Part of the IA.rbre Consortium

Technical Report : Sunlight calculations on 3D Urban data

Author:
Marwan Ait Addi-Russier
LIRIS, Lyon

Supervisors:
Gilles Gesquière
John Samuel

Abstract

Cities are more and more affected by climate change, with frequent episodes of heat waves. Mitigation of heat waves requires a pluridisciplinary effort of analysis, prevention and implementation of heat mitigating measures. To assist the decision makers, we developped tools to leverage 3D urban data and compute exposure to sunlight of terrains, roads, buildings and other city objects. With the objective to help identify the zones where heat mitigation is the most needed. These tools can be reused for other use cases, such as, solar panel installation studies, comparison between potential heat mitigation scenarios or helping choose tree types for revegetalisation projects. We will present different propositions, aiming to update, upgrade and repair the current pySunlight framework. First by reworking key features such as the method to export results, to increase stability and reusability of those results. Second by debugging and creating much needed tests suites. And finally by showing methods by which we can analyse and visualise the results.

With the objective being to produce a reproductible and maintainable method to calculate sunlight on 3D Urban Data.

Contents

Glossary

- **GIS :** GIS stands for Geographic information system. It consists of software that is capable of storing, managing, analysing, editing and visualising geographic data.
- **CRS :** Coordinate Reference System is a framework, used to measure locations on the surface of the globe. Thousands of standards exists depending on the studied area and desired projection with geodetic, earth-centered and planar variants. In short, it allows for precisely positioning data on the globe, and translating between different reference points.
- **IGN :** The National Institute of Geographic and Forest information is a French public establishment that produces and maintain geographical information for France. They provide a wide variety of products from roadmaps to Li-DAR data.
- **Workflow :** A workflow is a set of computational steps, that can be parallelized or not, that takes care of the whole process of fetching data, sanitizing it, doing the desired computation and outputting a formatted result.

1 Introduction

All the code associated with this report can be found on GitHub `https://github.com/VCityTeam/UD-IArbre-Research/tree/master/sunlight-shadow`

1.1 Context

For decades, we have known of the impact human activity has on the environments[1] with an increasing emergency to act on it[2]. This leads to an increasing push towards reducing our CO2 emissions. However, damage has already been done, and temperatures are going to increase drastically in only a few decades[3]. City planners and policymakers need to find solutions, that can be put in place relatively fast, and have a long-lasting effect on citizens' well-being. Streets are overly artificialised, causing problems with water circulation, but also with thermal insulation. It has been determined that Physiological Equivalent Temperature (PET)[Hoppe, 1999] is highly correlated with solar energy [Fahy et al., 2025]. Takebayashi et al.[Takebayashi and Moriyama, 2012] demonstrated that surface temperature distribution is dominated by direct sunlight, and that roof and road surfaces have a higher priority when considering thermal comfort and heat mitigation. Which is why calculating the amount of sunlight streets receive is interesting. It will help us determine where and when citizens are the most exposed to uncomfortable temperatures, and help decision makers in prioritising various projects, such as re-vegetation or sunshades.

1.1.1 The LIRIS Laboratory

The LIRIS (Laboratoire d'InfoRmatique en Image et Systemes d'information) is a computer science Laboratory based in the Lyon metropolis. The structure is co-owned by INSA Lyon, CNRS, Université Claude Bernard Lyon 1, Université Lumière Lyon 2 and Ecole Centrale de Lyon. It is composed of 6 areas of research :

[1] Overview of the IPCC's first asssesment report in 1990
[2] IPCC Synthesis Report, 2023
[3] State fo the global climate 2024, WMO

- **Security, data, and systems:** For data collection, managment and Cyber-security in complex systems.
- **Computer graphics and geometry:** For 3D objects manipulation, visualisation and creation.
- **Computer vision and learning:** Machine learning for computer vision and image manipulation.
- **Interactions and cognition:** Human computer interaction.
- **Algorithms and combinations:** Analysis and creating of complex algorithms and combinatory systems.
- **Simulation and Biology:** Simulating the living.

The workforce is composed of more than 300 employees, divided into 12 teams, as descibed in figure 1. I myself am part of the Origami team, which is specialised in Computer Graphics, but also has expertise in Computer vision and Algorithms.

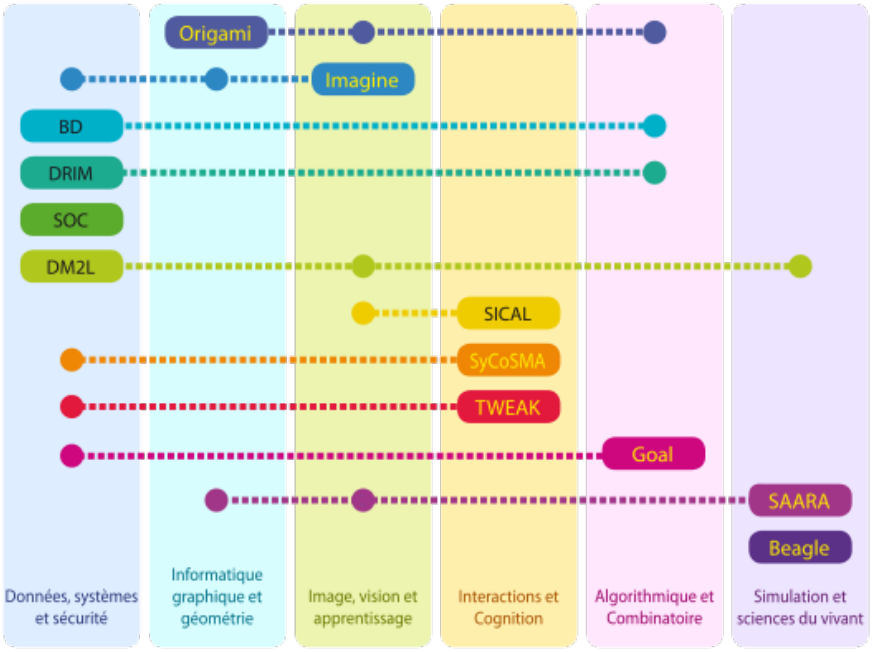

Figure 1: LIRIS' various teams

1.1.2 The VCity project

The VCity project leads a cross-specialty research effort, with researchers from various LIRIS teams, mostly Origami and BD. The research topics are anchored in the fields of geometric modeling and data sciences, applied to modeling analysing, and visualising the city. These efforts are supported by the wide availability of urban data, thanks to efforts by the municipalities in trying to understand and document their territories. The data is geometric and generaly in 3D and has related semantics and topology which allows for in depth analysis. The project is also involved with the Open Geospatial Consortium (OGC) in the creation, definition and maintenance of standards to be used in Geospatial applications. Some of the current main research topics are :

- **Versioning of city models :** The Versioning of city models allows for deep analysis of its evolution. Policies, urban projects and renovations can be contextualised and analysed to give decision makers precious insights on their impact. It is done through CityGML data, knowledge graphs and other information aggregation methods.
- **Viewing of geo-referenced documents :** Linking documents to buildings or places allows for in-context exploration of data. It is an ongoing research topic notably through the AGAPE ANR.
- **Measuring the city :** Through in depth analysis of the impact of urbanization, such as skyline visibility, measuring the permeability of the increasingly artificialised ground, and sunlight and shadow impact.

1.1.3 The IA.rbre project

In a context where government agencies have to face the consequences of climate change and loss of biodiversity, IA.rbre's objective is to develop analysis and visualisation tools based on Artificial Intelligence and Data science, to help government agencies with re-vegetation projects and more generally to rationalise and facilitate our adaptation to climate change. Lyon's metropolis, its innovation lab, its data collection service, and other (re-vegetation service, water management service, ERASME...) researchers and multiple other actors are united around a Cooperative Production Corporation *TelesScoop*[4]. They will create a stack of tools to analyse, visualise and compare existing territorial data. These tools aim to locate the best suited zones to plant vegetation, where vegetation could be densified or zones where urban projects combining planting greenery, de-waterproofing and refreshing the city are possible.

1.2 Problem statement

A solution to calculate sunlight exposition of streets and roofs, that can take into account buildings and vegetation in a 3D environment, can help detect the urban elements most affected by sunlight exposure.

3D Urban data is widely available in the whole of France, with both terrain and buildings. While Li-DAR data can be used, and has already been used, to create catagorised vegetation point cloud. By that it is meant that the point clouds produced by Li-DAR's automaticaly detects if it is scanning vegetation, returning a very accurate point cloud representation of it. The vegetation point cloud can then be triangulated, using for example convex hulls[Aurenhammer, 1991]. 3D Data can help us calculate this shade more accurately, by representing changes in terrain elevation and accurate roof shapes. During this internship we theorised and built a ready to use, scalable and maintainable pipeline. This pipeline will process 3D Urban data and calculate sunlight exposition for various types of city objects.

The tool will be based on the work of Vincent Jaillot and Wesley Petit, who produced respectively a 3D-USE (Which is a deprecated GIS application created by the VCity group) plug-in written in C++ for sunlight calculation, and a port of this plugin as a Library to be used in Python through SWIG[5]. We will build on the python wrapper to propose a stable data pipeline that will process 3DTiles data to produce ready to use Shadow calculations for further analysis.

The objective of this work is to compute sunlight exposure on roads and buildings, eventually taking into account vegetation.

[4] https://www.telescoop.fr/

[5] `https://www.swig.org/`

1.3 Theoretical Method

The paper[Jaillot et al., 2017] demonstrates a method to calculate the impact an object has on its neighbours' exposition to the sun (close and far). The approach is generic in that it works on any type of city object as specified in the CityGML standard. It also works on arbitrarily large city models thanks to its clever loading and unloading of geometry. The purpose of this is to help city planners and citizens of those cities in decision-making, for example in understanding the impact of a large construction and its shadow on its surroundings, the sunlight exposition of a future terrace at a café or *more importantly for the I.Arbre project* the sunlight exposure, and subsequently the shading needs, of an urban area can be calculated and analysed.

To handle an arbitrarily large city model a tiling process was used [Pedrinis and Gesquière, 2017] to enable loading of one tile at time, circumventing memory limitations.

