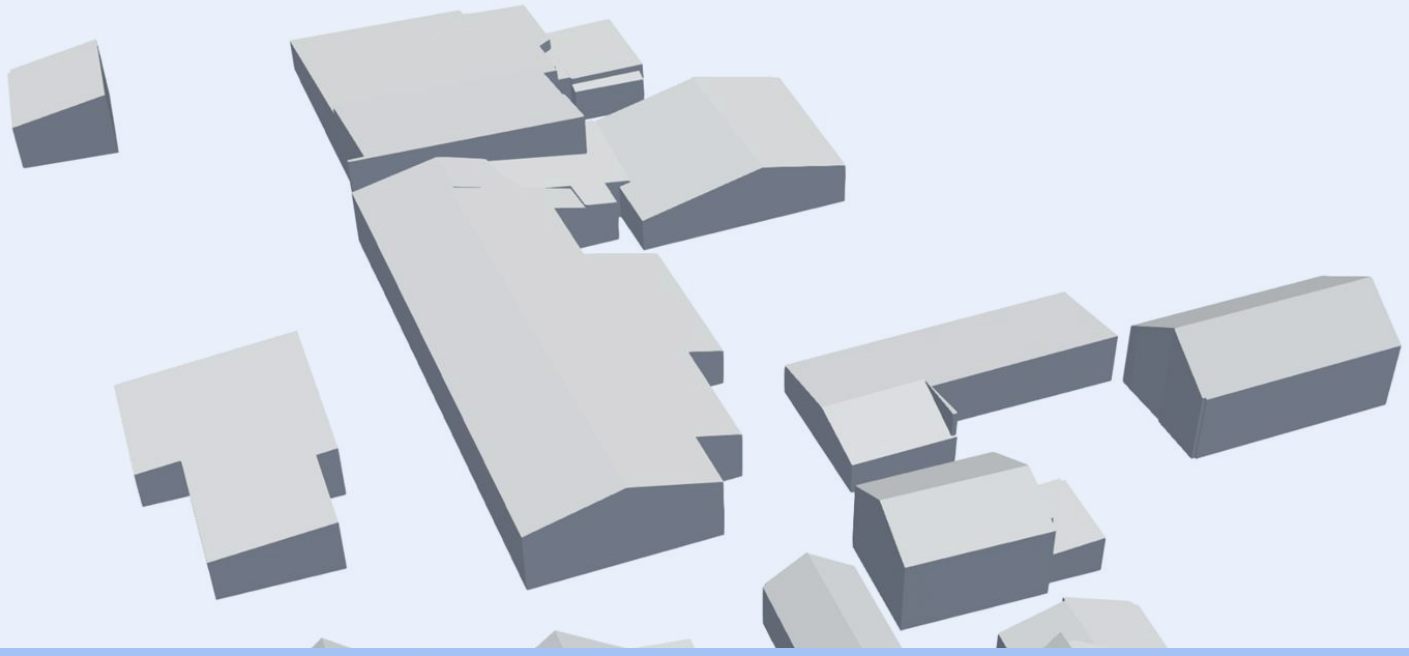
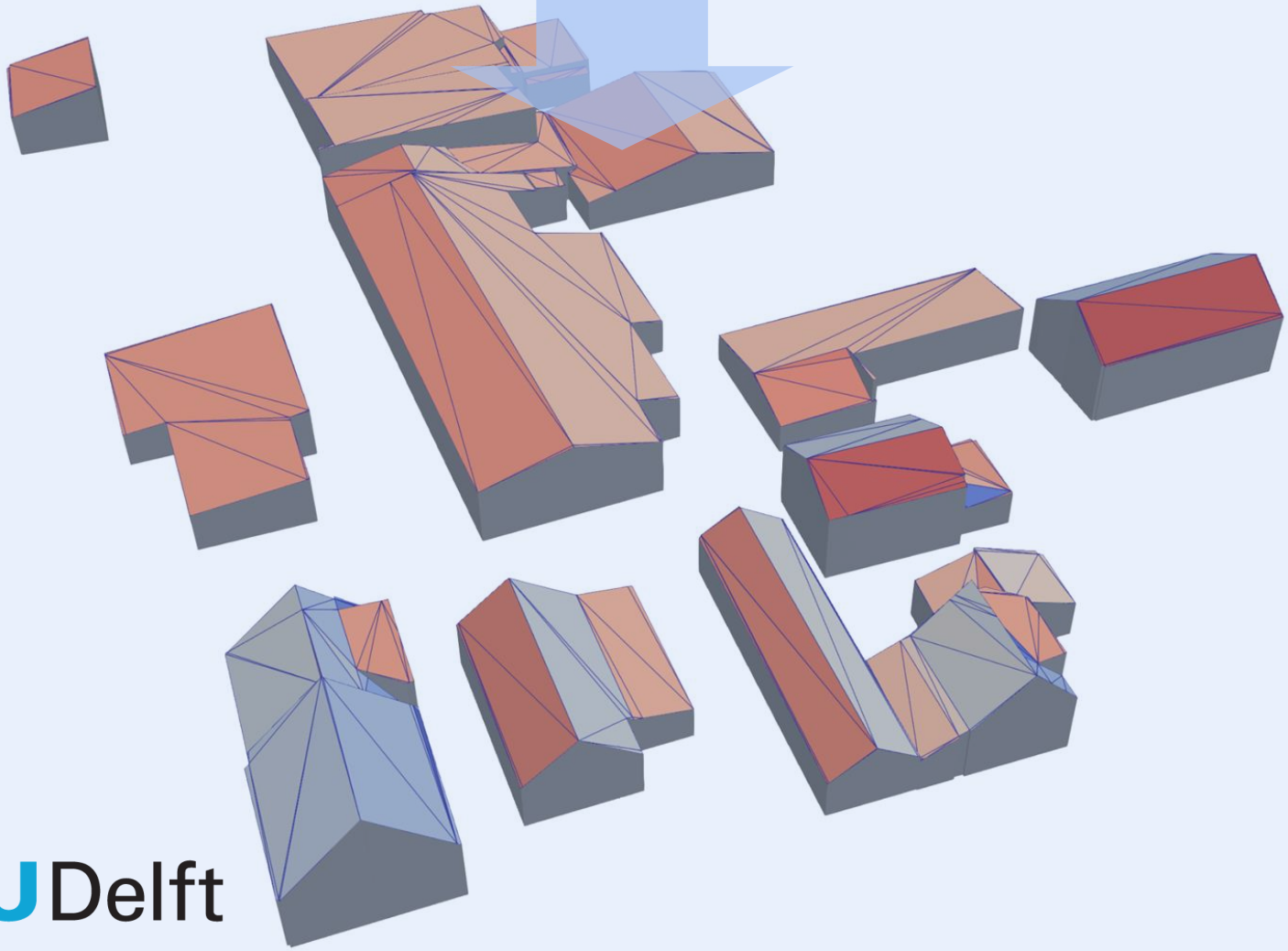


MSc thesis in Geomatics

Efficient Solar Potential Estimation of 3D Buildings: 3D BAG as use case

Robin Hurkmans
November 2022



TU Delft

MSc thesis in Geomatics

Efficient Solar Potential Estimation of 3D Buildings: 3D BAG as use case

Robin Hurkmans

November 2022

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors:	Hugo Ledoux
	Jantien Stoter
Co-reader:	Camilo León-Sánchez

Abstract

Solar energy is an important renewable energy source that is already generated by millions of solar panels attached to building roofs throughout The Netherlands. Whether a roof is suitable for solar panels relies on its type, orientation and whether it is put in shadow by a neighbouring object. This means, the higher the solar potential of a roof, the better.

A solar radiation model is used to determine the solar potential of a roof. Well-known Geographical Information Systems, such as ArcGIS and GRASS GIS provide solar radiation models for raster data. However, raster data does not model the 3D urban environment accurate enough. Vector data is better capable of representing 3D buildings models, but solar radiation models for vector data are not widespread available and are computationally inefficient, because of the shadow casting step.

This research aims at providing an efficient solar radiation model for processing large-scale 3D city models. The 3D BAG data set, containing all the buildings in The Netherlands as vector data, is taken as use case. Their building models are stored in CityJSON format, subdivided into smaller tiles based on the spatial extent. The implemented model in this research, called SolarBAG, takes one or multiple tiles as input, processes the building geometries to compute the solar potential, and outputs an enriched CityJSON file where each building roof consists of yearly solar potential values. Within the process, the building geometries are stored in an R-tree to allow fast retrieval when filtering neighbouring buildings that potentially cast a shadow over another building. The roofs of buildings are sampled into a grid of points to account for the variable solar radiation values on a roof surface caused by shadows of neighbouring buildings. A ray-box intersection method is used to find the neighbouring buildings casting a shadow on another building.

To assess the quality and scalability of the implemented solar radiation method, experiments are conducted. For the quality assessment, the `solarpy` module used to compute the beam solar radiation, is compared to ground truth values, and the solar radiation model is compared to the solar radiation tool in ArcGIS. For the scalability assessment, the solar radiation model is tested for an increasing number of tiles. Based on the assessments, it can be concluded that the implemented solar radiation model can successfully enrich building roofs with solar potential values for one or multiple CityJSON files. However, there are still some bugs and inconsistencies present in the solar radiation model, and performance gains could still be achieved by neighbour filtering improvements.

Acknowledgements

I would like to thank my former supervisor Stelios Vitalis and my current supervisors Hugo Ledoux and Jantien Stoter for their support during the whole, sometimes tedious, process. During our weekly meetings they helped me making progress, shared knowledge from their respective domains and provided me of useful feedback. Additionally, I would like to thank my co-reader for pointing out inconsistencies in my research and giving constructive feedback.

Besides, I would like to thank my study advisor for his support on emotional level and his helpful advice on tackling an MSc thesis.

Furthermore, I would like to thank my study friends of Geomatics for listening to my struggles and giving me meaningful advice and help, based on their own experiences in the graduation process. Finally, I appreciate my mother and sweet girlfriend for always supporting me.

Contents

List of Figures

List of Tables

Acronyms

1. Introduction

Solar energy is the sustainable and renewable energy source with the most promising future prospects to meet global energy demand [Kabir et al., 2018]. Attaching solar panels to building roofs is one such way to generate solar energy and is already being applied extensively in practice. In fact, in The Netherlands, 18.8% of the buildings have already solar panels installed on their roofs by May 2022 [van Groesen, 2022]. This means almost 3 million buildings, which is almost 2.5 times as much as 5 years ago, as seen in the graph in Figure 1.1.

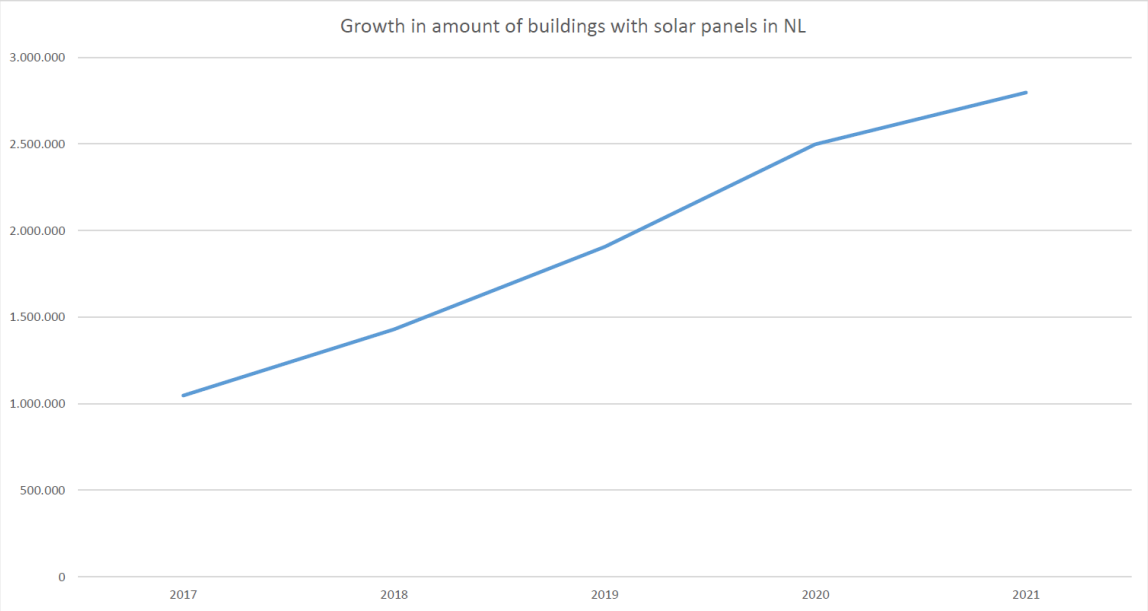


Figure 1.1.: Graph showing the growth in the amount of buildings with solar panels installed in The Netherlands from 2017 until 2021. Data from Kerkhof [2022].

Whether individual roofs of buildings are suitable for solar panels depends on various factors, such as the type of the roof [Song et al., 2018], the orientation of the roof surface, and whether the roof's visibility by the sun is obstructed by another object, for instance a building or a tree [Fu and Rich, 2002]. So, correctly oriented building roofs are ideal locations for solar panels because the generated energy can directly be applied to the household or company and as roofs are elevated, they have a higher chance of avoiding shadows cast on the panels than locations on ground-level. Facades of buildings can also be suitable locations for solar panels. Although they generate less solar energy than roofs, the area to attach solar panels onto is larger, yielding sufficient solar energy [Jaugsch and Löwner, 2016].

To determine the solar potential of a building one can manually utilise solar radiation sensors[1] such as a pyranometer or a pyrheliometer to gather raw solar radiation values for the actual building. However, for large areas it is much more efficient to automate this process and use a solar irradiance model to determine the solar potential of buildings. Modules of

[1] https://www.hukseflux.com/products/pyranometers-solar-radiation-sensors

well-known Geographical Information System (GIS) packages such as Solar Radiation Tools in ArcGIS [ArcGIS, 2016b] and r.sun in GRASS GIS [Hofierka and Suri, 2007] offer solid and fast solar irradiance modelling tools for raster data like a Digital Elevation Model (DEM). However, raster data does not model the 3D urban environment accurate enough. This is specifically noticeable for vertical surfaces such as facades, as a DEM models the environment in at most 2.5D [Redweik et al., 2012]. Another downside is that raster data with a low resolution causes a loss of information and raster data with a high resolution takes more time to process.

To capture the 3D urban environment with more spatial detail, buildings can be stored as vector data. As a benefit, vector data needs less computer memory than raster data. Moreover, vector data is more versatile than raster data and it is built up of geometric primitives like polygons and lines making vector data suitable to accurately represent 3D building meshes. However, techniques that take only 3D vector data as input for solar irradiation modelling are not widely implemented or available yet. The ones that are described or implemented are mostly applied to small scale data sets and are generally not fast enough for large scale data sets [Hofierka and Zlocha, 2012; Liang et al., 2015; Wieland et al., 2015]. This is mostly due to the necessary and computational-intensive shadow casting step to determine whether neighbouring objects such as trees or buildings are blocking the sun rays and therefore casting a shadow on another building as shown in Figure 1.2. Shadow casting is an important topic in this research and will be elaborated on in Sections 2.4 and 4.2.2.

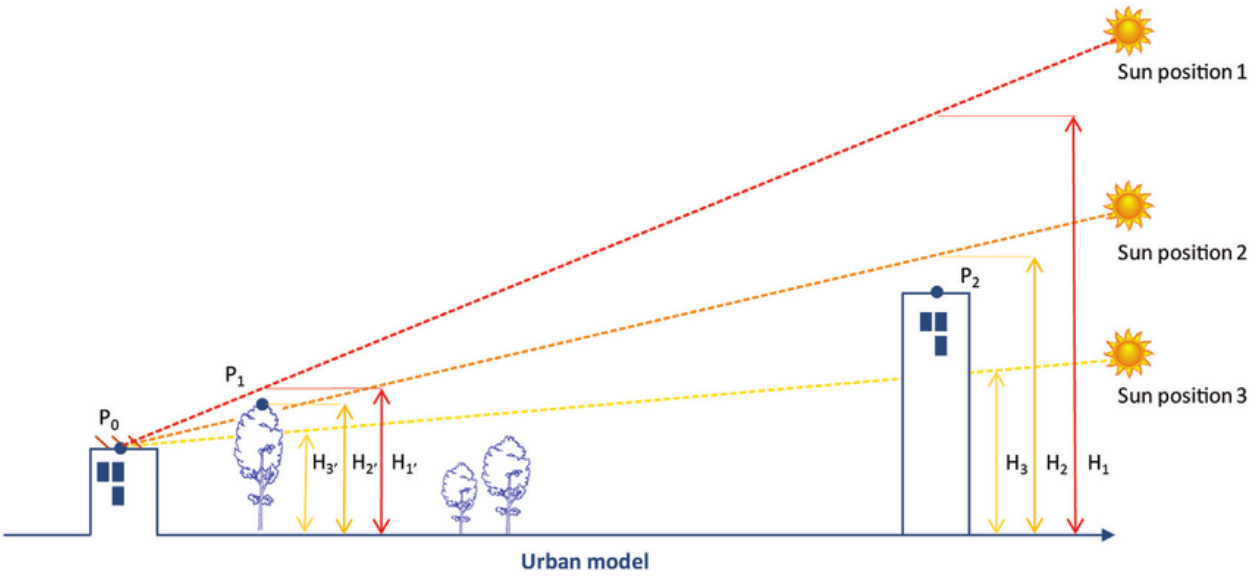


Figure 1.2.: The principle of shadow casting in an urban environment. Adapted figure from Desthieux et al. [2018].

Nowadays, more and more 3D city models are openly available. An example of this is the massive 3D model of all the buildings in The Netherlands [Peters et al., 2022]. It is openly available as the data set 3D BAG[2], where BAG is an abbreviation for 'Basisregistratie Adressen en Gebouwen', meaning Register of Buildings and Addresses. This data set contains a 3D representation of all the buildings as vector data in a LoD of at most 2.2, which includes wall surfaces and sloped roof surfaces. The buildings are stored into squared tiles covering smaller geographic areas.

[2]https://3dbag.nl/en/viewer

1.1. Research objectives

The main objective of this research is to develop a methodology to compute the solar potential of buildings in large 3D city models, such as 3D BAG, efficiently in terms of performance and memory usage. The accompanying main question is:

How can the solar potential of vector buildings in large 3D city models, such as the 3D BAG data set, be computed efficiently?

Sub-questions to help answering the main questions are:

- *How can spatial indexing be used to speed up shadow casting computations on the 3D BAG vector data set?*
- *What simplifications in the solar irradiation model can be applied?*
- *How can the solar irradiation model be implemented in Python by using open source libraries and open data?*
- *How should the computer memory be managed while processing the buildings stored in tiles in the 3D BAG data set?*

1.2. Scope and goal

The goal of this research is to implement a computationally efficient methodology to compute the solar potential for buildings in large 3D city models by taking the 3D BAG data set as a use case. As previously mentioned, determining whether a building roof is put into shadow by neighbouring objects, is a computational-intensive process. Together with the fact that the 3D BAG data set is a large vector data set, a trade-off between computational efficiency and accurate solar potential values is inevitable.

The scope of this thesis is as follows.

- The solar potential will only be computed for roof surfaces as these are in general more suitable for solar panel installation than wall surfaces;
- Only the direct component of the solar radiation will be computed as this component accounts for the largest percentage of solar radiation emitted. Diffuse and reflected radiation are out of scope for this research;
- To determine whether a building roof is put into shadow by neighbouring objects, only buildings from within the 3D BAG data set are taken into consideration as these neighbouring objects. Trees and other street furniture are out of scope for this research;
- For the computation of the solar radiation on roof surfaces, an existing Python library is used;
- Only a couple of tiles from the 3D BAG data set are used in experiments to assess the quality and scalability of the implemented system. Running the system for the whole of The Netherlands is out of scope for this research.

As limited by the scope of this research, the solar potential values computed for the building roofs in the 3D BAG data set serve as an indication to determine which building roofs have the highest potential to install solar panels. As the solar radiation is based on the direct component only, the determination of shadows is an important factor influencing solar radiation. To actually install solar panels on these roofs, the computation of diffuse radiation, further site investigation and power yield processing is necessary. Solar energy needs to be converted to other forms to make it suitable for household appliances. This conversion is also out of scope for this research.

1.3. Reading guide

The report is structured as follows. Chapter 2 explains the most import theory as a background for the rest of the report. Chapter 3 gives an overview of the relevant related work in the field of solar radiation. Chapter 4 presents the methodology and argues its design choices. Chapter 5 lists the tools and data sets used in this research, and provides the implementation of the proposed methodology. Chapter 6 shows the general results and the results for the experiments on quality assessment and scalability assessment. Chapter 7 concludes the thesis by discussing the limitations of the research, answering the research questions and giving recommendations on future work.

2. Theoretical background

To get a better understanding of the theoretical background needed for this research, this chapter illustrates the most important concepts and techniques.

2.1. Solar radiation

Solar radiation is the general term referring to electromagnetic radiation emitted by the sun, simply known as sunlight. In the scientific field of solar radiation various terms and definitions for solar radiation exist. Duffie and Beckman [2013] list several useful definitions of which these are relevant for this research:
Solar irradiance is "the rate at which radiant energy is incident on a surface per unit area of surface", measured in unit W/m^2.
Solar irradiation is the solar irradiance per unit area on a surface, which is found by integrating solar irradiance over a specified time period, for instance an hour or a day, measured in unit J/m^2.

Nowadays, solar irradiation is also represented in watt-hour with unit Wh/m^2 instead of Joule with unit J/m^2. In theory $1W$ is equal to $1J/s$ [Stouch Lighting Staff, 2015]. This means that integrating the solar energy over one hour (3600 seconds) results in $1Wh = 3600J$. So, J/m^2 refers to solar irradiation per second, while Wh/m^2 refers to solar irradiation per hour.

Throughout this research, the various terms for solar radiation are used interchangeably. These are solar radiation, solar irradiation and solar potential. In general, they all refer to the same concept unless stated otherwise. In Chapter 3, existing solar radiation models that take the theory presented in this section into practice, are discussed.

