logiciels de conception assistée par ordinateur (CAO) et des logiciels graphiques.

La plateforme a été développée à l'origine par Safe Software qui offre des licences de quatre mois pour des fins éducatives.

Visual Studio Code

Visual Studio Code, plus connu sous le nom de VS Code, est un éditeur de texte open source gratuit de Microsoft, disponible pour Windows, Linux et macOS. Bien que l'éditeur soit relativement léger, il comprend des fonctionnalités puissantes qui ont fait de VS Code l'un des outils d'environnement de développement les plus populaires.

VS Code prend en charge de nombreux langages de programmation, entre autres Java, C++, Python et d'autres syntaxes (HTML, CSS, etc.). Il comprend également des débogueurs et un support pour le développement web et en cloud.

MongoDB Compass

MongoDB Compass est une interface graphique, intuitive et flexible, pour gérer et manipuler des bases de données MongoDB. Il s'agit d'un outil open source qui offre des visualisations détaillées des schémas de données, des mesures de performance en temps réel, des capacités de

faire des requêtes sophistiquées, de validation de données JSON, de gestion des index et d'autres fonctionnalités importantes.

3.4 Méthodologie détaillée

Dans les sections suivantes, nous détaillons la méthodologie suivie pour la modélisation 3D des infrastructures routières selon le standard CityGML 3.0 et son encodage CityJSON.

3.4.1 Acquisition de données routières

L'acquisition des données routières a été réalisée par un système de cartographie mobile installé sur un véhicule en mouvement. La mission peut être subdivisée en deux parties : collecte de données sur terrain et post-traitements des données LiDAR collectées.

La collecte des données correspond à la mission d'acquisition sur terrain et en temps réel par le système LiDAR. D'autres informations sont acquises durant la mission, entre autres les observations GNSS/IMU et les photographies panoramiques 360°.

Les post-traitements sont nécessaires pour obtenir un nuage de points précis, colorié et de haute résolution. Ils concernent d'abord la trajectoire du véhicule, puis le traitement des nuages de points bruts et des photographies panoramiques. Durant la mission, le système de positionnement permet d'enregistrer la trajectoire absolue de la plateforme. Ainsi, la position du véhicule doit être corrigée pour une précision augmentée. La correction de la trajectoire se fait avec des observations GNSS d'une base de mesure en mode statique placée sur un point de coordonnées connues (X,Y,Z) durant la mission. La Figure 3.5 résume le processus de traitement des données LiDAR :

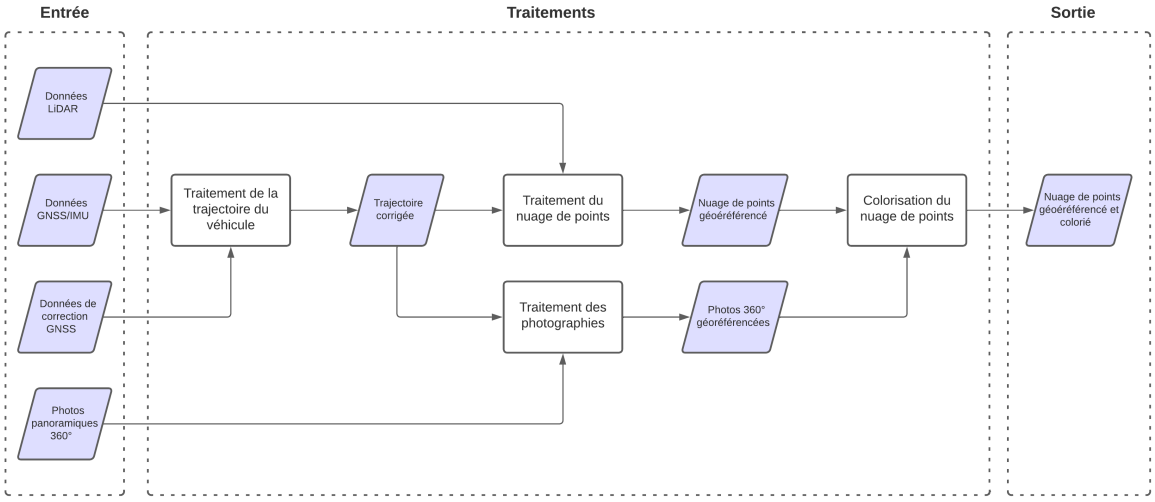


Figure 3.5 – Processus de traitement des données acquises par MMS

3.4.2 Modélisation 3D

L'approche proposée pour la modélisation 3D du tronçon routier, selon le module Transport de CityGML 3.0, consiste initialement à construire la surface terrain à partir du nuage de points.

Les points de la surface de la route appartiennent nécessairement aux points sols. Cela exige une segmentation préalable du nuage de points pour séparer les points sols des points sur-sols, et donc créer un modèle numérique du terrain.

Les caractéristiques linéaires de la route peuvent être extraites sur la base de la surface terrain générée. Cette structure linéaire permettra par la suite une modélisation géométrique et sémantique de l'espace routier.

Selon le module Transport de CityGML 3.0, à partir du niveau de détails LoD2, la représentation des différentes surfaces de l'espace routier devient possible. Il s'agit des surfaces sémantiques qui peuvent être des *TrafficArea* comme les zones piétonnes et les voies de circulation, ou des *AuxiliaryTrafficArea* telles que les espaces verts.

De plus, les objets de la classe abstraite *TransportationSpace* peuvent être subdivisée en sections, intersections et ronds-points.

Il est donc nécessaire de modéliser séparément chaque section ou intersection de la route, et chaque surface sémantique, pour pouvoir générer un modèle 3D à un niveau de détails relativement élevé (LoD2) tout en considérant les relations topologiques entre les différents objets de la route (Figure 3.6).

La solution proposée est d'établir un système de codification qui sert à attribuer un code prédéfini à chaque entité linéaire extraite, selon sa nature (limites du trottoir, bordures de la chaussée, etc.). Par conséquent, il sera possible de séparer les différentes sections de la route et de modéliser toutes les surfaces sémantiques.

La modélisation géométrique consiste à recréer en 3D les différents objets de l'espace routier. Cela peut se faire en combinant les lignes extraites pour générer des polygones 3D. Ensuite, une extrusion selon la normale sur chaque polygone permet d'avoir des solides, ou simplement des géométries volumétriques composées de plusieurs surfaces.

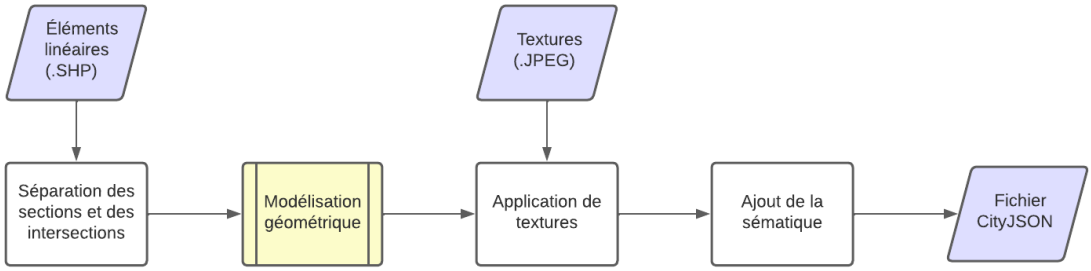


Figure 3.6 – Processus de modélisation 3D du tronçon routier

CityJSON supporte également les textures. En termes simples, une texture est une image qui est appliquée à un modèle 3D afin de le colorer.

Il serait donc possible d'appliquer des textures aux différentes surfaces sémantiques de la route. Ces textures peuvent être extraites des photographies panoramiques 360° prises durant la mission d'acquisition des données LiDAR.

3.4.3 Traduction en fichier CityJSON

En réalité, la modélisation 3D est réalisée dans un environnement processus ETL spatial pour reconstruire l'espace routier en trois dimensions, puis traduire ce modèle 3D en fichier CityJSON.

Par définition, un ETL spatial, comme son nom l'indique, est un processus par lequel les données spatiales sont extraites à partir de sources de données homogènes ou hétérogènes, transformées

dans n'importe quel format ou structure puis charger dans un système cible, qui peut être une base de données.

La modélisation 3D de l'espace routier est réalisée sur base des données vectorielles linéaires, extraites à partir d'un nuage de points. Les transformations appliquées permettent une modélisation géométrique et sémantique de la route, telle que décrite dans la section 3.4.2.

Ainsi, le modèle 3D de route généré est transformé en format CityJSON. Un correctif peut être appliqué pour se conformer aux spécifications CityJSON de la dernière version publiée.

3.4.4 Conformité au schéma de données

La validation d'un fichier CityJSON signifie que l'on doit s'assurer qu'il respecte les spécifications et les définitions standardisées telles que décrites dans la section 1.6.5.

La première étape pour valider un jeu de données CityJSON est de vérifier la syntaxe du fichier, c'est-à-dire la conformité au schéma de données.

La validation du schéma de données peut se faire à l'aide de *cjval*, qui est le validateur officiel des fichiers CityJSON par rapport aux schémas de données.

Il est disponible en tant qu'une application web ou avec *cjio*, une application Python à une interface en ligne de commande (ou CLI pour l'acronyme anglais) pour traiter et manipuler les fichiers CityJSON.

cjval permet de vérifier premièrement la syntaxe JSON, puis la conformité aux schémas JSON de CityJSON,

Les schémas JSON sont utilisés pour exiger qu'un document JSON donné, appelé également une instance, réponde à un certain nombre de critères.

D'autres éléments sont vérifiés pour valider la syntaxe d'un fichier CityJSON. Le Tableau 3.3 résume l'ensemble de ces éléments.

Un fichier qui n'est pas conforme aux spécifications CityJSON doit être corrigé. À titre d'exemple, l'opérateur *clean* de *cjio* permet de supprimer automatiquement les sommets en double et les sommets qui ne sont pas référencés dans le fichier.

Les autres erreurs du schéma de données peuvent être corrigées en appliquant un correctif ou, en termes simples, un script pour corriger la syntaxe du fichier.

Tableau 3.3 – Éléments de validation de la syntaxe d'un fichier CityJSON

Syntaxe JSON	Le fichier est-il un fichier JSON valide ?
Schémas CityJSON	Validation des schémas CityJSON
Schémas d'extensions	Validation des schémas supplémentaires en présence d'une extension
Cohérence parents-enfants	Vérifier les relations parents-enfants, c'est-à-dire si un objet fait référence à un autre dans ses *enfants* ou ses *parents*
Mauvais indice de sommets	Tous les indices de sommets existent-ils dans la liste des sommets ?
Sémantique	Est-ce que les listes des surfaces sémantiques ont la même dimension que celle de la géométrie ? Les valeurs sont-elles cohérentes ?
Propriétés supplémentaires	Les propriétés supplémentaires doivent être documentées dans une extension
Sommets en double	Les sommets en double sont autorisés, mais ils occupent de l'espace et diminuent les relations topologiques explicitement dans le fichier
Sommets non utilisés	Les sommets qui ne sont pas référencés dans le fichier prennent un espace supplémentaire

3.4.5 Validation des primitives géométriques

La conformité aux schémas de données n'est pas suffisante pour dire qu'un fichier CityJSON est complètement valide.

Selon Bendiksen (2021), la validité des géométries est également nécessaire pour pouvoir exploiter correctement les modèles 3D de route. Il est indispensable que les surfaces 3D de base, définissant l'espace routier en trois dimensions, soient conformes aux normes internationales.

Les règles relatives à la structure des primitives 3D sont spécifiées dans la norme internationale ISO 19107 et ont pour but de garantir la cohérence lors du transfert et de la transformation des jeux de données (Ledoux, 2018).

Les objets 3D sont appelés solides dans la norme ISO 19107. Un solide est créé en combinant des surfaces 2D. Ces surfaces sont créées par des courbes 1D qui, à leur tour, sont créées à l'aide de points, qui ont des dimensions nulles, ou 0D.

Les primitives géométriques de même dimension peuvent être combinées pour former une autre primitive : *aggregate*, qui est une collection de primitives de même dimension, qui peuvent se chevaucher ou être détachées, utilisée pour regrouper des géométries ; *composite* dont les primitives ne peuvent pas se chevaucher ou être disjointes.

Ledoux (2018) préfère la nomenclature GML, et donc *aggregate* est remplacée par *Multi*, et *composite* reste *Composite*. La Figure 3.7 illustre les différentes primitives géométriques tridimensionnelles :

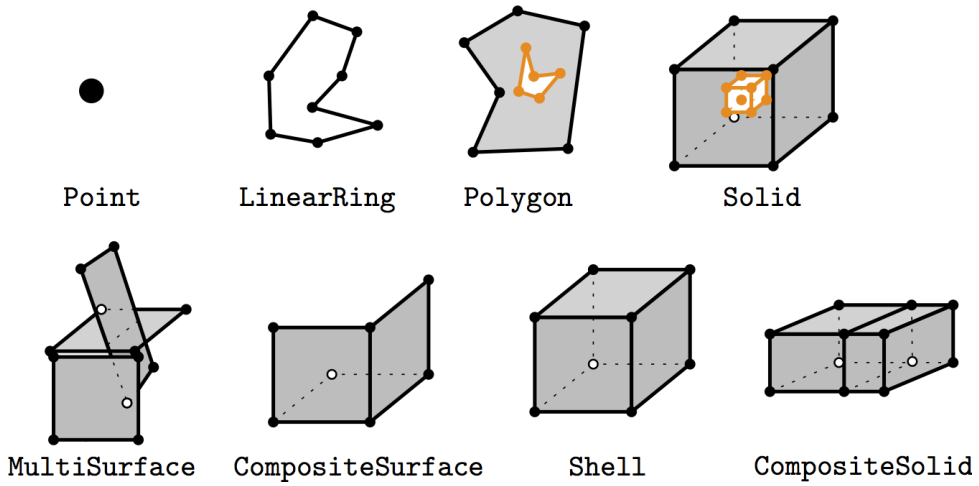


Figure 3.7 – Primitives géométriques 3D (Ledoux, 2018)

Les erreurs géométriques et topologiques peuvent entraîner des résultats faux lors de l'exécution des requêtes spatiales et des algorithmes sur le modèle 3D. Un polygone qui s'auto-intersecte, à titre d'exemple, peut entraîner un calcul de surface inexacte.

Selon Ledoux (2018), ces erreurs ne sont pas visibles à l'échelle à laquelle les jeux de données sont visualisés et, par conséquent, les praticiens ne sont pas conscients du problème. En fait, même à des fins de visualisation, les erreurs peuvent être problématiques car l'ombrage des surfaces est souvent basé sur l'orientation de leurs normales.

Pour pouvoir détecter ces erreurs géométriques et topologiques, Ledoux (2018) a proposé une nouvelle version de *Val3dity*, un logiciel open source pour valider les primitives 3D selon les définitions de la norme ISO 19107.

En plus des règles fixées par la norme ISO 19107, Val3dity inclut deux restrictions : les courbes doivent être linéaires et les surfaces doivent être planes.

Val3dity peut détecter de nombreuses erreurs dans un modèle 3D de ville (Figure 3.8). L'une de ces erreurs est la présence de surfaces non planes dont un ou plusieurs sommets ne sont pas situés sur le même plan que les autres (Bendiksen, 2021).

Si une erreur de géométrie est signalée, l'approche proposée est de détecter initialement l'objet qui porte l'erreur et les sommets qui provoquent l'invalidité. La correction de ces erreurs nécessite de revoir les transformations appliquées pour générer le modèle 3D.

Le transformateur *GeometryValidator* de Safe FME, Feature Management Engine, permet de détecter certaines erreurs géométriques selon la norme OGC Simple Features qui fournit une description des données vectorielles. Il permet également de corriger certaines erreurs géométriques et topologiques, au moment de la modélisation 3D de l'espace routier. D'autres approches peuvent être considérées pour résoudre les problèmes de planarité (voir la section 4.5.2).