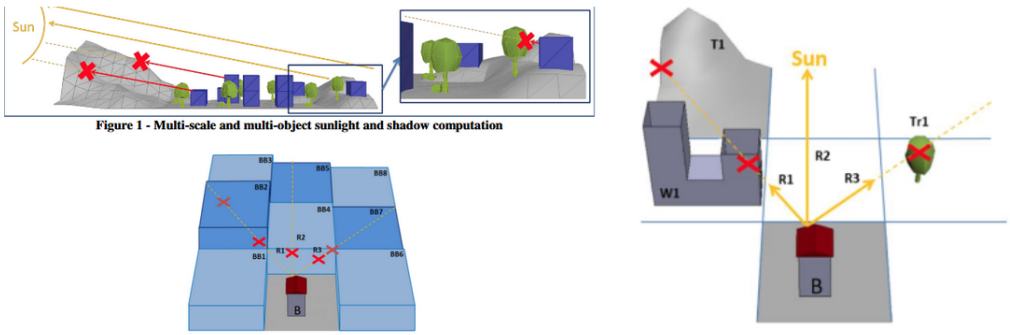

Figure 2: Sunlight calculation (Credit: Jaillot et al. 2017)

To calculate if an object is hit by direct sunlight, the sun's position at every analysed time frame need to be calculated. Michalsky's algorithm is used for this purpose[Michalsky, 1988]. Rays are constructed, with their origin being the centroid of the triangles composing the geometry. Their direction is derived from the sun's position, that was determined previously, and changes for each timestamp with the sun's movement. The rays are then tested against the surrounding geometry for interesection. To limit the number of triangles loaded during the calculation, a special type of Bounding Volume Hierarchy called a semantic BVH is used (see figure 3). Each level is defined by a semantic level of the city, each with its own axis aligned bounding box (AABB), starting with the tile's AABB. It is first used to reduce the number of complex Ray-Triangle intersection calculations but also to detect which tile and which objects to load, allowing the program to ignore part of the geometry and reduce memory usage.

2 Preliminaries

2.1 Formats

2.1.1 Obj

The obj file format is a simple, plain text format for storing 3D Meshes. It supports multiple objects per file, textures and polygons. The format is composed of a list of vertices, followed by two optional lists of texture coordinates and vertex normals. Each polygonal face element is composed of at least 3 vertices, supplemented by texture coordinates and normals if needed. The obj file fomat supports parametric curves in some cases, but they are not necessary for our use case.

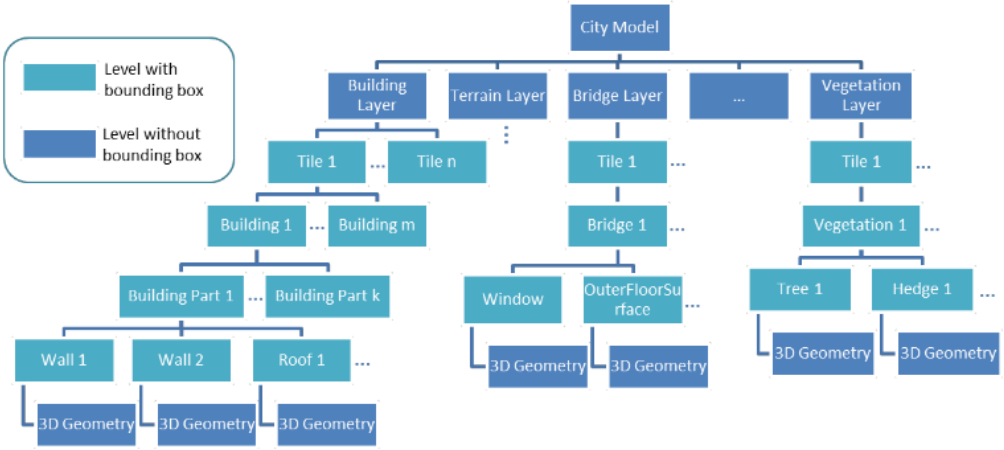

Figure 3: Nd-Tree representing an sBVH

2.1.2 CityGML

CityGML is an Open Geospatial Consortium (OGC)[OGC, 2025b] standard that is used to store semantic and geometric urban data. It is an extension of the Geospatial Markup Language (GML) that focuses on genericity. It contains built-in modules for common city objects (Buildings, roads, bridges, etc...). Each type of objects has different expected attributes (Height, lenght, position, etc...) but the geometry is defined using point clouds, polygons, solids or surfaces. CityGML data is comonly stored in a 3DCityDB[6] Database.

CityGML supports **five**[7] levels of detail (see Figure 5) :

- LOD0 is non-volumetric, horizontal footprint and/or roof surface representation
- LOD1 is a cuboid model of a building (with a horizontal roof)
- LOD2 adds a generalised roof and installations such as balconies
- LOD3 adds, among others, windows, doors, and a full architectural exterior
- LOD4 models the interior of the building, potentially with pieces of furniture

2.1.3 3DTiles

3DTiles is another OGC standard [OGC, 2025a] designed for storing and displaying large Geospatial datasets. It is based on the glTF format [Khronos, 2025], a common standard for 3D asset storage, and extends it with features tailored to urban data, such as semantics (categorizing objects by type) and spatial tiling. These capabilities make 3D Tiles particularly well-suited for applications that require rendering complex city-scale environments. The format

[6]https://github.com/3dcitydb
[7]We will only be concerned with LOD1 and LOD2

```
# List of geometric vertices, with (x, y, z, [w]) coordinates, w is optional
and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, [v, w]) coordinates, these will vary
between 0 and 1.
v, w are optional and default to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit vectors.
vn 0.707 0.000 0.707
vn ...
...
# Polygonal face element
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
```

Figure 4: Example obj file

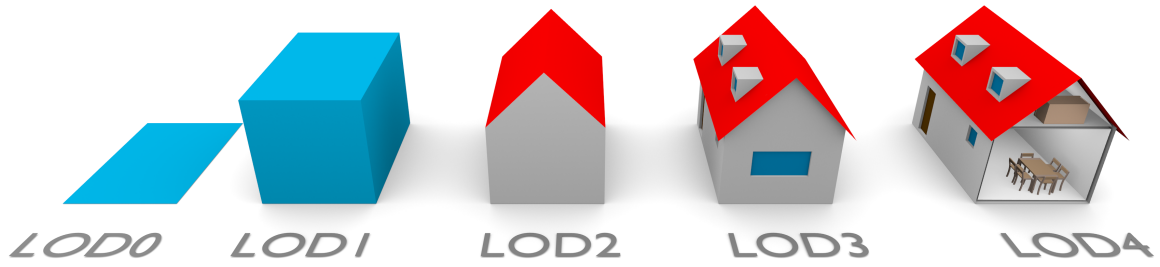

Figure 5: CityGML Various LODs

has been gaining traction in the open-source GIS community, notably with its integration into QGIS.

A key feature of 3D Tiles is its hierarchical tiling system, which enables large datasets to be subdivided into tiles and sub-tiles. This allows data to be selectively loaded based on the user's viewpoint or computational needs, significantly improving performance. This approach aligns with the proposed method[Jaillot et al., 2017] for calculating sunlight and shadow impact, which also relied on a custom tiling mechanism. Tools for converting other formats into 3D Tiles have been developed by the LIRIS laboratory, notably the py3DTilers project[8], which is now maintained by the open-source GIS company Oslandia.

[8]https://github.com/Oslandia/py3dtilers

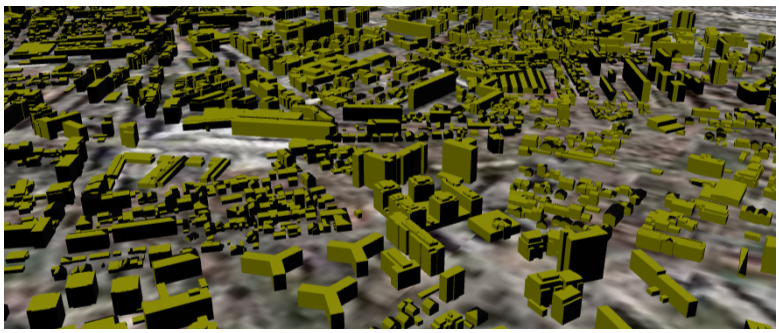

Figure 6: Rendering of a 3DTile tile set, produced by pySunlight, in UD-Viz

Each tile set (see figure 7) is formed of multiple Tiles, their relationships are descirbed in a Tileset.json file the tiles can be aranged in an octree, a grid, a nd-tree or linearly as additions or replacement to/for each other. In our usecase, tiles are on a grid, none of them are child to one another, and they all represent a different space.

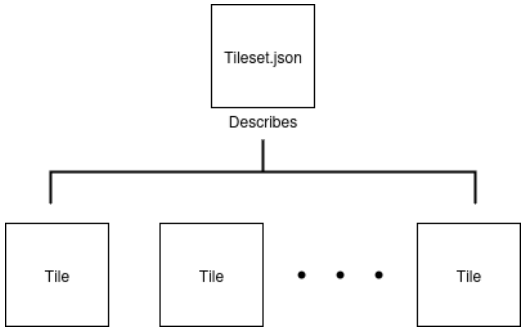

Figure 7: 3D Tiles tileset

The specification provides multiple options for our tiles file format, for example batched 3D Models, *glTF* or even *pnts* for point clouds. The Batched 3D Model format (*B3DM*) was chosen for its ability to add custom metadata to our geometry, wich is very important for us to store shading information. Each tile is composed of a FeatureList, storing individual features (Buildings, roads, etc...), metadata about the tile including its bounding volume (Boxes[9] in our case), transform and name, and additionnaly a BatchTable containing user defined information about individual features in the feature list (see figure 8). BatchTables have a few limitations, first each feature needs to store the same type of data, for example you can not store variable size lists in batch table. You could store a list if its size is eactly the same in all features.

[9]Rectangular cuboids

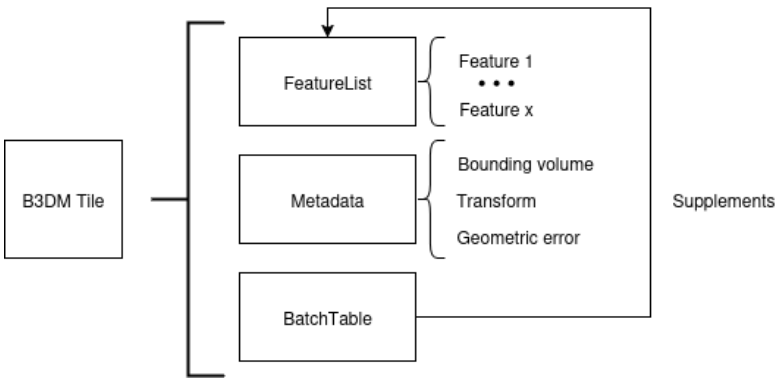

Figure 8: *B3DM* Tile

Finally, each feature stores its own geometry (see figure 9), which is a mesh comprised of triangles, in the glTF format. The features also include metadata about their bounding volume, transform, centroid and others.

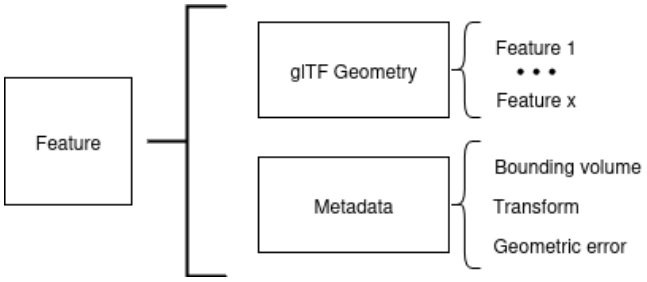

Figure 9: 3DTiles Feature

2.2 Software

2.2.1 3DUSE

3D-USE (Urban Scene Editor) is a software being developed by the VCity project since 2013. 3D-USE allows the user to display and edit various file formats such as CityGML, 3ds, obj or shapefile. The software is a standalone executable that can be run on Windows, Mac and Linux. Its capabilities can be extended by plug-ins, written in C++. For example, as can be seen in figure 10, it is capable of displaying a textured CityGML model of the 1st district of Lyon, the user can select buildings and see the associated metadata.