2.1.1. Types

Solar radiation can be distinguished by several types. The important ones are:

1. Direct radiation
2. Diffuse radiation
3. Reflected radiation
4. Global radiation

The direct radiation, also called beam radiation, is the solar radiation received directly from the sun without being scattered by particles in the atmosphere. On the contrary, diffuse radiation is the solar radiation received indirectly from the sun as its direction is changed by particles in the atmosphere. Reflected radiation is the solar radiation received indirectly from the sun after it is being reflected by a surface such as a window or the ground. Global radiation is the sum of the direct, diffuse and reflected radiation. The various types of solar radiation are shown in Figure 2.1.

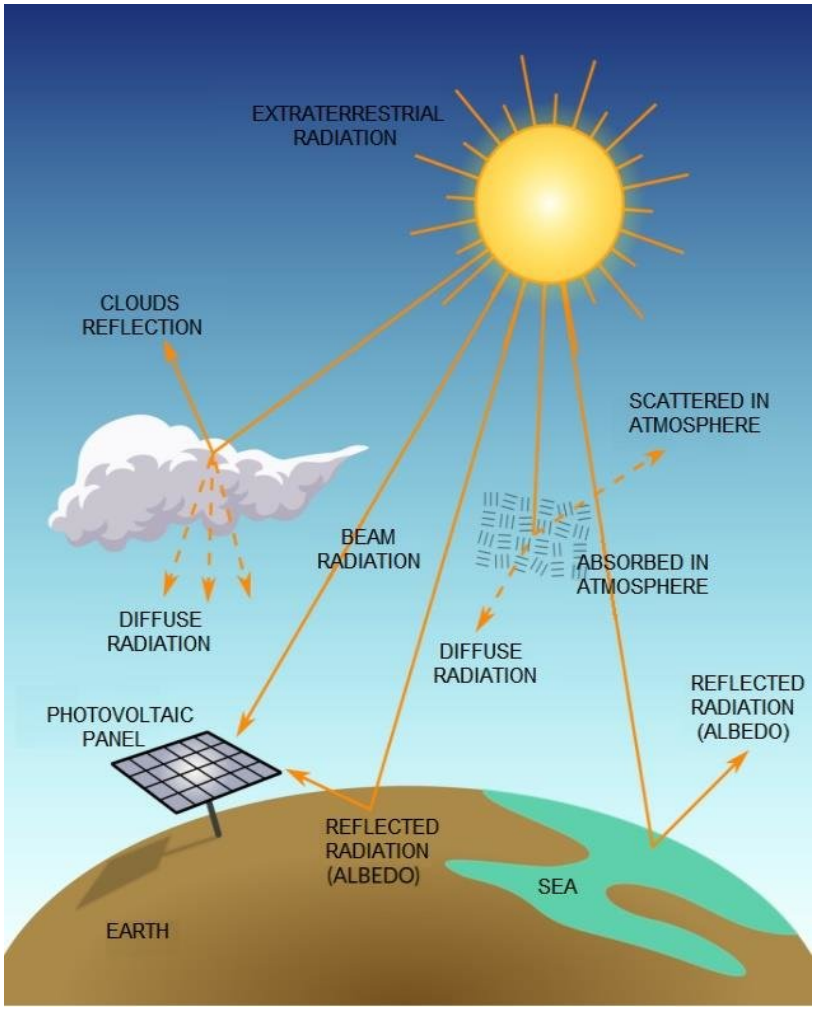


Figure 2.1.: Various types of solar radiation. Figure from Souza et al. [2019].

Under a standard clear sky (without clouds), direct radiation accounts for 77% of the global radiation and the other 23% is accounted by diffuse radiation. When the sky is overcast (only clouds), the global radiation is only diffuse [Spitters et al., 1986]. The proportion of reflected radiation to global radiation is so small that it can be neglected.

2.1.2. Factors influencing solar radiation

In order to compute the solar radiation for a whole region or a building roof, one needs to take into account several factors that influence the solar radiation [U.S. Department of Energy, 2022]. The important factors to consider are:

- Latitude;
- Date and time;
- Orientation of the surface;
- Potential obstructions (causing shadows);
- Height;
- Weather and atmospheric conditions.

As example to explain these factors, imagine a building with a sloped roof surface located in Delft such as shown in Figure 2.2. The latitude determines the north-south position of the roof surface on the earth. Latitude is specified by an angle ranging from 0° at the equator to 90° at the poles. In Delft, for instance, the latitude is approximately $52°N$. During the day, the position of the sun in the sky changes. This has as effect that the sun strikes the surface of the earth at different angles. At sun rise and sun set, when the sun is just above the horizon, this angle is 0° and during the day this angle might increase to maximal 90° depending on the latitude and time of the year. At 90° the sun is directly overhead striking vertical rays at the earth's surface, generating the most solar energy possible. At lower angles, the rays are more slanted meaning that they need to travel longer to hit a surface resulting in a lower generation of solar energy. For Delft, the angle at which the sun strikes the earth is higher on days in the summer than on days in the winter, meaning that solar energy is more effective in the summer.

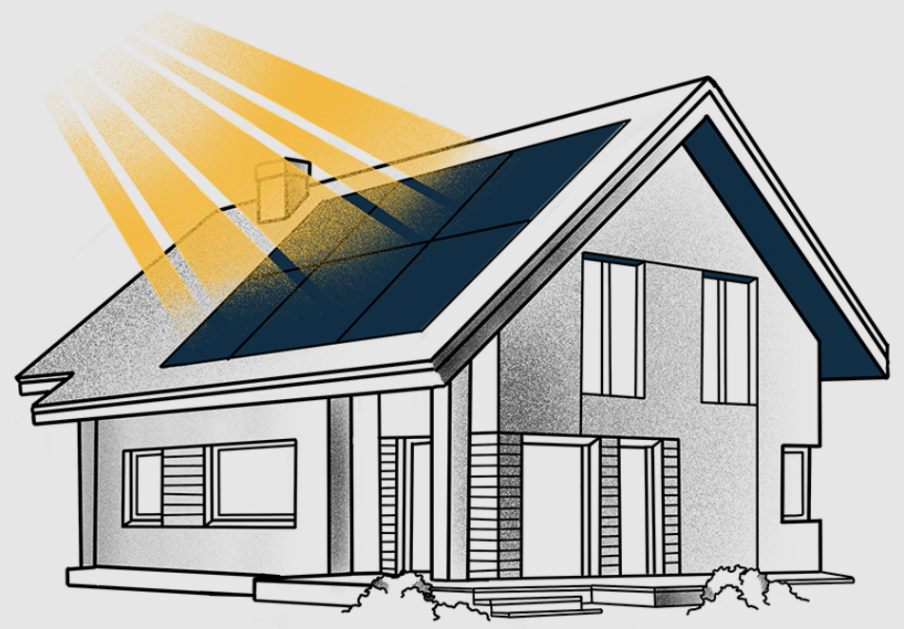


Figure 2.2.: Building with a sloped roof surface

The orientation of the roof surface is a very important factor influencing solar radiation. For locations in the northern hemisphere, the sun rises in the east and turns via the south to eventually set in the west. This means that surfaces oriented to the south receive more solar radiation than surfaces oriented to the north as the angle of incidence between the sun ray and the roof surface is higher for surfaces oriented to the north than for surfaces oriented to the south. The lower the angle of incidence, the more sunlight is hitting the roof surface as the projected area orthogonal to the incoming sunlight is larger [Rudisill, 2010]. When the roof surface is perpendicular (90°) to the sun ray, it receives the most solar radiation possible as the projected area orthogonal to the incoming sunlight is the largest possible. This is shown in Figure 2.3. Potential obstructions such as neighbouring buildings or trees might cast shadows on the building reducing the solar radiation.

Another factor determining the solar energy is the height value. The higher in the atmosphere, the thinner the atmosphere will be. This has as effect that less solar energy is absorbed higher in the atmosphere than near the surface of the earth. At building level, the height values does not significantly influence the solar potential for roof surfaces. Other atmospheric conditions affecting solar energy are the particles such as dust or rain droplets in clouds. These factors

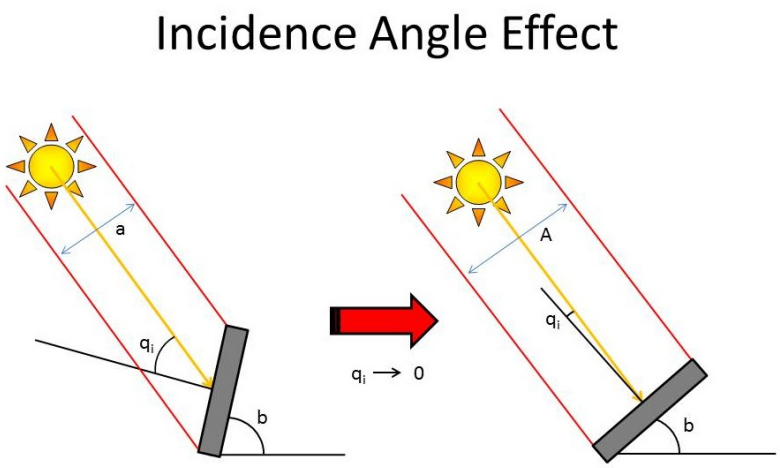


Figure 2.3.: Incidence Angle Effect where the incidence angle is denoted by q_i. Figure from Rudisill [2010].

reduce the solar energy generated by direct radiation, but increase the solar energy generated by diffuse radiation. In Figure 2.4 the so called 'Earth's Energy Budget' is shown, illustrating how much energy is being received by the Earth and how much energy is reflected or radiates back to space.

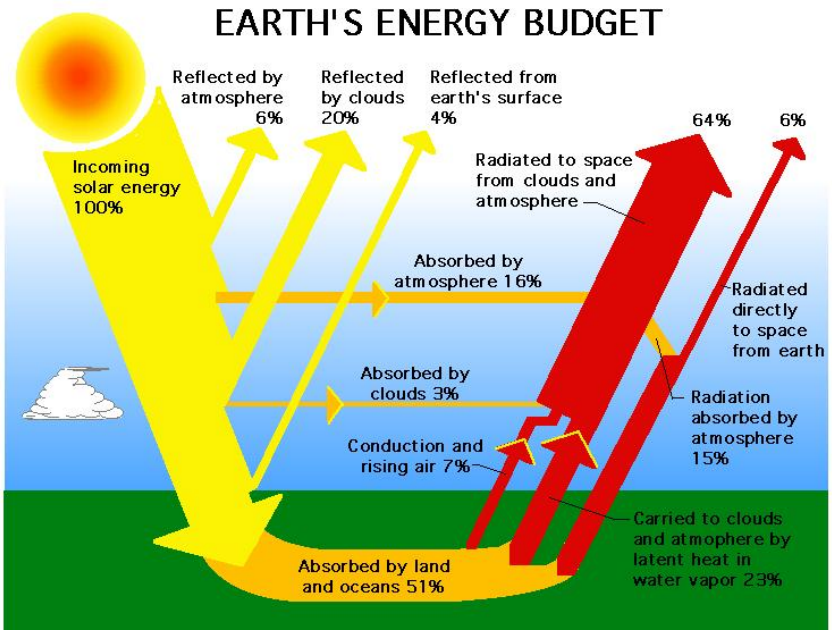


Figure 2.4.: Influence of the height in the atmosphere on the solar energy. Figure from NASA.

2.2. Spatial indexing

Spatial indexing is an optimisation technique by which a list of spatial elements, such as geometries, are stored in such a way that these can be requested efficiently by using spatial

queries. The performance gain is achieved by the fact that not the whole list needs to be traversed when looking for the requested geometry. Instead, the data can be directly accessed from memory. In most cases, a hierarchical tree structure is used to store spatially referenced bounding boxes containing a selection of geometries. When querying such a tree, first the bounding box containing the geometry is selected after which it is traversed to find the requested geometry. This means that the traversal of the other bounding boxes can be skipped, saving a lot of processing time. Van Oosterom [1999] explains the importance of spatial access methods which includes spatial indexing and discusses a few types.

Common spatial indexing structures are Bounding Volume Hierarchy (BVH), kd-tree and R-tree. For the R-tree several variants exist such as the Hilbert R-tree, R+ tree and R*-tree. The R-tree is the most popular structure for spatial indexing [Güting and Schneider, 2005]. In an R-tree, the spatial objects are wrapped in bounding boxes and are grouped in MBRs as shown in Figure 2.5. The root node of the tree contains MBRs A to C. A level lower, the nodes contain the bounding boxes of spatial objects within each MBR.

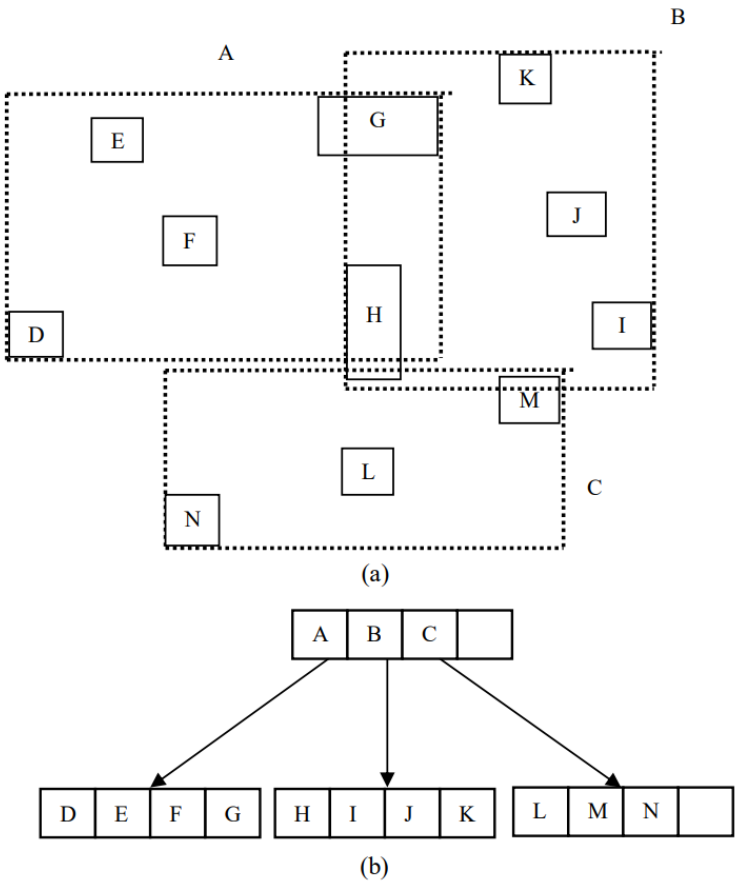


Figure 2.5.: Example of spatial data with their own bounding boxes (D - M) and grouped MBRs (A - C) in (a). The corresponding R-tree is shown in (b). Figure from Mon and Than [2015]

R+ tree is a variant on the R-tree. In order to convert the example in Figure 2.5 to an R+ tree, the nodes of the MBRs would store references to all the bounding boxes contained within the MBR. This would mean that a reference to H would also occur in MBR A and C, and a reference to G and M would also occur in MBR B. This allows more efficient querying at the cost of storage memory as duplicate bounding boxes might exist between nodes. The R*-tree variant uses a different insert algorithm with as goal to minimise the the amount of duplicates between the nodes to make the nodes as square as possible [Van Oosterom, 1999].

2.3. Multiprocessing

Multiprocessing refers to the usage of two or multiple Central Processing Units (CPUs) on a single computer. This is beneficial for processing times as the processing operations are spread across the CPUs. There exist various ways to implement multiprocessing. The two most important types are multiprocessing and multithreading. With multiprocessing multiple processors are run concurrently, while for multithreading multiple threads on the same processor are run concurrently [Wong, 2022]. The difference is shown in Figure 2.6.

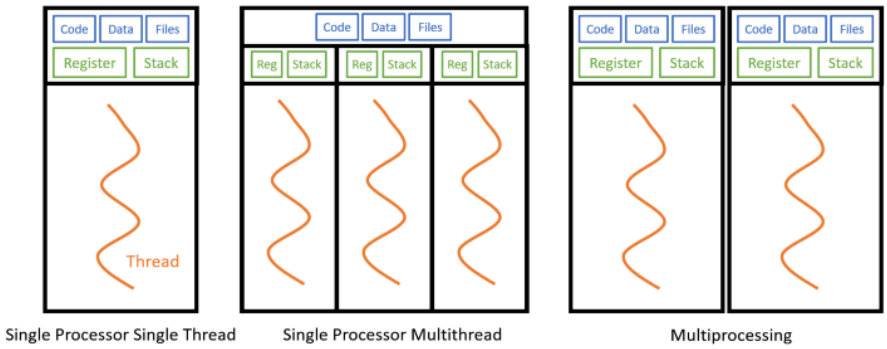


Figure 2.6.: Multiprocessing versus multithreading. Figure from Wong [2022]

2.4. Ray tracing & intersection detection

In computer graphics, ray tracing is the technique used to render 3D objects in scenes while taking lightning, surface types, orientation and reflections into consideration. Intersection detection is part of ray tracing, where it is determined whether a ray cast from a source to an object is intersected by another object in the scene. A technique to do this is ray-box intersection [Majercik et al., 2018], shown in Figure 2.7. In this case, a ray is cast from a position on the building (the origin 'or') to the sun as light source (the destination 'D'). In between, the ray intersects another object. The goal of intersection detection in this research is to find out whether buildings are put into shadow by neighbouring objects.

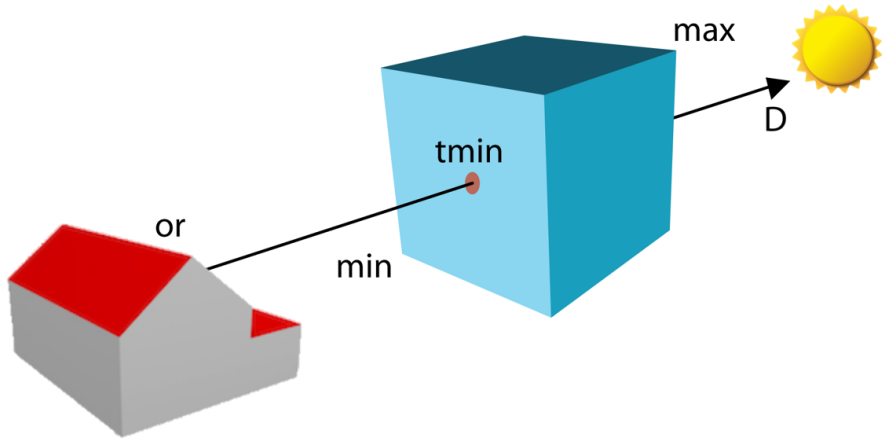


Figure 2.7.: Ray-box intersection

2.5. Storage of 3D city models

3D city models could be stored in various formats. They can be stored in CityGML and CityJSON format, but they can also be stored and processed in a geospatial DBMS, such as 3DCityDB.

CityGML is an exchange format to store 3D city models. It is also a data model and it is standardised by the Open Geospatial Consortium (OGC) [OGC, 2012]. It covers common 3D objects found in cities, such as buildings, roads, vegetation and bridges. As stated by Ledoux et al. [2019], CityGML is not user-friendly as there are no easy ways to read and write CityGML files. Therefore, CityJSON was developed.

CityJSON is presented as "a compact and easy-to-use encoding of the CityGML data model" [Ledoux et al., 2019]. In the CityJSON format, city models containing building models can be efficiently stored. Extracting data from this format is easy as the contents are stored as JavaScript Object Notation (JSON) objects instead of hierarchical Extensible Markup Language (XML) encodings. Furthermore, CityJSON files are 6x more compact than CityGML files and they support semantic surfaces.

Another encoding of CityGML is 3DCityDB. It is a free spatial database solution in which 3D city models based on CityGML can be managed, analysed and visualised [Yao et al., 2018]. It is seen as an alternative to GISS when used for large data sets.

3. Related work

When presenting the existing methods for computing the solar irradiation of a building roof, one can distinguish the methods on its input data type, namely raster and vector data. These data types both need a different approach when computing solar irradiation. Figure 3.1 shows a rasterisation of a vector polygon. It clearly shows that a smaller cell size results in a higher resolution raster and a larger cell size results in a lower resolution raster. A high resolution raster has more spatial detail, but takes more time to process than a low resolution raster. Vector data does not have a resolution, but models the polygon with geometric primitives such as lines and points without loss of spatial information.

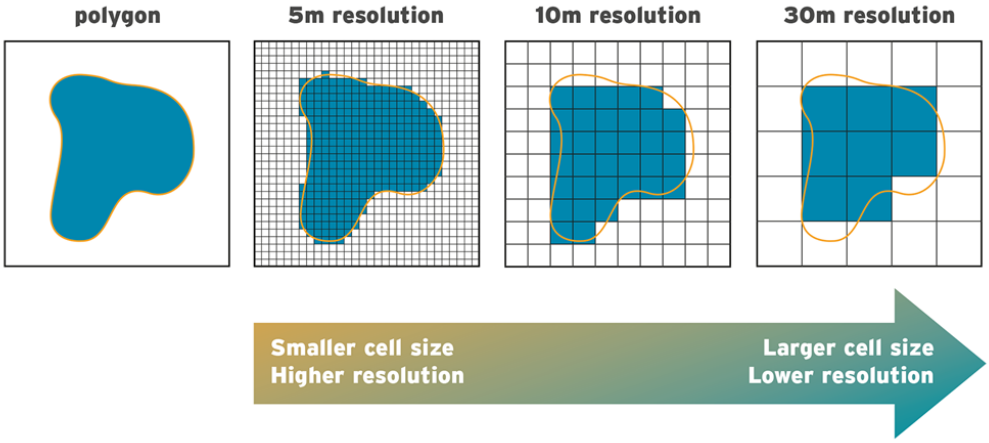


Figure 3.1.: Relation of cell size to resolution. Figure from [Alexandre, 2018]

Table 3.1 gives an overview of some of the existing tools and methods for solar irradiation modelling addressed for this research. For each tool or method it states the publication year, the type of input data for the solar irradiation method, a key feature and the citation. The entries are first ordered on type (raster, vector, other) and then ordered on year. Freitas et al. [2015] also performed a state-of-the-art review of available solar radiation models.

3.1. Raster methods

One of the first solar irradiation models was SolarFlux [Hetrick et al., 1993]. This method used simple formulas and was unfortunately not suitable for large area raster maps.