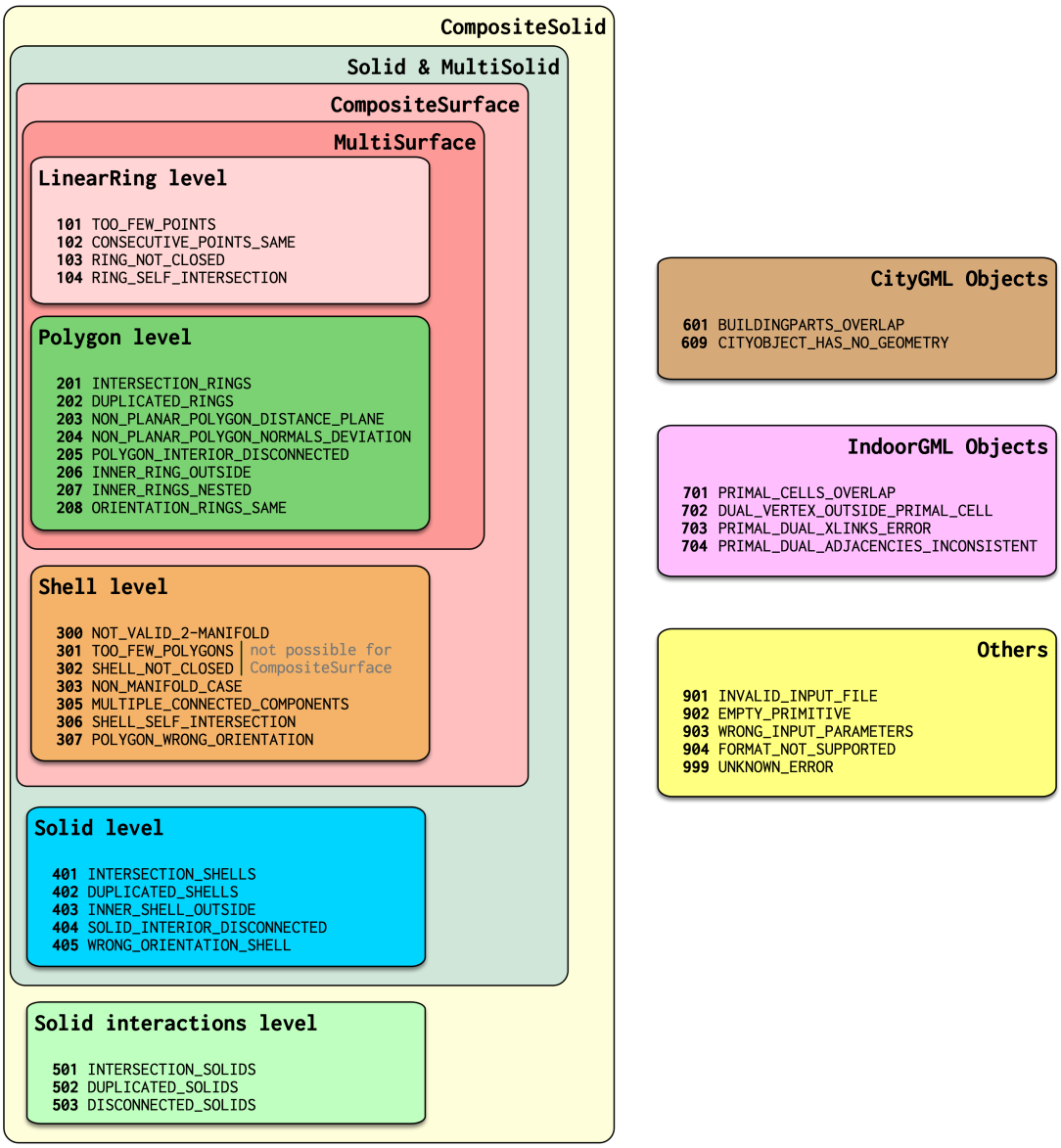


Figure 3.8 – Erreurs géométriques et topologiques et leurs codes (Ledoux, 2018)

3.4.6 Stockage des données routières

Comme expliqué dans la section 2.3, il est possible de stocker un fichier CityJSON dans un seul document dans une base de données MongoDB. Cependant, pour faciliter l'accès aux données stockées et leur manipulation, il est préférable de décomposer le fichier en plusieurs collections, puis un modèle CityJSON récupéré est constitué par la jointure des documents correspondants.

Dans MongoDB, le schéma n'est pas établi à l'avance et chaque document peut avoir sa propre structure. Cependant, il est préférable de garder une structure minimale commune afin de faciliter la manipulation des données. Chaque collection doit avoir donc un schéma de données prédéfini. Les schémas CityJSON sont utilisés, mais d'autres informations utiles, comme la taille des fichiers et la date d'insertion, peuvent être stockées.

La définition d'un schéma de données implique leur validation pendant les mises à jour et les insertions. Les règles de validation s'appliquent à chaque collection.

La librairie Mongoose prend en charge la définition et la validation des schémas JSON. Elle permet de déterminer, au préalable, le degré de rigueur avec lequel les règles de validation sont appliquées aux documents existants lors d'une mise à jour, ainsi que l'action de validation qui détermine si le système doit émettre une erreur et rejeter les documents qui ne respectent pas ces règles ou avertir des erreurs mais autoriser les documents non valides.

Measur3D, comme indiqué dans la section 2.4.4, permet le stockage, la gestion et la visualisation des jeux de données CityJSON. Les schémas JSON sont définis à l'aide de Mongoose. Chaque schéma de données correspond à une collection MongoDB et définit la structure des documents de cette collection.

3.4.7 Mise à disposition via une application web

Les modèles 3D de route peuvent servir pour différentes applications, entre autres la planification et la gestion urbaine des infrastructures routières, la conduite autonome et les simulations de trafic.

La plupart de ces applications ont besoin initialement de pouvoir visualiser et inspecter ces modèles 3D de route pour les comprendre, les exploiter et les communiquer.

La dernière partie du projet consiste alors à mettre à disposition les modèles stockés via une application web à trois couches. Les fonctionnalités de Measur3D peuvent être utilisées et améliorées pour mettre en place un système complet de gestion des fichiers CityJSON, permettant à la fois de stocker les modèles 3D, les récupérer à tout moment et les visualiser via une interface interactive.

Conclusion

Dans ce chapitre, nous avons développé une méthodologie pour la production des plateformes routières 3D cohérents géométriquement qui repose sur l'extraction des éléments linéaires de l'infrastructure routière à partir d'un nuage de points. Cela permet une modélisation tant géométrique que sémantique de l'infrastructure routière. Le modèle 3D généré est traduit en fichier CityJSON via un processus ETL spatial. Le jeu de données en sortie doit se conformer aux schémas de données et la cohérence géométrique soit vérifiée en corrigeant les erreurs géométriques et topologiques.

Un fichier CityJSON valide est ensuite chargé dans une base de données NoSQL orientée documents qui est MongoDB, et récupéré pour visualisation et inspection, via d'appels API d'une application web à trois couches, Measur3D.

Dans le chapitre suivant, nous présenterons l'implémentation détaillée de la méthodologie proposée et les résultats de traitement.

Chapitre 4

Implémentation et résultats

Introduction

Ce dernier chapitre est consacré à l'implémentation de la méthodologie présentée. Nous présenterons premièrement les étapes d'implémentation détaillée de l'approche proposée, ainsi que les résultats de chaque étape. A travers ces résultats, nous fournissons des perspectives de recherche qui peuvent être utiles aux futurs travaux sur la modélisation et l'exploitation des infrastructures routières en 3D.

4.1 Acquisition de données routières

L'acquisition de données routière avec un système de cartographie mobile est réalisée en deux phases principales : une acquisition de données sur terrain et des post-traitements. Nous présentons ces étapes dans les sections suivantes :

4.1.1 Collecte de données par MMS

La première phase d'acquisition des données routières consiste en une mission de collecte des données LiDAR par un système de cartographie mobile.

Le système de cartographie mobile utilisé pour réaliser la mission est Trimble MX2. Il permet de collecter 72000 points par secondes à une portée de 250m. La caméra de prise de vues permet de capturer, tous les cinq mètres, 6 photos qui sont converties par la suite en photographies panoramiques 360°.

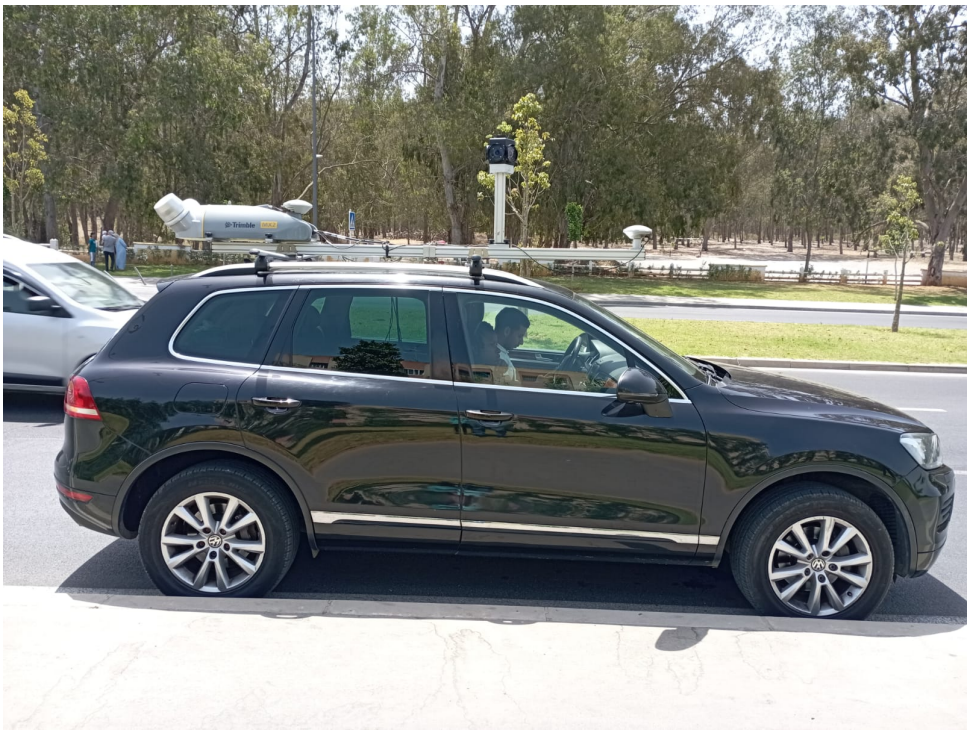


Figure 4.1 – Système de cartographie mobile Trimble MX2

En plus des données LiDAR et des photographies panoramiques, des observations GNSS/IMU sont enregistrées durant la mission. Il s'agit à la fois des données acquises par le système de cartographie mobile, et des données de correction enregistrées par un récepteur GNSS, en mode statique, placé sur un point connu en coordonnées Lambert (X,Y,H). Le nuage de points est donc rattaché en post-traitement par rapport à la base de mesure.

Une étape d'initialisation est nécessaire pour améliorer la précision des données de positionnement. Une première initialisation statique, faite pendant deux minutes, permet de résoudre l'ambiguïté de la mesure de phase, c'est-à-dire le nombre entier de cycles écoulés au début de la mesure. Ensuite, une initialisation dynamique durant trois minutes permet de synchroniser le récepteur GNSS mobile avec le système de navigation inertielle.

Le système est alimenté par le moteur du véhicule qui ne doit donc s'arrêter qu'à la fin de la mission pour assurer le fonctionnement de toutes les composantes.

En outre, la densité du nuage de points est fonction de la vitesse du véhicule. Plus la vitesse est élevée, plus la distance entre les points est grande. Une vitesse comprise entre 10 et 30 Km/h permet d'obtenir un nuage de point d'une densité moyenne inférieure ou égale à 5 cm.

Deux solutions logicielles sont utilisées durant la mission de capture des données : Applanix LV POSView et Trimble Trident Capture.

Applanix LV POSView est utilisé pour la navigation, plus précisément pour le positionnement et l'orientation du système durant la mission, tandis que Trimble Trident Capture sert à la collecte des données LiDAR et des photographies panoramiques.

4.1.2 Traitement de la trajectoire du véhicule

Avant de procéder au post-traitement des nuages de points et des photographies panoramiques, un post-traitement des observations GNSS/IMU est exigée pour calculer la trajectoire optimale du véhicule durant la mission, et par conséquent les coordonnées correctes et précises des points collectés.

Les observations GNSS/IMU du véhicule et les données de correction capturées par la base de mesure sont traitées avec le logiciel POSPac MMS.

Le traitement de la trajectoire du véhicule correspond généralement à la compensation induite par la perte des signaux engendrée par la concentration des bâtiments élevés dans les environnements urbains. Cela est réalisé en calculant les vecteurs tridimensionnels qui séparent la trajectoire de la base de mesure fixe. Ainsi, la précision de la trajectoire post-traitée est de 2 cm en planimétrie et de 5 cm en altimétrie.

4.1.3 Traitement du nuage de points et des photos

Sur la base de la meilleure trajectoire calculée, le nuage de points brut et les photographies panoramiques sont rattachés au système de coordonnées Lambert. Cela est réalisé grâce à la solution logicielle Trimble Imaging Hub.

En outre, les photographies panoramiques permettent d'affecter une couleur RVB (rouge, vert, bleu) à chaque point du nuage relevé. La Figure 4.2 montre le nuage de points résultant après post-traitement.

Le nuage de points est exporté par la suite sous format de fichier LAS (LASer), qui est un format open source d'échange de nuages de points LiDAR.

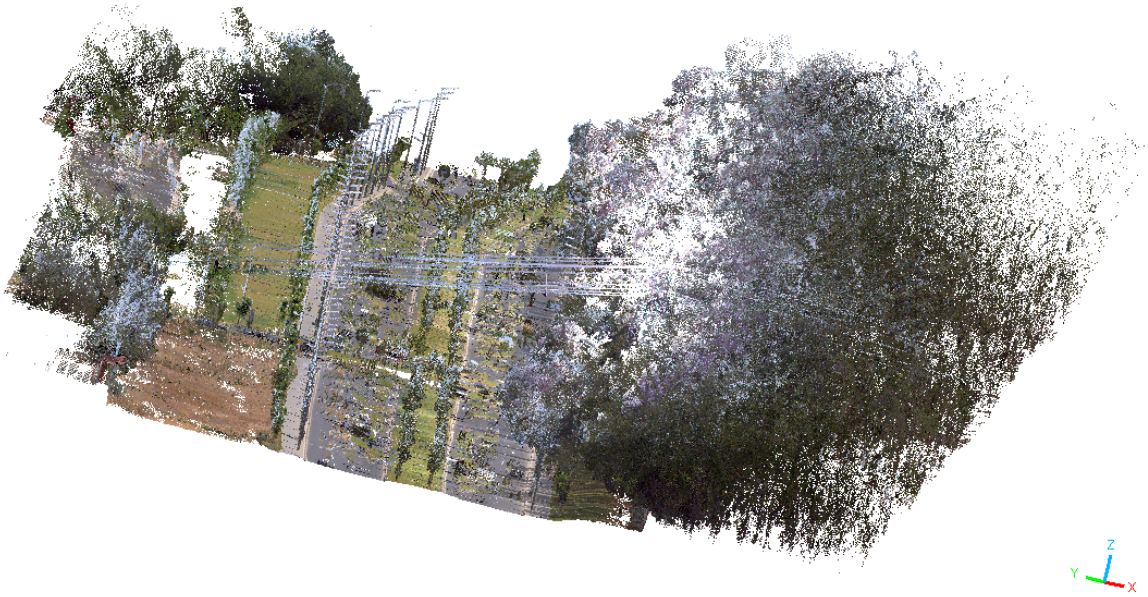


Figure 4.2 – Nuage de points 3D géoréférencé et colorié

4.2 Prétraitement

Le prétraitement des données LiDAR est primordiale pour pouvoir les utiliser correctement et efficacement dans le processus de modélisation 3D de l'espace routier. Ceci est réalisé avec le logiciel CloudCompare.