It is now being replaced by UD-viz wich is web based. It can display very large and varied datasets via their transformation into 3DTiles, which allows streaming of the data and various layers.

2.2.2 THREE.js

THREE.js is a javascript library dedicated to displaying 3D content on a web page. It is not to be confused with webGL, THREE.js uses webGL (although not exclusively), but webGL is a very low-level system that only draws points, lines and triangles and requires a lot of work to create complex scenes. It handles scenes, lights, shadows, materials, textures, 3d transformations, complex primitives, meshes and more, which allows the user to focus on higher-level concerns. Here is an example of the structure of a THREE.js application.

Figure 11 describes the typical structure of a THREE.js application. The renderer is the main overseer, it parses and renders the 3D objects defined in the Scene graph. Most renderers

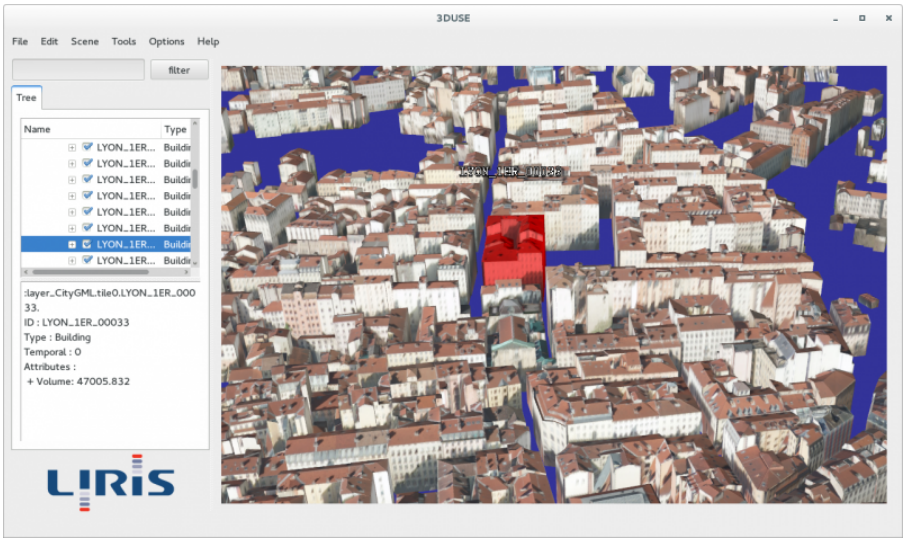

Figure 10: CityGML model displayed in 3D-USE

use openGL, but there are exceptions. A scene graph is built, containing assets, which can be groups, 3D objects, meshes, lights or cameras. It is then passed onto the renderer for displaying. In the schema, the camera is not directly connected to the scene graph, as it is an independent object, however is can be attached to any objects of the scene graph, its position and controls becoming relative to the objects it is attached to. More importantly for the use case described in this report, **materials** can be assigned to meshes, the type of material that will be of interest is the *ShaderMaterial* which allows us a custom GLSL[10] shader to be defined.

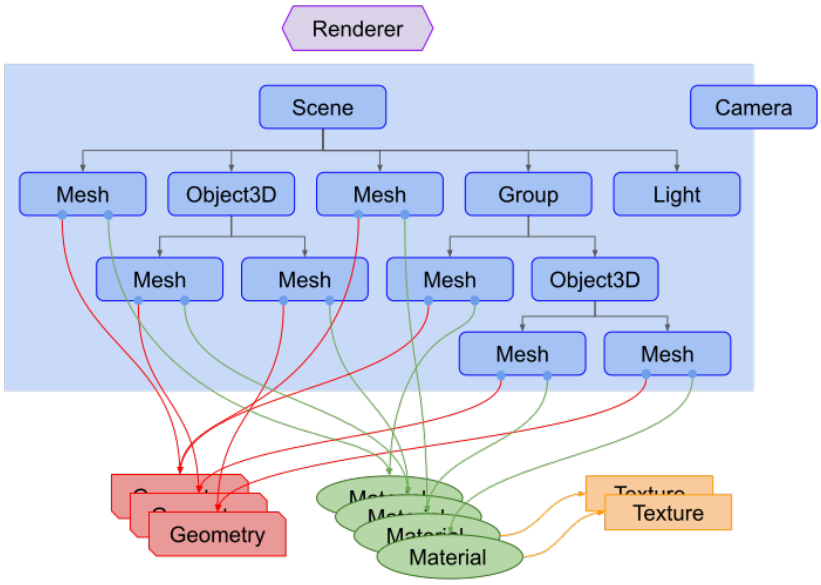

Figure 11: Sctructure of a THREE.js application

[10] https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)

2.2.3 UD-Viz

UD-viz is a Web-based Geospatial visualisation tool. It is a javascript web application, based on ITowns[11], a THREE.js based javascript framework for visualising 3D geopsatial data. It supports a variety of data formats (Including 3DTiles). UD-viz proposes an abstraction layer, permiting users to deploy their visualisations faster, with the caveat of less customisation being available. Layers can be selected, hidden and modified independently. The data is usually processed by UD-Serv[12], LIRIS' in-house backend solution.

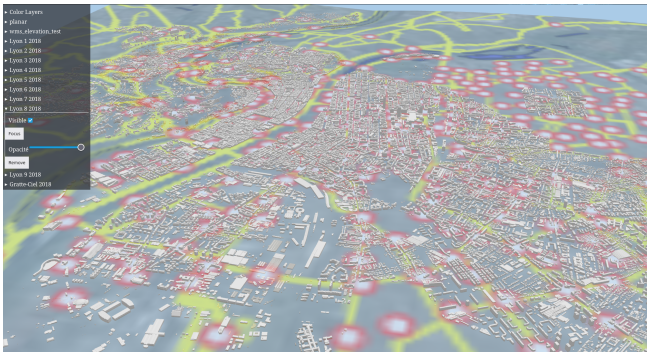

Figure 12: Layered 3DTiles model of the city of Lyon displayed in UD-viz

2.2.4 Argo workflows

Argo workflows is a workflow engine based on Kubernetes, where each computational step of a workflow is defined as a Docker container. With it, you can create a large variety of workflows, from simple linear ones to complex parallelized data treatment. The workload is automaticaly distributed as needed between the different containers thanks to Kubernetes. Workflows can be viewed as graphs, where going from one node to another requires the success of the current docker container. The different workflow types include, but are not limited to, directed acyclic graphs (DAG) and step-based workflows, conditional execution enabling branching into the graph, automatic retries of failed containers, and more. An example of a DAG workflow, visualised in argo's graphical interface environment, can be seen in figure 13.

The LIRIS laboratory hosts its own instance of argo workflow on the Pagoda[13] platform.

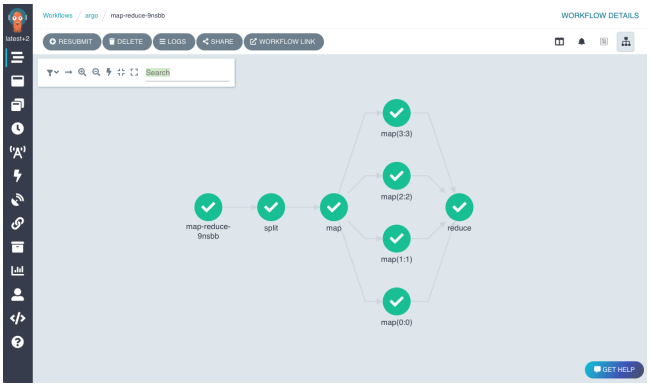

Figure 13: Example of a directed acyclic graph workflow

[11] https://github.com/iTowns/itowns
[12] https://github.com/VCityTeam/UD-Serv
[13] https://projet.liris.cnrs.fr/pagoda/latest/

3 Related works

Shadow computation is a well studied subject in Computer Graphics. Williams proposed an efficient way to calculate shadows by rasterisaton in 1978[Williams, 1978], McGuire et al. presented several methods for computing real time shadow rendering by rasterisation [McGuire et al., 2003]. These methods allow fast shadow computation for real-time graphics, however they focus on visualisation and make sacrifices in accuracy to attain real-time rendering. They also focus on screen-space, or view frustum calculations.

For shadow calculations on urban data, two approaches exist, shadow calculation from 2D+Height data and computation from full 3D data (Usually CityGML). For example, Leduc et al. have been using 2D + Height data with a similar goal of helping mitigate heat[Leduc et al., 2021] they talked about supplementing their tool with a 3D based approach to make it more relevant.

As for 3D Data, Wieland et al. [Manfred et al., 2015] used CityGML data to compute solar radiation on Buildings faces and roofs. Jaillot et al. developed a generic, tile based approach that works with terrain, roads, vegetation and buildings, it uses a custom tiling process and format. More recently Miranda et al. developed a method to interpolate between time stamps to calculate the accumulation of shadow over time [Miranda et al., 2019], however their method only works on flat terrain, which is not applicable to most cities.

As for industrial solutions, *Google solar API*[14] leverages google extensive mapping of the earth to help decision-making in solar panel orientation. *ArcGIS*[15] proposes tools to calculate solar radiation on a smaller scale with a focus on solar pannel efficiency and insulation. We can also mention *jveuxdusoleil*[16] that focuses on streets, but the method is not documented or open source.

3.1 Existing work in VCity

Work has been done on the subject previously by two interns (at the time) Vincent Jaillot and Wesley Petit. Jaillot [Jaillot et al., 2017] worked on a plugin for the 3D-USE[17] platform called Sunlight, which was writen in C++ and is not supported anymore. It pulls from CityGML data, triangulate the polygons and tiles everything to allow calculations on very large datasets. There were some known issues notably with the triangulation step, which created small artifacts, however the solution was deemed reliable and accurate.

Due to the deprecation of 3DUse, this implementation needed to be reworked to fit into the new UD-viz/Itowns[18] framework. Wesley Petit was responsible for this transition. The application called SunLight was entirely reworked as a C++ library, with functions to calculate intersections between ray and triangles, and between rays and a triangle soup. The library is then used inside a python script, to calculate sunlight exposure on 3DTiles[19] urban data. This extended application is called pySunlight.