A first approach in implementing solar irradiation models with faster speeds, higher accuracy and open source availability was done by Fu and Rich [2002]. They created the tool called Solar Analyst, which is still implemented, although improved, within the ArcGIS software package. Their method takes the topography, elevation and surface orientation of a DEM into account. Obstructions by surrounding topographic features, like hills and trees are also considered, by using a hemispherical viewshed algorithm.

Tool/Method	Year	Type	Key feature(s)	References
SolarFlux	1993	Raster	One of the first tools	Hetrick et al. [1993]
Solar Analyst	2000	Raster	Widely used, reliable	Fu and Rich [2002]
GRASS GIS *r.sun*	2002	Raster	Widely used, reliable	Hofierka and Ri [2002]; Hofierka and Suri [2007]
Solar Potential on multiple building rooftops	2013	Raster	Uses Solar Analyst. Combines high-resolution DEM with upward-looking hemispherical viewshed	Kodysh et al. [2013]
Solar Potential Remote Sensed Rooftops	2018	Raster	Takes variable rooftops into account	Song et al. [2018]
GPU-Enabled Shadow Casting (Geneva)	2020	Raster	Computationally efficient for high resolution, but shadow casting for rasters	Stendardo et al. [2020]
v.sun	2012	Vector to voxel	Vector-voxel approach, lacks benchmarking	Hofierka and Zlocha [2012]
SORAM	2013	Vector	Focused on diffuse radiation	Erdélyi et al. [2014]
Solar3DCity	2015	Vector	About error propagation, takes CityGML as input but does not consider shadows	Biljecki et al. [2015]
Solar radiation on CityGML building data	2015	Vector	Takes CityGML as input, does consider shadows. Uses point sampling on the surfaces	Wieland et al. [2015]
SURFSUN3D	2015	Vector to raster	Rasterises 3D triangular meshes, but not fast enough for higher LoD	Liang et al. [2014, 2015]
Sparse Voxel Octree (SVO)	2017	Vector to voxel	Extension of SURFSUN3D, sparsely voxelises the 3D triangular meshes in an octree	Liang and Gong [2017]
Skyline-based method	2019	Vector	Computes SVF and SCF. Also uses the GPU to speed up	Calcabrini et al. [2019]
3D shadow cast vector-based model	2020	Vector	Accurate; different projection techniques used based on the tilt of the surfaces	Viana-Fons et al. [2020]
3D Solar Photovoltaic Potential Mapping in Urban Environment	2021	Vector	Follow-up on Viana-Fons et al. [2020] by applying the skyline-based method on a use case	Zhou et al. [2021]
Vertical 3D walls	2016	Raster to 3D points	Uses observer point columns to efficiently ray cast 3D vertical walls	Jaugsch and Löwner [2016]
Solar3D	2020	OAP3D	Very accurate, it uses the cube mapping technique on 3D photographs of buildings	Liang et al. [2020]

Table 3.1.: Overview of various solar radiation modelling tools or methods

The resulting insolation maps include both direct and diffuse radiation. Their resolution back then was 30 meter, which is quite low for building roofs but sufficient for large geographic areas. This work is a good starting point for solar irradiation studies and is therefore cited a lot by other authors as well.

A more recent methodology utilising the Solar Analyst tool as described by Kodysh et al. [2013] also makes use of a hemispherical viewshed. It is raster-based and has a resolution of less than 1 meter, which is necessary to capture all features of a roof. A similar tool as the Solar Analyst was implemented for GRASS GIS, called *r.sun* [Hofierka and Ri, 2002; Hofierka and Suri, 2007]. It is also raster-based and widely used nowadays with reliable accuracy.

The way these tools work to compute the solar irradiation for specific buildings is to combine the raster data set with a 2D vector data set of the building footprints. The building footprints are then used as a mask laid over the pixels in the raster data set. Only these pixels are taken into consideration when computing solar irradiation values for the building roofs. As one can imagine, the higher the resolution of the raster grid, the better the geometry of the building roofs can be represented.

The work by Song et al. [2018] focuses more on the various shapes of the rooftops. It states that the solar potential differs per rooftop type and it selects the most suitable roof types to use for solar irradiation computations based on some filters. Stendardo et al. [2020] incorporate the Graphics Processing Unit (GPU) to speed up the shadow casting algorithm.

3.2. Vector methods

Vector data is more accurate than raster data as it is built up of geometric primitives such as polygons and lines, representing the 3D environment with more spatial detail. This makes modelling of for instance vertical facades easier. Moreover, vector data is more efficient to store in computer memory than raster data. However, raster data is simpler and faster to use as the resolution can be determined beforehand. Therefore, the challenge is to find a way to process the vector data computationally efficient to incorporate the higher spatial detail into the resulting solar potential of the individual 3D buildings.

An early approach in using 3D vector data in complex urban environments as input for solar irradiation computations is described by Hofierka and Zlocha [2012]. The module *v.sun* is created, which is based on the existing module *r.sun* to compute solar irradiation values. It takes a vector-voxel approach to segment the vector objects into smaller polygons. Then, for each voxel the proposed algorithm is applied: calculating the normal vector, solar ray directions for various time and the shadows cast by neighbouring buildings. However, the performance of the method is not assessed and it is not implemented in the GRASS GIS package as was proposed to be.

A tool taking 3D city models stored in CityGML files as input for solar irradiation computations is Solar3DCity [Biljecki et al., 2015]. The tool extracts the roof surfaces for each building and computes their inclination, area and orientation. Afterwards, it estimates the yearly solar irradiation for each surface by using the solpy library [Charles, 2015] and writes the enriched 3D city model back to the CityGML file. A notable shortcoming of this tool is that it lacks an implementation for shadows cast by neighbouring buildings. This is omitted because of computational reasons.

The study described by [Wieland et al., 2015] also takes 3D city models stored in CityGML format as input for solar irradiation computations. The improvement is that it does incorporate

the determination of shadows. Their workflow is as follows. For each roof and wall surface of a building they establish a point grid as shown in Figure 3.2. Then for each point they cast a line towards the direction of the sun for specific points in time. By using an intersection tool in PostGIS they check whether this line intersects with surrounding buildings to determine whether the corresponding point is in shadow or not. Afterwards, the solar irradiance (beam and diffuse) is computed for each surface point. Lastly, the values are integrated over time and area to arrive at monthly values per building. This is a simple and effective way to compute the shadows cast by surrounding buildings. However, it is quite inefficient because of the high computation costs. This can be explained by the inefficient way the buildings are processed; just brute force.

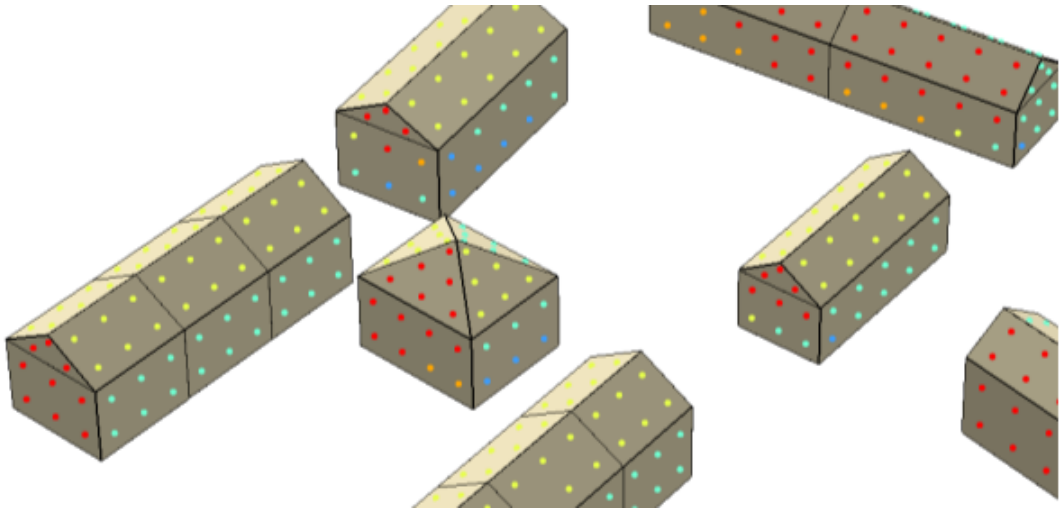


Figure 3.2.: Point sampled building surfaces. Figure from Wieland et al. [2015].

The work by Liang and Gong [2017] also takes a 3D vector-voxel approach to compute solar irradiation like Hofierka and Zlocha [2012], but this time in combination with a SVO to store the geometries. The strong point of using an SVO as storage model is that the computation time of shadow casting does not slow down for more complex geometries. Its time complexity stays the same with increasing LoD. Its predecessor is about just rasterising the 3D meshes in 2D [Liang et al., 2014], which does slow down shadow casting with increasing LoD. However, storing a 3D city model into an SVO [Liang and Gong, 2017] is memory inefficient, making it unusable for large 3D city models. Therefore, a better storage model than the SVO is needed as indexing structure to speed up obstruction detection for shadow casting.

The work described by Calcabrini et al. [2019] uses the indicators Sky View Factor (SVF) and Sun Coverage Factor (SCF) to represent the skyline profile for a building in a 3D city model. This skyline-based method is implemented for direct, diffuse and reflected solar irradiation making the model quite complex, but accurate. It also makes use of the GPU to speed up computations for large areas.

A follow-up of this work is presented by Zhou et al. [2021] which makes use of the skyline-based method to compute annual solar irradiation, direct current (DC) yield and alternating current (AC) yield maps. The model provides a quick and accurate assessment of solar potential on a large-scale and complex urban environment. As a use case the campus of TU Delft is chosen.

Viana-Fons et al. [2020] describe an efficient 3D shadow cast vector-based model. They create a 3D city model out of Light Detection And Ranging (LiDAR) data and building footprints. Then they apply their shadow model to the surfaces of a building, distinguishing whether a surface

is vertical or not. This is necessary as projection techniques are used for different surfaces to obtain a correct shadow profile. Their resulting values have low errors compared to real data, yielding a high accuracy.

3.3. Other methods

The tool Solar3D is mainly focused on using a cube mapping technique to accurately recreate 3D scenes based on Oblique Airborne Photogrammetry-based 3D city models (OAP3Ds) [Liang et al., 2020]. For this, real life photographs of the scene are needed, which is unfeasible for large data sets. For this to work, the tool also relies on DEMs, Digital Surface Models (DSMs) and feature layers as input data. The tool yields a high accuracy.

To enlarge solar energy generation, solar panels can also be installed on the facades of buildings. Jaugsch and Löwner [2016] implemented a method that takes 2D maps with additional height information as input. 3D block models are extruded from this input. They introduced the concept of observer point columns which determines whether a whole vertical part of the facade is illuminated or not. To speed up shadow computations they take the maximum shade length determined by the tallest building. As the building blocks are extruded from 2D data resulting in 3D mesh data, this method is considered as a hybrid approach between raster, points and vector to model the solar potential.

3.4. Simplifications and acceleration structures

Further methods to speed up computation times are simplifications in the solar irradiation models and to apply acceleration structures. A possible simplification is the usage of the characteristic declination of the sun [Brito et al., 2021]. This is the declination for the day on which the daily solar irradiation on a horizontal surface is identical to its monthly average value. This value, instead of daily values, can be used to reduce computing demand by a factor of 30. But, accuracy errors are introduced ranging from +5% to +12% depending on the latitude. The higher the latitude, the higher the errors. This should be kept in mind when applying this simplification to the solar irradiation model.

Another way to achieve better computation times is the usage of acceleration structures, indexing the buildings in the 3D city model. This is beneficial when performing shadow casting as obstructing neighbouring buildings can be efficiently found based on the index. The SVO as used by Liang and Gong [2017] is an example of such an acceleration structure. However, the SVO and voxelisation in itself raises some limitations and deficiencies as it is memory inefficient to voxelise a massive 3D city model. The memory usage increases proportionally with the size of the 3D city model. Another downside is that data management is challenging, because the 3D city model loses its spatial relations and semantics after the voxelisation.

Therefore, another acceleration structure is necessary for massive 3D city models. The BVH is a tree structure to spatially organise data by wrapping the geometries into bounding boxes [Wald et al., 2007]. Spatial relations and semantics of the geometries are not lost and it is also an efficient way to detect intersections of neighbouring objects when doing ray tracing. The time complexity of BVH is $O(log(n))$. This is achieved by the fact that geometries stored in children nodes do not have to be checked for intersections when the bounding box of the parent node is not intersected by the ray. Moreover, the BVH is already used in many ray tracing algorithms.

The two acceleration structures described can be distinguished by their purpose in two classes: space subdivision methods and object subdivision methods. SVO is a space subdivision method while BVH is an object subdivision method. Space subdivision methods subdivide the 3D space into smaller regions by using planes, where the geometry might belong to multiple regions. Object subdivision methods subdivide the geometry into smaller objects, resulting in tight volumes wrapped around the object [Pharr et al., 2004 - 2021]. The two method classes are both successful at speeding up the ray intersection method, hence SVO and BVH are both suitable for shadow casting.

4. Methodology

This chapter represents the methodology used to fulfil the research objectives listed in Section 1.1. First, an overview of the methodology is given and then the methodology is explained step-by-step in more detail.

4.1. Overview

In short, the methodology can be described as follows. First, building geometries stored in one or multiple CityJSON files from the 3D BAG data set, will be loaded in memory. Then, a 3D R-tree is created in which the buildings needed to process one file are stored for fast access. Afterwards, each building will be processed. First, the neighbouring buildings potentially casting a shadow on the current buildings will be found by querying the R-tree. Afterwards, each building in the file is processed in parallel on multiple CPUs where each triangular roof surface will be sampled into a grid of points. For each point in the grid, the shadowing is determined by intersection detection and the yearly beam solar irradiation is computed accordingly. The solar radiation values are then aggregated to surface level and statistics such as average, standard deviation and several percentiles are computed. Lastly, the 3D city model of each file is written back to CityJSON, enriched with the different solar irradiation values as attribute per triangular roof surface. This whole workflow for processing one CityJSON file is shown in Figure 4.1.

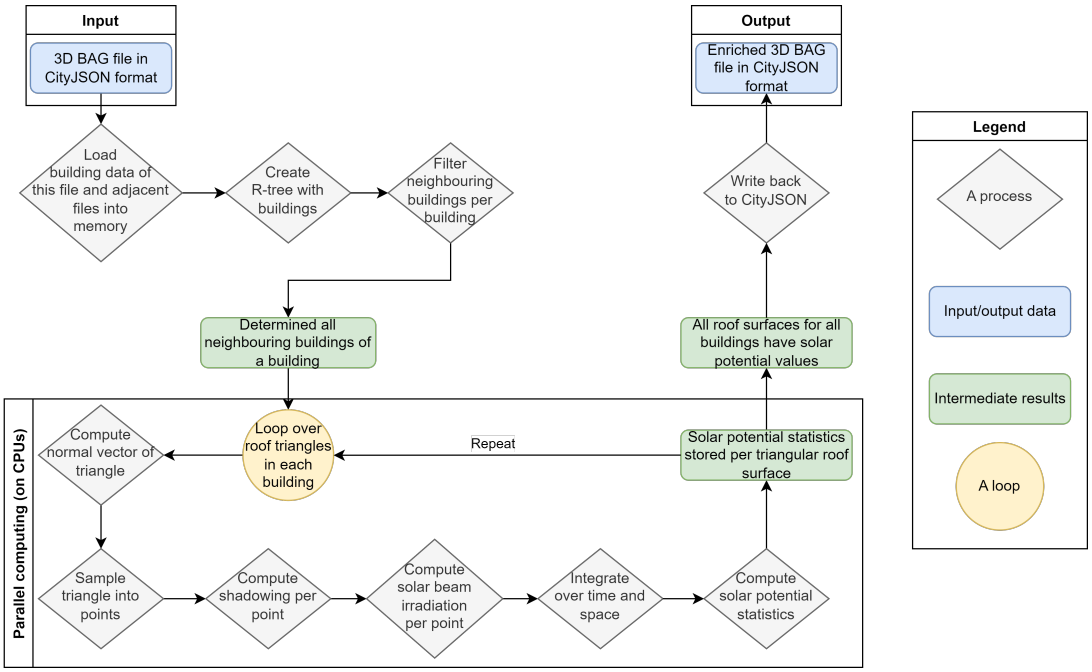


Figure 4.1.: Methodology workflow for one CityJSON file.

4.2. Step-by-step explanation

The proposed methodology will be discussed and justified in more detail below. Its structure can be roughly divided into three parts: preparing input data, processing the building geometries and exporting output data. For some parts, more detailed information on reasons and choices can be found in Chapter 5.

4.2.1. Input data preparation and loading

The first part of the methodology is to prepare the input data. The input data are building geometries stored in one or more CityJSON files. CityJSON is chosen as input data format because it is easy-to-use and it stores city objects such as buildings in a compact way. The input files are taken from the 3D BAG data set. This data set is explained in more detail in Section 5.1.4.

For this methodology, the geometric primitives of the buildings need to be triangles. Triangles are convenient primitives as they are always planar, meaning that all corners of a triangle lie on one plane. This makes triangles suitable to model complex 3D models. In the case that the buildings in the city model are not triangulated, they need to be triangulated beforehand. The triangulation might yield triangles that are not usable. At this point in the process, the triangles are not further inspected on inconsistencies. This is done later on in the process.

As this research is focused on large data sets containing large 3D city models, it needs to be taken into account that it is not possible to store all the data directly into memory. The computer memory of a normal PC will not be sufficient enough to store large amounts of data. Therefore, the CityJSON files are processed one at a time. So, only the building geometries that are needed to process one CityJSON file are loaded into memory. To determine the solar radiation of a roof surface, it is important to know whether the building roof is put in shadow by another building. For buildings at the borders of the CityJSON file, a building from the adjacent file could cast that shadow. Therefore, to process one CityJSON file, the buildings that could potentially cast a shadow on buildings in the current CityJSON file need to be loaded into memory as well. These buildings are chosen by applying a buffer. More details on the selection of all building geometries needed to process a CityJSON file is written in 5.2.2.

Then, the selected building geometries are inserted into a 3D R-tree. An R-tree is used because it is a spatial indexing structure that allows fast retrieval of spatial data when requested. This R-tree will be used in a later step to retrieve the neighbouring building geometries that potentially put another building into shadow.

4.2.2. Processing building geometries

When all the building geometries necessary to process a CityJSON file are stored within memory and the accompanying R-tree is created, each building geometry in a CityJSON file is individually processed in order to compute its solar potential value. First, the neighbouring buildings that might cast a shadow on a building geometry are filtered on their distance, height and orientation with respect to the building geometry in order to reduce processing time. More information on how these filters work and how their thresholds are chosen is found in Section 5.2.3. To retrieve these neighbouring buildings, the R-tree is queried before multiprocessing. The reason for this is explained in more detail in Section 5.2.3.

As the solar potential of one building is not influenced by the solar potential of another building, but only by its existence in the neighbourhood, each building is further processed in parallel on multiple CPUs. The benefit of performing parallel computing is that it reduces the computation time for a whole city model by a factor equal to the number of cores used for parallel computing.

Then, there are a couple of steps to go through to compute the yearly solar potential for each triangular building roof surface, based on the work by Wieland et al. [2015]. For each building their triangular roof surfaces are extracted and filtered on undesirable thin and elongated triangles. For each triangle its normal vector is computed and it is sampled into a uniformly distributed grid of points. The normal vector is needed as a parameter for the solar radiation computation and sampling is necessary to account for variable solar irradiation values on a triangle, especially on larger triangles. Some parts of the triangle may be blocked by a shadow from a neighbouring object, while other parts are not. An example of sampled roof surfaces is shown in Figure 4.2.

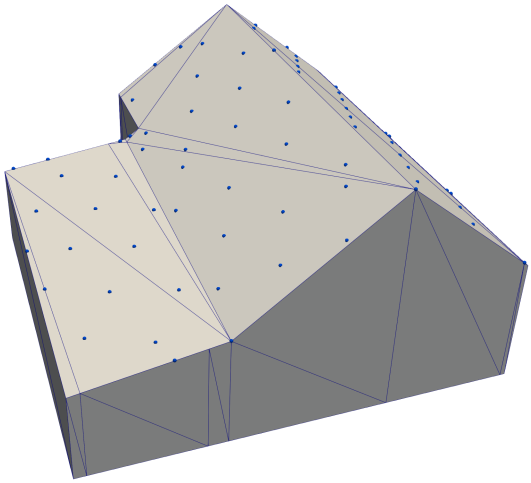


Figure 4.2.: Point sampled building surfaces.

For each point on a surface it is determined whether it is illuminated or not, based on the shadows cast by the filtered neighbouring buildings. Rays originating at each point on the surface are cast in the direction of the sun at a specified time. If an intersection with a neighbouring building is encountered, the grid point at the origin of the ray is considered to be in shadow for this specific timestamp. Otherwise, the grid point is considered to be illuminated.

Now, that it is determined when each grid point is illuminated or in shadow, the yearly solar potential can be computed. To speed up computation time, the shadow casting and solar potential are computed at time intervals of 1 hour, for 12 days a year. Each day is the 21st day of a month, meaning an average solar potential is computed per month. More explanation on these numbers is given in Section 5.2.3. The solar potential per month is then aggregated to arrive at a yearly solar potential value. This simplification will drastically improve processing times as the solar potential values are not computed for each day of the year individually. However, the downside is that the solar potential values might be less accurate.

To compute the solar potential for each illuminated grid point for a specified time, a couple of parameters related to the building roof geometry are needed. These are the normal vector of the surface the point belongs to, the height of the point, the position of the point, represented by the latitude, and the timestamp. The direct solar radiation for each grid point based on these

parameters is computed. As limited by the scope of this research (Section 1.2), the diffuse and reflected radiation is not considered.