4.2.1 Nettoyage du nuage de points

Tout d'abord, une étape de nettoyage du nuage de points s'avère nécessaire pour éliminer les points indésirables présentant un bruit dans la mesure.

Deux opérations de nettoyage peuvent être mises en œuvre : un nettoyage grossier effectué manuellement avec les outils de ségmentation et de manipulation des nuages de points, et un autre automatique à l'aide des filtres basés sur la distance entre les points.

Le nettoyage grossier permet d'éliminer les détails n'appartenant pas à l'espace routier. En outre, l'élimination du bruit pour un grand jeu de données est plus compliquée, et donc deux filtres peuvent être utilisés pour un nettoyage automatique :

SOR Filter

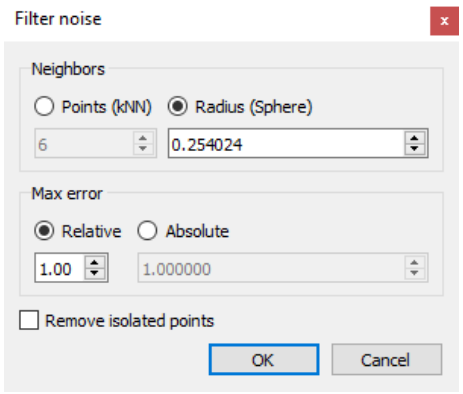
Le filtre SOR, Statistical Outlier Removal, permet d'éliminer les valeurs aberrantes dans le nuage de point en calculant la distance moyenne de chaque point à ses voisins, puis rejette les points qui sont plus éloignés que la distance moyenne plus un nombre de fois l'écart type.

Deux paramètres sont fixés : tout d'abord, le nombre de voisins qui sera utilisé pour calculer la distance moyenne D_{moy}, puis le coefficient multiplicateur n de l'écart-type σ (Figure 4.3(b)).

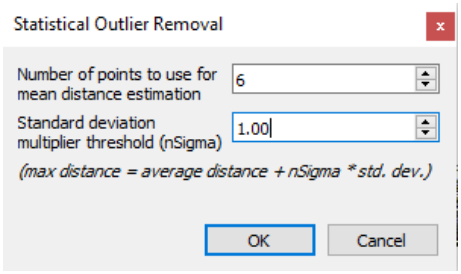
Noise Filter

Le filtre de bruit, Noise Filter en anglais, fonctionne de la même manière que le filtre SOR mais il considère la distance à la surface sous-jacente au lieu de la distance aux voisins.

Autour de chaque point du nuage, un plan est ajusté localement, à partir d'un rayon ou d'un nombre constant de voisins, puis tout point trop éloigné du plan est supprimé (Figure 4.3(a)).



(a) Noise Filter



(b) Filtre SOR

Figure 4.3 – Paramètres des filtres de bruit dans CloudCompare

La Figure 4.4 montre les résultats du nettoyage grossier et automatique du nuage de points :

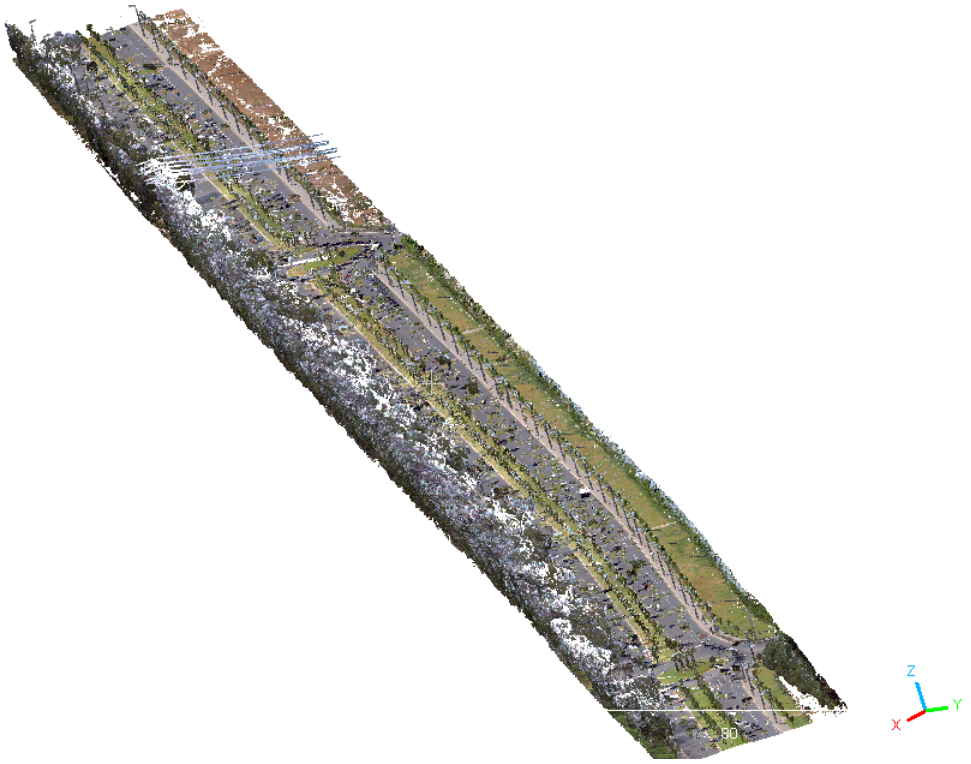


Figure 4.4 – Nuage de points après nettoyage du bruit

4.2.2 Extraction de la surface terrain

La séparation des nuages de points en points sol et sur-sol est une étape essentielle pour générer un modèle numérique de terrain (MNT) à partir de données LiDAR (W. Zhang et al., 2016).

Le filtre CSF, Cloth Simulation Filter, est un algorithme qui permet d'extraire les points sol à

partir d'un nuage de points. Il peut être ajouté et utilisé comme plugin dans CloudCompare.

Selon Yilmaz et al. (2017), le principal avantage de l'algorithme CSF, par rapport aux autres algorithmes d'extraction des points sol, est qu'il utilise moins de paramètres.

Le filtre est basé sur la méthode Cloth Simulation, qui est un algorithme utilisé dans l'infographie 3D pour simuler un tissu attaché à un objet. Le principe de l'algorithme est simple : un nuage de points LiDAR est inversé, puis un tissu rigide est utilisé pour couvrir la surface inversée. En comparant les nœuds du tissu aux points LiDAR correspondants, ces nœuds peuvent être utilisés pour générer une approximation de la surface du sol. Enfin, les points sol peuvent être extraits du nuage de points LiDAR en comparant les points LiDAR originaux et la surface générée (W. Zhang et al., 2016).

Deux types de paramètres sont à fixer avant d'exécuter le filtre CSF sur le nuage de points. Le Tableau 4.1 résume l'ensemble de ces paramètres :

Tableau 4.1 – Paramètres du filtre CSF

Paramètres généraux	**Scènes**	Définir le type de scènes du nuage de point : pente raide, relief ou plat
	Post-traitement des pentes pour les terrains isolés	S'il n'y a pas de pentes raides dans le nuage de points, il suffit de les négliger.
Paramètres avancés	**Résolution du tissu**	Taille de la grille du tissu utilisé pour couvrir le terrain
	Itérations maximales	Nombre maximum d'itérations
	Seuil de classification	Distance maximale entre les points et le terrain simulé pour classer le nuage de points en points sol et sur-sol

Le tronçon de route balayé par LiDAR est d'une faible pente. Il s'agit d'un terrain relativement plat. Les paramètres avancés ont été choisi après plusieurs tests sur le nuage de points. La Figure 4.5 montre les valeurs adoptées :

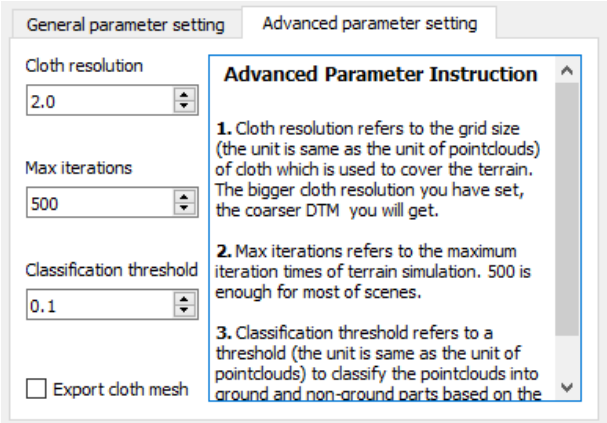


Figure 4.5 – Configuration adoptée pour extraire les points sol par le filtre CSF

La Figure 4.6 illustre les résultats de la ségmentation :

Figure 4.6 – Résultats d'extraction des points sol par le filtre CSF

4.3 Modélisation 3D des routes

Le modèle numérique du terrain (MNT) extrait à partir du nuage de points contient les points appartenant à la surface de la route.

Pour pouvoir modéliser l'espace routier, les caractéristiques linéaires de la route sont extraites sur la base du MNT. Cela est réalisé avec Autodesk InfraWorks. Ensuite, les surfaces sémantiques sont construites à partir de ces éléments linéaires.

4.3.1 Extraction des éléments linéaires

Autodesk InfraWorks fournit des outils pour pouvoir générer la surface terrain à partir du nuage de points, puis en extraire automatiquement ou manuellement des caractéristiques linéaires.

Pour pouvoir importer le nuage de points sur InfraWorks, il est nécessaire de convertir le fichier LAS en format RCP (acronyme de ReCap-Project) qui stocke les informations des métadonnées d'un projet Auotdesk ReCap Pro, associé à un fichier RCS, ReCap-Scan, qui correspond à un nuage de points enregistrés sous ReCap (Figure 4.7).

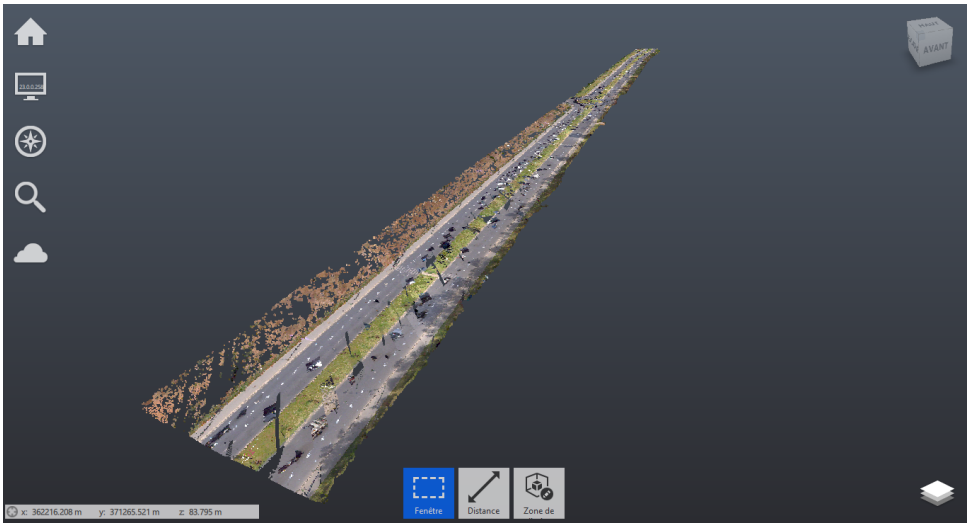


Figure 4.7 – Nuage de points importé dans Autodesk ReCap Pro

Une fois importées dans InfraWorks, les données peuvent être assignées à un système de coordonnées (Projection conique conforme de Lambert Zone I - EPSG :26191), une échelle et une translation.

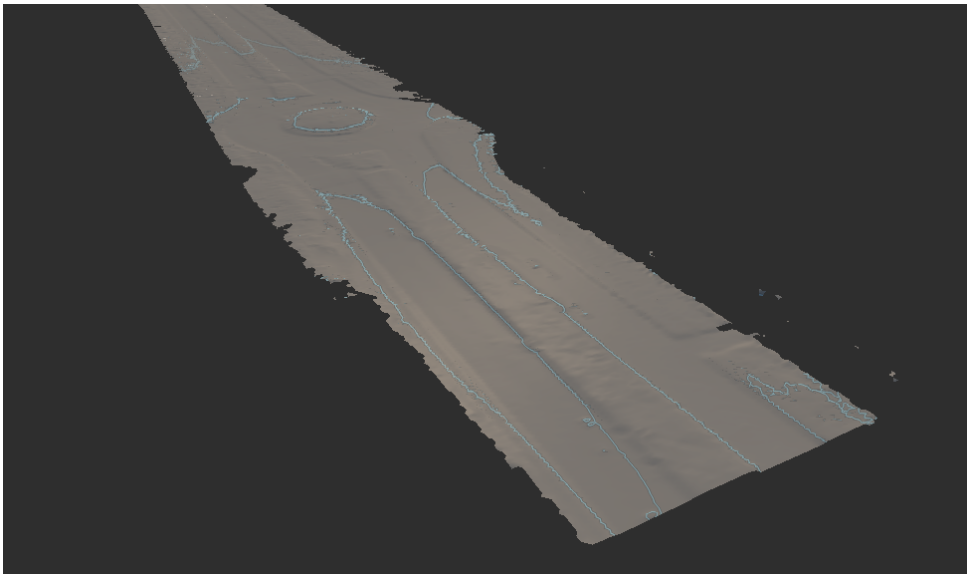


Figure 4.8 – Surface terrain générée dans InfraWorks

Pour extraire les caractéristiques linéaires de la route à partir du nuage de points importé, il faut tout d'abord générer la surface terrain. Comme illustré dans la Figure 4.8, l'outil *Terrain par nuage de points* permet de générer la surface terrain sous format raster.

Sur la base de la surface terrain générée, il est possible d'extraire les caractéristiques linéaires de l'espace routier, notamment les lignes de rupture. Comme illustré dans la Figure 4.9, cela est réalisé avec l'outil *Extraction d'objets linéaires*.

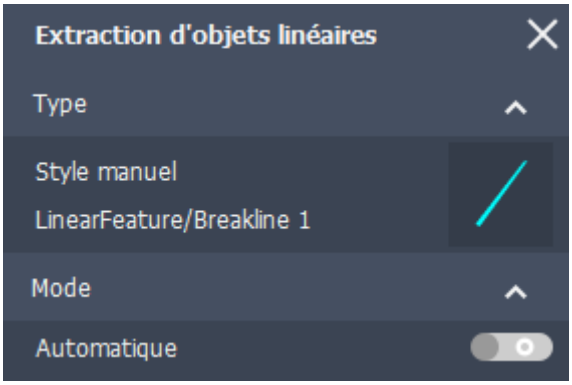


Figure 4.9 – Outil d'extraction d'objets linéaires dans InfraWorks

L'extraction des objets linéaires effectuée dans InfraWorks peut être qualifiée de semi-automatique. Il suffit de choisir au moins deux points le long de la ligne de rupture pour commencer l'extraction. Cette méthode fonctionne correctement sur des données très homogènes, sinon il y aura des lacunes dans l'extraction.

Toutefois, les imperfections dans les lignes extraites peuvent être corrigées avec la *vue du profil en travers* (Figure 4.10). Elle permet une extraction et un contrôle qualité rapides et simples en contrôlant l'emplacement des sommets.