We will describe in greater detail all the technologies in the following sections

3.1.1 py3DTiles

py3DTiles is a python library that implements the 3DTiles OGC standard [OGC, 2025a] in python. It allows for Reading, writing and modifying 3DTiles 1.0 tile sets. It is co-developed

[14]https://developers.google.com/maps/documentation/solar/overview

[15]https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/how-solar-radiation-is-calculated.htm

[16]https://jveuxdusoleil.fr

[17]https://projet.liris.cnrs.fr/vcity/tools/3d-use/

[18]https://projet.liris.cnrs.fr/vcity/tools/ud-viz/

[19]https://www.ogc.org/standards/3dtiles/

by LIRIS[Gaillard, 2018] and Oslandia and is maintained by Oslandia, an open source software company. The in memory representation of tile sets, tiles and features greatly facilitate tile set manipulation. Multiple features are of great use for the use case presented here :

- Reading mutliple 3Dtiles specifications (.b3dm, .pnts, .gltf)
- Creating 3DTiles tile sets from scratch
- Manipulating batch table data

3.1.2 py3DTilers

py3DTilers is a python library developed inside the VCity project of the LIRIS laboratory. It is being used by numerous organisations such as the National Geographical Institute (IGN), the Natioal Spacial Studies Center (CNES), Berger Levrault, Carl software, CIRIL Group and Oslandia.

It provides multiple converters, called tilers, allowing the user to convert geojson, cityGML, OBJ and IFC files to the 3DTiles format. The conversion process is done via the Command Line Interface and is highly customisable. It also allows for manipulation of existing tile sets, including CRS changes, translation and rotation of the geometry, and combination of multiple tile sets into one.

Maintenance has been suspended and ownership is being transfered to Oslandia as of now. Its features are currently being integrated into py3DTiles, following a collaborative effort between Oslandia and the LIRIS laboratory.

3.1.3 pySunlight

PySunlight is a python wrapper for the sunlight c++ Library that allows it to interface with 3DTiles. It runs on python3.9 and uses **py3DTilers** as well as py3DTiles to read and convert 3DTiles to a format that is usable by the sunlight library. After calculating sunlight values, it can export the results as 3DTiles or a CSV data (figure 14).

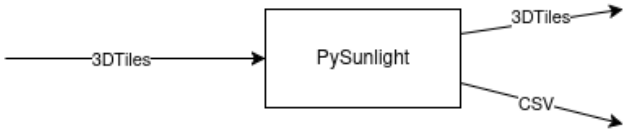

Figure 14: Pysunlight data flow

The sunlight calculation process is done via ray tracing as described in [Jaillot et al., 2017]. A ray is cast from each triangle of the geometry, towards the current position of the sun depending on the current date and time. The sun's position is read from pre-computed results, calculated using Michalsky's algorithm [Michalsky, 1988] for each hour of each day. Each ray will fist intersect with the bounding box of each tile, if a tile is intersected, the ray is tested against the tiles' geometry. If a ray is not intersected by the geometry, the sun directly lights the corresponding triangle, which is recorded by pySunlight.

The fact that pySunlight is a wrapper to a c++ library implies the use of an interface, between the C++ library and python code. The Simplified Wrapper Interface Generator (SWIG) is being used here, it is a generic tool used to connect C/C++ code to various high level languages, including python. It requires minimal setup, and parses the given C++ intefaces, generating the appropriate functions for the target language (e.g. python) to access the C/C++

code. It is very useful in our usecase, allowing pySunlight to run much faster C++ code when doing intensive tasks such as ray tracing.

During conversion between the 3DTiles format, and a triangle soup that can be processed by Sunlight, the geometry is extracted for each tile and each triangle is given an ID corresponding to its original tile, feature and index in that feature. This is later reflected in the CSV export, wich is a list of Tile, feature and triangles ID associated with lighting data. For the 3DTiles export, each triangle becomes its own feature, with information about its lighting, before being inserted in its original Tile. This means the whole tileset is rewriten for each timestamp, we will see later on why that is problematic, for performance and accuracy.

As an example, figure 15 contains a visualisation of a 3DTiles tile set, containing all the roads in the Lyon metropolis. The data comes from the IGN, it was originaly stored as a Graph in the geoJSON format and converted into a 3DTiles tileset using py3DTilers' geoJSON tiler.

Figure 15: Original roads tileset

And in figure 16, the same tileset is displayed after being processed by pySunlight. A shift in the geometry can be seen, as all the roads seem to lay on top of each other. This is probably due to faulty read and write logic, that will be described later on.

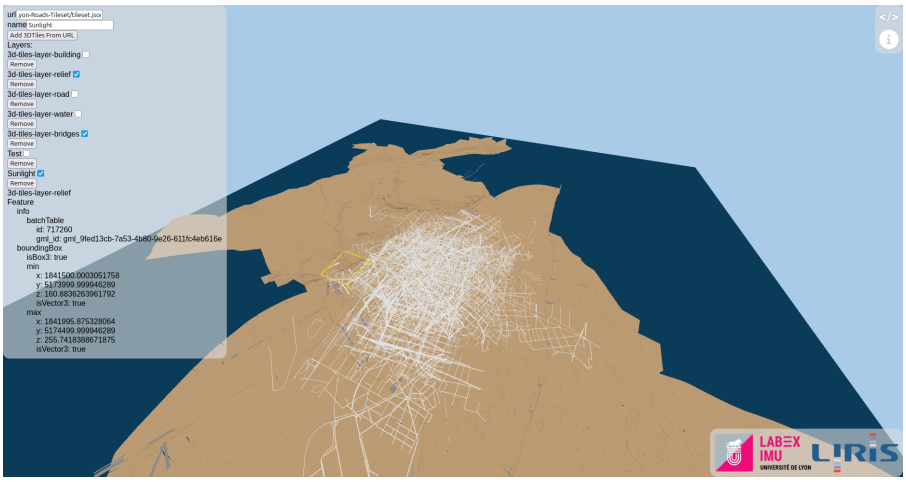

Figure 16: Roads tileset processed by pySunlight

The method of building new tile sets, comprised of single-triangle feature, will be called "triangle level features" tile sets. They make displaying the tileset in UD-viz easier, as there

are built in methods to change the way each feature is displayed. Having multiple triangle per feature however, does not permit using UD-Viz' methods, implying the manipulation of the geometry and its color using THREE.js[20].

3.2 Sourcing data

Obtaining data from trusted and reliable sources is often a critical challenge in the field of urban modeling. Fortunately, in France, the "Institut Géographique National" (IGN) and the Lyon Metropolis (Grand Lyon) offer access to high-quality, 3D urban data, which forms the foundation of our methodology.

The IGN provides a valuable resource through its BDTopo package, which offers highly accurate, geo-referenced datasets in geoJSON format. These datasets contain detailed information about roads, terrain, and various other geographic features, which are essential for creating precise urban models. With BDTopo, we can rely on comprehensive data for most of the French territory, ensuring that our model can represent a large portion of the country, although there may be slight resolution differences for smaller towns or more remote areas.

Additionally, the Lyon Metropolis offers its CityGML dataset, which represents a digital twin of the urban environment in Lyon. This dataset is based on the CityGML standard and provides a 3D representation of the city's buildings, infrastructure, and topography, offering valuable data for urban planning, simulations, and visualisation. The CityGML format, while highly suitable for urban modeling, does present a challenge due to the fact that the most recent dataset available is from 2018. This limits our ability to model changes that have occurred in the city since then, such as new construction, infrastructure updates, or changes to urban zoning.

To address the issue of working with different data formats, we leverage the Py3DTilers tool, which allows us to convert both the geoJSON data from the BDTopo package and the CityGML data from Lyon Metropolis into the more widely used 3DTiles format. 3DTiles is a highly efficient format for streaming and visualising large-scale 3D city models, making it an ideal choice for our work. The ability to transform these different data sources into a unified format ensures smooth integration and efficient handling of the data.

By sourcing data from IGN, we can reasonably assume that our methodology will be applicable to most of France. This extensive coverage is one of the key advantages of using IGN data. However, the outdated CityGML dataset for Lyon presents a notable challenge. The dataset, being from 2018, may no longer accurately reflect the current urban landscape of the city. This discrepancy highlights the need for more frequent updates to CityGML datasets, especially for rapidly evolving urban environments like Lyon.

In summary, while we are fortunate to have access to reliable data from IGN and the Lyon Metropolis, the gap in CityGML data poses a limitation in terms of up-to-date urban representation. As urban environments continue to change, obtaining timely updates will be crucial to maintaining the accuracy and relevance of 3D urban models. Until more recent datasets become available, alternative approaches, such as integrating additional data sources or utilising remote sensing technologies, may be necessary to ensure our models reflect the current state of the urban environment.

3.2.1 Description of the data flow

Figure 17 represents the flow of data of the methodology. py3DTilers is particularly important, as the gateway to this pipeline, it serves to nomalise and triangulate the data. The geoJSON data in particular, that we use to define the road system, is not completely 3D and does not

[20]https://threejs.org/manual/#en/fundamentals

contain geometry. Instead, it consists of a bidirectional graph, with each node representing a road crossing or the end of a road, with coordinates. Each vertice of the graph contains the road name and its width, allowing py3DTilers to recreate the geometry. A similar process is done for the terrain, wich constist of 2D + height data, which is triangulated by py3DTilers to create the 3DTiles tile set. The 3DTiles conversion process also includes tiling (as the format name suggests), fitting particularly well with the methodology.

The second step is managed by py3DTiles, which reads the 3DTiles tileset and provides a convenient in-memory representation of it. It allows for the manipulation of the tile set and its conversion into the Triangle Soup representation, necessary for the Sunlight library. The triangle soup is a uni-dimensional vector of triangles, stripped of semantics (ownership, etc) but associated to a unique identifier, permiting the semantic to be retrieved later in the pipeline. It allows for fast memory access, accelerating the sunlight calculation process.

Sunlight exports back a triangle soup, with each triangle containing its lighting information, and occluding triangle if it is occluded. pySunlight reads the triangle soup, and writes the lighting information in the CSV format for further treatment.

One possible use case, which is shown in the chart, is data visualisation with iTowns. Here the CSV is used, in combination with the original 3DTiles tileset to display the shading information, directly on the geometry.

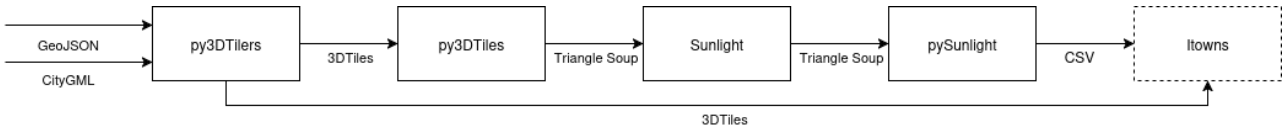

Figure 17: Data flow chart

3.3 Limits of existing works

During the initial stages of my internship, numerous challenges arise while attempting to use the software, primarily related to installation difficulties, usability constraints, and unreliable computational results. At the outset, several technical limitations and bugs were identified. One significant issue is the inability to select the desired export type directly from the command-line interface. Users are required to manually comment or uncomment sections of the source code to switch between export formats.

Another major concern involved inconsistencies in the sunlight computation results. These discrepancies, illustrated in Figure 16, prompted an extensive debugging process. Given the complexity of the system, potential sources of error spanned several modules, including data visualisation, format conversion, geometry manipulation, and ray tracing algorithms. This process was made more difficult by the limited documentation and the size of the codebase, with which I was initially unfamiliar.