The solar radiation model described by Duffie and Beckman [2013] is used to compute the direct solar radiation under clear-sky conditions. In their literature it is called clear-sky beam normal radiation, given in equation 4.1, where G_{on} refers to the extraterrestrial radiation defined by equation 4.2 and τ_b refers to the atmospheric transmittance for beam radiation given in equation 4.4.

$$G_{cnb} = G_{on}\tau_b \tag{4.1}$$

The extraterrestrial radiation on the nth day of the year is defined by

$$G_{on} = G_{SC}(1.000110 + 0.034221cosB + 0.001280sinB + 0.000719cos2B + 0.000077sin2B) \tag{4.2}$$

where G_{SC} refers to the solar constant of $1367W/m^2$ and B is defined by

$$B = (n-1)\frac{360}{365} \tag{4.3}$$

The atmospheric transmittance for beam radiation is defined by

$$\tau_b = a_0 + a_1 exp(\frac{-k}{cos\theta_z}) \tag{4.4}$$

where a_0, a_1 and k are constants determining the atmospheric transmittance for the standard atmosphere with 23 km visibility. The constants are dependant on the height for which the beam radiation is computed with a maximum height of 2.5km. The parameter θ_z is the zenith angle. This is the angle between the zenith and the sun ray on a roof as shown in Figure 4.3 which is derived from the timestamp (date) and the latitude as this angle changes throughout the day and is different per latitude.

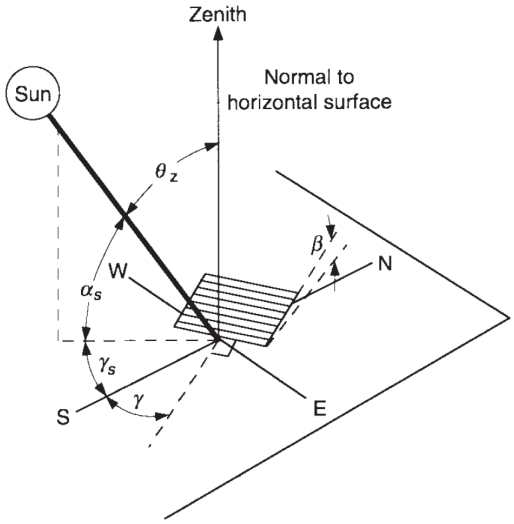


Figure 4.3.: A tilted surface with a couple of parameters, such as the zenith angle θ_z. Figure from Duffie and Beckman [2013].

As the direct radiation as defined by equation 4.1 is computed as if the surface is perpendicular to the sun, the equation needs to be adapted to equation 4.5 in order to account for tilted building roofs.

$$G_{cnb} = G_{on}\tau_b cos\theta \tag{4.5}$$

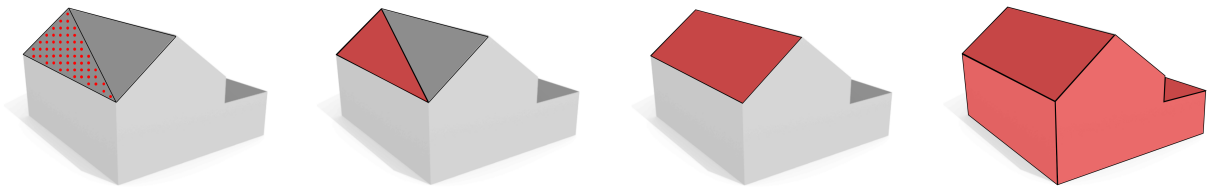
where θ refers to the angle of incidence between the sun ray on a roof surface and the normal vector of that roof surface. Just like the zenith angle, the angle of incidence is also different throughout the day and per latitude. The effect of the angle of incidence is shown in Figure 2.3.

4.2.3. Exporting enriched output data

In the last part of the process, the solar potential of the buildings need to be exported to CityJSON. From the solar potential at surface and building level some statistics can be deduced. These are average, minimum, maximum, standard deviation, 50th percentile and 70th percentile.

The various statistics derived from the solar potential of a grid of points can be written to CityJSON at different levels as shown in Figure 4.4. The levels that are considered including their pros and cons are described below:

1. *Per point*. This is an accurate representation of the computed values per point. The downside is that it is inefficient to store, because of the many values per triangle and building. For this choice, no aggregation over space is needed.
2. *Per triangular roof surface*. This is a less accurate representation than per point, but simpler to store in a CityJSON file. For this choice, the statistics described above need to be computed over all the points in the triangle.
3. *Per polygonal roof surface*. This is a less accurate representation than per triangular surface. The triangular surfaces for which the solar potential is computed need to be converted (by detriangulation) to polygonal surfaces and the statistics computed accordingly. This is the original way the buildings in the 3D BAG data set are represented.
4. *Per building*. This is the least accurate way to represent the computed values, as it is not known which part of the building surfaces has the highest solar potential. It is a too simplified representation, but the building geometries are very easy to extend with an extra attribute. For this choice, the solar potential statistics over all the roof surfaces in the building need to be computed.



(a) Per sampled grid point. (b) Per triangular roof surface. (c) Per polygonal roof surface. (d) Per building as a whole.

Figure 4.4.: Visualisation of the 4 different ways to store solar potential of a building geometry.

For the use case of 3D BAG, it is chosen to store the solar potential statistics *per triangle*. The values are stored as extra attributes to triangular roof surfaces and written as such to CityJSON. This is the easiest way to enrich the roof surfaces in the city model with solar potential values as the geometric primitives are already triangles. The benefit is that no extra point geometries need to be created that take up more memory. Also no detriangulation needs to be done or additional aggregation to building level needs to be performed. However, one needs to take into account that the actual solar potential value per location on the roof surface might be different.

4.3. Quality Assessment

In order to prove that the implemented vector-based solar radiation model is correct in terms of accurate solar radiation values, quality assessment needs to be performed. Comparing the results to ground truth data says something about the reliability and quality of the implemented model. However, ground truth data of solar radiation is not widely available at each roof surface. But, weather stations throughout The Netherlands do exist which can be used for ground truth comparison at an exact location.

Another way to perform quality assessment is to compare the implemented solar radiation model to another existing solar radiation model. In this way, one is not restricted to a comparison of one specific location only, but a comparison of a whole city model incorporating shadows can be performed. Both ways to perform quality assessment will be executed and their approach and results will be discussed in Section 6.2.

4.4. Scalability Assessment

In addition to quality assessment, the implemented solar radiation model is assessed on its scalability. This means that the system should not terminate during processing due to a lack of available memory. The input of the solar radiation model is a folder with a number of tiles stored in CityJSON files. For the scalability assessment, there will be experiments with increasing number of tiles given as input to the model to prove that the system is scalable. The experiments themselves and their results will be discussed in Section 6.3.

5. Implementation

This chapter explains the implementation of the methodology. First, the tools and data used for the implementation are discussed. Then, each step of the software implementation itself is discussed in depth.

5.1. Tools and Data

5.1.1. Hardware

The specifications of the computer used for the implementation of the methodology and the execution of the experiments in Chapter 6 is as follows.

- Processor: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
- RAM: 16.0 GB
- GPU: Intel(R) UHD Graphics 630 and NVIDIA Quadro P1000
- Operating System: Windows 10 Home x64

Besides this, it is important to note that the developed implementation is run on Windows Subsystem for Linux (WSL)[1]. Directly running it on a Windows machine is not possible, but running it on a Linux machine is possible.

5.1.2. Python

The programming language used to implement the methodology proposed in Chapter 4 is Python[2]. Python is an open source and user friendly programming language that is very suitable for data science. It has a large community contributing to the knowledgebase of Python. Many additional libraries and packages for numerous fields are freely available. The libraries used in this research are listed below:

- NumPy[3] is a library used for storing data in arrays and matrices. It is also used for simple mathematical computations.
- PyVista[4] is an Application Programming Interface (API) for the Visualisation Toolkit (VTK)[5]. It is used to wrap the data in its native PolyData datastructures and perform some computations. The library is also used to visualise intermediate results.

[1] https://learn.microsoft.com/en-us/windows/wsl/
[2] https://www.python.org/
[3] https://numpy.org/
[4] https://docs.pyvista.org/
[5] https://vtk.org/

- solarpy[6] is a package that provides a reliable solar radiation model, based on the work by [Duffie and Beckman, 2013]. It is able to compute the solar beam irradiance on any plane, any place on earth and any day in the year. In this research this package is used as core solar radiation model to compute the direct radiation for each roof surface.
- cjio[7] stands for CityJSON input/output. This library is used to read, write and manipulate CityJSON files.
- Rtree[8] is a library that provides spatial indexing for Python. In this research it is used to store 3D BAG tiles and building geometries in R-trees and provide fast access to them when respectively performing neighbouring tile selection and intersection detection.
- concurrent.futures[9] is a module that provides a high-level interface for asynchronously executing callables. In this research it is used to process building geometries in parallel.

5.1.3. GIS and visualisation programs

In this research the GISs QGIS and ArcGIS are used. QGIS was initially used to load and inspect the CityJSON files that were used as input. ArcGIS is used to compute the solar radiation for a raster and to compare the results with the implemented method. This comparison is presented in Section 6.2.2.

Programs used for visualisation of (intermediate) results are ParaView and Ninja. ParaView[10] is an open-source, multi-platform data analysis and visualisation application. Users can interactively explore data in 3D. In this research, ParaView is used to visualise intermediate results stored in PyVista's PolyData format that are written to a .vtm or .vtk file. In this research, a seperate script is written that reads a CityJSON file and outputs a .vtk file where each triangular roof surfaces contains an attribute with the average solar potential value.

Ninja[11] is a web-based visualisation application to visualise city models stored in CityJSON files. It can show the city models at each LoD available, distinguish between their semantic surface and list the attributes at surface and building level. In this research it is used to check whether the resulting city model enriched with the solar potential is correct and to explore the results. In addition, the software cjval[12] is used to validate CityJSON files against the CityJSON schemas.

5.1.4. 3D BAG data set

3D BAG[13] is an open data set covering all the buildings in The Netherlands as 3D building meshes [Peters et al., 2022]. The data set is fully automatically generated by combining two Dutch open data sets, namely the building footprints from the 'Basisregistratie Adressen en Gebouwen' (BAG)[14] and the elevation data as a point cloud from 'Actueel Hoogtebestand Nederland' (AHN)[15]. An example of some buildings, with the faculty of Architecture and the

[6] https://pypi.org/project/solarpy/
[7] https://cjio.readthedocs.io/
[8] https://pypi.org/project/Rtree/
[9] https://docs.python.org/3/library/concurrent.futures.html
[10] https://www.paraview.org/
[11] https://ninja.cityjson.org/
[12] https://github.com/cityjson/cjval
[13] https://3dbag.nl/en/viewer
[14] https://www.kadaster.nl/zakelijk/registraties/basisregistraties/bag
[15] https://www.ahn.nl/

Built Environment of the TU Delft as central point, is shown in Figure 5.1.

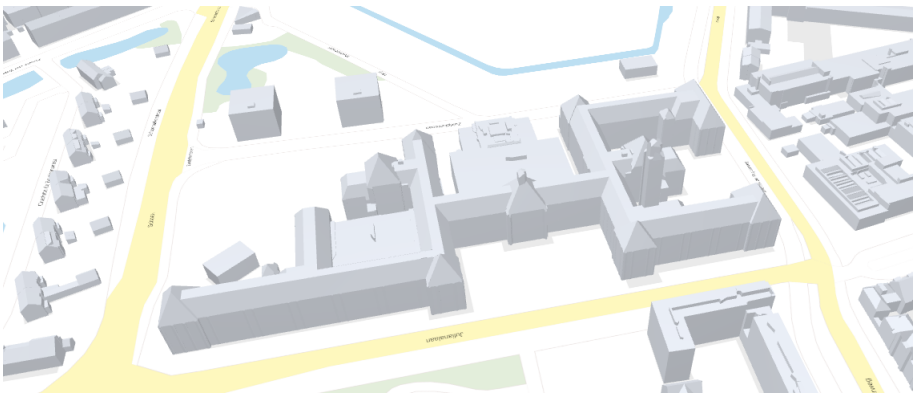


Figure 5.1.: Sample of 3D BAG with LoD 2.2 (taken from the 3D BAG viewer).

The buildings within 3D BAG are individually stored as a 3D mesh with LoD 0, LoD 1.2, LoD 1.3 and LoD 2.2. LoD 0 is the building footprint and does not include walls. LoD 2.2 can include sloped building roofs whereas LoD 1.2 and LoD 1.3 can only store the buildings as block models with flat roofs. An overview of the available LoDs for CityJSON files as presented by Biljecki et al. [2016] is shown in Figure 5.2. The building footprints at LoD 0 contain a list of attributes such as its id, building year and roof type. The buildings at LoD 1.2, LoD 1.3 and LoD 2.2 contain semantic surfaces for ground, wall and roof alongside possible semantic attributes.

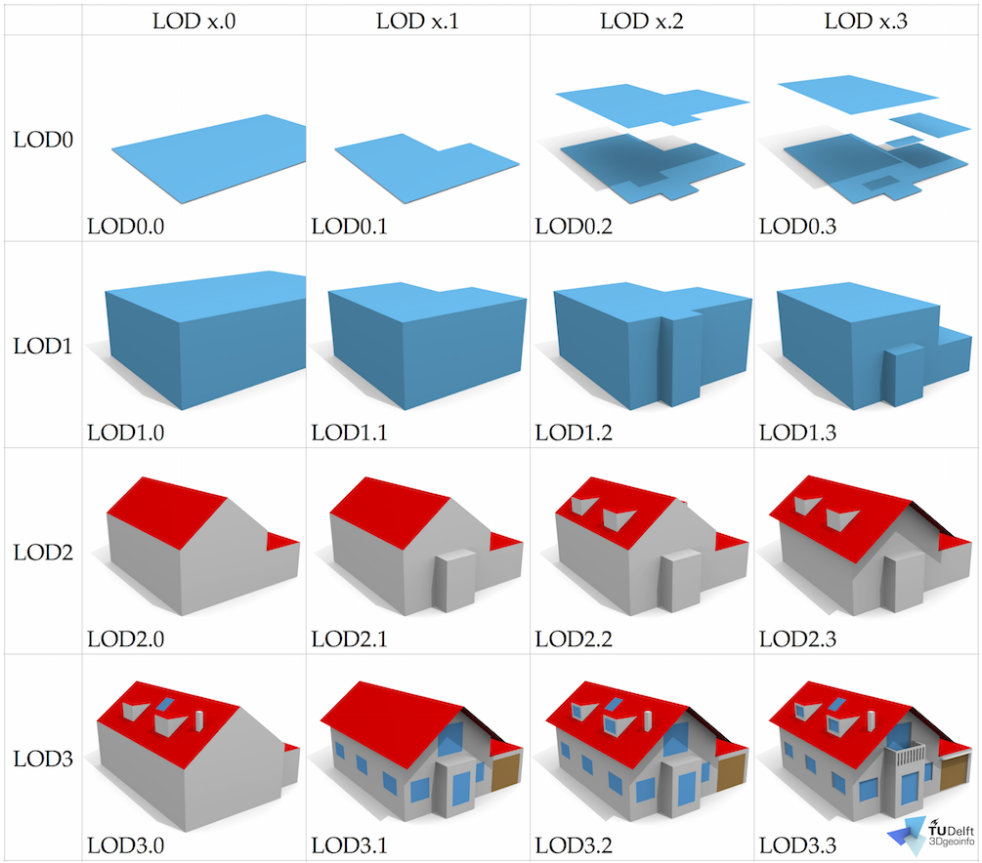


Figure 5.2.: Overview of LoDs. Figure from Biljecki et al. [2016]

The geometric primitives of the 3D building models in the 3D BAG data set are polygonal

surfaces defined by its vertices. These buildings need to be triangulated to be able to process them. The details about the triangulation can be read in Section 5.2.1.

Because the 3D BAG data set consist of more than 10 million buildings, the data is subdivided into tiles covering smaller geographic areas, as shown in Figure 5.3. The amount of buildings in a tile are determined by a quad tree data structure. The tiles are available for download in several formats: CityJSON, GeoPackage and Wavefront OBJ. The entire data set is also available via web services WMS and WFS. Lastly, the raw 3D BAG data for the whole of the Netherlands can also be downloaded as PostgreSQL data dump.

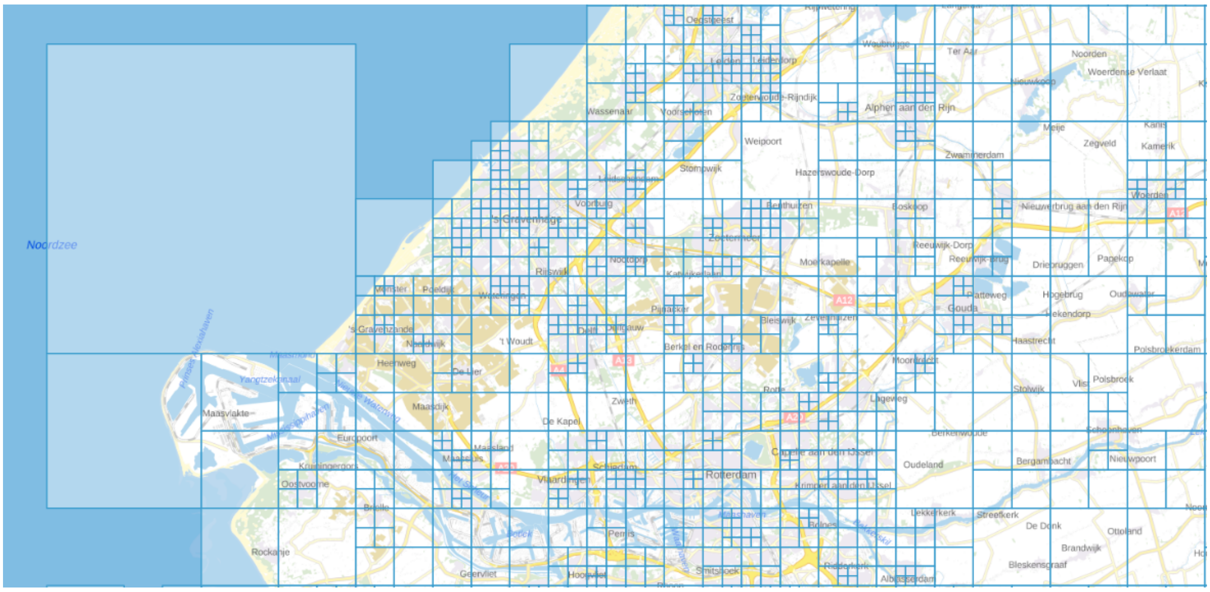


Figure 5.3.: 3D BAG tiles as quad tree

As 3D BAG is a large data set covering vector data, it is very suitable to use for this research. The fact that the data set is available in LoD 2.2 and the buildings have semantic surfaces, ensures that sloped roofs are represented, and that roof surfaces can easily be distinguished from other surfaces. Moreover, it is advantageous that the city models stored as tiles are available in CityJSON format. CityJSON is a compact and easy-to-use encoding of the CityGML data model very suitable to use when programming in Python [Ledoux et al., 2019].

For this research it is chosen to download the tiles used for the experiments presented in Chapter 6 in CityJSON format. For the solar potential computation only buildings with LoD 2.2 are taken into account.

5.2. Software implementation

In this section the software implementation of the methodology is explained in detail. The technicalities are explained in such a way that it could be reproduced. Also the challenges that were encountered and how they were solved are discussed. The code is made available in a GitHub repository[16].

[16] https://github.com/robinjo78/solarBAG

5.2.1. Input preparation

This section describes the steps that need to be taken to prepare and pre-process the input data before it can be given as input parameter to the implemented program called SolarBAG. The program takes as input parameter a folder of CityJSON files. This could be one file or multiple files as long as these are contained within a folder. The CityJSON files are tiles containing a city model that can be downloaded from the 3D BAG dataset. The CityJSON file should be at least version 1.1. The file needs to be triangulated by using the `triangulate` function in cjio:

```
$ cjio [file.json] triangulate save [new_file.city.json]
```

To check the triangulation, visualise the resulting CityJSON file in Ninja. Most probably there will be some buildings containing triangles of which the orientation is flipped, yielding a hole as a visual artefact. This could give problems for computing the solar radiation as the normal vectors are incorrect.

Now, the folder containing one or multiple CityJSON files can be used as input to SolarBAG by the following command:

```
$ python3 solarBAG_CityJSON.py [folder_with_cityjson_files]
```

The reason that the input to SolarBAG is a folder containing CityJSON files is because this makes it possible to process multiple tiles at once and to make the process scalable. It also makes it possible to consider buildings at the edges of a tile as neighbouring buildings potentially casting a shadow over a building in another tile.

5.2.2. Data loading and parameter settings

After the input is prepared, the CityJSON files in the folder given as input parameter to the program need to be loaded into memory. It need to be taken into account that the data might be too large to be stored in memory at the same time. Therefore, the CityJSON files in the folder are processed one by one. As the buildings on the edges of the tile could potentially cast a shadow over a building in another tile, these buildings also need to be loaded into memory for the specific tile.

This goes as follows. For each tile, its bounding box is extracted, which is needed to determine what the neighbouring tiles of a tile are. This extraction is done by manually reading through the raw CityJSON file and searching for the 'metadata' tag in which the 'geographical extent' property, representing the bounding box of a tile, is stored. This search is much faster than first loading all the CityJSON files and retrieving the 'metadata' object by using a 'get' function. The retrieved bounding boxes of each tile are inserted into a 2D R-tree, by using the 'Rtree' library.

The next step is to process each CityJSON file separately. The 'cjio' library is used to load the city model stored in a CityJSON file into memory. All the buildings in the city model are already triangulated in the pre-processing step, but it would be better to do this when loading the city model into memory, avoiding the overhead.

Then, from the city model the city objects, which are the buildings, are extracted. These are stored in a native Python's dictionary datastructure. This dictionary will be extended with buildings from neighbouring tiles that may potentially cast a shadow on buildings in the current tile. To actually find the neighbouring tiles of a tile, the R-tree containing the bounding

boxes of the tiles, is queried by the bounding box of the current tile, enlarged with a buffer of 100 meters.

For each tile returned by the query, the city model is loaded into memory and their building objects are inserted into an R-tree. On this R-tree, the same query as for finding the neighbouring tiles (the bounding box with a buffer), is applied. This query will return buildings that are located within the buffer extent that could potentially cast a shadow on the buildings at the edges of the currently processed CityJSON file. A schematic of this process for a tile with 8 neighbouring tiles is shown in Figure 5.4. The selected buildings from the neighbouring tiles within the buffer limits together with the buildings in the current tile are first stored in a common dictionary and then inserted into an R-tree.