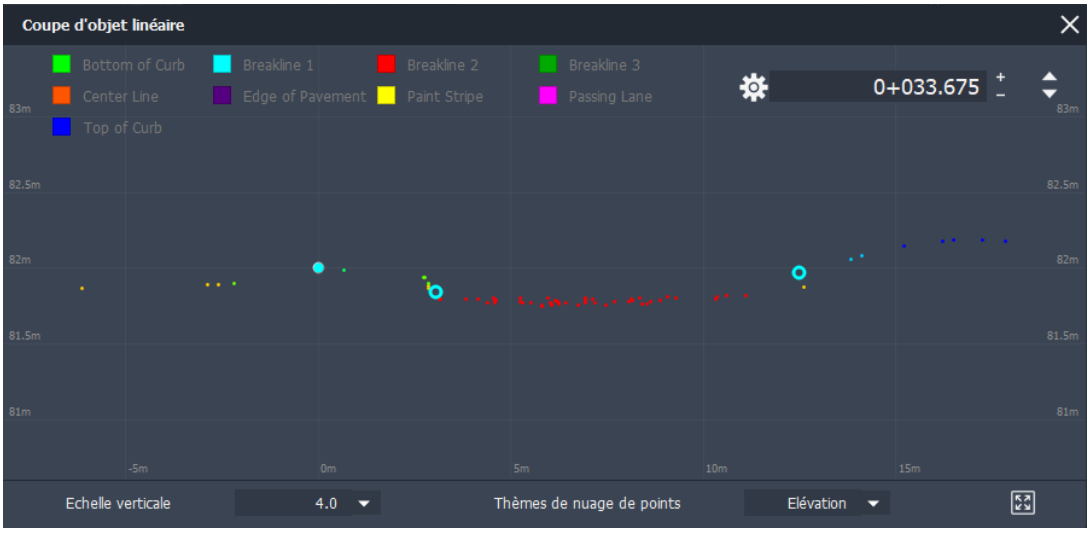


Figure 4.10 – Vue en coupe transversale des lignes extraites

D'autres lignes sont ajoutées manuellement pour pouvoir modéliser l'espace routier selon les spécifications de CityJSON.

Les résultats de l'extraction des éléments linéaires sont illustrés dans la Figure 4.11 :

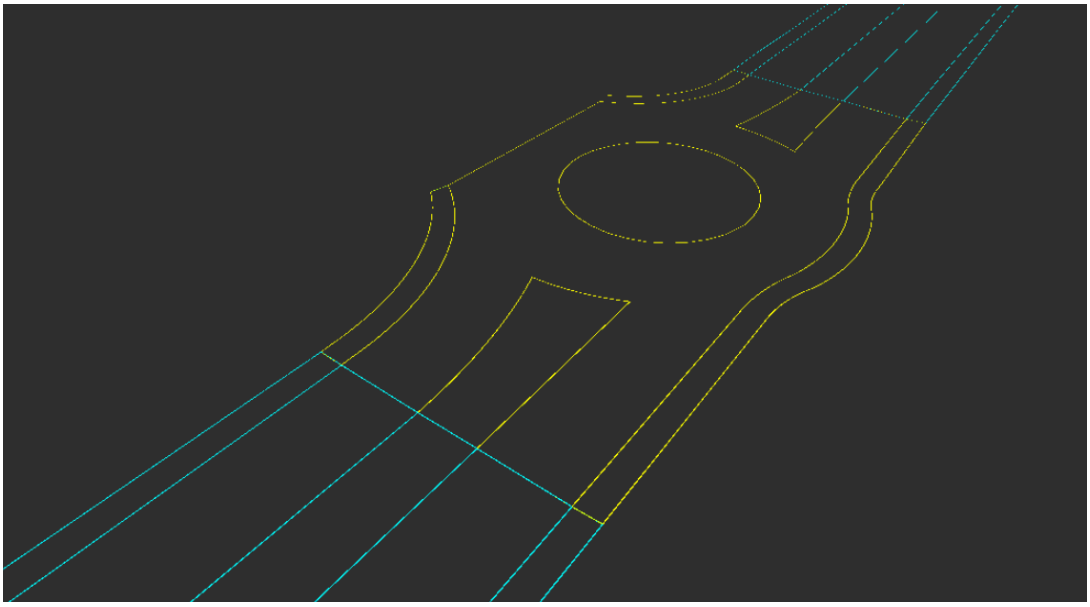


Figure 4.11 – Éléments linéaires extraits à partir de la surface terrain

4.3.2 Établissement d'un système de codification

La modélisation 3D de l'espace routier est effectué dans FME. Il s'agit d'une modélisation à la fois géométrique et sémantique.

Pour se conformer aux spécifications du module Transport de CityGML 3.0 telles que décrites dans la section 1.10, il est nécessaire de séparer les lignes appartenant aux différentes sections, puis aux différentes surfaces sémantiques.

Les éléments linéaires extraits dans InfraWorks peuvent être exportés au format Shapefile. Chaque élément peut enregistrer des attributs, entre autres des étiquettes et des codes d'entités (Figure 4.12).

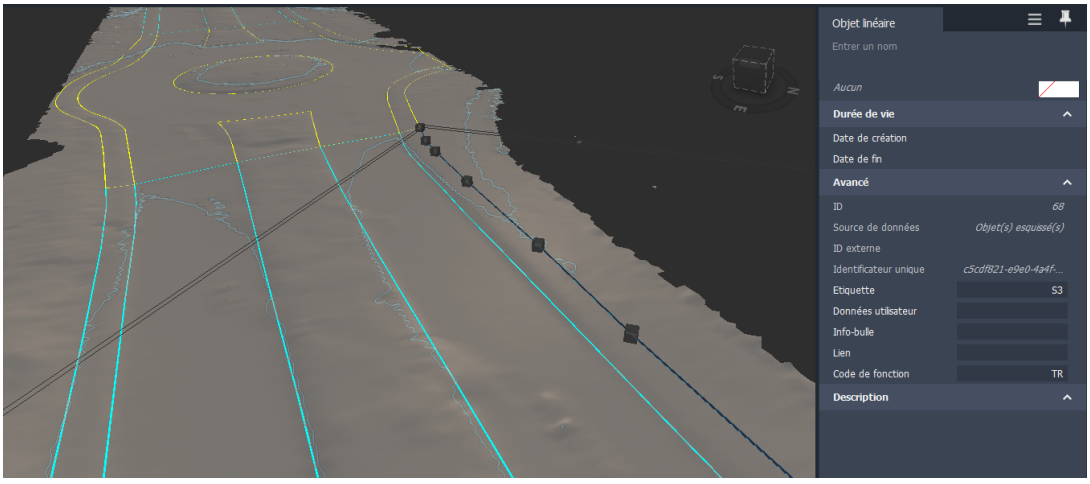


Figure 4.12 – Attributs d'un objet linéaire

Les étiquettes, *Tags* en anglais, contiennent la section, l'intersection ou le rond-point auquel appartient chaque objet linéaire. Par exemple, les lignes de la première section peuvent avoir *S1*

comme valeur d'étiquette, *S2* pour la deuxième section, *R1* pour le premier rond-point et ainsi de suite.

Les surfaces sémantiques peuvent être séparées sur la base d'un système de codification. L'attribut *FeatureCode* contient un code prédéfini selon la nature de la ligne de rupture.

Deux caractères sont utilisés pour chaque ligne de rupture appartenant à une seule surface sémantique. S'il s'agit d'une limite commune entre deux surfaces sémantiques, quatre caractères sont utilisés, en référence aux deux surfaces.

Le Tableau 4.2 résume les codes assignés aux différentes lignes :

Tableau 4.2 – Système de codification des éléments linéaires

Code	**Élément linéaire**
TR	Limite d'un trottoir
CH	Limite d'une chaussée
VT	Limite d'un espace vert
CHTR	Limite entre un trottoir et une chaussée
CHVT	Limite entre un espace vert et une chaussée
CHVTH	Limite entre un espace vert central d'un rond-point et une chaussée (H pour *Hole*)

4.3.3 Modélisation 3D

La modélisation géométrique et sémantique du tronçon routier est réalisée à l'aide de FME Workbench. Les données d'entrée correspondent au Shapefile des objets linéaires de la route extraits à partir de la surface terrain.

La première étape consiste à séparer les différentes classes (sections, intersections et ronds-points), pour pouvoir les traiter individuellement.

Dans FME, il est possible de filtrer un jeu de données, vers un ou plusieurs ports de sortie, en fonction de conditions de test en utilisant le transformateur *TestFilter*. Une condition de test consiste en une ou plusieurs clauses de test et une méthode de comparaison spécifiée.

Comme illustré dans la Figure 4.13, il est possible de séparer les différentes classes sur la base du premier caractère de la valeur d'attribut *Tag*. Si l'étiquette commence par *S*, il s'agit donc d'une section, sinon *I* pour intersection et *R* pour rond-point.

	Test Condition	Output Port
If	@Value(Tag) BEGINS_WITH S	Section
Else If	@Value(Tag) BEGINS_WITH I	Intersection
Else If	@Value(Tag) BEGINS_WITH R	Roundabout
Else If		
Else	<All Other Conditions>	<UNFILTERED>

Figure 4.13 – Conditions de test pour séparer les sections des intersections et ronds-points

Ensuite, les objets de chaque classe sont séparés à l'aide du transformateur *AttributeFilter* qui dirige les entités en entrée vers différents ports de sortie en fonction de la valeur d'un attribut. L'ensemble des valeurs d'attributs possibles peut être saisi manuellement, ou extrait d'une source d'entrée dans la boîte de dialogue des propriétés. Dans ce cas, l'attribut utilisé est *Tag*, qui correspond à l'identifiant d'une section ou d'une intersection.

La Figure 4.14 résume le processus de séparation des sections, intersections et ronds-points :

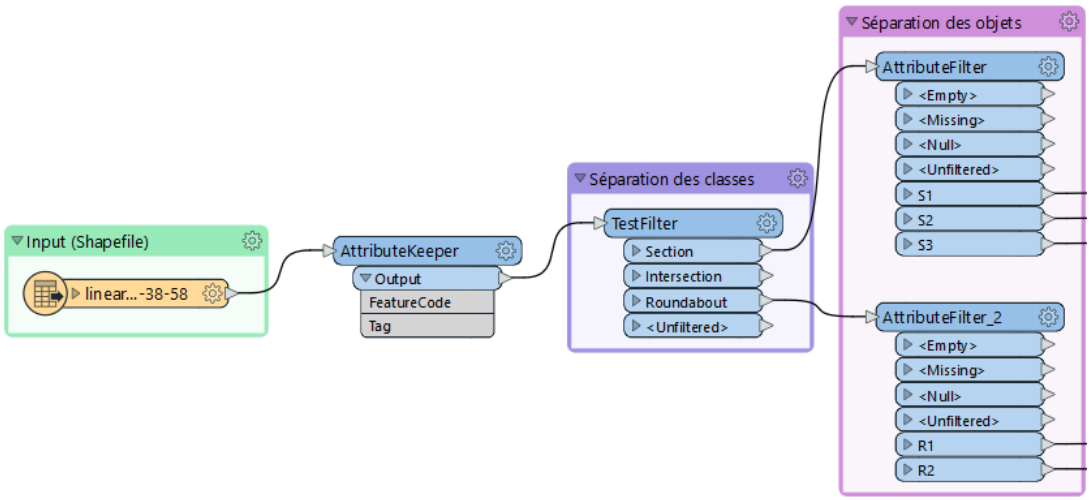
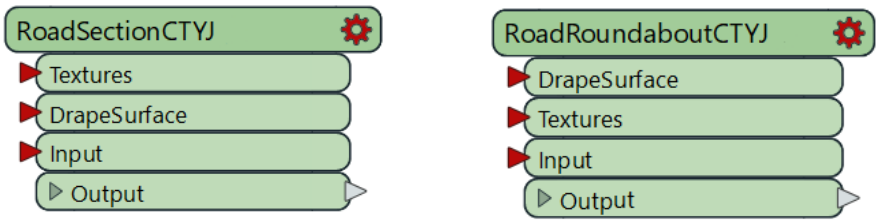


Figure 4.14 – Processus de séparation des sections, intersections et ronds-points

Le transformateur *AttributeKeeper* permet de supprimer tous les attributs à l'exception de ceux spécifiés comme devant être conservés : *FeatureCode* et *Tag*.

Chaque objet est modélisé individuellement. Des transformateurs personnalisés sont créés pour pouvoir traiter deux classes différentes : section de route et rond-point (Figure 4.15). Ils prennent en entrée les éléments linéaires de la route, des textures sous format image pour transférer tous les détails visuels de l'objet aux surfaces modélisées et des surfaces planes pour corriger les erreurs de planarité si nécessaire (Section 4.5.3).



(a) Section de route (b) Rond-point

Figure 4.15 – Transformateurs personnalisés

Les surfaces de la route sont modélisées séparément. Les objets linéaires sont filtrés sur la base du code attribué puis combinés pour construire trois surfaces sémantiques : espace vert, chaussée et trottoir.

Le transformateur *LineCombiner* permet de connecter les lignes formant chaque surface pour créer des polylignes plus longues. Sur les sommets communs entre les lignes combinées, il conserve les valeurs z et les mesures de la ligne subséquente.

Une fois connectés, les objets linéaires d'entrée sont transformés en polygone en liant leur point de départ au point d'arrivée avec le transformateur *LineCombiner*.

Ensuite, le transformateur *Orientor* permet d'ajuster l'orientation des polygones générés. Cela permet de contrôler le sens d'extrusion pour générer des solides 3D dans le sens correct.

Si le type d'orientation *Left hand rule* est choisi, les sommets de la bordure extérieure du polygone sont classés dans le sens inverse des aiguilles d'une montre. Ainsi, les polygones sont orientés vers le sens positif de l'axe Z. Cela peut être vérifié en calculant la normale à la surface à l'aide du transformateur *PlanarityFilter*.

La Figure 4.16 illustre le processus de transformation des éléments linéaires en polygones :

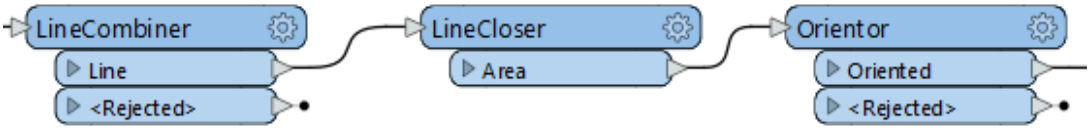


Figure 4.16 – Transformation des objets linéaires en polygones

Par la suite, le transformateur *extruder* est utilisé pour créer des solides à partir des polygones générés. Le paramètre *Direction* permet de spécifier la direction que le vecteur d'extrusion doit suivre, qui peut être la normale sur la surface. De plus, le paramètre *Distance* permet de spécifier la distance d'extrusion.

Les solides créés par extrusion sont convertis en *CompositeSurface* à l'aide du transformateur *GeometryCoercer* qui permet de réinitialiser le type de géométrie des entités en entrée. Dans ce cas, le type *CompositeSurface* est préféré pour pouvoir, d'une part se conformer aux types de géométries de la classe *Road* de CityGML 3.0 et d'autre pour appliquer des textures sur les géométries construites.

Les informations sémantiques sont attribuées à chaque surface modélisée en ajoutant les attributs correspondants, selon les spécifications CityJSON.

La Figure 4.17 illustre le processus d'extrusion et d'ajout de la sémantique aux surfaces modélisées :

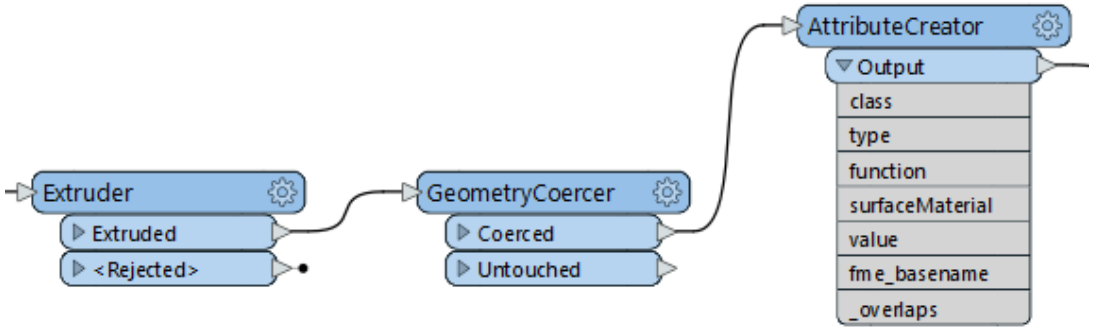


Figure 4.17 – Processus de modélisation des surfaces sémantiques de la route

Les informations sémantiques ajoutées doivent être traitées et enregistrées correctement dans le fichier CityJSON. Cette tâche ne peut pas être réalisée facilement à l'aide des transformateurs de FME vu la complexité des solides créés par extrusion et le nombre de surfaces générées.