A particularly intricate issue arose in connection with the 3DTiles export functionality, where exported tile sets exhibited geometric distortions. This problem appeared to be multifaceted. Several hypotheses were considered, including potential mismatches in coordinate reference systems (CRS), incorrect handling of the conversion between the 3DTiles specification and the "triangle soup" format expected by pySunlight, and the use of custom tile-loading routines that failed to account for hierarchical dependencies between parent and child tiles. The last one being the curent culprit, although no final solution has been found as of now.

The installation process itself was outdated and no longer functional in its current form, largely due to the accumulation of technical debt and a lack of ongoing maintenance, particularly within the py3DTilers module, which had several broken or incompatible dependencies. These

dependency issues also extended to pySunlight, which could not be built using the existing Docker wrapper.

Further complications were encountered when attempting to compile the underlying C++ sunlight library. The build process failed due to dependency conflicts with spdlog, a logging library, compounded by changes in CMake configuration and evolving version requirements. Finally, the JSON export feature was found to be non-functional, further limiting the software's utility in its original state.

4 Propositions

In this section we will detail design propositions that arose during the internship. First within a Conceptual design perspective, and second through their implementation.

4.1 Conceptual Design

4.1.1 Rework of triangle feature tile sets

In an effort to upgrade pySunlight's exports consistency and accuracy, a rework of the batch table usage was proposed. In its current form, pySunlight creates a feature for each triangle of the tile set and assigns batch table data, with the lighting info, and the occluding triangle if it was occluded, to each of the triangle level features. This makes sense, because batch table data is assigned to features and not triangles, so assigning batch table data to individual triangles requires features with only a single triangle. However, having triangle level features causes multiple problems.

1. A large amount of the semantic associated with the features is lost, even though it could be rebuilt via triangle IDs and reusing the original tile set.
2. The geometry is often misaligned, because of the unstable implementation of py3DTilers.
3. Poor performance when reading and displaying the tile set, since the geometry is not bundled together efficiently, with a single object per feature, it requires a lot of *THREE.js 3DObjects* instances, greatly affecting performance.

To tackle the problem a description of how batch tables are defined is needed. According to Cesium's documentation *"A Batch Table is a component of a tile's binary body and contains per-feature application-specific properties in a tile. These properties are queried at runtime for declarative styling and any application-specific use cases such as populating a UI or issuing a REST API request"*.

In all cases, all the features of a given tile share the same properties (ex. Height, IDs) with different values stored in them. These properties cannot be lists of indeterminate length. So, for example, creating a property *Lighting* for each of the features, associated to a list of boolean the size of the features' geometry, would not be allowed. One workaround would be to find the feature with the most triangles in its geometry and make all the lists of Booleans this size. This is costly (counting a list of triangles in each feature + finding the biggest one), so another workaround could be implemented. Strings derogate from this rule, as you can assign different sized string to each feature (for example for a name property). This property of strings enables an alternative encoding strategy.

We proposed a simple translation from list to boolean to string with each *true* becoming a one and each *false* becoming a zero. This would result in a conversion as such :

This is much more efficient and cost-effective, as well as being very easy to parse for data visualisation. However, this raises the consideration of wether sunlight information should be decoupled from the geometry.

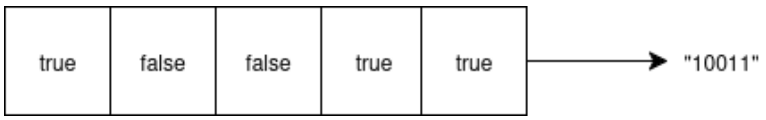

Figure 18: List to string conversion

4.1.2 CSV export of sunlight results

Exporting the results into a CSV (Comma Separated Values) file would be solution to decouple the data from the geometry, allowing for flexibility in its usage. CSV's simplicity implies that the format is easy to manipulate in any situation, any programming language capable of reading text files will sufice. Initially, the export contained four columns, ID, date, blighted and occluding. With ID and Occluding being a string representing a Triangle ID. It was decided to change the way triangles are referenced to facilitate correspondence between the results and the original data. Table 1 is an example of what you can now find in a CSV export.

Tile ID	Feature ID	Triangle ID	Date	Lighted	Occluding Tile ID	Occluding Feature ID	Occluding Triangle ID
0	10	21	2016-01-01	True			
1	2	15	2016-01-01	False	0	10	21

Table 1: CSV File example

As you can see some fields remain empty under certain condition (when a triangle is lighted). On the first line, the triangle is lighted, and correspondence with the initial geometry can be achieved via the tile, feature and triangle ID. The triangle ID corresponds to the triangles position in the triangle list of the feature, each feature possesses a list of the triangles that are contained in its geometry. This list of triangles is fixed, as per the specification, so any tool that reads 3DTiles should represent them in the same order, if it is compliant with the specification. Please note that a feature's ID is only unique inside the same tile, so for example you could have two features with ID *2* if they are in different tiles. The second line is a record of an unlit triangle, (Lighted is False), alongside the ID's necessary to find the occluding triangle. Storing this data is interesting for in depth Urban studies, allowing the user to study the impact of an object on top of knowing which objects are impacted.

We will propose later a way to visualise our results via this CSV export.

4.1.3 Obj version of Sunlight

This proposition is motivated by the need for extensive debugging of the whole (py)Sunlight software. For this purpose we propose a stripped down version of the pySunlight library, operating on geometry from Obj files. This allows us to isolate the components of Sunlight, and verify their integrity independently of the python wrapper (pySunlight). Eliminating possible points of failures allows us to detect errors more accurately, and isolating each component is a common way to do that.

The Obj file format was chosen for its relative simplicity, the great availability of test data in this format and the ability of py3DTilers to convert cityGML data in the Obj format, allowing us to test on real world data later on. The fact that a multitude of well maintained c++ libraries capable of parsing the format exist also played a part in that decision. Working on smaller, easily readable geometry also allows us to iterate efficiently.

To handle Obj data, we chose to use rapidOBJ[21], a lightweight library designed for fast and efficient parsing of large Obj files. The main advantage of rapidObj is its single header file

[21] https://github.com/guybrush77/rapidobj

implementation, which minimizes setup complexity. Additionally, its efficiency in reading large datasets made it ideal for our use case [Pranckevičius, 2025], where Obj files can contain tens of thousands of vertices and faces. However, while rapidOBJ excels in parsing the raw data, the way it stores and structures the information is not immediately compatible with Sunlight's data format.

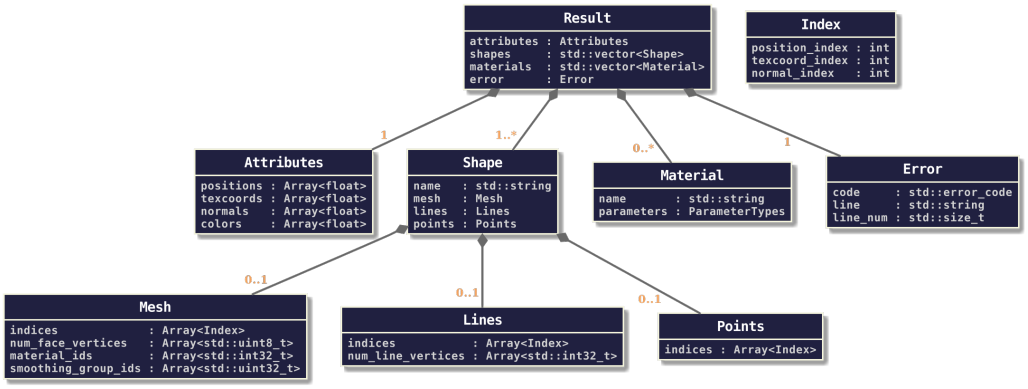

Figure 19: rapidOBJ data layout

Obj files typically store geometrical information in a series of shapes, where each shape has its associated vertices and indices, and in some cases, normals. In contrast, Sunlight requires a different data structure, specifically Triangle Objects that store both vertex positions and normals, for further computations. The conversion process from rapidOBJ's structure to the format needed by Sunlight involved several steps:

- **Parsing Obj Data**: We started by parsing the Obj file to access individual shapes and their associated position indices. Each shape can contain multiple triangles, and we had to ensure that each triangle's geometry is correctly identified and how it is represented in rapidOBJ.
- **Creating Triangle Objects**: From this data, we generated Triangle Objects compatible with Sunlight's geometry representation. This step requires converting the raw data into a more structured format with precise vertex positions and face indices, which can be fed into the sunlight computation algorithms.
- **Handling Missing Normals**: A significant challenge with Obj files is that normals are optional. Some Obj files may not include normals at all, which is common in cases where the mesh does not need to store surface orientation information (e.g., for rendering purposes). Therefore, we had to implement a system to check for the presence of normals in the parsed Obj data. If normals were missing, we computed them on the fly using the geometric properties of the triangles (i.e., by calculating the cross-product of the triangle's edges). This computation ensures that our Triangle Objects have all the necessary information for sunlight computations, even when the input data is incomplete.

Once these steps were completed, we had a set of Sunlight-compatible Triangle Objects ready for further processing. At this point, the next logical step was to begin debugging and testing the sunlight calculation algorithm. Given the complexity of ray-triangle intersections, we needed a structured approach to debugging. We decided to start by testing the simplest and most basic scenario to ensure the foundational algorithm worked as expected. The best strategy is to start with the most simple situation, for us that would be a ray with origin $O = (0, 0, 0)$ and pointing along an axis $D = (0, 0, 1)$ and a triangle right in front, with its centroid on $C = (0, 0, 1)$. This simple test case was the ideal starting point because it allowed us to validate

the core ray-triangle intersection logic before adding additional complexity. By ensuring this test passed, we confirmed that the basic framework for sunlight calculation was functioning correctly. Once the basic test passed, we gradually introduced more complex interactions to cover edge cases, such as:

- A triangle that intersects exactly at one of its vertices (testing precision and handling of degenerate cases).
- A triangle located behind the ray origin (testing for correct intersection directionality and handling of backward-facing geometry).
- Multiple triangles intersecting with the ray at once (testing performance and robustness under more complex scenarios).

Throughout the debugging process, several errors were identified, especially related to the ray-triangle intersection algorithm. The initial implementation of the ray-triangle intersection algorithm was faulty, leading to incorrect results in some edge cases. We refined the algorithm based on the results from our unit tests, improving its accuracy in handling degenerate cases (e.g., when the ray intersects exactly on a vertex or edge). The conversion process itself also introduced a few errors, primarily due to incorrect handling of normal vectors. Some of the converted Triangle Objects had inaccurate normals, causing incorrect sunlight calculations. This was addressed by double-checking the normal computation and ensuring the correct application of the cross-product formula when normals were missing. The detailed conversion process, along with systematic debugging and unit testing, ensured that Sunlight was capable of handling Obj files with various complexities. Each step—starting from parsing Obj data to handling missing normals and debugging complex intersection cases—was essential for achieving reliable sunlight calculations.