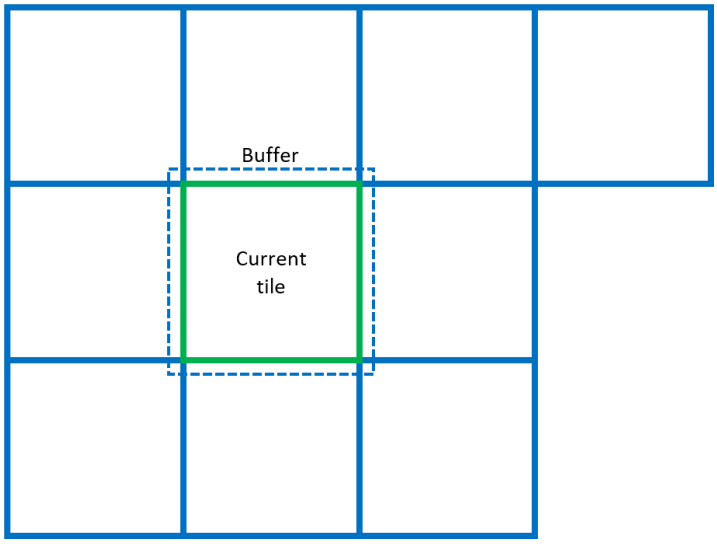


Figure 5.4.: Neighbouring tiles selection based on a buffer.

The R-tree is created by using the 'Rtree' library. The R-tree will have a dimension of 3, meaning it will be in 3D. For each building a bounding box is computed based on its surfaces. This bounding box will be inserted into the R-tree. Its entry will also contain the original building's id with which it is stored in CityJSON to easily look up the building's city object in the dictionary containing all building geometries. The R-tree will later be used to efficiently extract building geometries for intersection detection when performing shadow casting.

Before each building will be processed to compute its solar potential, some parameters need to be set. These parameters are:

- *Number of cores:* the number of cores used for performing multiprocessing on each building model.
- *LoD:* the LoD of each building model being processed. LoDs above 2.0 are supported as their roof surfaces can be distinguished from other semantic surfaces.
- *Neighbour offset:* the offset in meters considered when selecting neighbouring buildings that potentially cast a shadow on the building currently being processed.
- *Sampling density factor:* the sampling density used when sampling a roof surface triangle into grid points, given in meters.
- *The dates:* a list of dates for which the solar potential will be computed. This list will contain a date for each month. Also, the number of days the corresponding month will last is stored.

5.2.3. Processing each building model

After the data is loaded in memory and the parameters are set, it is time to process each building in order to compute its solar potential. This is done in parallel on multiple CPUs by using a multiprocessing library in Python.

The asynchronous multiprocessing library used is 'concurrent.futures'. The asynchronous execution can be performed with threads or separate processes. Within the context of this research, it is necessary to parallelise processes by using the '`ProcessPoolExecutor` instead of threads for which the `ThreadPoolExecutor` could be used. This is because the implemented process for each building can be executed independently. More on the different ways to parallelise processes can be found in Section 2.3.

But before submitting a building to the `ProcessPoolExecutor`, the neighbouring buildings with a high chance of casting a shadow on this building need to be selected. The neighbouring buildings are filtered on a couple of factors:

- Neighbour offset: a Euclidean distance in x, y and z directions in meters measured from the considered building;
- Maximum height;
- Azimuth (orientation) range.

To apply these filters the semantic roof surfaces of the considered building are extracted from its geometry and stored in PyVista's PolyData data structure. For the roof surfaces their center of mass can be computed, which is used as center point for a cubic bounding box that will be created based on the neighbour offset parameter. This bounding box is used as input to the intersection function of the R-tree to determine which buildings fall within this bounding box. The offset parameter has a large influence on the running time of the system. For a large neighbour offset, more neighbouring buildings will be subject to shadow casting than for a small neighbour offset. This also has effect on the accuracy of the resulting solar potential values. Therefore a trade-off is found by trial-and-error. The neighbour offset can be chosen by the user, but by default it is set at 100 meters.

The returned buildings are then filtered on their height. Neighbouring buildings of which their maximum rooftop height is lower than the minimum rooftop height of the considered building's roof are filtered. Lastly, the orientation of the remaining neighbouring buildings with respect to the considered building is computed. If the orientation of a neighbouring building falls within the azimuth range for which the sun does not illuminate the earth at the specific location and date, this building is filtered as well. In the end, only buildings that potentially cast a shadow on the considered building are kept and subjected to shadow casting later on. Visualisations of the filter process is shown in Figure 6.3 in Section 6.1.1.

The reason that the selection of neighbouring buildings is not done while multiprocessing is because the multiprocessing executor cannot store a pointer to the R-tree in memory. This means that the R-tree cannot be queried while multiprocessing. In practice, querying the R-tree does not give an error, but it does not find the neighbouring buildings either. The query result will be an empty list.

Now that the neighbouring buildings are selected, the considered building is submitted to the parallel processing pool and processed further. This can be done for each processing core that is set available for multiprocessing. When a core has finished processing its result will be saved and a new building will be submitted to the pool.

The process a building undergoes while multiprocessing is as follows. From the building geometry the roof surfaces and their individual triangles are extracted. For each triangle it is checked whether it is a sliver triangle. A sliver triangle is a very thin and elongated triangle as shown in Figure 5.5. They are undesirable because a simple geometric computation such as computing the normal vector yields an unreliable result. The thinness is determined by the ratio between the triangle's perimeter and area. When the thinness ratio of the triangle is lower than 0.001, the triangle is considered a sliver triangle and excluded from the solar potential computations. This number is chosen based on computing the thinness ratio for known sliver triangles and determining a ratio accordingly.

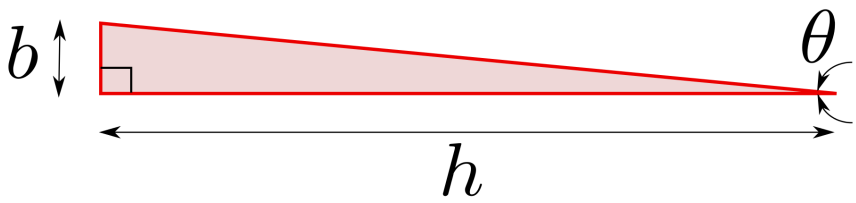
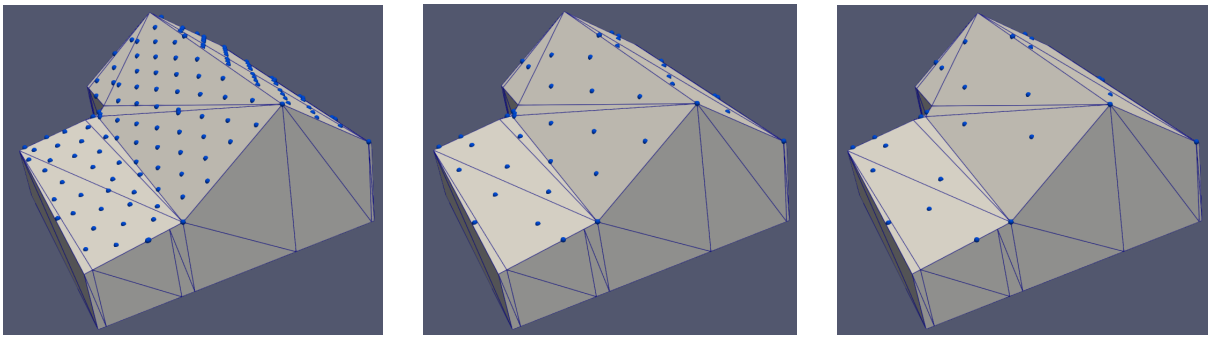


Figure 5.5.: Sliver triangle.

When a triangle is suitable for further computations, its normal vector is computed by using the PyVista library. The returned normal vector is pointing inward and is in East North Up (ENU) frame. Normally one would expect a normal vector to point outward. The ENU frame is equal to x, y, z notation. However, the function that is used to compute the beam solar radiation is expecting a normal pointing outward and stored in North East Down (NED) frame, equal to y, x, z notation. Therefore, a conversion of the normal vector is needed. The conversion of the normal vector from ENU to NED frame and from pointing inward to outward is done by swapping and negating x and y:

$$NED : [-y, -x, z] = ENU : [x, y, z] \tag{5.1}$$

Afterwards, on the triangle a uniformly distributed grid of points is sampled. For this, the function `create_surface_grid()` written by former supervisor Stelios is used and adapted. It can take a mesh with multiple surfaces as input, but for this implementation only one surface, a triangle, is given as input. Also the density parameter, defined at the parameter settings, is given as input to the function. In Figure 5.6, the effect of the density factor on the sampled grid points is shown. A lower density factor yields a higher resolution grid than a higher density factor.



(a) Grid sampled with a density of 1m

(b) Grid sampled with a density of 2m.

(c) Grid sampled with a density of 3m.

Figure 5.6.: Grids sampled with increasing density values.

The vertices of the triangle are projected to 2D and a polygon is created from it. On this 2D polygon, a 2D grid of points according to the density is sampled. Only the sampled points that are located within the 2D polygon are kept. Then, these points are projected back to 3D with a small positive offset for the z-value. This ensures that the grid point is always above the triangle, preventing an intersection with the triangle the grid point belongs to when the grid point is subjected to intersection detection for shadow casting. Each of these points will be put in a PolyData data structure and an empty attribute is added to store a list of solar potential values for each month. Now, the triangle is enriched with a grid of points ready for solar potential computations.

The next step is to compute the solar potential for each grid point on the triangle for each date in the date list. For this research, for each month in a year, the 21st day is used as the day to compute the solar potential. This value represents the average solar potential for that month. As the 21st of December is the shortest day in the year and the 21st of June is the longest day in the year in terms of sunlight hours, they will compensate each other. This value is then aggregated to the length of the month to get an average solar potential for the month. Lastly, all the months will be summed up to obtain a solar potential per year.

The actual solar radiation computation will be done by using the `irradiance_on_plane()` function in the solarpy module. Code excerpts of this algorithm are given in equation 6.2 and 6.3 in Section 6.2.2. It is included in that chapter in order to easily compare the algorithm with the solar radiation algorithm in ArcGIS.

The `irradiance_on_plane()` function takes as parameters:

- Normal vector
- Height
- Date
- Latitude

The input normal vector is the normal vector of the triangle. The height is the average height of all the grid points on the triangle. The date is the day of the month in hourly intervals. So each day will have 24 hours. The latitude value is computed by taking the coordinates of the center point of the triangle and projecting it to a latitude. The height and the latitude of a grid point is chosen to be the average value for the triangle's roof, because at such a small local scale, the height and latitude does not have much influence on the resulting solar radiation value. Another reason to use the average height and latitude value of the triangle for a grid point is to save processing time.

At first, the solar radiation is computed for the whole triangle for each date. Then, for each grid point on the triangle it needs to be determined whether it is in put in shadow or not by a neighbouring building. As this might change throughout the day, the path the sun travels on the day at hourly intervals is determined. For each time stamp the solar vector, also called sun beam, is computed. This vector determines the direction of the sun ray in NED frame. To model the position of the sun, not the real distance is used, but a distance far enough to cover buildings in the scene potentially blocking the sun's rays. This distance needs to be more than the neighbour offset distance of 100 meters. So, a distance of 500 meters is chosen.

After the sun path corresponding to a grid point is determined, a ray intersection test between the position of the sun at each time stamp in the sun path and the grid point is performed. This is done by applying the `ray_trace()` function available in PyVista to each of the neighbouring buildings of the building currently processed. In the case of a hit, the corresponding timestamp is stored. Then, the daily solar potential for the grid point is recalculated while setting the solar

potential at the time stamp of a hit to 0 and stored in the list attached as field to the PolyData object of the grid point.

Now, when the solar potential for a grid point is calculated for each 21st day of a month, the daily solar potential will be multiplied by the number of days in the specific month to arrive at a solar potential for a month. Then, these monthly values will be summed up to arrive at a yearly solar potential for a grid point.

Writing the various statistical values of the yearly solar potential per triangle to file will be discussed in the next section.

5.2.4. Writing to CityJSON

During the main process, where a city object is being processed, the city object is prepared to be written back to a CityJSON file. It is important to write the file back to CityJSON as it was read, but with the solar radiation attributes added to each suitable roof triangle. These are:

1. Number of samples: reffering to the number of points that are used to sample this triangle of a roof surface.
2. Average solar potential;
3. Maximal solar potential;
4. Minimal solar potential;
5. Standard deviation: describing the amount of variation of the solar potential of the grid points;
6. 50th percentile: the solar potential value below which 50% of the solar potential values of the grid points may be found;
7. 70th percentile: the solar potential value below which 70% of the solar potential values of the grid points may be found.
8. Solar potential unit: the unit in $Wh/m^2/year$.

The number of samples is saved per roof surface in order to make sense to the rest of the attributes. The average solar potential value is meaningful for each surface, while the maximum, minimum and standard deviation is only meaningful for roof surfaces with 2 or more grid points, otherwise the values do not make sense. The 50th and 70th percentile is becoming useful for roof surfaces with 3 or more grid points as these statistics are meaningful for a larger amount of values.

The `save()` function of the 'cjio' library is used to write the city model, enriched with solar potential statistics per roof surface, back to a CityJSON file. Unfortunately, the `save()` function does not write a transformed version of the city model to file. The vertices of the geometries are just written to the file uncompressed. This also means that the transform object is missing leading to an invalid file when validating the CityJSON file with the CityJSON validator cjval[17]. In order to make the validator succeed, a transform object needs to added manually to the CityJSON file. Visualising such a CityJSON file in Ninja does not give any problems.

[17] https://validator.cityjson.org/

6. Experiments and Results

This chapter gives the general results after running the implemented model for a tile. Besides, it lists the experiments and results for the quality assessment and scalability assessment.

6.1. General results

Figure 6.1 shows a resulting city model after running the implemented model for tile number 5873 of the 3D BAG dataset[1]. It covers a neighbourhood in The Hague. The CityJSON file in which the city model is stored is loaded into visualisation program Ninja of which the screenshot is taken. The roof surfaces of each building are enriched with solar potential values. The list of attributes is shown for a specific building. You can see that the number of sampled points for the specific triangle is 2 in this case. The solar radiation values of these two points are different as the minimum and maximum solar potential values of this triangle are different. This also gives varying values for the average, 50th and 70th percentile and a standard deviation of non-zero.

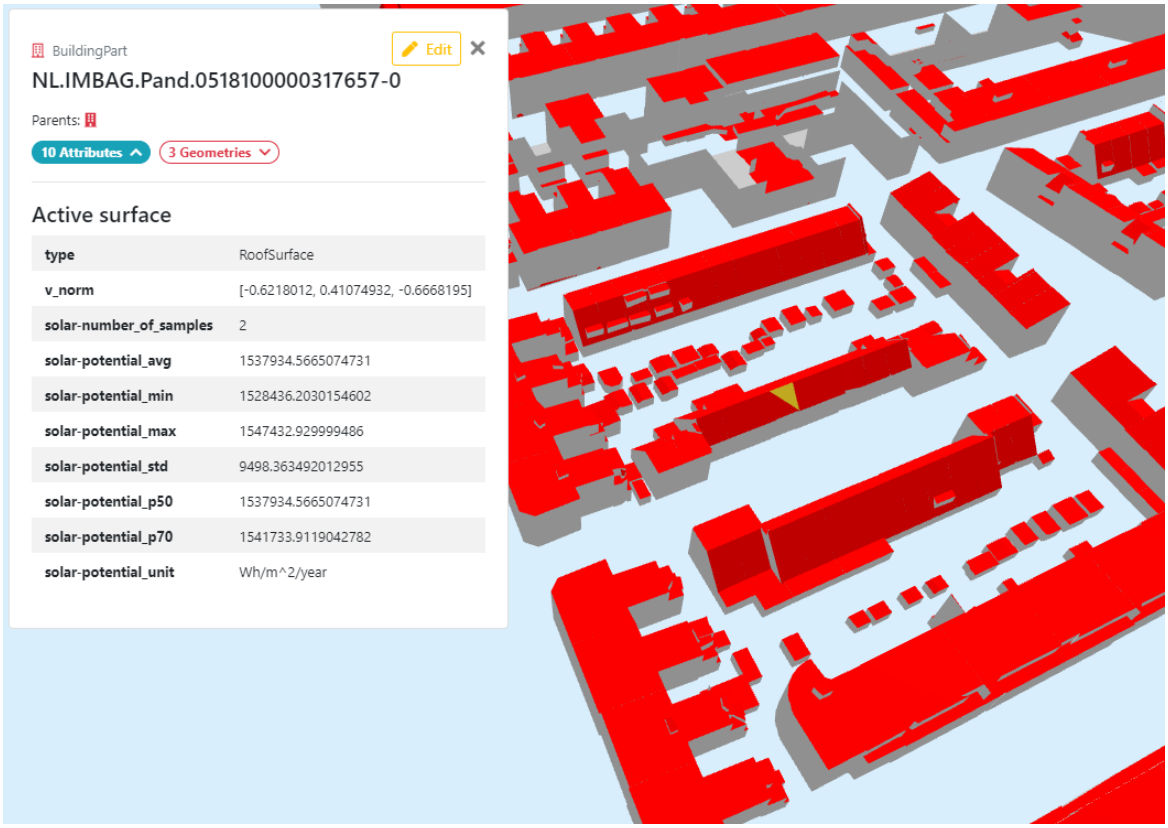
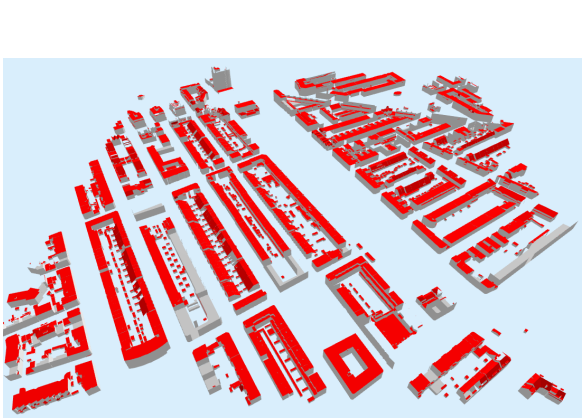


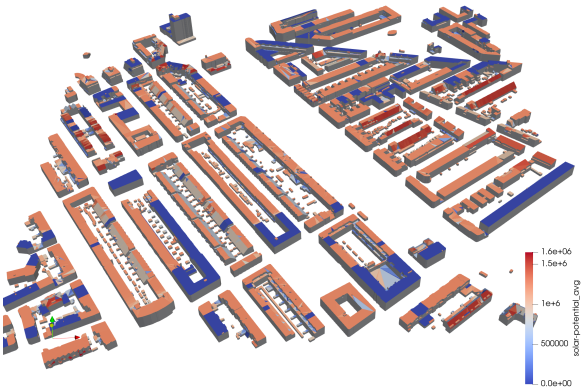
Figure 6.1.: Building enriched with solar potential values, shown in Ninja.

[1] https://3dbag.nl/en/download?tid=5873

When visualising the city model in this tile as a whole in Ninja alongside a visualisation of the average solar potential values per triangle shown in ParaView, something interesting can be observed. As already mentioned previously, the orientation of some triangular surfaces might be flipped, which is visible as missing roof surfaces in Figure 6.2a. This has as effect that the normal vectors of these surfaces point downwards resulting in a solar potential value of 0. In Figure 6.2b this is shown by the blue coloured roof surfaces. A larger image of Figure 6.2b is shown in Figure B.4 in Appendix B.1.



(a) Visualisation of the city model in Ninja.



(b) Visualisation of average solar potential values (in Wh/m^2) per triangle in ParaView.

Figure 6.2.: Visualisation of tile 5873.

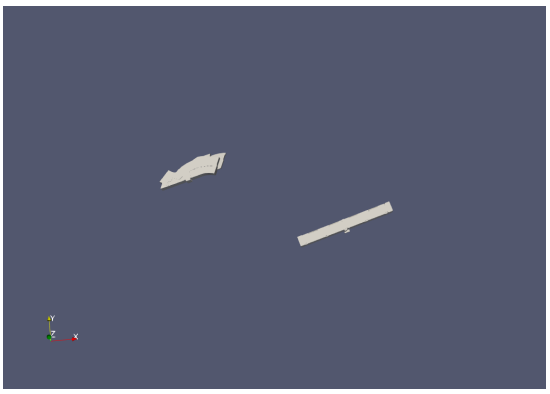
Unfortunately, this issue could not have been solved during this research. The triangulation function used from 'cjio' is responsible for this behaviour. The issue could be solved manually by flipping the orientation of the wrongly oriented roof surfaces. The validation of the solar potential values for the correctly oriented triangles is executed in Section 6.2.2.

6.1.1. Visualisation of intermediate steps

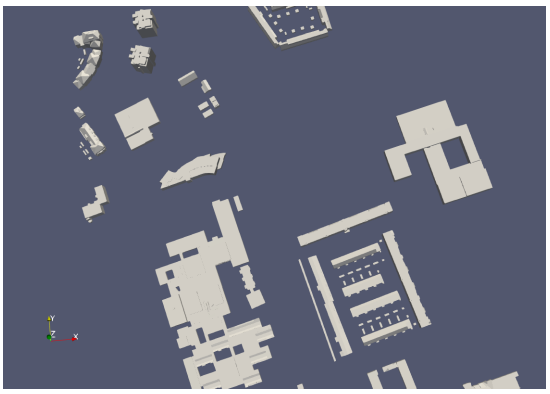
In this section, some of the steps of the methodology of this research are visualised. The concerned steps are neighbour filtering, point sampling and intersection detection. These steps are visualised by using the software ParaView.

Figure 6.3 shows the remaining neighbouring buildings after applying filter operations based on the distance, height and orientation of the neighbouring buildings. The two buildings meshes for which the neighbours are filtered are shown in Figure 6.3a. Their neighbouring buildings are shown in Figure 6.3b. In Figure 6.3c, the distance filter is applied to remove buildings that are too far to cast a shadow. In Figure 6.3d, the neighbouring buildings are filtered on the height of their roofs. If the maximum height of a neighbouring building roof is lower than the height of the currently processed building, this neighbouring building is removed. In Figure 6.3e, the neighbouring buildings are filter on their orientation with respect to the currently processed building. The buildings are filtered for a certain azimuth range at which the sun will not be positioned during a day.