L'utilisation de Python permet d'étendre les fonctionnalités de FME et d'effectuer des manipulations complexes. Cela est possible en utilisant le transformateur *PythonCaller* (Figure 4.18). Ainsi, un script est écrit pour pouvoir traiter les surfaces sémantiques (Annexe 2).

Ces informations sont par la suite ajoutées comme attribut à la section ou à l'intersection modélisée, puis à leur géométrie en appliquant un correctif (voir la section 4.4.2).

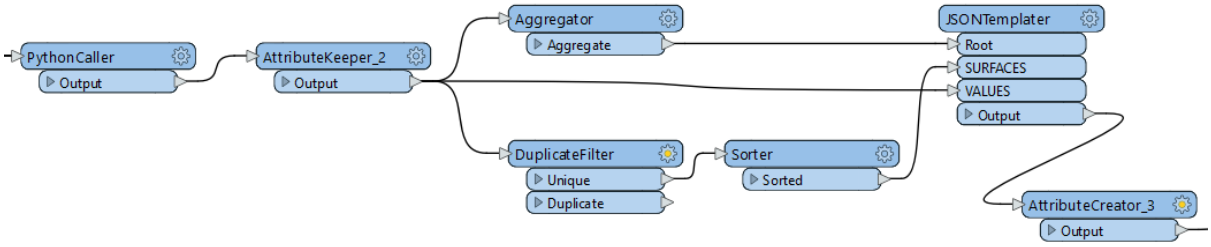


Figure 4.18 – Traitement des surfaces sémantiques

Le transformateur *Aggregator* permet de combiner les géométries de toutes les surfaces sémantiques modélisées. La géométrie en sortie est de type *MultiSurface*.

DuplicateFilter est utilisé pour éliminer les valeurs en double des surfaces sémantiques pour pouvoir les enregistrer par la suite dans la propriété *surfaces* de l'objet *semantics*.

Les surfaces sémantiques des carrefours circulaires sont modélisées de la même manière que les sections, à l'exception des espaces verts.

L'espace vert du carrefour est entièrement enfermé dans la surface de la chaussée. Le transformateur *DonutBuilder* est donc utilisé pour créer un trou dans la surface chaussée.

Des textures peuvent être appliquées pour transférer les détails visuels de la route aux surfaces modélisées.

Les textures sont extraites à partir des photographies panoramiques (Figure 4.19), puis appliquées aux différentes surfaces sémantiques de l'espace routier.

Figure 4.19 – Exemple d'une parie de photographie panoramique 360°

Les textures sont ajoutées à l'aide du transformateur *AppearanceSetter* qui permet de définir des styles d'apparence aux surfaces. Le processus d'habillage de texture est illustré dans la Figure 4.20 :

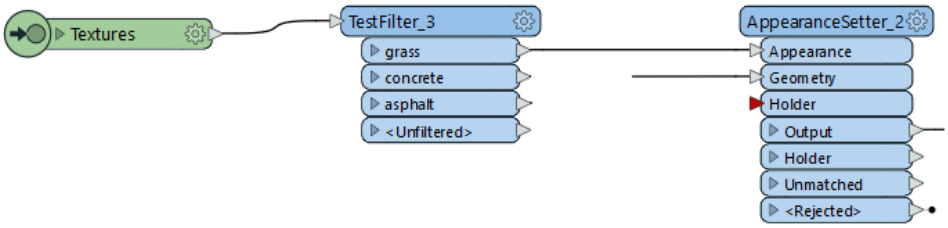


Figure 4.20 – Modèle 3D de route texturé

Le résultat de modélisation et d'habillage de texture est illustré dans la Figure 4.21 :

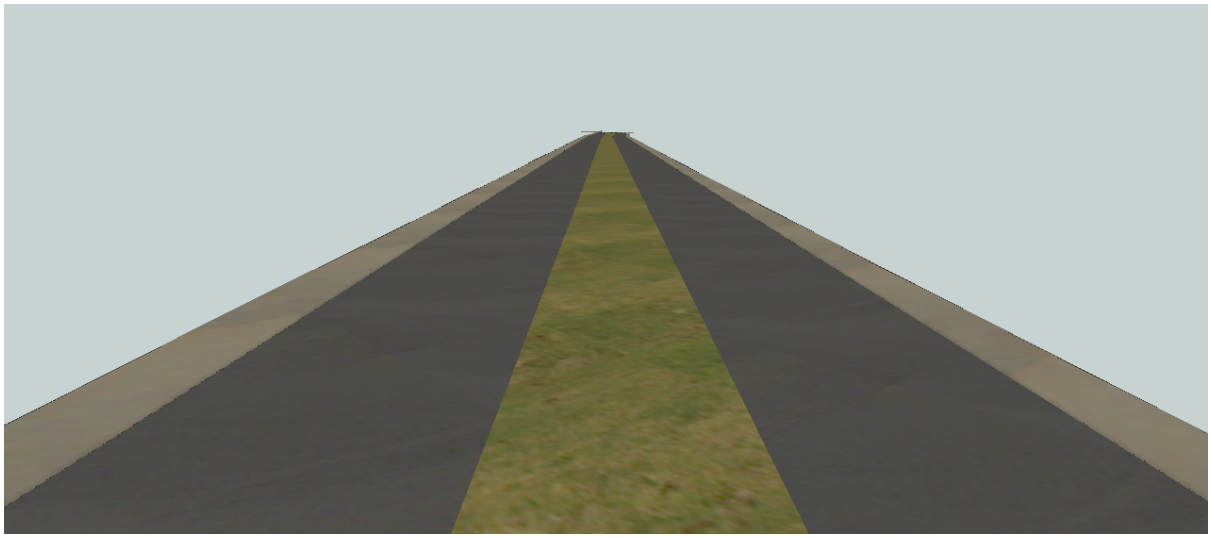


Figure 4.21 – Modèle 3D de route texturé

4.4 Traduction en fichier CityJSON

Après modélisation géométrique et sémantique de l'espace routier, le modèle 3D généré peut être traduit en fichier CityJSON. Les étapes d'écriture d'un jeu de données CityJSON conforme aux spécifications de la dernière version publiée sont présentées dans les sections suivantes :

4.4.1 Écriture d'un fichier CityJSON

FME Workbench est capable de lire et d'écrire plusieurs formats de données spatiales et non spatiales, entre autres CityGML et son encodage CityJSON.

Des identifiants uniques universels (UUID pour l'acronyme anglais) sont attribué, à l'aide de *UUIDGenerator*, à chaque objet avant la traduction en fichier CityJSON (Figure 4.22).

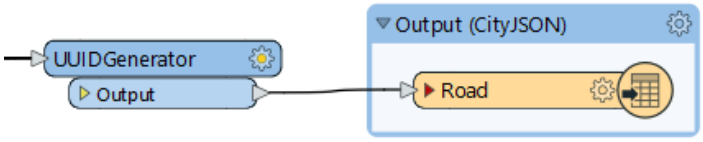


Figure 4.22 – Traduction du modèle 3D en fichier CityJSON

4.4.2 Application d'un correctif

Bien que FME permette de traduire un modèle 3D en format CityJSON, seule la version 1.0.1 est supportée.

Pour se conformer aux spécifications CityJSON 1.1, il faut appliquer un correctif sous forme d'un script à exécuter pour modifier la syntaxe du fichier.

Le correctif est écrit en JavaScript dans un environnement Node.js (Annexe 1) . Le module *fs* est utilisé pour interagir avec le système de fichiers.

Les modifications appliquées concernent :

— La version du fichier CityJSON (*version*) ;

— Le système de coordonnées (*referenceSystem*) ;
— La matrice de transformation (*transform*) ;
— Le type de la géométrie des objets *Road* (MultiSurface au lieu de MultiLineString)
— Surfaces sémantiques (*semantics*).

Comme illustré dans la Figure 4.23, une première visualisation du fichier CityJSON peut être effectuée avec Ninja Viewer.

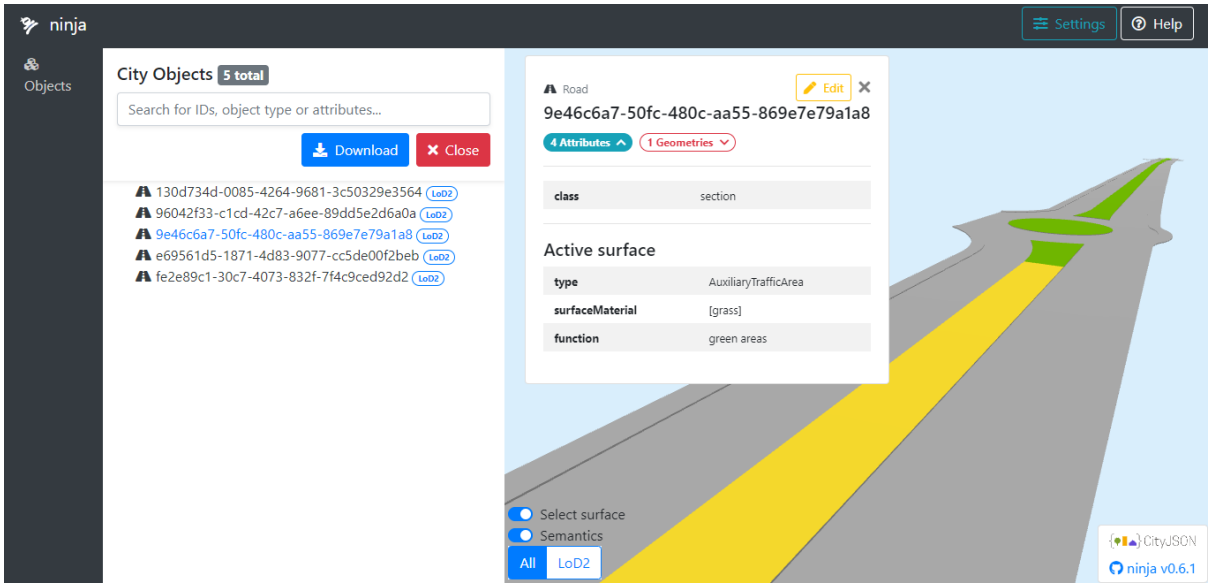


Figure 4.23 – Jeu de données CityJSON visualisé avec Ninja Viewer

4.5 Validation du fichier CityJSON

Le fichier CityJSON généré doit être valide avant de le charger dans la base de données.

Comme indiqué dans la section 3, la validation est réalisée en deux étapes : une validation du schéma de données, puis une validation des primitives géométriques.

Toute non-conformité aux spécifications CityJSON, et toute erreur topologique et/ou géométrique doivent être corrigées.

4.5.1 Validation du schéma de données

La conformité au schéma de données CityJSON est vérifiée à l'aide de *cjval* disponible comme une application web : `https://validator.cityjson.org/`.

Le fichier est valide à 100%, c'est-à-dire que tous les critères de validation indiqués dans la section 3.4.4 sont satisfaits.

La Figure 4.24 montre les résultats de la validation du schéma de données :

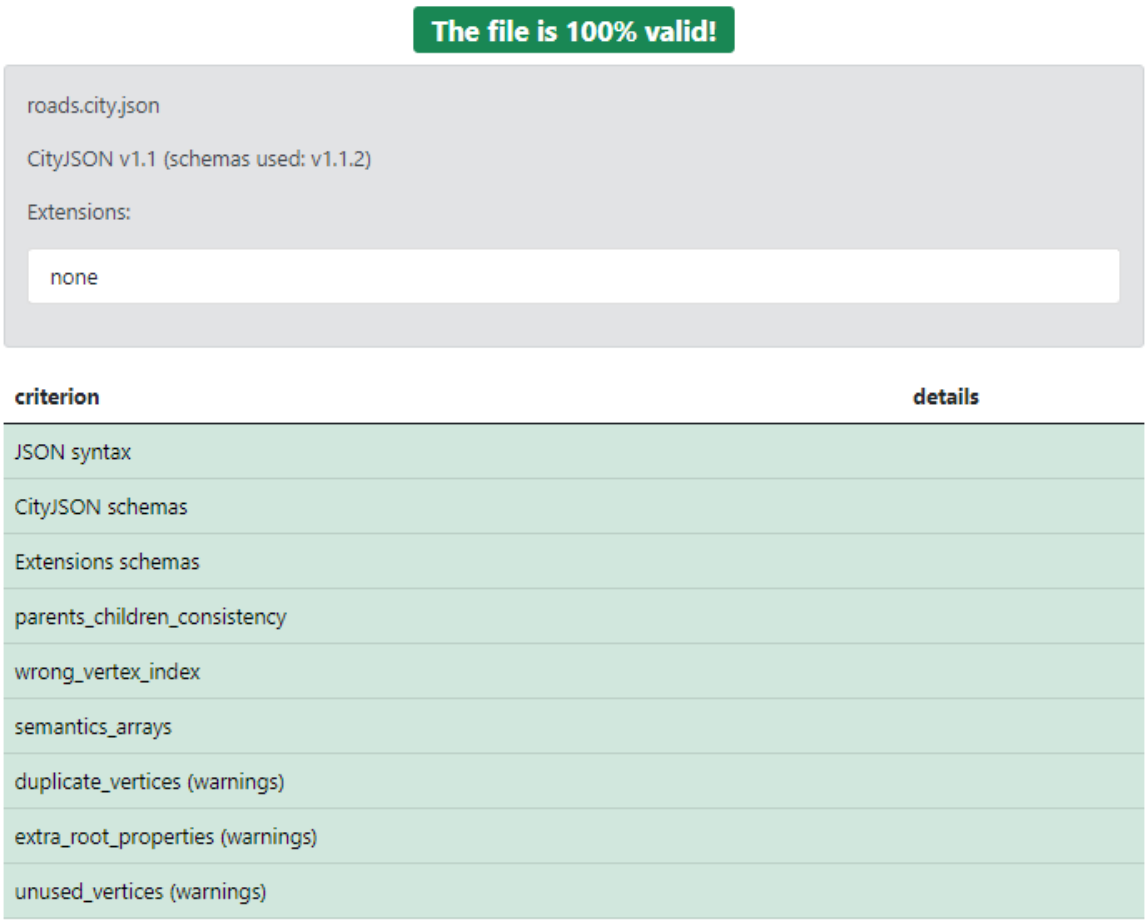


Figure 4.24 – Résultats de la validation du schéma de données

4.5.2 Validation des primitives géométriques

Un fichier CityJSON n'est totalement valide que si les primitives géométriques du modèle 3D sont valides et conformes à la norme internationale ISO 19107 (voir la section 3.4.5).

Val3dity est utilisé pour valider les géométries du modèle 3D de route et détecter les erreurs géométriques et topologiques. Il est disponible comme une application web : `http://geovalidation.bk.tudelft.nl/val3dity/`.

Comme détaillé dans le Tableau 4.3, deux erreurs de géométrie ont été détectées :

Tableau 4.3 – Erreurs de géométrie détectées dans le modèle 3D

Code	Erreur	Description
203	NON_PLANAR_POLYGON_DISTANCE_PLANE	Erreur de non-planarité (tolérance de 1 cm)
104	RING_SELF_INTERSECTION	Erreur d'auto-intersection

La première erreur correspond à une erreur de non-planarité. En fait, un polygone doit être plan, c'est-à-dire que tous les sommets doivent se trouver sur un plan, avec une tolérance de 1 cm. Ce

plan est ajusté aux sommets du polygone par la méthode des moindres carrés, et si la distance d'un sommet est supérieure à la tolérance, une erreur de planarité est détectée.