4.1.4 Multi-layer Sunlight computation

Multi-layer 3D Tiles computation refers to the integration and interaction of multiple 3D Tiles datasets—such as buildings, terrain, roads, and vegetation—for advanced environmental simulations, in our case sunlight exposure analysis. Each layer represents a distinct semantic category of objects, and when combined, they create a comprehensive 3D urban model where the presence and geometry of one layer can influence the others.

Crucially, our method allows for selective sunlight computation. This means that while all layers can interact geometrically, casting shadows and influencing light exposure, we can control which specific layers are evaluated for sunlight access. For example, we might want to calculate how much sunlight reaches a road surface while accounting for shadows cast by buildings and trees, but without needing to determine whether the trees themselves are exposed to sunlight. This selective computation helps us manage processing resources efficiently, focusing only on the layers relevant to the current analysis.

While merging multiple tile sets into a single dataset is already technically feasible using tools like py3DTilers, this approach has significant limitations. Merging would result in the loss of semantic granularity—after the merge, it becomes difficult or impossible to distinguish between different object types (e.g., building vs. vegetation), which is critical for nuanced analysis. For instance, determining the shading effect of a tree on a pedestrian path is fundamentally different from assessing the sunlight access of a building's façade.

To address this, we have developed a new approach that maintains the separation of tile sets while enabling them to interact during sunlight calculations. Our method introduces the concept of "secondary datasets"—independent 3D Tiles layers that influence the computation by casting shadows without themselves being evaluated for sunlight exposure. These secondary

datasets are fully integrated into the scene but are treated as passive elements in the analysis. This allows us to preserve semantic detail, retain analysis flexibility, and reduce unnecessary computation.

This approach is particularly valuable for extending existing urban datasets with new thematic layers. For example, by generating a 3D Tile set representing vegetation, we can incorporate it into an existing urban model to simulate its shading effect—without needing to recalculate sunlight access for the vegetation itself. This modularity allows urban planners and researchers to construct complex and realistic models while maintaining performance and analytical clarity.

4.1.5 Argo Workflows

For our workflow, we propose a DAG (Directed Acyclic Graph). This allows us to have a few preliminaries such as data collection and converting to 3DTIles thanks to py3DTilers. We would then have a diamond shape, representing the division of work on the data, with one node for each timeframe, working in parallel to accelerate our workflow. This would result in a more reproducible, ready to use and stable data treatment, however argo workflows is a very resource intensive tool to deploy, we have to recognize that although it is a nice inclusion, its complexity might be prohibitive for smaller organisations. Thankfully the LIRIS has its own argo workflows implementation, running on our servers, which spares us from having to deploy it ourselves.

Here is a representation of what our workflow would look like :

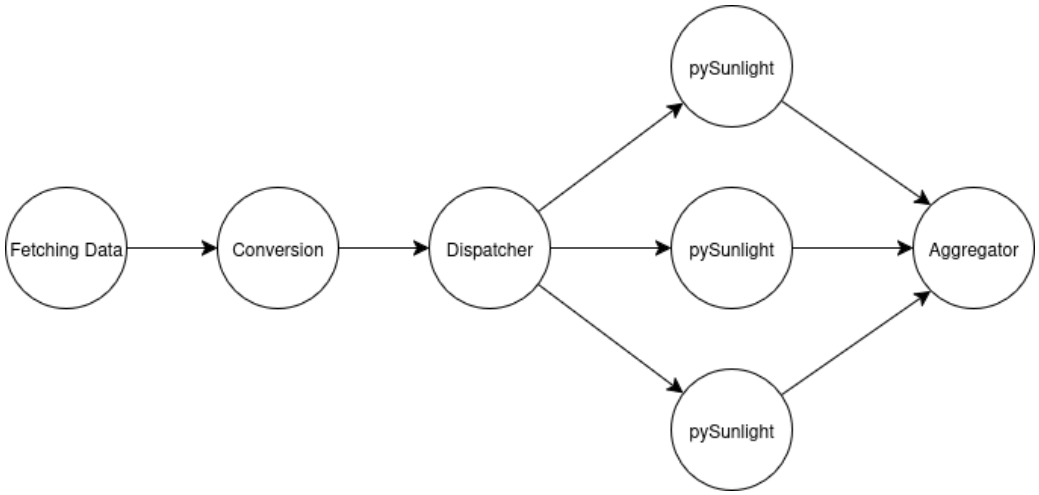

Figure 20: Proposed workflow diagram

In this example, we parralelise the computation, allowing us to make a better use of our ressources and give out results for large time intervals faster. The dispatcher is tasked with creating the adequate number of pySunlight instances, corresponding to the number of timeframes we want to compute. This approach is generic and adapted to calculations on large datasets, with a large amount of timeframes. It would be counterproductive on smaller workloads, but very beneficial for larger ones.

4.2 Implementation

4.2.1 Upgrading pySunlight's python version and dependencies

One of my initial objectives was to upgrade pySunlight from python 3.9 to python 3.12 as part of a broader effort to modernise the software and align its dependencies (including, NumPy,

py3DTilers, and py3DTiles) with their most recent stable versions. Multiple unsolvable problems arose, leading us to reconsider this proposition. First of all, within the two-year period separating Wesley's internship and mine, a lot of the architecture of py3DTiles had changed, notably the internal representation of *gltf* objects and their associated geometry underwent significant restructuring, alongside widespread renaming and refactoring of key functions. These changes were affecting both pySunlight *and* py3DTilers. The latter could not directly be altered, as it is maintained by Oslandia, a private company. Rewriting pySunlight to take into account the new paradigm would have been time-consuming, while providing little to no value to the project. Moreover, py3DTilers' latest version (and most versions released after the one used in pySunlight) are unstable. This is partly due to the py3DTiles changes discussed above, and bugs introduced during development that have not been corrected due to a lack of support. This issue prevents upgrading pySunlight's dependencies as we are now forced to use a legacy version of py3DTilers.

For these reasons, the effort to upgrade pySunlight's dependencies and python version support were reconsidered.

4.2.2 Obj sunlight

This section follows the Conceptual design one pertaining to the conversion of Sunlight (the original C++ library) into a standalone application.

The requirements for the new sunlight version are as follows :

1. Run as a standalone application.
2. Parse Obj files.
3. Convert the results into sunlight objects (*TrianlgeSoup and Triangle*).
4. Calculate sunlight exposure on the geometry with real sunlight position.
5. Export the results in the Obj file format for further analysis.
6. Contain unit tests.

Allowing it to run as a standalone is relatively straightforward. It requires modifying the `CMakeList.txt`, replacing `add_library()` by `add_executable()`, and creating a `main.cpp` file, which serves as an entrypoint. To parse obj files, rapidOBJ was chosen, as detailed in the Conceptual Design section. Including the library's header in the `main.cpp`, and using its provided `ParseFile()` function was sufficient to enable parsing of Obj files. The library also provides a robust triangulation tool, which was used to normalise the input data, some buildings being composed of polygonal faces.

Once the parsing and triangulation steps were completed, the data needed to be added to `Triangle` objects, and stored in a vector of triangles for sunlight to be able to process it. Objects are stored in multiple lists (see figure 19), namely, Attributes, Shapes and Materials.

- **Attributes** store vertex specific information, such as Position, Normals and Colors.
- **Shapes** store object specific information, such as which vertices are used to define each face and how many are used per face.
- **Materials** contains information about external `.mtl` files and to which vertices they are linked. They were not used in the implementation as this information is not necessary for Sunlight's computations.

Together, these components describe the geometry contained in an OBJ file. Shapes define objects, but the vertex data is stored inside the Attributes list. By linking the two, we are able to rebuild a geometry of `Triangle` objects, stored in a vector, so that Sunlight can process the geometry consistently with its internal data structures.

For the sunlight computation, the sun's position is parsed from a CSV of precalculated positions. A normalised vector is created from the position of the tileset (latitude and longitude), to the sun, and reused for each ray. Each triangle is iterated over, and a ray is generated, starting from the centroid of each triangle, and pointing in the precomputed direction. Each ray is tested for intersection with the geometry and the results are recorded, shading the associated triangle in the process.

The results are output as two different obj files. One for the occluded triangles, and one for the triangles that receive sunlight. This separation allows for direct visual comparison of sunlit versus occluded surfaces in *MeshLab*.

To ensure the correctness of the geometric computations, a lightweight test suite was implemented in a header-only file. This suite validates the behavior of the ray–triangle and ray–AABB intersection methods under a variety of scenarios, including standard intersection, no intersection, and edge cases such as rays starting inside a box, originating on a face, or passing exactly through a vertex. While the current setup relies on simple inline functions and console output for validation, it provides essential coverage of the core geometric operations used by Sunlight. A more formal testing framework (e.g., GoogleTest or Catch2) could be integrated in the future to improve scalability, automate regression testing, and provide richer diagnostic feedback.

There tests directly support the sunlight computation process described earlier. They allowed for detecting a faulty implementation of the ray-triangle intersection test. The Möller–Trumbore ray-triangle intersection algorithm [Möller and Trumbore, 2005] was implemented as a functional replacement.

4.2.3 Data visualisation

To visualise the results, the first effort was put into reusing what was already available through UD-Viz. The objective was to fetch and parse the data from the CSV result and use it to shade the tile set. However, as will be described later, UD-Viz being built on top of ITowns, although it contains very useful pre-built routines, obfuscates some of ITowns' and *Three.js*' features that are required for the implementation. The effort was then put towards creating a standalone ITowns web app from the ground up.

The first part of data collection was to parse the CSV data with JavaScript and create an object for us to reuse it later in the script. At first, JavaScript's MAP objects were used. They are dictionaries, having a feature map for each tile, containing a list of shading value for their triangles like described in the next figure.