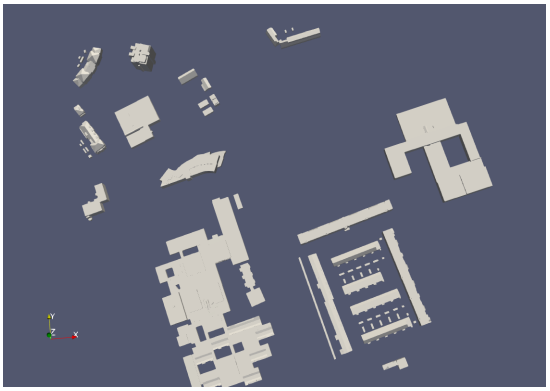
These filters are applied when the neighbouring buildings of a building need to be found. The filters have a performance benefit when neighbouring buildings potentially casting a shadow over a building need to be found. The more buildings are filtered, the less buildings need to be considered for the compute-intensive intersection detection operation.



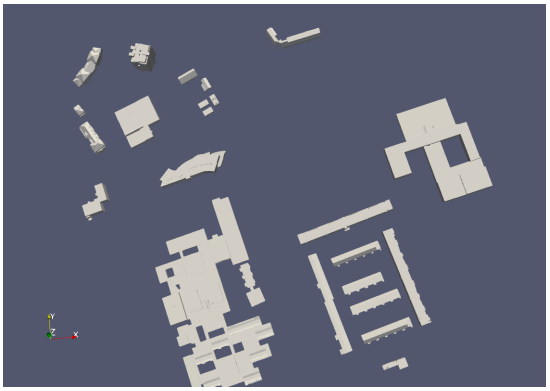
(a) Two starting meshes.



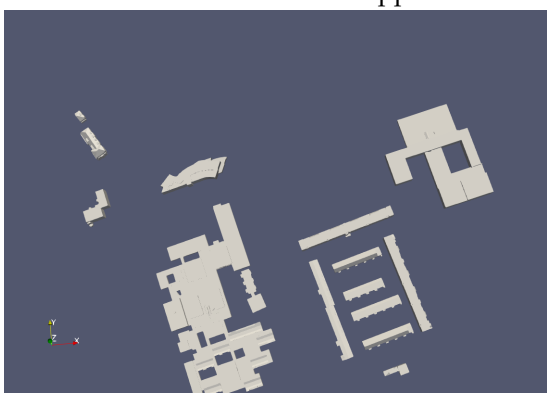
(b) All neighbouring buildings meshes.



(c) Filter on distance to neighbouring buildings applied.



(d) Filter on height of neighbouring buildings applied.



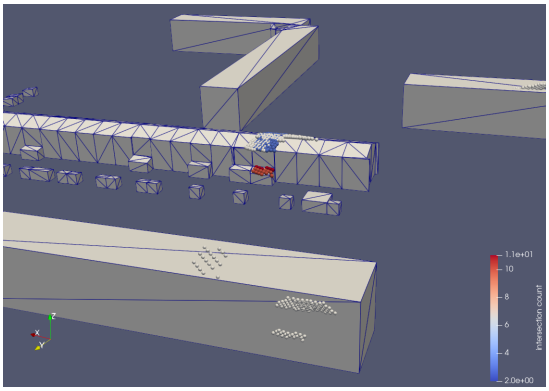
(e) Filter on orientation of neighbouring buildings applied.

Figure 6.3.: Applying filters on neighbour selection for intersection detection. Screenshots taken in ParaView.

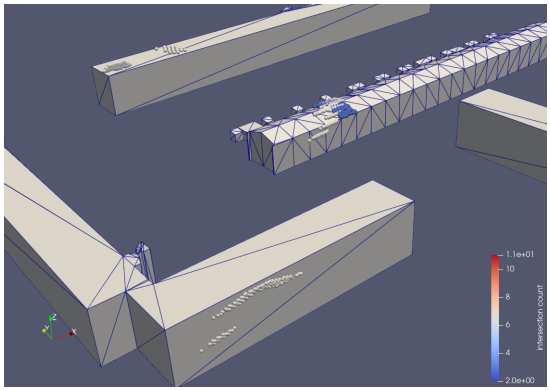
Figure 6.4 gives a visualisation of the intersection detection operation that is carried out for each sampled point in order to find neighbouring buildings casting a shadow on a building. Figures 6.4a and 6.4b show the locations of the intersections, represented by the light gray points, from two perspectives. The coloured points show the number of times a sun ray cast in the direction of that sampled point is intersected. It can be seen that the lower roof has a higher intersection count than the higher roofs. This is as expected as lower roofs have a higher chance to be put in shadow than higher roofs.

Figures 6.4c and 6.4d show the effect of the intersections on the resulting solar potential values

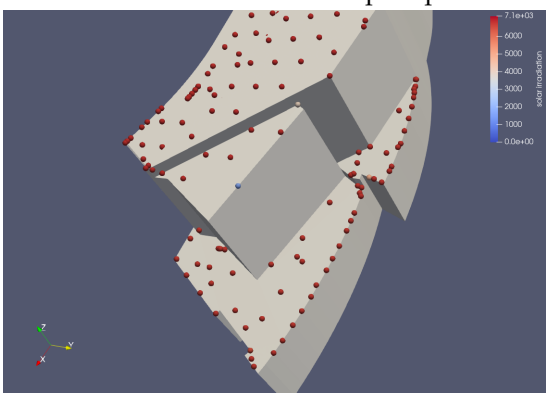
per grid point on a building roof. Figure 6.4c shows the solar potential per sampled point without taking intersections into account, while in Figure 6.4d, the intersections are taken into account for the sampled points. As expected, the solar potential for some of the sampled points is lower after taking intersection detection into account as the building itself puts a shadow on its lower roofs. In Appendix B.1, additional screenshots of early visualisations or intermediate steps are provided.



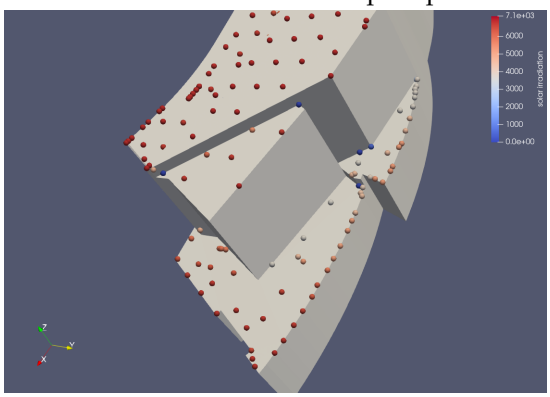
(a) Intersection locations (gray points) and intersection counts on the sampled points.



(b) Intersection locations (gray points) and intersection counts on the sampled points.



(c) Solar potential value per sampled point without intersections taken into account.



(d) Solar potential value per sampled point with intersections taken into account.

Figure 6.4.: Visualisation of intersection detection and the effects. Screenshots taken in ParaView.

6.2. Quality assessment

In Section 4.3 the ideas behind the ground truth comparison and ArcGIS comparison were already shortly introduced. These two comparisons will serve as quality assessment methods to assess the solarpy solar model utilised in the implemented method, and to assess the implemented method itself. In this section, the main workflow, results and discussion for both comparison methods will be presented.

6.2.1. Ground truth comparison

To verify whether a simulated model is accurate and reliable, it can be compared with the ground truth values of the phenomena, in this case the solar radiation. In The Netherlands

there are 48 weather stations that can measure global solar radiation [KNMI]. For the ground truth comparison three of these weather stations will be chosen and its global radiation values will be compared to the direct radiation computed by the `beam_irradiance()` function in the solarpy package in Python for the location of the weather stations. The workflow of the ground truth comparison is presented in Figure 6.5 and explained in more detail below.

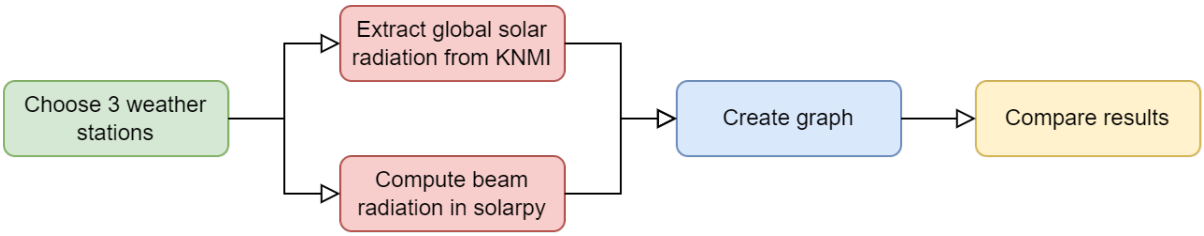


Figure 6.5.: Workflow of ground truth comparison.

To start the ground truth comparison three weather stations spread throughout The Netherlands need to be chosen. The three weather stations chosen are:

1. Voorschoten;
2. Wilhelminadorp;
3. Nieuw Beerta.

In Figure 6.6 the locations of the three weather stations and the typical lay-out of such a weather station are shown. These locations are chosen as such in order to account for various factors influencing the resulting solar radiation values, such as humidity and latitude. For each weather station the raw weather data for a specific month is extracted from the Koninklijk Nederlands Meteorologisch Instituut (KNMI) data platform[2].

(a) Map showing the location of the chosen weather stations.

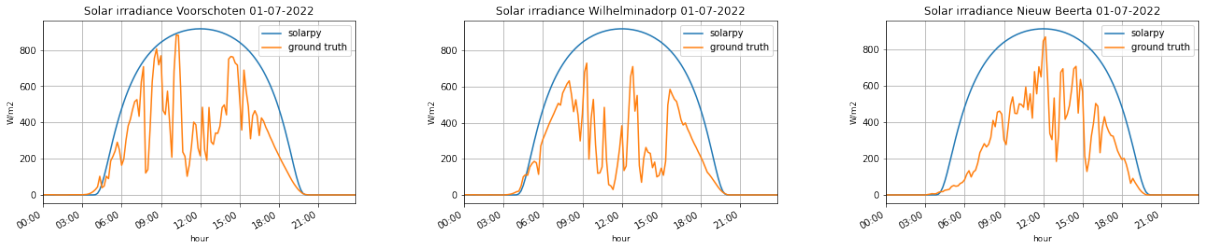
(b) Lay-out of a weather station.

Figure 6.6.: Map of weather stations and their lay-out.

[2]https://dataplatform.knmi.nl/dataset/zonneschijnduur-en-straling-1-0

In such a weather file the data is organised row by row. Each row covers 10 minutes and contains values for the date, timestamp, location id, location name, latitude, longitude, altitude, average global solar radiation, minimum global solar radiation, maximum global solar radiation and sunshine duration. The values to be used for the comparison are taken from the column containing the average global solar radiation. As the values are averaged, they are somewhere in the middle between the minimum and maximum values. As direct radiation has the largest contribution to global radiation with approximately 77% [Spitters et al., 1986], it is most logical to choose the average global solar radiation. Therefore, it is chosen to compare the average global solar radiation as ground truth to the direct solar radiation computed by solarpy.

Figure 6.7 shows the graphs for the solar radiation values of the three chosen weather stations in The Netherlands. In each graph, the orange curve shows the ground truth average global solar radiation as measured by the weather station, whereas the blue curve shows the beam (direct) radiation as computed by the solarpy package in Python. Each graph covers a day (01-07-2022) at 10-minute intervals for the corresponding weather station.



(a) Solar irradiance in Voorschoten.

(b) Solar irradiance in Wilhelminadorp.

(c) Solar irradiance in Nieuw Beerta.

Figure 6.7.: Graphs showing the beam and global irradiance for the three weather stations.

As can be seen in the graphs, the curves do not really fit each other, but the curve development is following more or less the same trend. If we take Figure 6.7a, showing the graph of Voorschoten as example, we can see that the ground truth curve has a lot of peaks and valleys. This can be explained by the time the sun shines at each time interval. Where there is a peak, the sun most likely shines more or less the complete time interval, whereas at a valley, the sun most likely shines barely or not at all. The blue curve representing the computed solar radiation, does not have peaks and valleys, because the model always considers a clear sky condition, meaning that the sun shines the complete time range. Therefore this curve can be considered to show the potential direct solar radiation for each timestamp at clear sky conditions.

Furthermore, the comparison is not quite fair, because global solar radiation is compared to direct solar radiation. It would be a more fair comparison if the global radiation is decomposed into diffuse and direct direction. But, this is not done as the data is then not ground truth anymore but also derived. This would also have as effect that the curve shape would be more or less the same but with lower peaks and deeper valleys as the diffuse radiation part is not included anymore. In the end, the curves give an indication of the potential direct solar radiation for each time interval throughout a day compared to the ground truth.

An idea for a better ground truth comparison with actual measured direct solar radiation, instead of global solar radiation measured at fixed locations, could be by manually placing pyrheliometers at building roofs and compare their measurements with the direct solar radiation computations by solarpy. A pyrheliometer is a device that can measure only direct radiation as it is designed in such a way that only solar energy from direct sun rays are being

captured. In order to capture usable direct solar radiation data, the device needs to be facing towards the direction of the sun.

6.2.2. ArcGIS comparison

State-of-the-art GIS packages such as ArcGIS [Redlands, 2011] and GRASS GIS [GRASS Development Team, 2020] contain solar radiation modules to compute the solar radiation of raster data sets. To verify the correctness of the implemented vector model using solarpy as core solar radiation model, it is chosen to compare the vector model with a raster model as for both data formats it should be possible to compute accurate solar radiation values.

Giannelli et al. [2022] performed a comparison analysis on existing solar radiation modules within GIS programs. They tested the solar radiation modules in GRASS GIS, ArcGIS, SimStadt, CitySim and Ladybug. In their results it is shown that the solar radiation module within ArcGIS is the fastest one and that the yearly solar radiation values computed by the ArcGIS module differ the least from the ground truth. Therefore the raster-based module in ArcGIS is chosen over the module in GRASS GIS for the comparison with the implemented vector model described in this research.

As ArcGIS processes raster data and the implemented model in this research processes vector data, a couple of pre-processing steps need to be performed in order to do a comparison. The workflow of the pre-processing steps and the comparison is shown in Figure 6.8.

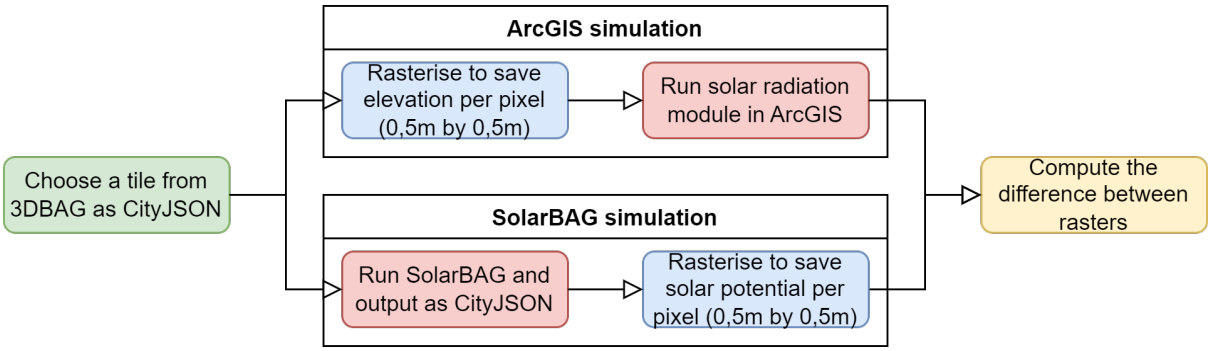


Figure 6.8.: Workflow of the ArcGIS comparison.

ArcGIS simulation

To be able to run the solar radiation module in ArcGIS a sparse raster is created based on a tile in CityJSON format containing 3D building models downloaded from the 3D BAG data set. As these building models are used for the vector-based method, this ensures that the raster has the same elevation data as the 3D BAG data set: only the building models. As an alternative, raster data from AHN could have been used, but this data also includes other objects like vegetation which is not included in the 3D BAG dataset. This would result in an unfair comparison as the features are different.

For the creation of the raster based on the 3D BAG dataset, a separate script is written. Its workflow is as follows.

1. Load a 3D BAG tile, extract buildings and convert to PyVista's PolyData;
2. Create a grid of resolution $0.5m$ by $0.5m$ on the extent of the 3D BAG tile;

3. Loop over each pixel in the grid and use PyVista's `ray_trace()` function to check whether a building is visible from this pixel with a downward vector;
4. Store an elevation value for each pixel in the grid belonging to a building, otherwise this pixel will be stored as 'NoData';
5. Write the grid to a '.tif' file.

The resulting raster file is then used as input for the solar radiation module in ArcGIS. Within the ArcGIS environment a geoprocessing tool to compute solar radiation for raster data sets is available. This tool is called "Area Solar Radiation"[3]. Its description reads: "Derives incoming solar radiation from a raster surface" [ArcGIS, 2016a]. The relevant input/output parameters are listed in Table 6.1. Only the direct radiation raster is considered for the comparison.

Parameter	Value in ArcGIS	Value in SolarBAG
Input	Raster (.tif)	Vector (CityJSON)
Latitude	52	Different per building
Sky size / Resolution	200 x 200 pixels	N.A.
Time configuration	Whole year	Whole year
Hour interval	1	1
No. of cores	1	10
LoD	N.A.	2.2
Neighbour offset	N.A.	150m
Sampling density	N.A.	3m
Calculation directions	32	N.A.
Transmittivity	0.7	Value unknown
Radiation type	Global, direct and diffuse	Direct
Output	Raster (.tif)	Vector (CityJSON)

Table 6.1.: Comparison of input/output parameters for the solar radiation model in ArcGIS and SolarBAG.

SolarBAG simulation

To create a raster containing the resulting direct solar radiation values of the implemented vector method, SolarBAG, the process is reversed as compared to the ArcGIS simulation. First, SolarBAG is executed for a tile stored in CityJSON format, taken from the 3D BAG data set. The relevant parameters are listed in Table 6.1.

The resulting CityJSON file enriched with the solar potential values is then rasterised to store a solar potential value for each pixel. The creation of the raster file is similar as the one created for the ArcGIS simulation. The workflow is as follows, where the differences are shown in **bold**.

1. **Take a with solar potential enriched tile**, extract buildings and convert to PyVista's PolyData;
2. Create a grid of resolution $0.5m$ by $0.5m$ on the extent of the 3D BAG tile;
3. Loop over each pixel in the grid and use PyVista's `ray_trace()` function to check whether a building is visible from this pixel with a downward vector;

[3] https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/area-solar-radiation.htm

4. Store the **average solar potential value** for each pixel in the grid belonging to a building **by extracting this from the building geometry**, otherwise this pixel will be stored as 'NoData';

5. Write the grid to a '.tif' file.

Comparison

Now, that the direct solar radiation rasters from both methods are created, they can be compared to each other. This is done by computing the absolute difference between both rasters. The visual effect of the absolute difference raster is that the large differences in solar radiation values are easily visible with an appropriate colour scheme.

For this computation two geoprocessing tools in ArcGIS are needed, namely "Minus"[4] and "Abs"[5]. First, the Minus tool is used for the two rasters. Afterwards the Abs tool is used on the raster resulting from the Minus operation.

The raster comparison is performed for tile number 3476 taken from the 3D BAG dataset[6]. It is situated to the south west of Leiden and contains only a few buildings so processing it would not take too much time. The resulting solar radiation rasters for a part of the tile as computed by ArcGIS and the implemented vector model are shown in Figure 6.9.

(a) Solar radiation raster for ArcGIS method.

(b) Solar radiation raster for vector method.

Figure 6.9.: Resulting solar radiation rasters.

By comparing the images in Figures 6.9a and 6.9b it can be seen that for most of the roof pixels the colours representing the solar radiation values are similar. However, at some edges or corners the values do differ significantly. This can easily be seen in Figure 6.10 where the difference between the two rasters is shown. An example is visible in the top part of the image where a band of yellow/green pixels is present in Figure 6.9a. This difference is caused by the resolution. The raster used as input for the solar radiation tool in ArcGIS has a resolution of 0.5m by 0.5m, while in the vector method the solar radiation value is stored as average per

[4]https://pro.arcgis.com/en/pro-app/latest/tool-reference/3d-analyst/minus.htm
[5]https://pro.arcgis.com/en/pro-app/latest/tool-reference/image-analyst/abs.htm
[6]https://3dbag.nl/en/download?tid=3476

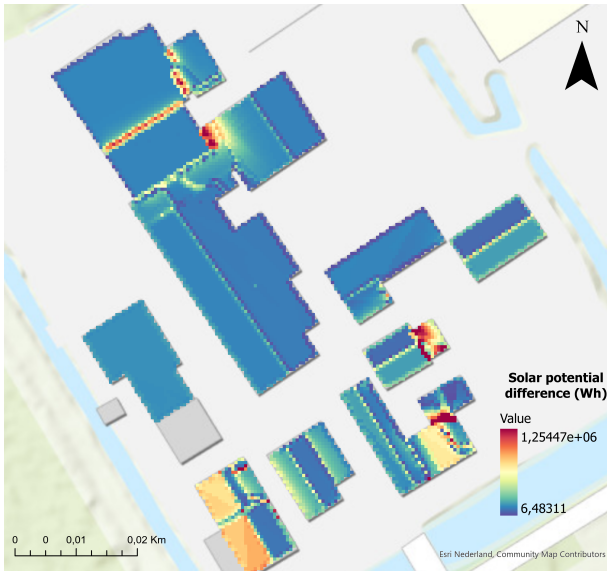


Figure 6.10.: Absolute difference raster between the two methods.

triangle. So practically, the size of the triangle determines its resolution. To get to an average solar radiation value per triangle, point sampling with a certain density is performed. In this example the density factor represents a point sampling resolution of 3.0m by 3.0m. As the solar radiation value is eventually stored as average per triangle, this means that a higher density (of for instance 0.5m by 0.5m) will not result in a higher resolution per triangle for the vector method. It may only result in a slightly different average solar radiation value per triangle as the average is then taken from more sampled points that may potentially be in shadow. Of course, the resolution per sampled point is higher, but to emphasise once more, this information is lost with the aggregation to triangle level. A last point to make, related to the resolution, is that the raster created for the vector method has a resolution of 0.5m by 0.5m while the actual resolution is determined by the size of each triangle.