L'erreur d'auto-intersection peut être due à des nœuds en double ou un polygone qui se coupe en un point. Pour un polygone 3D, l'auto-intersection est vérifiée par rapport à sa projection sur le plan le mieux ajusté (par la méthode des moindres carrés) aux sommets du polygone.

4.5.3 Correction des erreurs de géométrie

Les erreurs de géométrie détectées doivent être corrigées. Chaque type d'erreur peut être traité différemment :

Correction des erreurs d'auto-intersection

Pour pouvoir corriger les erreurs d'auto-intersection, il est nécessaire de détecter les objets qui portent l'erreur et les sommets qui provoquent l'invalidité.

Le transformateur *GeometryValidator* peut être utilisé pour identifier ces objets, tout long du processus de modélisation géométrique de l'espace routier.

En fait, les erreurs d'auto-intersection détectées sont dues à la combinaison des lignes et leur fermeture, à l'aide du transformateur *LineCloser*, pour former des poylgones.

Cette erreur est corrigée en calculant les intersections entre les lignes formant le polygone, avant de les connecter à l'aide de *LineCombiner*, en créant des noeuds partout où se trouve une intersection (Figure 4.25).

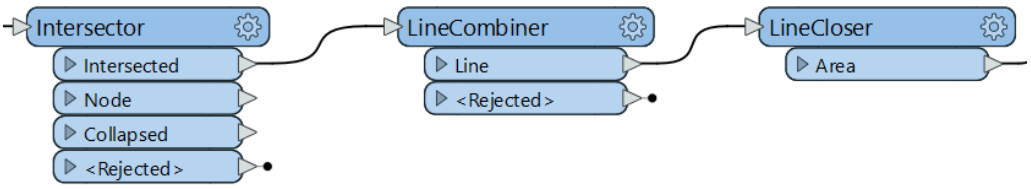


Figure 4.25 – Processus de correction des erreurs d'auto-intersection

Correction des erreurs de non-planarité

La correction des erreurs de non-planarité des polygones 3D semble plus difficile. Cela revient à l'information sémantique attribuée à chaque surface, ou polygone, de l'espace routier.

L'une des solutions possibles est de créer, par triangulation, une surface TIN (Triangulated Irregular Network) à partir des sommets des polylignes combinés, au lieu de les fermer pour former un seul polygone 3D.

Une triangulation est, dans sa forme la plus simple, la subdivision d'un objet géométrique en triangles.

Selon Ohori et al. (2012), un polygone est triangulé en créant des segments de ligne entre les sommets du polygone sans passer par son extérieur, qui subdivisent le polygone en triangles (Figure 4.26).

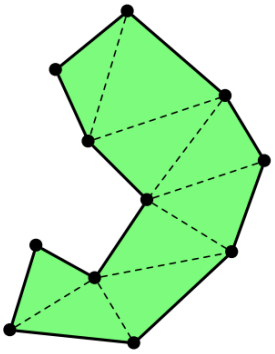


Figure 4.26 – Illustration d'un polygone triangulé

Une deuxième approche peut être considérée. Au lieu de trianguler le polygone non-plan, les sommets du polygone sont superposés à une surface plane, qui peut être le plan le mieux ajusté par la méthode des moindres carrés au nuage de points. Le polygone résultant est donc plan.

Dans FME, le transformateur *SurfaceDraper* permet de superposer les polygones générés sur la surface plane.

Les résultats de la validation des primitives géométriques après correction des erreurs sont illustrés dans la Figure 4.27.

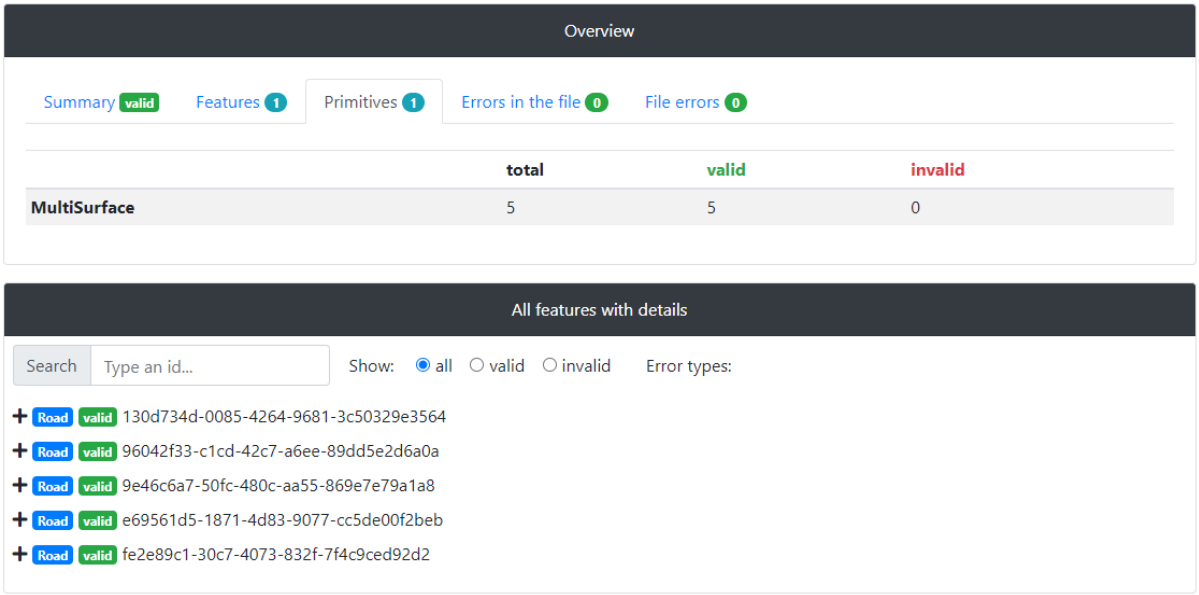


Figure 4.27 – Résultats de validation des primitives géométriques après corrections

4.6 Stockage et visualisation des modèles 3D de routes

Un jeu de données CityJSON valide est stocké par la suite dans une base de données MongoDB. Les fonctionnalités de Measur3D permettent le stockage, la récupération et la visualisation des fichiers CityJSON. La démarche suivie pour le stockage et la gestion des modèles sémantiques de routes est détaillée dans les sections suivantes :

4.6.1 Chargement dans la base de données

Measur3D propose un RESTful API (voir la section 2.4.2) qui permet de manipuler des modèles 3D de ville sur une base de données orientée document. Par défaut, il est déployé sur localhost :3001/measur3d.

Le point de terminaison *uploadCityModel* permet d'importer un modèle CityJSON dans la base de données. La méthode utilisée est *POST* qui envoie des données au serveur.

En cas de succès, le code de statut HTTP 201 est renvoyé indiquant que la requête a réussi et qu'une ressource a été créée en conséquence. Sinon, une erreur 500 est renvoyée qui indique que le serveur a rencontré un problème inattendu qui l'empêche de répondre à la requête.

La définition des schémas de données, la connexion à la base de données et la gestion des données stockées sont assurées par la librairie Mongoose.

Un modèle Mongoose fournit une interface avec la base de données pour créer, interroger, mettre à jour et supprimer des documents.

La création d'un modèle Mongoose comprend principalement deux parties :

Un schéma qui définit les propriétés du document par le biais d'un objet dont le nom de la clé correspond au nom de la propriété dans la collection. Ensuite, le modèle est exporté en transmettant le nom de la collection et une référence à la définition du schéma.

L'exemple suivant correspond à la création d'un schéma de données pour l'insertion des objets de la ville de la classe *Transportation* :

```
let Transportation = mongoose.model("CityObject").discriminator(
  "Transportation",
  new mongoose.Schema({
    type: {
      type: String,
      required: true,
      enum: ["Road", "Railway", "TransportSquare"]
    },
    geometry: [mongoose.Schema.Types.Mixed]
  })
);

module.exports = {
  insertTransportation: async (object, jsonName) => {
    object["CityModel"] = jsonName

    var temp_geometries = [];

    for (var geometry in object.geometry) {
      var authorised_type = ["MultiSurface", "CompositeSurface", "
          MultiLineString", "MultiPoint"];
      if (!authorised_type.includes(object.geometry[geometry].type)) {
        throw new Error(object.type + " is not a valid geometry type.")
            ;
        return;
      }

      temp_geometries.push(await Geometry.insertGeometry(object.
          geometry[geometry], jsonName));
    }
```

```
28
29      object.geometry = temp_geometries;
30
31      var transportation = new Transportation(object);
32
33      try {
34        let element = await transportation.save();
35        return element.id;
36      } catch (err) {
37        console.error(err.message);
38      }
39    },
40    Model: Transportation
41  };
```

4.6.2 Mise à disposition via une application web

En plus du stockage des jeux de données CityJSON, Measur3D permet de récupérer les données stockées, les gérer et les visualiser dans un navigateur web.

L'interface utilisateur de Measur3D est créé avec React.js. Elle comprend trois composantes principales : une liste des fichiers CityJSON stockés, un visualisateur des modèles 3D et un gestionnaire des attributs.

Cependant, la version actuelle ne supporte pas les surfaces sémantiques. Le visualisateur et le gestionnaire des attributs permettent d'inspecter les objets d'un modèle 3D de ville et leurs attributs, sans accès à la sémantique.

Les améliorations proposées servent donc à supporter l'information sémantique et l'accès aux différentes surfaces de l'espace routier.

4.6.2.1 Récupération d'un jeu de données CityJSON

Le point de terminaison API *getCityModelsList* permet de récupérer dans une liste tous les noms de modèles de villes stockés dans la base de données. La méthode utilisée est *GET* qui sert à demander des données au serveur. Deux réponses sont possibles :

Succès de la requête avec un code de statut de réponse HTTP 200. Les données récupérées est une liste de de modèles 3D de ville stockés. Dans ce cas, un seul modèle est stocké dans la base de données :

```
[{"name":"roads","nbr_el":5,"filesize":"61.71 KB"}]
```

En cas d'**échec de la requête**, une erreur 404 est envoyé par le serveur et indique qu'aucun modèle n'est stocké dans la base de données.

Un deuxième point de terminaison *getNamedCityModel* permet d'obtenir un modèle de ville spécifique à partir de la base de données. Cette méthode prend comme paramètre le nom du modèle à récupérer et renvoie deux réponses : succès ou échec de la requête. En cas de succès, un objet JSON correspondant au jeu de données CityJSON est renvoyé.

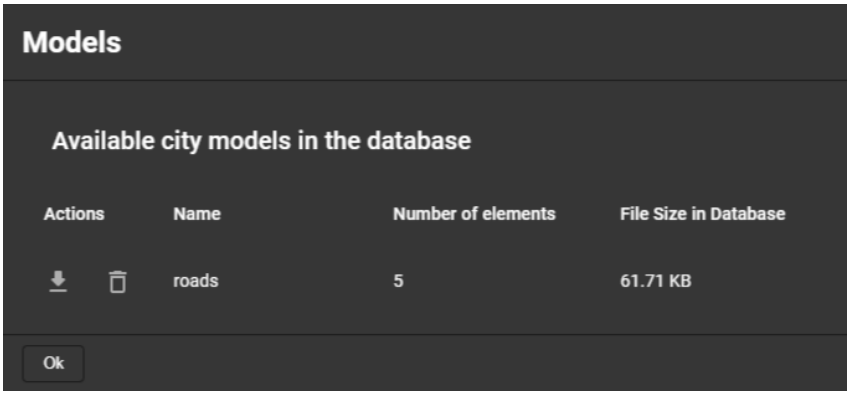


Figure 4.28 – Liste des modèles de villes disponibles dans la base de données

Le Tableau 4.4 résume les deux méthodes de récupération des jeux de données CityJSON :

Tableau 4.4 – Méthodes utilisées pour récupérer les jeux de données CityJSON (Nys & Billen, 2021)

	getCityModelsList	getNamedCityModel
Méthode	GET	GET
URL	/measur3d/getNamedCityModel	/measur3d/getCityModelsList
Paramètres	Aucun paramètre requis	name : string (nom unique du modèle de ville)
Succès	Code : 200 - [String]	Code 200 - [{ JSON Object }]
Échec	Code 404 - { error : "There is no CityModels in the DB." }	Code 404 - { error : "There is no CityModel with this name in the DB." }

Les requêtes sont exécutées depuis l'interface utilisateur à l'aide du module client *axios*, qui envoie la requête au serveur et renvoie les réponses HTTP au client.

```
1 axios
2     .get("http://localhost:3001/measur3d/getNamedCityModel", {
3       params: {
4         name: cm_name,
5       },
6     })
7     .then()
```

4.6.2.2 Visualisation des modèles 3D de route

La visualisation des modèles 3D de ville est réalisée à l'aide de Three.js qui permet de créer des scènes 3D dans un navigateur web.

Une fois un modèle de ville est récupéré, les objets *CityObjects* sont convertis en maillage et ajoutés au visualisateur.

La version actuelle de Measur3D transforme les objets du modèle en maillages pour pouvoir les

ajouter à la scène 3D. Chaque maillage comprend toutes les primitives géométriques combinées de l'objet. Par conséquent, il n'est pas possible d'accéder aux surfaces sémantiques.

La solution proposée consiste à parcourir les surfaces qui composent la géométrie de l'objet, au lieu de convertir la géométrie entière en maillage.

Les modifications sont apportées au fichier **client/src/Measur3DComponent/functions.js**. Deux fonctions sont modifiées : *loadCityObjects* et *parseObject*.

La fonction *parseObject* est remplacée par une autre fonction *parseSurface* qui permet de parcourir la géométrie de toute surface sémantique (Annexe 3)).

Dans *loadCityObjects*, chaque objet de ville est traité individuellement. Les primitives géométriques sont extraites à partir de la propriété *boundaries* (Annexe 4). Différentes couleurs sont attribuées aux surfaces sémantiques de l'espace routier :

```
var ROADCOLOURS = {
  'sidewalk': 0xC9CBC9,
  'road': 0x848784,
  'green areas': 0x32AC2E
}
```

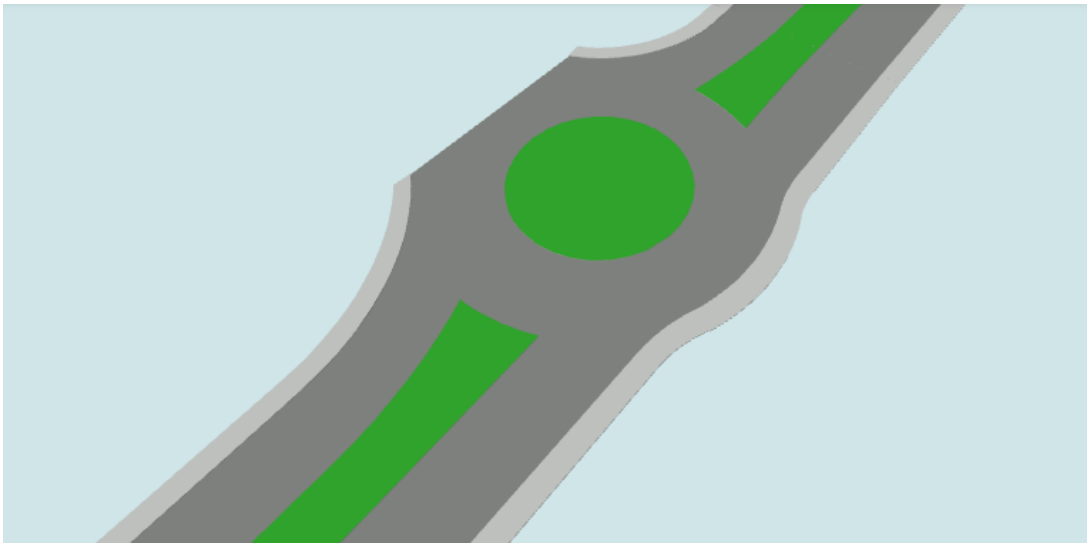


Figure 4.29 – Couleurs des surfaces sémantiques

Contrairement à Ninja Viewer, les surfaces sémantiques sont colorées sur la base de la propriété *fonction* (Figure 4.29). Cela permet de différencier entre les *TrafficArea* qui peuvent être des trottoirs ou des chaussées, et les *AuxiliaryTrafficArea*.