With lighting data taken care of, the next step consisted on displaying 3DTiles tile set(s). ITowns is well-equipped for 3DTiles rendering, with a specific *OGC3DTIles* layer, that imports the geometry and stores it for us in a *Three.js* Group. Shading each individual triangle with sunlight information is the next objective, however the styling options given to us by ITowns can only affect a whole feature, as described before. A new way of affecting the geometry was needed to implement the feature. This comes in the form of a *Three.js* Material, materials can be applied to a geometry, and they will change the way they are displayed. The implementation employs a *ShaderMaterial*, it allows the user to define a customisable vertex and fragment shader, using the GLSL[22] language. This allows manipulation of the position of vertices, and the color of triangles at runtime on the geometry. This feature was used to read vertices color

[22] `https://learnopengl.com/getting-started/shaders`

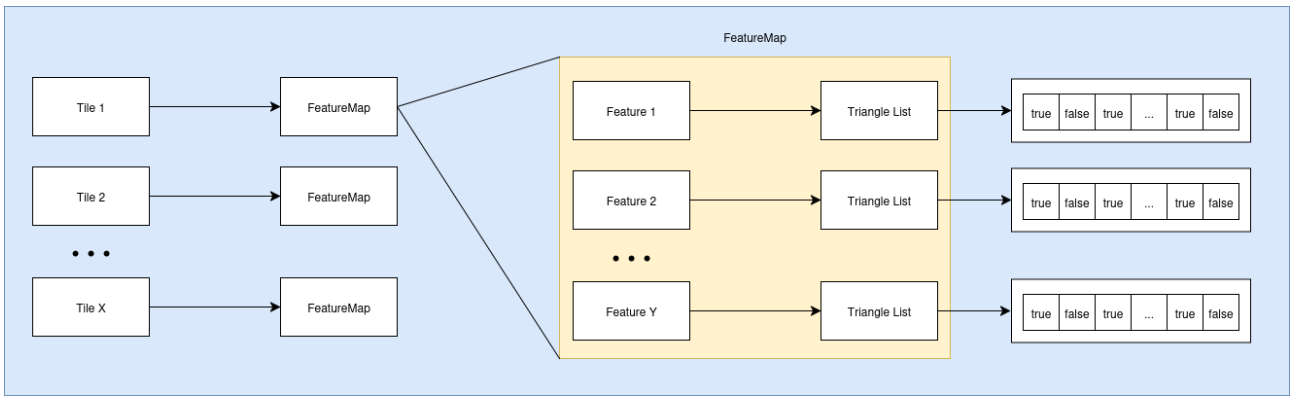

Figure 21: Original data structure of lighting data

attributes and display colored triangles accordingly. The color attributes were extracted from the CSV data, produced by pySunlight, and injected into the geometry. To implement it, we first have to understand how the geometry is stored inside Three.js, and how to add the lighting information to it, in a way that can be used by the shader.

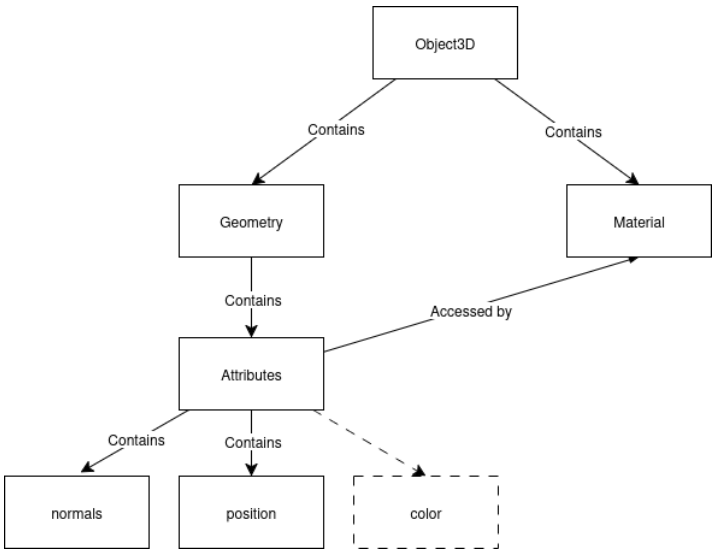

Figure 22: Partial representation an Object3D's data structure

Here the three lists in attributes, are of the same size and contains floats. They represent vertex normals, positions and colors. The objective is to inject the color into the attributes for it to then be accessible in the vertex shader. To do so the load-model event was used, inside which a new *BufferAttribute* object was created, and added to the objects attributes. The *BufferAttribute* class stores data for an attribute (such as vertex positions, face indices, normals, colors, UVs, and any custom attributes) associated with a *BufferGeometry*, which allows for more efficient passing of data to the GPU. The color in the vertex shader is accessed as such :

```
#include <itowns/precision_qualifier>
#include <common>
#include <logdepthbuf_pars_vertex>

out vec3 vColor;

void main() {
#include <begin_vertex>
#include <logdepthbuf_vertex>
    vColor = color;
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Figure 23: Vertex shader passing color attributes to the fragment shader

The Color is then passed onto the fragment shader, an alpha channel is added, and now the geometry is shaded, according to the sunlight computation, without breaking it into triangle level features.

```
#include <itowns/precision_qualifier>
#include <logdepthbuf_pars_fragment>

in vec3 vColor;