A large difference in the solar radiation values can be found in the bottom (left) part of Figure 6.10. The reason for this large difference is not found out, but is probably caused by bugs in the computations. But, after recalculation by extracting the normal vector from the roof surface the resulting solar radiation of that part is similar to that part in the ArcGIS raster. To get an indication of the sloped roof surfaces enriched with the average solar potential value computed by SolarBAG, the 3D city model of the rasters in Figure 6.9, is shown in Figure 6.11.

Another factor that may cause differences in resulting solar radiation values is the algorithm/formula that is used per method. The method in ArcGIS processes raster data while the implemented method in this research processes vector data. The difference in input data type has as effect that a different algorithm/formula is needed for both methods. The solar radiation calculation for the raster method is based on a viewshed algorithm. This algorithm determines the raster surface locations visible from another raster pixel, known as the observer. The solar radiation calculation for the vector method it is based on normal vectors and ray intersection detection. This also means that the parameters may differ per method and might therefore not completely match.

In equation 6.1 the formula for the computation of direct radiation for a raster in ArcGIS is given[7]. S_{const} refers to the solar constant, set at $1367W/m^2$, refers to the transmittivity of the

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

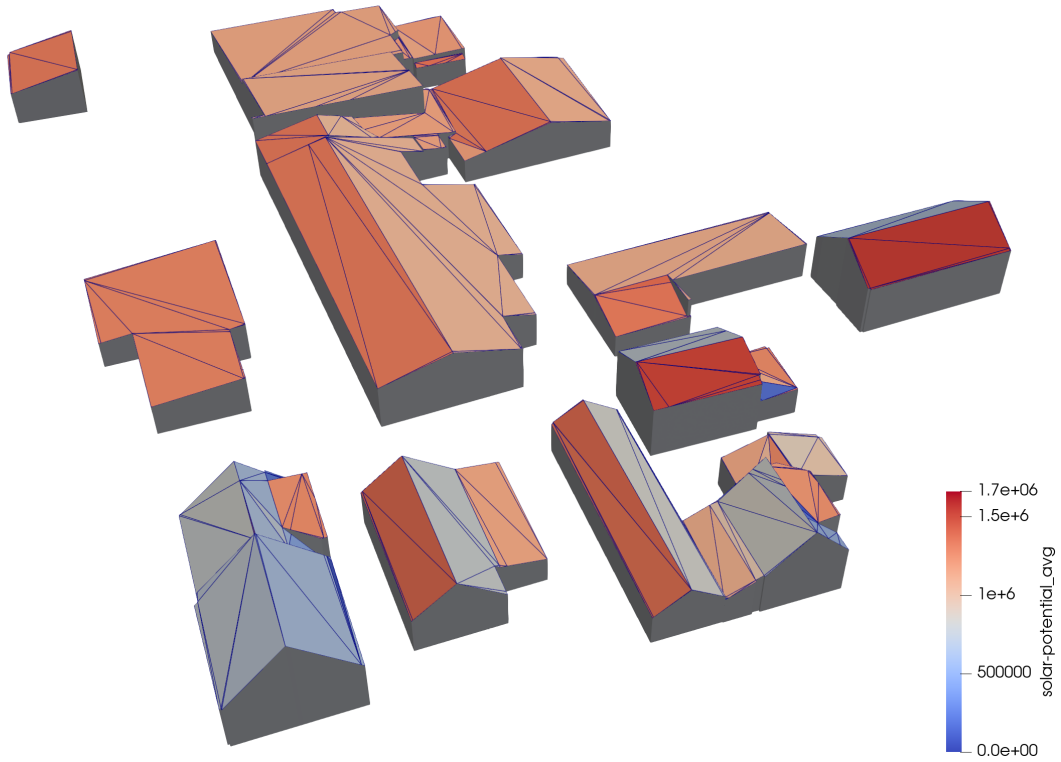


Figure 6.11.: 3D equivalent of the buildings in the rasters with solar potential in Wh/m^2 stored per triangle. Visualised in ParaView.

atmosphere, $m(\theta)$ refers to the relative optical path length, $SunDur_{\theta,\alpha}$ is the time duration represented by the sky sector, $SunGap_{\theta,\alpha}$ is the gap fraction for the sun map sector and $AngIn_{\theta,\alpha}$ is the angle of incidence.

$$Dir_{\theta,\alpha} = S_{const} * \beta^{m(\theta)} * SunDur_{\theta,\alpha} * SunGap_{\theta,\alpha} * \cos(AngIn_{\theta,\alpha}) \tag{6.1}$$

In equations 6.2 and 6.3 the code excerpts taken from the solarpy library[8] for the computation of the solar irradiance on a plane based on the beam irradiation within the solarpy library is given. The excerpts are a direct implementation of equations 4.1 and 4.5. $gon(date)$ refers to the extraterrestrial irradiance, $prel$ refers to the pressure relation, m refers to the air mass ratio, $alpha\_int$ refers to the atmospheric extinction parameter set to 0.32 and $\cos(\theta)$ refers to the angle of incidence to the plane.

$$beam\_irradiance = gon(date) * exp(-prel * m * alpha\_int) \tag{6.2}$$

$$irradiance\_on\_plane = beam\_irradiance * \cos(\theta) \tag{6.3}$$

Between equation 6.1 and the code excerpts in equations 6.2 and 6.3 similarities can be found. S_{const} and $gon(date)$ refer to the same parameter, namely the solar constant. However, in the equation in ArcGIS just the constant of $1367 W/m^2$ is taken while in the solarpy package the value referring to $gon(date)$ is a fraction of the solar constant influenced by the time of the day. The transmissivity factor $\beta^{m(\theta)}$ in the ArcGIS equation is related to the $exp(-prel * m *$

[8] https://pypi.org/project/solarpy/

$alpha\_int$) in the solarpy excerpt as these parameters are influenced by the weather and atmospheric conditions. The factors $SunDur_{\theta,\alpha}$ and $SunGap_{\theta,\alpha}$ in the ArcGIS equation are raster exclusive and are related to the viewshed operation performed on the raster pixels. The last parameter, $AngIn_{\theta,\alpha}$ is related to $\cos(\theta)$ referring to the angle of incidence on the plane. This concludes the comparison between the equations for the ArcGIS solar radiation module and the solarpy package.

Furthermore, the vector method in solarpy computes the beam solar radiation for clear-sky conditions, meaning that the sun shines the whole day between sun rise and sun set all year long. This results in best case solar radiation values. But in practice, the sun does not shine the whole day of course as it might be cloudy or raining. In ArcGIS cloudy days can be taken into account, reducing the amount of direct solar radiation reaching the earth. The parameter responsible for this is the transmittivity: it is a factor (between 0 and 1) denoting the amount of solar energy reaching the earth's surface. The rest is received at the upper limit of the atmosphere meaning that it does not reach the earth's surface. In the solarpy package which is used for the vector method, this transmittivity factor cannot be set, but is encapsulated by the pressure relation, air mass and atmospheric extinction. After some trial and error, with a transmittivity factor of 0.7 the resulting solar radiation values computed in ArcGIS are the closest to the results of the vector method. It can be expected that the factor needs to be 1 as the solarpy method computes beam solar radiation for clear-sky conditions. However, it does takes atmospheric conditions such as the air mass and atmospheric extinction into consideration. Therefore, the transmittivity factor of 0.7 is found to be suitable.

The discussed factors that may cause this difference are summarised below:

- Raster/sampling resolution;
- Different algorithm/formulas;
- Impossible to match input parameters exactly;
- Transmittivity.

6.3. Scalability Assessment

In order to show that the implemented method, SolarBAG, proposed in Chapter 4 is able to process growing amounts of big data, its scalability needs to be assessed. The scalability is addressed by executing a couple of scenarios in an experiment and by discussing some other scalability issues.

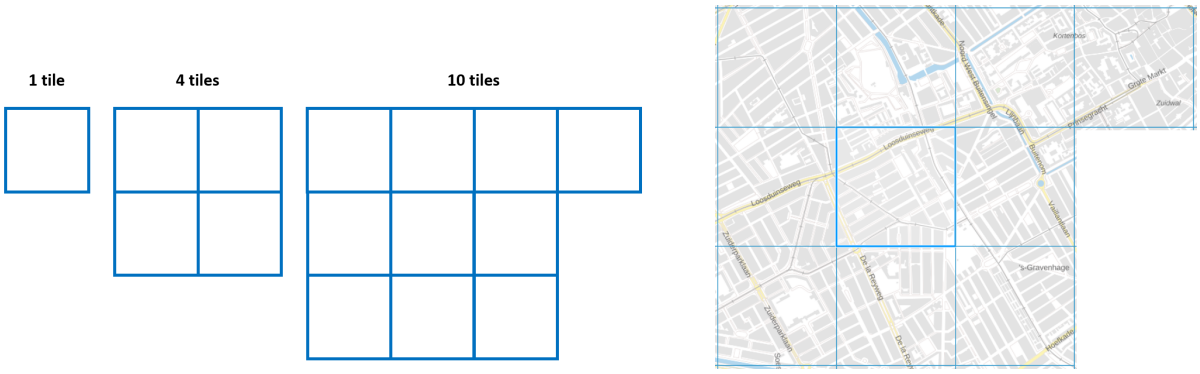
6.3.1. Scalability experiment

To assess the scalability for SolarBAG practically, three scenarios are executed. In each scenario, the model is run for a folder with a number of tiles. These numbers are respectively 1, 4 and 10 tile(s). A schematic of the spatial configuration of the tiles for each scenario is shown in Figure 6.12a. The spatial context of the scenarios is the city of The Hague. A map with the corresponding tiles overlaid is shown in Figure 6.12b. The program settings throughout all these scenarios are equal and listed in Table 6.2. To recap the meaning of these settings, refer to Section 5.2.2.

The specifications of the computer on which the three scenarios are run is given in Section 5.1.1.

Parameter	Value
No. of cores	10
LoD	2.2
Neighbour offset	20m
Sampling density	3m
Time configuration	1 year

Table 6.2.: The parameters of SolarBAG for the scalability assessment.



(a) Schematic configuration. (b) Map of the configuration.

Figure 6.12.: Configuration of the tiles.

The goal of the scalability assessment is to simply check whether the implemented program terminates. If for a certain amount of tiles, the program crashes, one can simply say that the program is not scalable. Another interesting part of the assessment is to keep track of the time it takes to process all the tiles for a scenario. The results per scenario are shown in Tables 6.3, 6.4 and 6.5. Alongside the time, some other characteristics per tile are stored. These are the tile ID as used in the 3D BAG data set, the spatial extent of the tile, the number of buildings the tile contains, the number of buildings inserted in the R-tree used for intersection detection and the number of neighbouring tiles.

Table 6.3 shows the results for the scenario with 1 tile. As expected, this scenario terminates. The number of buildings in the R-tree is of course the same as the number of buildings in the tile itself as no neighbouring tiles are present.

Table 6.4 shows the results for the scenario with 4 tiles. This scenario terminates as well. For a tile the number of buildings in the R-tree is not the same as the number of buildings in the tile anymore, because for each tile 3 neighbouring tiles are present. Therefore the execution of tile 1 in this scenario takes a bit longer than in the first scenario with only 1 tile. This shows that more buildings in the R-tree will result in slower processing times.

Table 6.5 shows the results for the most interesting scenario: the scenario with 10 tiles where the middle tile has neighbouring tiles in all directions. This scenario also terminates successfully. So, simply put, the implemented model is scalable. But there are some peculiarities that will be discussed.

When looking at the times, especially in Table 6.5, one can see that the execution times are not always relative to the number of buildings in the tile. Different building sizes are a factor that influences execution time. If we compare tile 5880 with tile 5873 in Table 6.5, we can see that tile 5880 contains less buildings than tile 5873, but takes about an hour longer to process. After visual inspection on the type of buildings in both tiles, it can be concluded that tile 5880

	Tile ID	Tile extent (m)	No. of buildings in tile	No. of buildings in R-tree	No. of neighbour tiles	Time (hh:mm:ss)
Tile 1	5801	652 x 686	1735	1735	0	02:18:53
Total time						**02:18:53**

Table 6.3.: Table showing the characteristics and the execution time per tile for a scenario with 1 tile.

	Tile ID	Tile extent (m)	No. of buildings in tile	No. of buildings in R-tree	No. of neighbour tiles	Time (hh:mm:ss)
Tile 1	5801	652 x 686	1735	2760	3	02:27:12
Tile 2	5802	652 x 638	1620	2294	3	01:12:54
Tile 3	5875	671 x 647	1236	1825	3	01:07:28
Tile 4	5876	658 x 639	955	1659	3	00:54:17
Total time						**05:41:51**

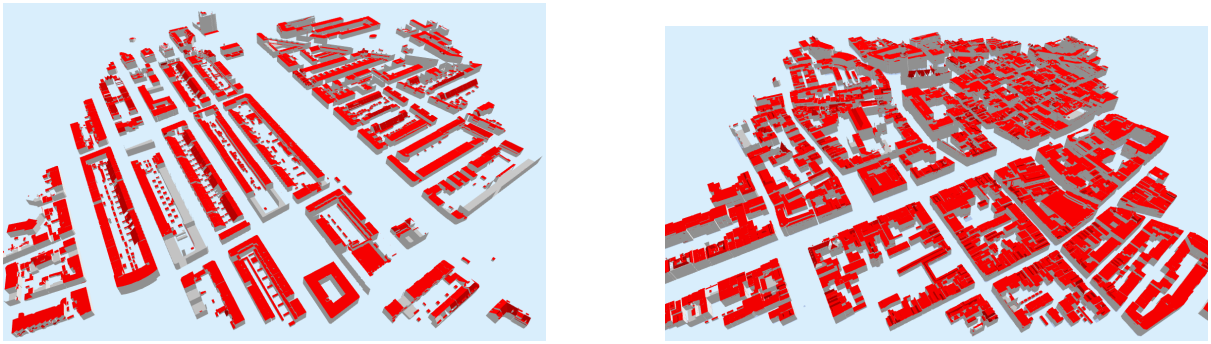
Table 6.4.: Table showing the characteristics and the execution time per tile for a scenario with 4 tiles.

	Tile ID	Tile extent (m)	No. of buildings in tile	No. of buildings in R-tree	No. of neighbour tiles	Time (hh:mm:ss)
Tile 1	5801	652 x 686	1735	2760	3	02:24:26
Tile 2	5802	652 x 638	1620	2646	5	01:11:06
Tile 3	5805	636 x 635	1688	2408	3	00:47:08
Tile 4	5872	644 x 654	1724	2644	5	00:48:23
Tile 5	5873	650 x 648	1400	1976	3	00:45:50
Tile 6	5875	671 x 647	1236	2412	8	01:06:04
Tile 7	5876	658 x 639	955	1951	5	00:52:54
Tile 8	5877	658 x 659	888	1667	4	00:44:12
Tile 9	5878	636 x 679	1352	1981	6	01:02:03
Tile 10	5880	694 x 646	1124	1475	2	01:46:05
Total time						**11:28:11**

Table 6.5.: Table showing the characteristics and the execution time per tile for a scenario with 10 tiles.

contains much larger buildings than tile 5873. Tile 5873 covers a neighbourhood with mainly house blocks and small sheds, while tile 5880 covers large building blocks in the city centre. In Figure 6.13, the buildings in both tiles are shown where the red surfaces represent the roof surfaces. In this case larger buildings mean larger roof surfaces. If a building has a larger roof surface this means that more points will be sampled leading to more computations for the building and altogether to more computations for the tile.

Furthermore the parameter settings, and especially the neighbour offset, are chosen as listed in Table 6.2 in order to make processing times relatively fast. A neighbour selection offset of 20 meters is quite low as compared to the neighbour offset of 150 meters used for the quality assessment. Tall buildings, located more than 100 meters away that might cast shadows on another building, will not be taken into account with these settings. However, the idea of the



(a) Buildings in tile 5873.

(b) Buildings in tile 5880.

Figure 6.13.: Tile 5873 versus tile 5880.

scalability assessment is not to compute accurate solar radiation values, but to assess whether SolarBAG is scalable for large amounts of data. Therefore, a neighbour offset of 20 meters is chosen to reduce execution times.

If the scenarios would be run on a PC with better specifications, such as a higher RAM or more multiprocessing cores the execution times would be lower, i.e. the program will run faster.

The runtime of the implemented model is also compared with the runtime of the 'Area Solar Radiation' tool in ArcGIS. For this, tile 5873 is taken and rasterised and used as input for the tool in ArcGIS. The settings of the ArcGIS tool are the same as in Section 6.2.2. The time it takes to process the raster and create the resulting raster is 00:06:55. Compared to the time for the entry of tile 5873 in Table 6.5, this is about 6,5 times faster. This is caused by the fact that executing the ray tracing operation on many building objects in the implemented vector model is more computational-intensive than the viewshed operation in the raster model.

6.3.2. Other scalability issues

In the previous section, it is shown that SolarBAG is scalable for multiple tiles in CityJSON format. However, scalability covers much more than just telling that a program can be executed for large data in a certain format. What happens when the tiles are not rectangles of the same size and what happens if the 3D building models are not stored in tiles in CityJSON format, but in a DBMS for instance? And, if one wants to actually use SolarBAG for all the tiles in the 3D BAG data set, is this possible right away and how much time would it take? These issues will be adressed in this section.

In the scalability experiments, the tiles are all of the same size. If the tiles would be of a different size, SolarBAG will not behave differently. It is still possible to process the tiles with their correct neighbouring tiles as the selection of the neighbouring tiles are based on the buffer applied to the current tile. As shown in Figure 6.14, the buffer will also cover the spatial extent of the tiles of different sizes. As explained in Section 5.2.2, the bounding box including the buffer of the current tile will be used as query to the R-tree to select the bounding boxes of the neighbouring tiles.

In the case the 3D building models are not stored in tiles, but all in one batch such as a DBMS a different approach in processing the 3D building models as this is not possible with the current implementation. A way to do this is to split up the buildings stored in the DBMS into smaller areas by using an R-tree indexing structure. Then, each smaller area can be processed as before.

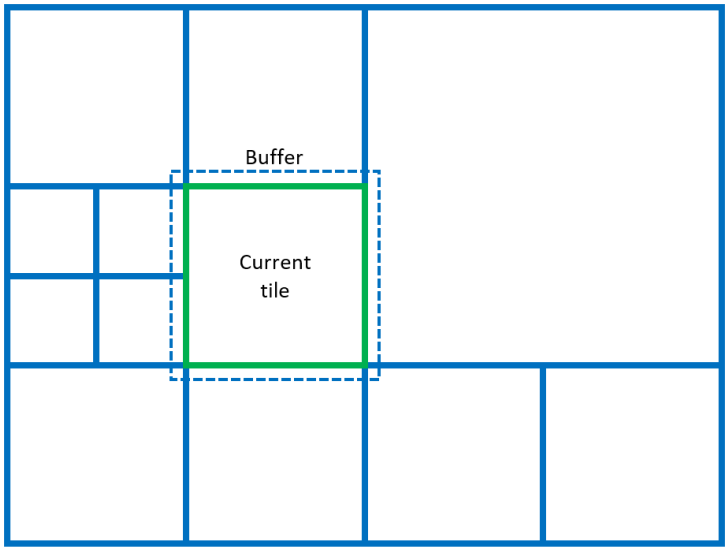


Figure 6.14.: Tile configuration with different tile sizes.

In the case, the building models are not stored in CityJSON files but in CityGML[9] or Wavefront OBJ[10] format, a different file parser and writer should be implemented, but the processing of the building geometries itself could be done more or less similar.

The question whether SolarBAG is capable to run for all the buildings in The Netherlands is interesting. In theory, the answer is yes. A folder with all the tiles could be given as input to the program and after (quite) a while, a folder with the enriched tiles will be given as output. The practical downsides, however, are that it is quite cumbersome to prepare a folder with all the tiles being pre-processed, that the process will take a lot of time and that there is a chance that the system will break on unforeseen exceptions or errors that are not covered in the scalability experiments in the previous section.

In order to compute the total execution time for processing all tiles in The Netherlands, the resulting time from the scalability experiment for 10 tiles could be extrapolated to estimate the execution time for all the tiles in The Netherlands. The easiest way to do this, is to count all the buildings in the 10 tiles and count all the buildings in all the tiles. Then, compute the factor between these two counts and multiply this factor by the execution time for the 10 tiles. The result is approximately **338 days**. One should be aware that extrapolation has a high uncertainty and a high risk of producing meaningless results. Therefore, this number is just seen as a rough estimation.

[9] https://www.ogc.org/standards/citygml
[10] http://paulbourke.net/dataformats/obj/

7. Conclusion

This chapter first discusses the limitations of the research, followed up by the conclusion revising and answering the research objectives introduced in Section 1.1. Afterwards, recommendations are listed as future work.

7.1. Discussion

Although the methodology carried out in this research is considered to be of sufficient quality, there are some comments to be made. In this section the limitations and challenges of the research are discussed.

First, it is worth noting that the ground truth comparison and the comparison with ArcGIS are not completely fair comparisons. For the ground truth comparison, global radiation from the ground truth is compared with direct radiation from the solarpy solar model, but in such a way that the comparison makes sense. However, it would be better to compare the same types of radiation with each other.

The comparison with ArcGIS has another reason of not being completely fair. Even though a density value, serving as resolution, can be set as parameter to do point sampling, this is not the actual resolution of the resulting solar potential values. In the experiments a sampling density of 3 meters is used, but in the end the values are aggregated per triangle leading to a loss of resolution. For example, if for a triangle 15 points are sampled, only the statistical values such as average, minimum and maximum are stored for the triangle. If for instance, a density of 1 meter is chosen, this will lead to more points sampled for the triangle, but in the end only the statistical values for the points on the triangle are stored, leading to a loss of resolution again. Simply put, the resolution is not very relevant for the final aggregated solar potential value. It is relevant for the processing time, because a higher sampling density results in slower processing times as more points are sampled that need to be processed.

Another limitation related to comparing results has to do with a lack of experiments on parameter tweaking. Parameter tweaking gives insights into the processing time of the implemented model and into the accuracy of the results for varying parameter settings. In this way a trade-off between accuracy and performance could be established.