4.6.2.3 Gestion des attributs et des informations sémantiques

Les attributs de tout objet de la ville sont récupérés de la base de données avec la méthode *getObjectAttributes*. Elle prend comme paramètre l'identifiant unique de l'objet, auquel peut s'ajouter le type de l'objet.

Comme pour Ninja Viewer, les modifications apportées servent à ajouter les informations sémantiques à la table des attributs :

```
var semantics = {
    "type": intersects[0].object.type,
```

```
    "surfaceMaterial": intersects[0].object.surfaceMaterial,
    "function": intersects[0].object.function
  };

  axios
    .get("http://localhost:3001/measur3d/getObjectAttributes", {
      params: {
        name: intersects[0].object.name,
        CityObjectType: cityObjectType,
      },
    })
    .then((response) => {
      if (semantics.type !== "Mesh") {
        var data = extend(response.data.attributes, semantics);
      } else {
        var data = response.data.attributes;
      }
      EventEmitter.dispatch("attObject", data);
    });
```

Le code complet peut être examiné dans : `https://github.com/GANys/Measur3D/tree/dev-anass`. La Figure 4.30 illustre toutes les composantes de l'interface utilisateur après modifications :

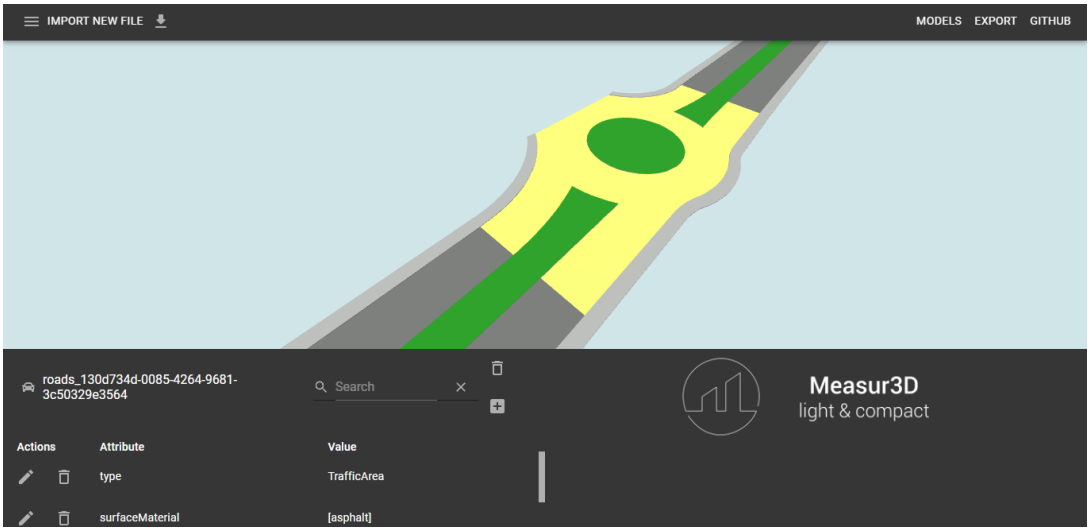


Figure 4.30 – Modèle 3D de ville visualisé dans Measur3D

Conclusion

L'utilisation d'une solution ETL spatial permet l'entière maîtrise du processus de modélisation tant géométrique que sémantique de l'infrastructure routière, et par conséquent la résolution rapide et efficace de toute erreur ou anomalie détectée dans le modèle généré.

La visualisation des modèles 3D dans Measur3D a été amélioré pour supporter la sémantique. Le traitement des primitives géométriques permet l'accès aux surfaces sémantiques. Les détails visuels des modèles 3D sont ainsi récupérés, ce qui permet une inspection plus compacte des fichiers CityJSON.

Discussion et perspectives

L'extraction des caractéristiques linéaires de la route à partir du nuage de points permet non seulement de modéliser l'espace routier en 3D mais aussi le traitement de la sémantique. Sur la base du système de codification établi, chaque surface sémantique est reconstruite et traitée individuellement pour pouvoir générer un modèle 3D de route à un niveau de détails élevé (LoD2).

La modélisation 3D via un processus ETL spatial dans FME Workbench permet de reconstruire l'espace routier en conformité avec le module Transport de CityGML 3.0, bien avant la traduction du modèle 3D en jeu de données CityJSON. La chaîne de traitement, qui est une succession de transformateurs et de scripts, permet l'entière maîtrise du processus de modélisation, et par conséquent la résolution rapide et efficace de toute erreur ou anomalie détectée dans le modèle généré.

CityJSON est une norme communautaire qui propose toujours de nouvelles versions, et donc des modifications dans le schéma de données. Bien que FME simplifie et rend plus efficace la conversion de données entre divers formats, un fichier CityJSON en sortie doit être corrigé pour se conformer aux spécifications de la dernière version publiée.

Un fichier CityJSON n'est entièrement valide que si les géométries des objets de la ville sont toutes valides. Certaines erreurs, comme les auto-intersections, peuvent être rapidement corrigées. D'autres erreurs sont plus difficiles à traiter. La non-planarité des polygones, à titre d'exemple, est corrigée soit par triangulation, ou par superposition des polygones sur une surface plane. Dans un terrain plat, avec une variation modérée d'altitude, le plan moyen ajusté par la méthode des moindres carrés à la surface terrain peut être utilisé. Dans le cas d'un terrain à grande variation d'altitude, les sections peuvent être subdivisées en sous-sections dont chacune est corrigée individuellement.

La gestion des jeux de données CityJSON avec un système de fichiers présente des inconvénients en termes de performances. En revanche, le chargement dans une base de données permet le stockage et la récupération rapides et efficaces des modèles 3D de route.

Attendu que les données CityJSON sont semi-structurées, l'utilisation d'une base de données NoSQL orientée documents présente la meilleure solution de stockage. Cependant, il est nécessaire que les documents à stocker respectent des schémas prédéfinis.

La dernière tâche consiste à mettre à disposition les modèles stockés via une application web à trois couches. Les fonctionnalités de Measur3D peuvent être utilisées. Il propose un API qui permet à la fois d'insérer des données CityJSON dans une base de données MongoDB et de récupérer les modèles stockés.

Une contribution non négligeable de ce travail consiste à améliorer la couche client de Measur3D pour permettre la visualisation et l'accès aux informations sémantiques des modèles 3D de route.

Les perspectives et les futurs travaux peuvent porter sur un benchmarking des algorithmes de détection automatique des caractéristiques linéaires à partir des nuages de points pour implémentation d'une méthode d'extraction et de codification automatiques des objets linéaires de la route. D'autres travaux peuvent s'intéresser à l'exploitation des jeux de données CityJSON dans divers domaines d'application, entre autres la conduite autonome et la planification urbaine. Des tests de simulation permettront de comprendre le fonctionnement des réseaux routiers dans le tissu urbain.

Conclusion générale

Les modèles 3D constituent la base pour le développement des jumeaux numériques de villes, représentant avec une grande précision les paysages et les zones urbaines ainsi que la dynamique de la ville en termes de processus et d'événements.

La norme CityGML de l'OGC sert à la représentation et l'échange des propriétés tant spatiales que sémantiques de ces modèles. Son encodage CityJSON offre plus de légèreté et de flexibilité quant à l'exploitation des modèles 3D dans un environnement web.

Cette étude s'inscrit dans un cadre de recherche sur la modélisation 3D, la structuration et l'optimisation des données spatiales 3D. Son objectif principal est le développement d'une approche pour la production des plateformes routières 3D cohérentes géométriquement en CityJSON.

La modélisation géométrique puis sémantique des infrastructures routières est effectuée sur la base des caractéristiques linéaires de l'espace routier via un processus ETL spatial qui permet la transformation de jeux de données spatiales et leur conversion en différents formats, entre autres CityGML et son encodage CityJSON.

Ces éléments linéaires peuvent être extraits à partir de données recueillies par télédétection. La technologie LiDAR fait partie des systèmes de télédétection utilisés pour la collecte des données routières.

Parmi les différents types de systèmes LiDAR, la cartographie des routes utilisent souvent un système monté sur un véhicule en mouvement. Il s'agit d'un système de cartographie mobile permettant de mesurer la géométrie de la route, y compris la chaussée et les bordures de trottoirs. Ainsi, les objets situés le long de la route peuvent être levés simultanément. La précision et la densité des nuages de points est fonction des performances du scanner laser utilisé et des conditions d'acquisition. Toutefois, les données LiDAR peuvent être améliorées avec des données acquises par des caméras de prise de vue, à partir des plateformes mobiles, statiques ou aéroportées.

Dans cette étude, les nuages de points fournis par cartographie mobile sont exploités pour l'extraction de la surface terrain, à partir de laquelle les objets linéaires de la route sont extraits. L'utilisation de InfraWorks permet une extraction semi-automatique de ces éléments. Cependant, dans un contexte urbain, les réseaux routiers s'étendent sur plusieurs kilomètres, et donc une approche supervisée peut être inefficace et épuisante en termes de temps et de performances. Ainsi, il est recommandé de développer une méthode d'extraction et de codification automatique des caractéristiques linéaires des routes dans le but d'optimiser la chaîne de traitement.

Certes, les fichiers CityJSON générés peuvent présenter des erreurs qui doivent être corrigées pour se conformer aux spécifications de la dernière version publiée. Par conséquent, tout jeu de données doit être validé en deux étapes : premièrement, une validation de la syntaxe du fichier, puis la validation des primitives géométriques pour détecter les erreurs géométriques et topologiques.

Ces erreurs de géométrie peuvent entraîner des résultats faux lors de l'exécution des requêtes spatiales et des algorithmes sur le modèle 3D. Ainsi, toute erreur identifiée est ensuite traitée et corrigée pour obtenir des modèles 3D de route cohérents géométriquement.

Bibliographie

Ali, W., Shafique, M. U., Majeed, M. A., & Raza, A. (2019, 10). Comparison between sql and nosql databases and their relationship with big data analytics. *Asian Journal of Research in Computer Science*, 1-10. doi: 10.9734/ajrcos/2019/v4i230108

Beil, C., & Kolbe, T. H. (2017). Citygml and the streets of new york - a proposal for detailed street space modelling. Consulté sur `https://doi.org/10.5194/isprs-annals-IV-4-W5-9-2017-corrigendum` doi: 10.5194/isprs-annals-IV-4-W5-9-2017-corrigendum

Beil, C., Ruhdorfer, R., Coduro, T., & Kolbe, T. H. (2020, 10). Detailed streetspace modelling for multiple applications : Discussions on the proposed citygml 3.0 transportation model. *ISPRS International Journal of Geo-Information*, *9*. doi: 10.3390/ijgi9100603

Bendiksen, T. A. (2021, 11). *Creating a workflow of 3d building data in a municipality context.* Consulté sur `https://lup.lub.lu.se/student-papers/search/publication/9058579`

Boersma, F. (2019, 07). Modelling different levels of detail of roads and intersections in 3d city models. Consulté sur `http://resolver.tudelft.nl/uuid:ebfc48f8-4704-47d3-9654-cd00c765e0af`

Celesti, A., Fazio, M., & Villari, M. (2019, 04). A study on join operations in mongodb preserving collections data models for future internet applications. *Future Internet*, *11*. Consulté sur `https://www.mdpi.com/1999-5903/11/4/83` doi: 10.3390/fi11040083

Chaplier, J., That, T. N., Hewatt, M., Gallée, G., & Sa, O. (2010). *Toward a standard : Roadxml, the road network database format.*

Cheng, L., Chen, S., Liu, X., Xu, H., Wu, Y., Li, M., & Chen, Y. (2018, 5). *Registration of laser scanning point clouds : A review* (Vol. 18). MDPI AG. doi: 10.3390/s18051641

Cho, W., Shin, H., Kim, S., Kim, K., & Choi, Y. (2017). *Applicability analysis of mms based road spatial information* (Vol. 21). Elsevier B.V. doi: 10.1016/j.trpro.2017.03.088

Clode, S., Rottensteiner, F., Kootsookos, P., & Zelniker, E. (2007, 5). Detection and vectorisation of roads from lidar data. Consulté sur `https://www.researchgate.net/publication/273129527` doi: 10.14358/PERS.73.5.517

Demirel, H. (2002, 2). *An integrated approach to the conceptual data modeling of an entire highway agency geographic information system (gis).*

Ghotiya, S., Mandal, J., & Kandasamy, S. (2017, 12). Migration from relational to nosql database. In (Vol. 263). Institute of Physics Publishing. doi: 10.1088/1757-899X/263/4/042055

Gneeniss, A. S. (2013, 11). *Integration of lidar and photogrammetric data for enhanced aerial triangulation and camera calibration.* Consulté sur `https://theses.ncl.ac.uk/jspui/handle/10443/2518`

Gruler, H.-C., Stubkjær, E., Axelsson, P., & Wikstrom, L. (2016, 12). *Ogc land and infrastructure conceptual model standard (landinfra).* Consulté sur `https://docs.ogc.org/is/15-111r1/15-111r1.html`

Gröger, G., Kolbe, T. H., Nagel, C., & Häfele, K.-H. (2012, 04). *Ogc city geography markup language (citygml) encoding standard.* Consulté sur `https://www.ogc.org/standards/citygml`

Henricsson, R., & Gustafsson, G. (2011). *Document oriented nosql databases - a comparision of performance in mongodb and couchdb using a python interface.*

Hussain, A., & Sharma, P. K. (2019, 12). Deployment of web application in lan based 3 tier architecture. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, 341-345. doi: 10.32628/cseit195661

Jaakkola, A. (2015). *Low-cost mobile laser scanning and its feasibility for environmental mapping* . Consulté sur `http://urn.fi/URN:ISBN:978-952-60-6198-6`

Klapper, T. (2020). *Comparison of uav-based lidar and photogrammetry case study : Lichfield supersite, australia.* Consulté sur `https://studenttheses.uu.nl/handle/20.500.12932/35883`

Kutzner, T., Chaturvedi, K., & Kolbe, T. H. (2020, 02). Citygml 3.0 : New functions open up new applications. *PFG - Journal of Photogrammetry, Remote Sensing and Geoinformation Science*, *88*, 43-61. doi: 10.1007/s41064-020-00095-z

Ledoux, H. (2018, 12). val3dity : validation of 3d gis primitives according to the international standards. *Open Geospatial Data, Software and Standards*, *3*. doi: 10.1186/s40965-018-0043-x

Ledoux, H., & Dukai, B. (2022, 02). *CityJSON Specifications 1.1.1.* Consulté sur `https://www.cityjson.org/specs/1.1.1/`

Ledoux, H., Ohori, K. A., Kumar, K., Dukai, B., Labetski, A., & Vitalis, S. (2019, 02). Cityjson : a compact and easy-to-use encoding of the citygml data model. Consulté sur `http://arxiv.org/abs/1902.09155http://dx.doi.org/10.1186/s40965-019-0064-0` doi: 10.1186/s40965-019-0064-0

Li, X. (2021, 07). *Cityrest : Cityjson in a database + restful access.* Consulté sur `http://resolver.tudelft.nl/uuid:e6bb2142-c113-4add-a466-41bf0fea3b11`

Mao, B., Harrie, L., Cao, J., Wu, Z., & Shen, J. (2014). Nosql based 3d city model management system. In (Vol. 40, p. 169-173). International Society for Photogrammetry and Remote Sensing. doi: 10.5194/isprsarchives-XL-4-169-2014