void main() {
    #include <logdepthbuf_fragment>
    gl_FragColor = vec4(vColor, 1.0);
}
```

Figure 24: Fragment shader, using the color attribute from the vertex shader

During this process, modifications to the way sunlight data was stored were required, as the *per-feature* triangle list was unnecessary and inconvenient. It was simplified so that each tile stores a list, corresponding to the triangles in the Three.js object.

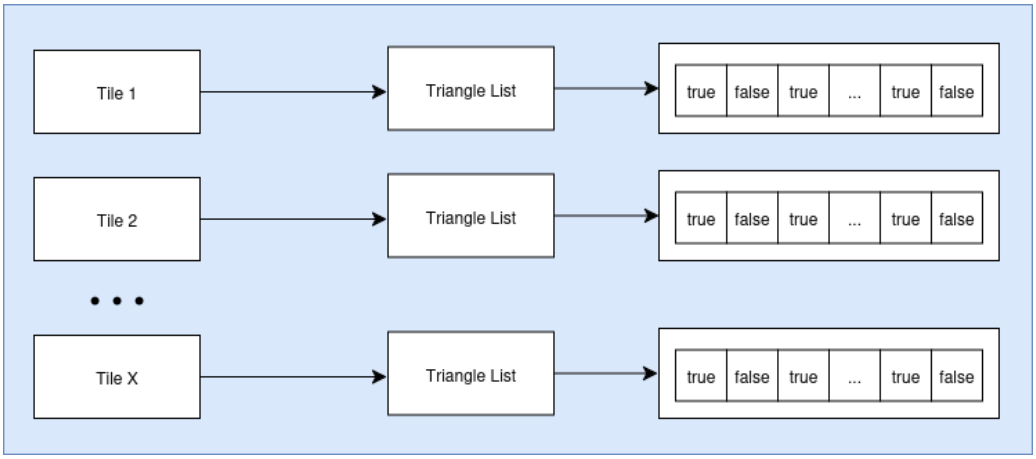

Figure 25: Revised data structure of lighting data

4.2.4 Multi-layer compatibility

Occlusion aware multi-layer sunlight computation refers to the use of multiple tile sets, referred to as Layers, in the same pySunlight computation. Layers that will receive shade are referred to as called Primary layers, as for the one that only provide shade, they are referred to as called secondary layers. The implementation needs to satisfy these requirements :

1. The layers should affect each other when calculating sunlight.
2. The user should be able to choose which layers should be shaded, and which layers only provide shade to the other.
3. The layers should not be combined into one, each one needs to remain independent and a separate output for all layers is required. This is to retain as much of the semantics as possible, such as object types and layer specific attributes, in order to facilitate further treatment of the results.

Such modifications required extensive modifications in the way pySunlight reads and processes tile sets. The first modification was to remove the current usage of py3DTilers merging capabilities, as it conflicts with the third requirement. This involved rewriting some of the importing logic, preventing the merging of the various tile sets. Each tile set is then added to one of two list, the list of tile sets to be shaded and the list of tile sets that only provide shade to the others. The latter being optional for the computation. A command line argument was added titled `--secondary` for passing secondary tile set. The `compute_3DTiles_sunlight`function was rewritten to take into account the new paradigm. At the beginning of the computation, it is determined whether secondary tile sets are in use, if they are, they are temporarily combined for intersection testing only, as their own sunlight results are not computed. The tile sets are integrated into the main ray-triangle intersection computation, respecting the distinction between primary and secondary layers. Unit tests were added to pySunlight as well to supplement the feature, and help with further maintenance and modifications of the project. The tests are composed of computing results with primary and secondary tile sets, for each export method. The setup relies on the industry standard *Pytest* framework, allowing for integration into most IDEs (Integrated Development Environment), and automatisation of unit testing.

5 Results

In this section, the OBJ and CSV results have been rendered, using either meshlab or ITowns. These renders show parts of Lyon that have been processed either by the OBJ version of Sunlight for figure 26, or by pySunlight for the other two.

In figure 26, meshlab has been used to render the computation results of Sunlight's obj version. The sun is positionned close to a zenithal position, with a slight tilt to the right from the persective of the camera (which corresponds to 12:00). The results look very binary, with little to no faces that are partially shaded or partially lighted. This is likely due to the geometry being composed of very big triangles, with little granularity. As a result most rectangular faces are composed of ony two triangles, meaning only two sample points per face. Additionally, harsh yellow color makes the angles of buildings hard to see in a still picture. Better renders could be made with some efforts, but it was deemed unnecessary as the current state of the meshlab visualisation is sufficient for debugging, which is its only intended purpose.

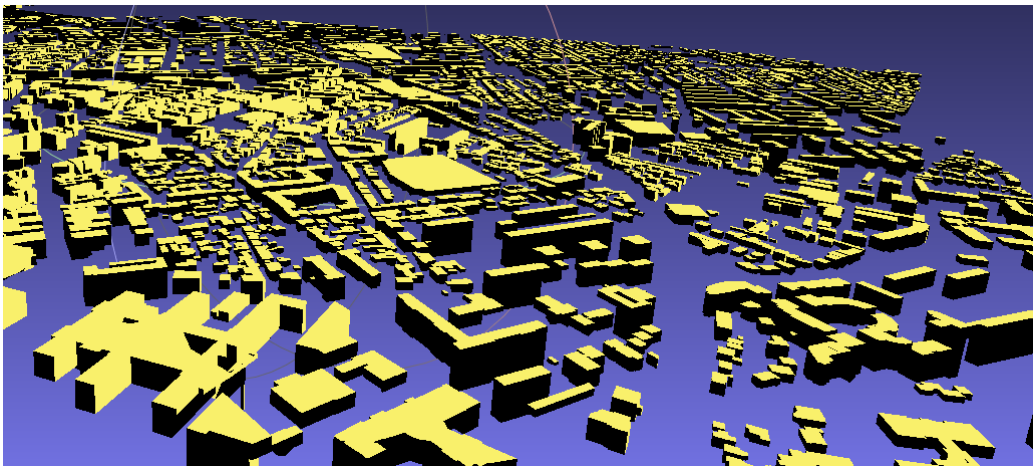

Figure 26: Render of sunlight calculation in obj for the 8th district of Lyon

Figure 27 shows a visualisation of shaded buildings in Lyon on top of plane, textured with a satelite photo of the area. This visualisation was performed with the custom ITowns web app described in section 4.2.3 about data visualisation. The visualisation is close to ground truth, however some buildings appear to be floating in front of others. This is due to a Depth-Buffer issue in the implemented GLSL Shader[23]. The depth buffer used is a custom ITowns built one, adapted to 3DTiles layers, debugging still needs to be done, as it seems it wasn't used correctly in the current ITowns web app.

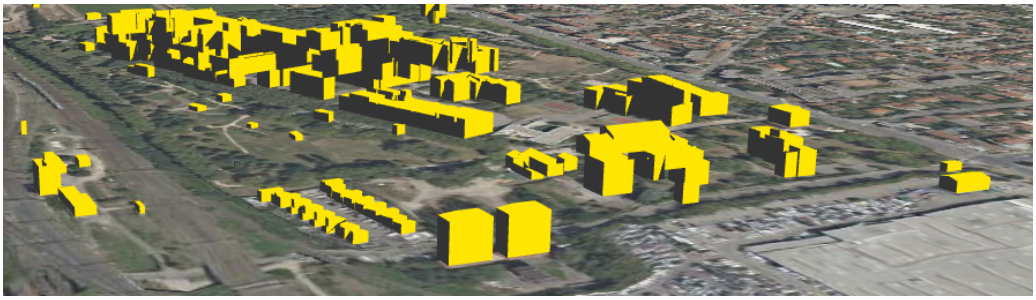

Figure 27: Visualisation of shaded buildings

This render of a roads tile set is only partly shaded, because only a small portion of the 1st disticts buildings was used. Which explains the fully lights road on the bottm left and bottom

[23] `https://www.opengl.org/archives/resources/faq/technical/depthbuffer.htm`

right. It is suspected that the completely occluded roads in the center are due to overlapping geometries.

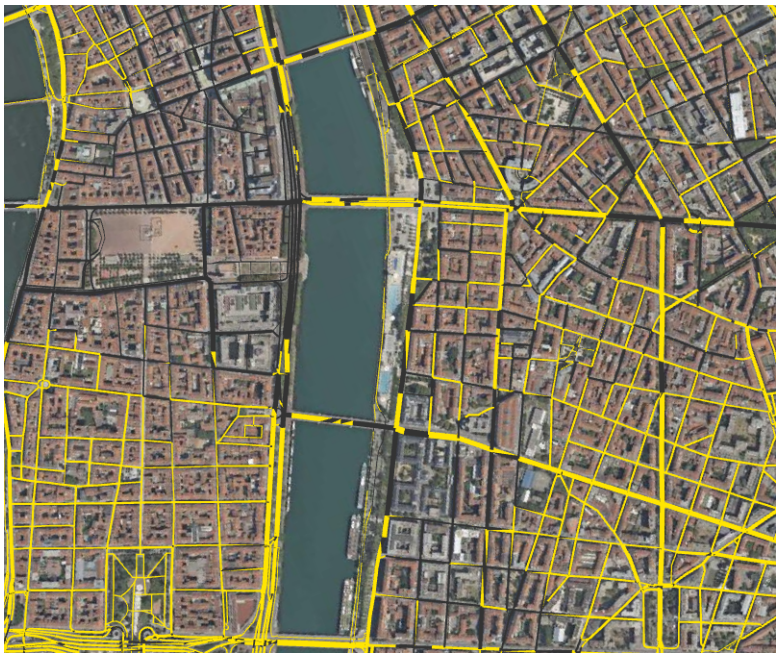

Figure 28: Visualisation of shaded Roads

The results can be heavily critisised, both in accuracy and legibility. Debugging is still due to be done before the end of the internship, in the visualisation tools and, most importantly, in the reading of the 3DTiles tilesets. There are known issue with geometric errors and wrongfully applied offsets when reading the geometry with py3DTiles that will be focused on for the rest of the internship.

6 Discussion and Conclusion

Throughout this internship, several propositions and implementations were explored to enhance urban sunlight modelling, highlighting both the potential and the limitations of current approaches. On the conceptual side, accessing high-quality datasets such as IGN's BDTopo and Lyon Metropolis' CityGML allowed us to construct detailed 3D urban models. However, differences in data formats and outdated datasets (e.g., CityGML 2018) presented challenges in maintaining accuracy and up-to-date representations. The need to integrate multiple sources and standardize them into the 3DTiles format proved essential for consistent modelling, but it also revealed the importance of timely data updates for dynamic urban environments.

From a technical standpoint, efforts to modernize pySunlight, particularly through dependency upgrades, highlighted the difficulties of maintaining legacy software alongside evolving libraries. While the direct upgrade was impractical, it led to insights on reworking tile set structures, batch table handling, and data export formats to improve performance, accuracy, and usability. The development of an Obj-based standalone version of Sunlight enabled systematic debugging and verification of core ray-triangle intersection algorithms, demonstrating that isolating components can significantly improve reliability and transparency in computational pipelines.

The multi-layer sunlight computation and visualisation efforts illustrate the practical implications of this work for urban analysis. By maintaining separation between primary and secondary layers, it became possible to assess sunlight exposure with greater semantic detail,

while also preserving computational efficiency. The visualisation pipeline, leveraging ITowns and Three.js shaders, further underscored the importance of structuring lighting data effectively, enabling interactive exploration of large-scale urban models without sacrificing detail or performance.

Some efforts still needs to be put into alternative results format, for example rasterised results. While improvement will be made during the rest of the internship to ensure stable reading of the tile set, preventing small shifts of the geometry.

In conclusion, this internship not only produced tangible improvements to urban sunlight modelling but also highlighted broader lessons for computational urban analysis: the necessity of flexible, modular software, the value of reliable and current datasets, and the benefits of systematic debugging and visualisation strategies.

Future work could focus on integrating additional dynamic data sources, further automating multi-layer computations, and extending visualisation capabilities to facilitate more interactive and decision-oriented urban planning applications. Overall, the methodology developed lays a solid foundation for scalable and accurate 3D urban sunlight simulations.

Future works could also focus on improving data visualisation, and data treatment. Better visualisation of occluding triangles could provide additional use cases for the descibed method. For example through soft lines, rendered between the occluded and occluding triangles. Or, translucent prisms, constructed between occluding and occluded triangles, that could give helpful visual aid in determining shadow impact of city objects. Additionally, the implementation of the proposed argo workflow pipeline would greatly improve reproductibility of our methodology, as well as its scalability. A complete rewrite of pySunlight could be relevant on the condition that py3DTilers functionalities are transferred into py3DTiles' codebase as planned by Oslandia. It would be preferable for it to happen after the new 1.1 specification of 3DTiles' specification is implemented as well, which is currently in the works according to Oslandia.

Acknowledgments

I wish to thank the IA.rbre Consortium for funding my internship. The Vcity project for all the resources. Gilles Gesquière and John Samuel for their mentoring and helpful insights, Corentin Le Bihan-Gauthier for being a great teacher and helping me find this internship in the first place, Diego Vinasco-Alvarez for helping me with UD-viz and other technical questions, Akin Gulfidan for the helpful design insights and the croissantage, and all the Vcity project's participants for being so welcoming and benevolent.

Funding

This internship was done in the LIRIS laboratory, as part of the IA.rbre project. It was funded by the Banque des territoires, following the DIAT (Démonstration d'IA frugal au service de la transisition écologique des Territoires/Demonstrating frugal AI to serve the ecological transition of territories) call for projects.

References

[Aurenhammer, 1991] Aurenhammer, F. (1991). Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405.

[Fahy et al., 2025] Fahy, J. C., Bachofen, C., Camponovo, R., Gallinelli, P., and Schlaepfer, M. A. (2025). Beyond land surface temperature: Identifying areas of daytime thermal discomfort in cities by combining remote sensing and field measurements. *Urban Climate*, 61:102460.

[Gaillard, 2018] Gaillard, J. (2018). *Représentation et échange de données tridimensionnelles géolocalisées de la ville*. PhD thesis, Université Lumière Lyon 2. Thèse de doctorat dirigée par Gesquière, Gilles et Peytavie, Adrien Informatique Lyon 2018.

[Hoppe, 1999] Hoppe, P. (1999). The physiological equivalent temperature – a universal index for the biometeorological assessment of the thermal environment. *Int J Biometeorol*.

[Jaillot et al., 2017] Jaillot, V., Pedrinis, F., Servigne, S., and Gesquière, G. (2017). A generic approach for sunlight and shadow impact computation on large city models. In *Proceedings of WSCG2017, 25th International Conference on Computer Graphics, Visualization and Computer Vision 2017*, Proceedings of WSCG2017, 25th International Conference on Computer Graphics, Visualization and Computer Vision 2017, page 10 pages, Pilsen, Czech Republic.

[Khronos, 2025] Khronos (2025). Gltf format. `https://www.khronos.org/api/index_2017/gltf`. Accessed: 2025-13-08.

[Leduc et al., 2021] Leduc, T., Stavropulos-Laffaille, X., and Requena-Ruiz, I. (2021). Implementation of a solar model and shadow plotting in the context of a 2D GIS: challenges and applications for the cooling effect of tree-covered based greening solutions in urban public spaces. In Davoine, P.-A., Josselin, D., Pinet, F., and Espace, U. ., editors, *SAGEO 2021, 16th Spatial Analysis and Geomatics Conference*, SAGEO 2021, 16th Spatial Analysis and Geomatics Conference, pages 89–100, La Rochelle, France. Alain Bouju and Christine Plumejeaud-Perreau.

[Manfred et al., 2015] Manfred, W., Alexandru, N., Monjur, M. S., and Jochen., W. (2015). Computing solar radiation on citygml building data.

[McGuire et al., 2003] McGuire, M., Hughes, J., Egan, K., Kilgard, M., and Everitt, C. (2003). Fast, practical and robust shadows. *Brown University Computer Science Tech Report CS-03-19*.

[Michalsky, 1988] Michalsky, J. J. (1988). The astronomical almanac's algorithm for approximate solar position (1950–2050). *Solar Energy*, 40(3):227–235.

[Miranda et al., 2019] Miranda, F., Doraiswamy, H., Lage, M., Wilson, L., Hsieh, M., and Silva, C. T. (2019). Shadow accrual maps: Efficient accumulation of city-scale shadows over time. *IEEE Transactions on Visualization and Computer Graphics*.

[Möller and Trumbore, 2005] Möller, T. and Trumbore, B. (2005). Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 7–es, New York, NY, USA. Association for Computing Machinery.

[OGC, 2025a] OGC (2025a). 3dtiles standard. `https://www.ogc.org/standards/3dtiles/`. Accessed: 2025-13-08.

[OGC, 2025b] OGC (2025b). Citygml standard. https://www.ogc.org/standards/citygml/. Accessed: 2025-13-08.

[Pedrinis and Gesquière, 2017] Pedrinis, F. and Gesquière, G. (2017). *Reconstructing 3D Building Models with the 2D Cadastre for Semantic Enhancement*, pages 119–135. Springer International Publishing, Cham.

[Pranckevičius, 2025] Pranckevičius, A. (2025). Comparing .obj parse libraries. https://aras-p.info/blog/2022/05/14/comparing-obj-parse-libraries/. Accessed : 2025-20-07.

[Takebayashi and Moriyama, 2012] Takebayashi, H. and Moriyama, M. (2012). Relationships between the properties of an urban street canyon and its radiant environment: Introduction of appropriate urban heat island mitigation technologies. *Solar energy*.

[Williams, 1978] Williams, L. (1978). Casting curved shadows on curved surfaces. *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*.