Nys, G.-A., & Billen, R. (2021, 07). From consistency to flexibility : A simplified database schema for the management of CityJSON 3D city models. *Transactions in GIS*. doi: 10.1111/tgis.12807

Ohori, K., Ledoux, H., & Meijers, M. (2012, 10). Validation and automatic repair of planar partitions using a constrained triangulation. *Photogrammetrie - Fernerkundung - Geoinformation*, *2012*, 613-630. doi: 10.1127/1432-8364/2012/0143

Prandi, F., Devigili, F., Soave, M., Staso, U. D., & Amicis, R. D. (2015). 3d web visualization of huge citygml models. In (Vol. 40, p. 601-605). International Society for Photogrammetry and Remote Sensing. doi: 10.5194/isprsarchives-XL-3-W3-601-2015

Rahman, A. (2021, 5). *Uses and challenges of collecting lidar data from a growing autonomous vehicle fleet : Implications for infrastructure planning and inspection practices.* Consulté sur `https://digitalcommons.usu.edu/etd/8048`

Rai, P. K., & Singh, P. (2015). *Studies and analysis of popular database models* (Vol. 4). Consulté sur `https://www.ijcsmc.com/`

Staring, K. (2020). *Combination of cityjson with postgresql, mongodb and graphql.* Consulté sur https://repository.tudelft.nl/islandora/object/uuid:7f9209f7-248f-4c93-9ba3-5866655e6040

Stoter, A. E., Ohori, J. E. A., Dukai, G. A. K., Labetski, B. ., Kavisha, A. ., Vitalis, K. ., & Ledoux, S. . (2020). *State of the art in 3d city modelling six challenges facing 3d data as a platform.* Consulté sur https://www.gim-international.com/content/article/state-of-the-art-in-

Vitalis, S., Labetski, A., Boersma, F., Dahle, F., Li, X., Ohori, K. A., ... Stoter, J. (2020, 9). Cityjson + web combining double low line ninja. In (Vol. 6, p. 167-173). Copernicus GmbH. doi: 10.5194/isprs-annals-VI-4-W1-2020-167-2020

Xu, S., & Aljarallah, M. (2014, 04). *Comparative study of database modeling approaches.*

Yilmaz, C. S., Yilmaz, V., & Gungor, O. (2017). Ground filtering of a uav-based point cloud with the cloth simulation filtering algorithm.. Consulté sur https://www.researchgate.net/publication/316881354

Zhang, W., Qi, J., Wan, P., Wang, H., Xie, D., Wang, X., & Yan, G. (2016). An easy-to-use airborne lidar data filtering method based on cloth simulation. *Remote Sensing*, *8*(6). Consulté sur https://www.mdpi.com/2072-4292/8/6/501 doi: 10.3390/rs8060501

Zhang, X., Zhong, M., Liu, S., Zheng, L., & Chen, Y. (2019, 08). Template-based 3d road modeling for generating large-scale virtual road network environment. *ISPRS International Journal of Geo-Information*, *8*. doi: 10.3390/ijgi8090364

Annexes

Annexe 1 : Correctif des fichiers CityJSON

```
"use strict"

const fs = require("fs")

var file_path = "../roads_1.json"
var model_version = "1.1"
var model_epsg = "https://www.opengis.net/def/crs/EPSG/0/26191"

let rawdata = fs.readFileSync(file_path)
let citymodel = JSON.parse(rawdata)

citymodel.metadata.referenceSystem = model_epsg

citymodel.version = model_version

citymodel.transform = {
  scale: [1.0, 1.0, 1.0],
  translate: [0.0, 0.0, 0.0],
}

for (var key of Object.keys(citymodel.CityObjects)) {

  for (var geom in citymodel.CityObjects[key].geometry) {
    citymodel.CityObjects[key].geometry[geom].lod = String(citymodel.
       CityObjects[key].geometry[geom].lod)

    if (citymodel.CityObjects[key].type == "Road") {

      citymodel.CityObjects[key].geometry[geom].type = "MultiSurface"
      citymodel.CityObjects[key].geometry[geom].semantics = JSON.parse(
         citymodel.CityObjects[key].attributes.semantics)

      var values = citymodel.CityObjects[key].geometry[geom].semantics.
         values
      var newValues = []

        values.forEach(el => {
          let string = el.toString()
          let b = []

          string = JSON.parse(string)

          string.forEach((a, i)=> {
            if (a == 'null') {
              b.push(null)
            } else {
              b.push(a)
            }
          })

          b.forEach(element => {
```

```
            newValues.push(element)
          })
        })


      var values = citymodel.CityObjects[key].geometry[geom].semantics.
         values = newValues
      delete citymodel.CityObjects[key].attributes.semantics

    }
  }

  if (Object.keys(citymodel.CityObjects[key].attributes).length == 0) {
      delete citymodel.CityObjects[key].attributes
    }
}

let file_citymodel = JSON.stringify(citymodel, null, 2);
fs.writeFileSync("roads.city.json", file_citymodel)

console.info("CityJSON file created successfuly")
```

Annexe 2 : Script de traitement des surfaces sémantiques dans FME

```
import fme
import fmeobjects
import numpy as np

class FeatureProcessor(object):

    def __init__(self):
        self.features = []
        self.semantics = []
        self.values = []

    def input(self, feature):
        self.features.append(feature)

    def close(self):
        for feature in self.features:
            for i in range(feature.getGeometry().numParts()):
                self.semantics.append(feature.getGeometry().getPartAt(i
                    ))

            for i in range(len(self.semantics)-1):
                self.values.append('null')

            self.values.append(feature.getAttribute('value'))

            feature.setAttribute("valeurs", str(self.values))
            self.pyoutput(feature)

            self.semantics.clear()
            self.values.clear()
```

Annexe 3 : Fonction *parseSurface*

```
1  //convert json file to viewer-object
2  async function parseSurface(object, surface, transform, cityObj, geoms)
       {
3    // CityObject JSON, transform, CityObject name, threeScene.Geoms
4
5    //create geometry and empty list for the vertices
6    var geom = new THREE.Geometry();
7
8    if (object.geometry[0] == null) return; // If no geometry (eg:
       CityObjectGroup (not always true))
9
10   //each geometrytype must be handled different
11   var geomType = object.geometry[0].type;
12
13   var object_vertices = object.vertices;
14   var face_vertices = [];
15
16   var boundaries = object.geometry[0].boundaries;
17
18     var boundary = [],
19         holes = []
20         if (boundary.length > 0) {
21           holes.push(boundary.length);
22         }
23
24           var new_boundary = decomposeFaces(
25             geom,
26             surface[0],
27             face_vertices,
28             object_vertices,
29             transform
30           );
31           boundary.push(...new_boundary);
32
33
34       if (boundary.length === 3) {
35         geom.faces.push(new THREE.Face3(boundary[0], boundary[1],
             boundary[2]));
36       } else if (boundary.length > 3) {
37         //create list of points
38         var pList = [],
39           k;
40
41         for (k = 0; k < boundary.length; k++) {
42           pList.push({
43             x: object_vertices[face_vertices[boundary[k]]][0],
44             y: object_vertices[face_vertices[boundary[k]]][1],
45             z: object_vertices[face_vertices[boundary[k]]][2],
46           });
47         }
48         //get normal of these points
49         var normal = get_normal_newell(pList);
50
51         //convert to 2d (for triangulation)
```

```
        var pv = [];
        for (k = 0; k < pList.length; k++) {
          var re = to_2d(pList[k], normal);
          pv.push(re.x);
          pv.push(re.y);
        }

        //triangulate
        var tr = earcut(pv, holes, 2);

        //create faces based on triangulation
        for (k = 0; k < tr.length; k += 3) {
          geom.faces.push(
            new THREE.Face3(
              boundary[tr[k]],
              boundary[tr[k + 1]],
              boundary[tr[k + 2]]
            )
          );
        }
      }


    //needed for shadow
    geom.computeFaceNormals();
    //geom.computeVertexNormals();

    geoms[surface] = geom;


  return object.children;
}
```

Annexe 4 : Fonction *loadCityObjects*

```
1  //convert CityObjects to mesh and add them to the viewer
2  export async function loadCityObjects(threescene, cm_name) {
3    await axios
4      .get("http://localhost:3001/measur3d/getNamedCityModel", {
5        params: {
6          name: cm_name,
7        },
8      })
9      .then(async (responseCity) => {
10       var json = responseCity.data;
11
12       var ext = json.metadata.geographicalExtent;
13       var avgX = (ext[0] + ext[3]) / 2;
14       var avgY = (ext[1] + ext[4]) / 2;
15       var avgZ = (ext[2] + ext[5]) / 2;
16
17       var z_dist = (ext[3] - ext[0]) / (2 * Math.tan((30 * Math.PI) /
             180));
18       var y_dist = (ext[3] - ext[0]) / (2 * Math.tan((30 * Math.PI) /
             180));
19
20       threescene.camera.position.set(avgX, avgY - y_dist, avgZ + z_dist
             ); // Can be improved
21       threescene.camera.lookAt(avgX, avgY, avgZ);
22
23       threescene.camera.updateProjectionMatrix();
24
25       threescene.controls.target.set(avgX, avgY, avgZ);
26
27       //enable movement parallel to ground
28       threescene.controls.screenSpacePanning = true;
29
30       //iterate through all cityObjects
31       for (var cityObj in json.CityObjects) {
32         var boundaries = json.CityObjects[cityObj].geometry[0].
               boundaries;
33         var cityObjectType = json.CityObjects[cityObj].type;
34
35         switch (cityObjectType) {
36           case "BuildingPart":
37             cityObjectType = "Building";
38             break;
39           case "Road":
40           case "Railway":
41           case "TransportSquare":
42             cityObjectType = "Transportation";
43             break;
44           case "TunnelPart":
45             cityObjectType = "Tunnel";
46             break;
47           case "BridgePart":
48             cityObjectType = "Bridge";
49             break;
50           case "BridgeConstructionElement":
```

```
51              cityObjectType = "BridgeInstallation";
52              break;
53            default:
54          }
55
56          var childrenMeshes = [];
57
58          var surfaces = json.CityObjects[cityObj].geometry[0].semantics.
                surfaces;
59          var values = json.CityObjects[cityObj].geometry[0].semantics.
                values;
60
61          var idx = 0;
62          //parse cityObj that it can be displayed in three js
63          boundaries.forEach(surface => {
64            var returnChildren = parseSurface(
65              json.CityObjects[cityObj],
66              surface,
67              json.transform,
68              cityObj,
69              threescene.geoms
70            );
71
72            var color;
73            var type;
74
75            //if object has children add them to the childrendict
76             for (var i in returnChildren) {
77              childrenMeshes.push(returnChildren[i]);
78            }
79
80            var value = values[idx];
81            if (value !== null) {
82              type = surfaces[value].function;
83              color = ROADCOLOURS[type];
84            } else {
85              type = json.CityObjects[cityObj].type;
86              color = ALLCOLOURS[type];
87            }
88
89
90            //set color of object
91            var material = new THREE.MeshStandardMaterial();
92            material.color.setHex(color);
93
94            //create mesh
95            var coMesh = new THREE.Mesh(threescene.geoms[surface],
                  material);
96
97            // Added by Measur3D
98            coMesh.name = cityObj;
99            coMesh.CityObjectClass = json.CityObjects[cityObj].type;
100           coMesh.jsonName = json.name;
101
102           if (value !== null) {
103             coMesh.type = surfaces[value].type;
```

```
104             coMesh.function = surfaces[value].function;
105             coMesh.surfaceMaterial = surfaces[value].surfaceMaterial;
106           }
107
108           coMesh.childrenMeshes = childrenMeshes;
109
110           coMesh.castShadow = true;
111           coMesh.receiveShadow = true;
112
113           threescene.scene.add(coMesh);
114           threescene.meshes.push(coMesh);
115
116           idx++;
117         })
118       }
119     })
120     .then(() => {
121       threescene.setState({
122         boolJSONload: false, //enable clicking functions
123         cityModel: true,
124       });
125
126       threescene.renderer.render(threescene.scene, threescene.camera);
127     });
128 }
```

ملخص

تُستخدم النماذج ثلاثية الأبعاد للمدن كأساس لتطوير التوائم الرقمية للمدن. ركزت معظم الأعمال حتى الآن على نماذج البنايات بشكل أساسي، إلا أن هذا الوضع قد تغير في السنوات الأخيرة بالنسبة لشبكات النقل، نظرًا لتوافر بيانات أكثر عنها. رغم ذلك، لا يزال هذا التطور حديثا باعتبار أن معظم المعايير الحالية ترتكز أساسا على تمثيل خطي للشبكات الطرقية.

يهدف هذا المشروع إلى تطوير نهج لإنتاج نماذج ثلاثية الأبعاد للطرق متماسكة هندسيًا، وفقًا لمعيار CityGML 3.0 وتشفيره CityJSON.

النهج المتبع يهدف أولاً إلى نمذجة الطرق بشكل ثلاثي الأبعاد، بمستوى عالٍ من التفاصيل، بناءً على الخصائص الخطية للطرق المستخرجة من سحب النقط ثلاثية الأبعاد. يتم بعد ذلك ترجمة النموذج ثلاثي الأبعاد الذي تم إنشاؤه إلى ملف CityJSON وفقا لمعايير تمثيل شبكات النقل في CityGML 3.0.

ملفات CityJSON المُنشأة يتم تخزينها في قاعدة بيانات موجهة للمستندات، هي مونغو دي بي، التي تقوم بتنظيم البيانات كمستندات بمخططات ديناميكية بدلاً من تخزينها في جداول كما هو معروف في قواعد البيانات المترابطة، والذي يجعل دمج البيانات في أنواع محددة من التطبيقات أسهل وأسرع.

أخيرًا، يتم استرداد البيانات من قاعدة البيانات لإتاحتها عبر تطبيق ويب. للقيام بذلك يمكن استعمال وظائف Measur3D وإدخال تعديلات على واجهة المستخدم للسماح بفحص الأسطح الدلالية للنماذج ثلاثية الأبعاد.

الكلمات المفتاح: النماذج ثلاثية الأبعاد للمدن، البنيات التحتية للطرق، CityGML 3.0، CityJSON، قاعدة البيانات الموجهة للمستندات، تطبيقات الويب، الأسطح الدلالية

تأطير

الأستاذة رفيقة حجي

السيد جيل أنوان نيس

المترشح

يروض أنس

المملكة المغربية
ROYAUME DU MAROC

INSTITUT AGRONOMIQUE
ET VETERINAIRE HASSAN II

معهد الحسن الثاني
للزراعة والبيطرة

مشروع نهاية الدراسات لنيل دبلوم مهندس في الطبوغرافية

النمذجة ثلاثية الأبعاد وتطوير نهج لهيكلة وتخزين واستغلال البنى التحتية للطرق وفقا لمعيار CityGML 3.0 وتشفيره CityJSON

قدم للعموم ونوقش من طرف:

يروض أنس

أمام اللجنة المكونة من:

الأستاذ موحا العياشي	رئيس	معهد الحسن الثاني للزراعة والبيطرة
الأستاذة رفيقة حجي	مقررة	معهد الحسن الثاني للزراعة والبيطرة
السيد جيل أنطوان نيس	مقرر	وحدة الجيوماتيك بجامعة لييج
السيد عبد الرزاق خروبي	ممتحن	وحدة الجيوماتيك بجامعة لييج

شتنبر 2022

العنوان: ص. ب 6202 الرباط المعاهد الرباط – المغرب
الهاتف: 59 / 58 17 77 0537 (00 212)
الفاكس: 45 58 77 0537 (00 212)
موقع الأنترنت:http://www.iav.ac.ma

Adresse : Madinat Al Irfane, B.P. 6202. Rabat – Maroc
Tél : (00 212) 0537 77 17 58/59
Fax : (00 212) 0537 77 58 45
Site web : http://www.iav.ac.ma