The second type of limitation to discuss is the usage of workarounds instead of sound solutions to solve a couple of issues. An issue that was posed with triangulating the 3D BAG tiles within the model, is solved by doing the triangulation as a pre-processing step. However, it would be much more convenient to have it integrated within the model, making the model easier to use. Another issue is the error raised when finding empty lists of grid points after performing point sampling on a triangle. This is now solved by just skipping this triangle instead of finding the cause of the problem. Furthermore, the implemented model is still sensitive to issues in geometry processing. There are still warnings raised when running the model. An example is the warning that raises an error on extracting eigenvalues in the `vtkMath` module of package

PyVista. According to other users, this gives unpredictable behaviour. Luckily, the program does not crash so for now it is not much of an issue.

The third type of limitation to discuss has to do with the processing time of the implemented model. It is still quite slow to process a whole tile. The neighbour filtering and data organisation are not optimal. At this moment, the bottleneck of the program is within the ray-tracing part. To determine whether a building is put into shadow by another neighbouring building, a ray trace operation needs to be performed on the neighbouring for a specific timestamp. To make this process faster, the number of neighbouring buildings for which ray tracing needs to be performed should be as small as possible. At this moment, a lot of unnecessary buildings are ray traced making the program slow. So, the improvement can be achieved by implementing a better neighbouring filter algorithm.

This improvement in neighbour filtering would be achieved by selecting neighbours from the R-tree to perform ray tracing on based on the extent of the ray of a certain timestamp represented by a number of bounding boxes. This should result in less neighbouring buildings to ray trace than with the current solution. However, it was not possible to do this while multiprocessing, because there is no way to store a pointer to the R-tree as the R-tree is not thread-safe[1]. In practice, querying an R-tree while multiprocessing will not give an error, but it will result in an empty R-tree query result. Instead another approach to only ray trace neighbouring building within the extent of a ray of a certain timestamp on a certain date was tried. This can be achieved by querying the R-tree with the extent of the ray for each specific timestamp before performing multiprocessing and storing the returned neighbouring buildings per timestamp. But, in practice this turned out to be even less efficient than the current method implemented, because the time-consuming operation of create sun rays to query on the R-tree was operated on a single processor for each building.

The fourth type of limitation to discuss has to do with the scalability of SolarBAG. The scalability for the whole of The Netherlands is questionable. Will there be unforeseen errors in the geometries of the buildings in other tiles and will the computer memory and processors be able to cope with all that data? This is considered an uncertainty. Also, SolarBAG is only capable of processing tiles from the 3D BAG in CityJSON format, so SolarBAG is not scalable to other file types. More discussion on these scalability issues is found in Section 6.3.2.

The fifth type of limitation is related to the physical circumstances of installing solar panels on roof surfaces. The fact that parts of the roof surfaces might not be suitable for installation of solar panels is not taken into account. Roof surfaces can consist of obstacles such as chimneys, antennas or dormers, or roofs are simply too small to install panels. Also, the solar radiation captured by solar panels need to be converted to DC and AC in order to be used by household appliances. With this conversion, there is a loss of energy which need to be taken into account.

Within this research, walls are not taken into account for the computation of solar radiation. SolarBAG could easily be extended to also compute the solar potential on walls. To compute it more efficiently, the method by Jaugsch and Löwner [2016] using observer point columns could be used. Just like roof surfaces, wall surfaces contain obstacles such as windows, doors and balconies on which solar panels cannot be installed. Furthermore, walls have a higher chance to be in shadow than roof surfaces.

A next limitation is related to the visualisation of the results. The solar potential values on each triangle in the resulting CityJSON file cannot directly be represented with a colour scheme in Ninja. For this, separate code should be written to convert the CityJSON file to '.obj' or '.vtk'

[1] https://gis.stackexchange.com/questions/304711/python-rtree-and-parallelized-code

format for instance. The conversion of CityJSON to '.vtk' is done in this research. Besides that, it might be better to change the the unit of the solar potential values from $Wh/m^2/year$ to $kWh/m^2/year$ as the yearly solar values are quite large. This could be done by dividing the current values by 1000. It would make the value better readable, but it does not change the value itself. On top of that, it would be useful to include an average solar potential value per triangular surface with unit $kWh/year$ in order to get insight into the aggregated value for the surface. But this is just a design choice and depends on the use case of the data.

The last limitation to discuss is not really a limitation, but more a discussion whether the implemented model could also be applied to other city models in the world. The title of this research suggests an efficient solar potential estimation of 3D buildings with the 3D BAG taken as use case. So, to what extent is the implemented model applicable to other data sets? This can be answered quite easily, as the only constraint is that the city model should be stored in a valid CityJSON file containing triangulated building geometries of LoD 2.0 or higher with roof surfaces as semantic surfaces. If a folder with one or multiple CityJSON files are then given as input to the program, a CityJSON file enriched with solar potential values per roof surface will be the output.

7.2. General conclusion

In this section, the main research question will be answered by giving answers to the 4 sub-questions. The main research question is:
How can the solar potential of vector buildings in large 3D city models, such as the 3D BAG data set, be computed efficiently?

The sub-questions are:

1. *How can spatial indexing be used to speed up shadow casting computations on the 3D BAG vector data set?*
2. *What simplifications in the solar irradiation model can be applied?*
3. *How can the solar irradiation model be implemented in Python by using open source libraries and open data?*
4. *How should the computer memory be managed while processing the buildings stored in tiles in the 3D BAG data set?*

Below, all the sub-questions are answered.

Sub-question 1:
How can spatial indexing be used to speed up shadow casting computations on the 3D BAG vector data set?

At first, a general BVH was intended to be used as indexing structure. However, it appeared to be more useful and easier to use a specific type of BVH, namely an R-tree. The R-tree hierarchically stores the geometry of each building in the 3D BAG data set encapsulated in a bounding box. When buildings in a certain geographic extent are requested, first the bounding boxes containing these buildings and located within the geographic extent are queried. This ensures fast and easy look-up of the actual buildings as many bounding boxes can be skipped and only the buildings in the relevant bounding boxes need to be traversed. Implementation-wise, the R-tree can be used as spatial indexing by using the 'Rtree' library in Python.

Sub-question 2:
What simplifications in the solar irradiation model can be applied?

Computing the solar potential for all buildings in the 3D BAG at hourly intervals for each day in the year takes a lot of time. A simplification applied in this research is to take one day for each month to compute the solar potential. This is the 21st day of each month as this will be the longest day of the year in June and the shortest day of the year in December, and therefore they compensate each other. Instead of computing the solar potential for each day in the year, taking one day per month, so only 12 days a year, will drastically decrease the computation time by a approximately factor of 30. The solar potential outcomes are still realistic as proved by the comparison with the solar radiation module available in ArcGIS. There are some differences detected at corners of building roofs, caused by resolution differences. Also, there are still some bugs, but the resulting solar potential values per roof surface are at least an indication.

Furthermore, the process of shadow casting takes a lot of computation time when this needs to be executed for a lot of buildings. Therefore the number of buildings for which shadow casting has to be applied, needs to be as small as possible. In this research, the neighbouring buildings of a building are considered as potentially putting the building into shadow. However, not every building in the neighbourhood will cast a shadow on this building. Therefore, filters are applied based on the distance of the neighbouring building to the considered building, the height difference and the orientation of the neighbouring buildings. After applying these filters, only neighbouring buildings with a high chance of putting the considered building into shadow are kept and subjected to shadow casting operations. This has as benefit for the model that the computation time is improved. Moreover, applying filters will not influence the resulting solar potential values of the building as only buildings that would not cast shadows to the building are removed. Therefore, the solar potential outcomes after applying these filters are still realistic.

Sub-question 3:
How can the solar irradiation model be implemented in Python by using open source libraries and open data?

The solar irradiation model is successfully implemented in Python with the usage of open source libraries within Python and the large open data set 3D BAG, containing 3D building models of all buildings in The Netherlands. The implemented methodology is presented in Chapter 5. It is possible to enrich a 3D BAG tile, stored in CityJSON format, containing building geometries of LoD 2.2 with statistics on yearly solar potential values per triangular surface. The most important open source libraries used are NumPy, PyVista, solarpy, cjio and RTree. NumPy and PyVista are both used for data organisation and mathematical computations. Solarpy is used to compute the direct radiation for each roof surface. Cjio is used to read and write CityJSON files and RTree is used to wrap the building geometries in R-trees.

Sub-question 4:
How should the computer memory be managed while processing the buildings stored in tiles in the 3D BAG data set?

Processing big data requires a lot of processing power and computer memory. Therefore smart solutions are necessary to make sure that the PC's resources are able to cope with this. For this research the solution for managing the computer memory lies in subsequently storing and processing the required spatial data for each tile. When processing a tile, only the buildings in the tile and buildings within a buffer from directly neighbouring tiles are stored in memory and in an R-tree. When getting to the next tile, the memory is cleaned and the buildings in the following tile are loaded, stored and processed. This ensures that the PC can always execute the implemented model for increasing number of tiles.

7.3. Future work

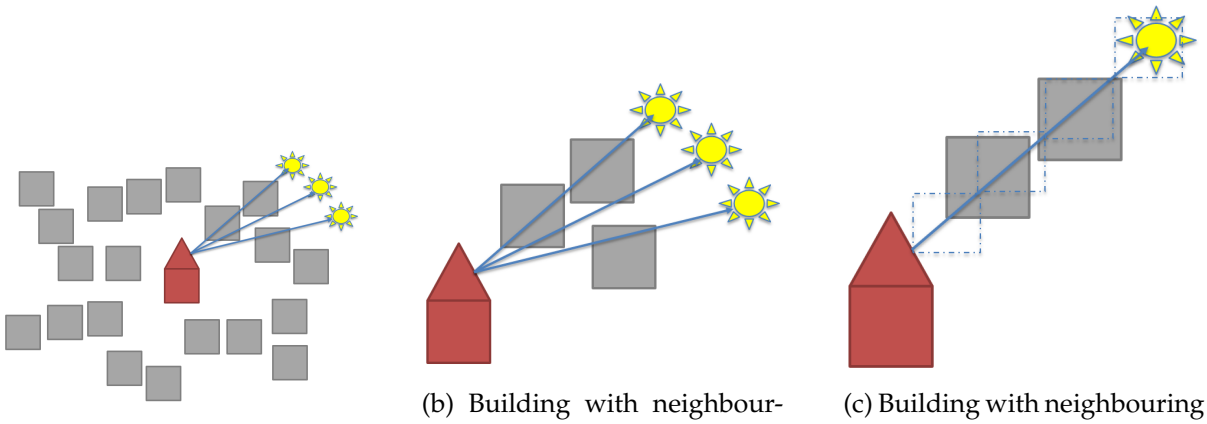
This section lists some recommendations for future work with respect to this research. From the discussion in Section 7.1, recommendations can be derived. The important topics for future work to outline are implementation improvements, solar radiation model extensions, and additional analyses.

7.3.1. Efficient implementation improvements

Efficient implementation improvements have as goal to make SolarBAG more computational and memory efficient. A couple of ideas to make SolarBAG more computational efficient are discussed.

The first straightforward computational improvement would be to make use of the GPU to manipulate geometries and perform ray tracing for shadow casting operations. As stated by Viana-Fons et al. [2020], usage of the GPU will accelerate shadow casting operations and simple geometric computations. A straightforward improvement on memory limitation would be to make use of a more powerful computer to run SolarBAG for a lot of tiles.

In order to reduce the amount of ray tracing operations to be performed on neighbouring buildings potentially casting a shadow on another building, better filters need to be applied. Currently, the neighbouring buildings are filtered on their distance, height and orientation with respect to the currently processed building. On these buildings the `ray_trace()` function is applied to find out whether they are actually blocking a sun ray towards a another building. This operation is very computational-intensive, so the amount of neighbouring buildings subject to ray tracing need to be as small as possible. A way to achieve this is to only perform the ray trace operation on neighbouring buildings that are within the spatial extent of a couple of bounding boxes representing the ray. A schematic visualisation of this process is shown in Figure 7.1. The resulting buildings can be retrieved by querying the R-tree with bounding boxes representing the ray.



(a) Building with neighbouring buildings.

(b) Building with neighbouring buildings close to a sun ray.

(c) Building with neighbouring buildings that are subject to ray tracing.

Figure 7.1.: Improvement of filter operation on neighbouring buildings

Another way to increase the performance of SolarBAG is to apply mesh simplification to reduce the number of triangles contained in 3D building models. Zhou et al. [2021] make use of this technique to remove unnecessary vertices or meshes. This has as effect that a building will consist of fewer triangular surfaces, increasing performance.

7.3.2. Solar radiation model extensions

In order to better simulate real-life situations, the solar radiation model could be extended in several ways. The first way is to incorporate diffuse and reflected radiation into the solar radiation model. Right now, only direct radiation is taken into consideration as it has a contribution of 77% to the global radiation. The remaining 23% is accounted by diffuse and reflected radiation. Incorporating diffuse and reflected into the solar radiation model, by extending the formula in equation 4.1 will give a better representation of the solar potential of a building roof.

As SolarBAG takes vector data as input, wall surfaces are also modelled. This means that the solar potential of buildings in the 3D BAG could also be computed for walls. This extension is quite easy to implement, as the same approach could be taken as for roofs. However, the chance that a wall surface is put into shadow by another building is higher than for roof surfaces. Therefore, smart and efficient approaches should be taken, such as described by Jaugsch and Löwner [2016]. They use observer point columns to efficiently execute shadow casting on wall surfaces.

Instead of downloading CityJSON files from the 3D BAG data set and use them as input to SolarBAG, the building data could also be accessed from a database. 3D BAG has the option to dump the building geometries in a PostgreSQL database. PostgreSQL has the PostGIS extension which is suitable to storing and managing large geospatial databases. Making use of a database, solves the cumbersome process of downloading and preprocessing the CityJSON files before inputting it to SolarBAG.

7.3.3. Additional analyses

Next to improvements and extensions, additional analyses on the current solar radiation model could be performed. Right now, an R-tree is created per tile in which buildings are hierarchically grouped together in bounding boxes. It might also be efficient to create an R-tree per building in which the triangles forming the building are grouped in an R-tree.

In this research, a lot of parameters can be set for the solar radiation model. It is not completely investigated what the effect of these parameters are on the accuracy and performance of the implementation system. Therefore a thorough analysis by parameter tweaking could be performed to illustrate the overall impact.

A. Reproducibility self-assessment

A.1. Marks for each of the criteria

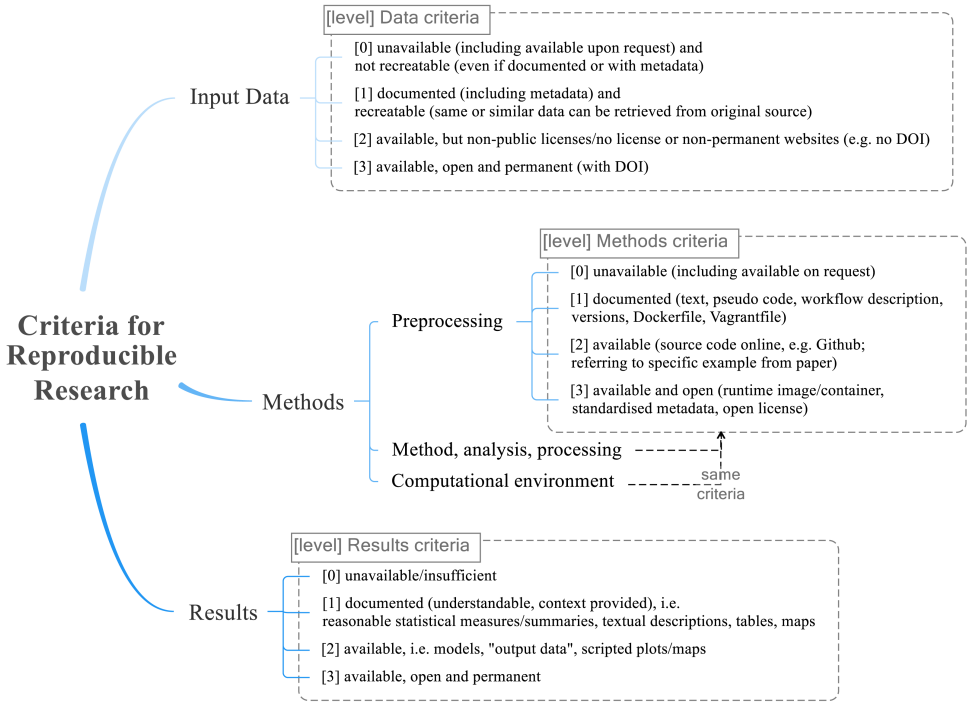


Figure A.1.: Reproducibility criteria to be assessed.

A.2. Reproducibility marks

Criteria	Mark
Input data	3
Preprocessing	2
Methods	2
Computational environment	2
Results	1

A.3. Self-reflection

In my research, I have made use of the large open data set 3D BAG containing all the buildings geometries of The Netherlands. The tiles in which the buildings are stored are available to download in numerous formats or to use in web based systems or databases. Therefore, the mark for input data is the highest possible.

The implemented methodology is made available on GitHub[1]. A readme with how-to-use the program is provided. The preprocessing steps are explained as well as what systems to use and how to run the code. Furthermore, the methodology is intended to explain the steps in such a way that a skilled reader and programmer could implement the system themselves.

The results are not all available in the GitHub repository. They are documented in the thesis. Statistical measures are not outstanding.

[1] https://github.com/robinjo78/solarBAG

B. Additional Figures

B.1. ParaView screenshots

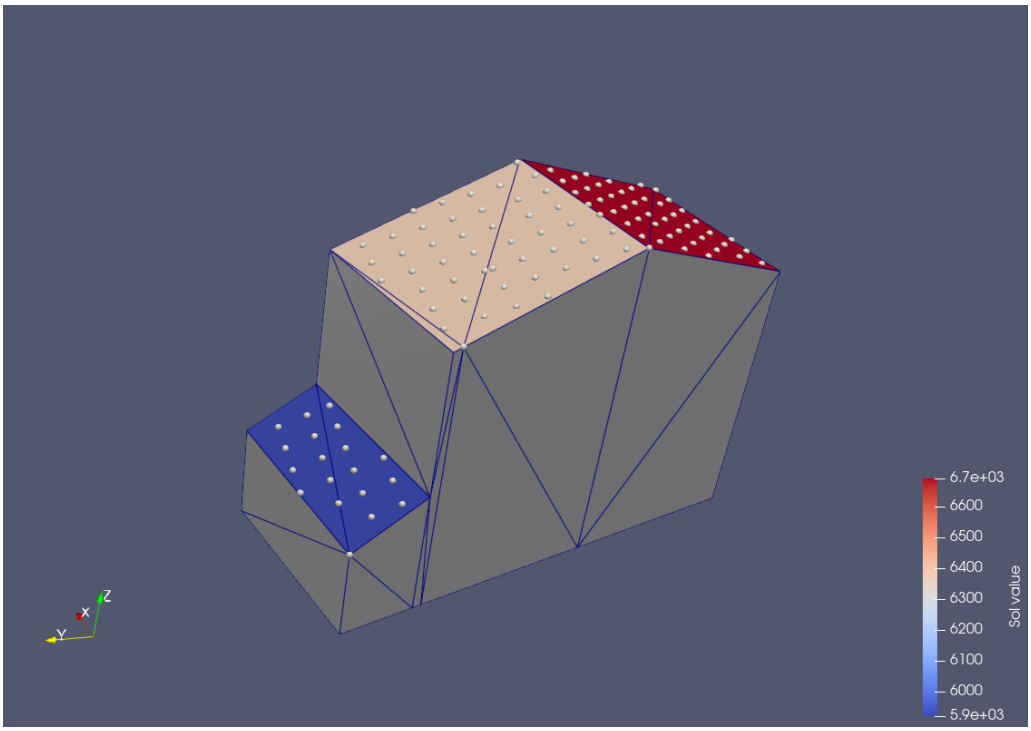


Figure B.1.: Early in research point sampling visualisation, with solar radiation stored per triangle.

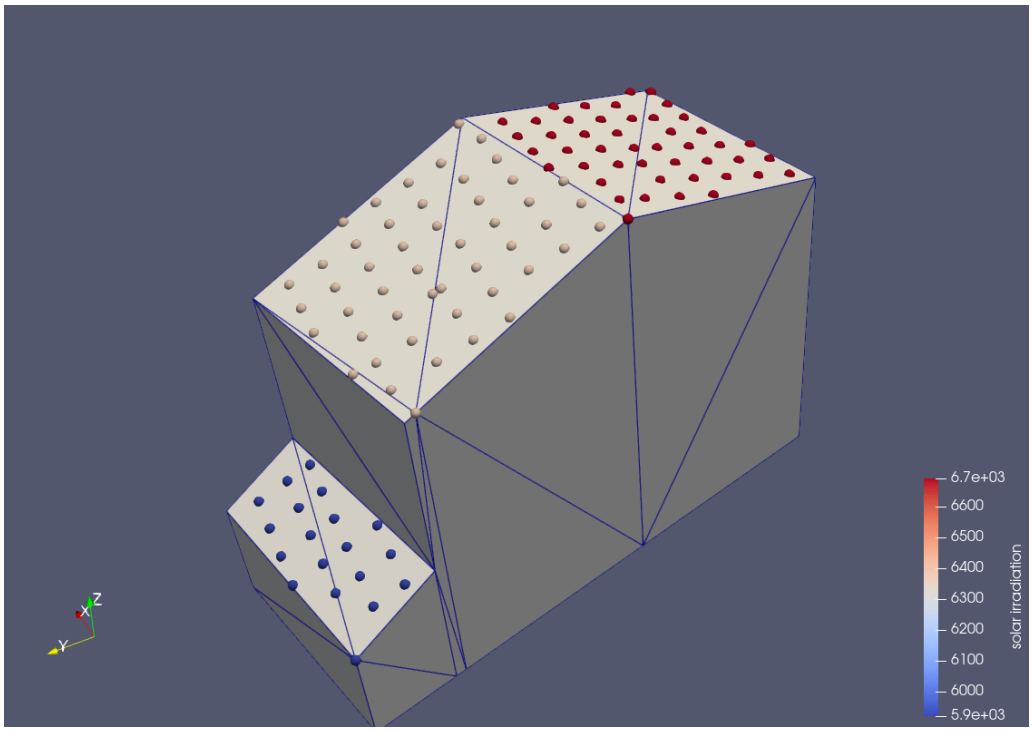


Figure B.2.: Early in research point sampling visualisation, with solar radiation stored per point.

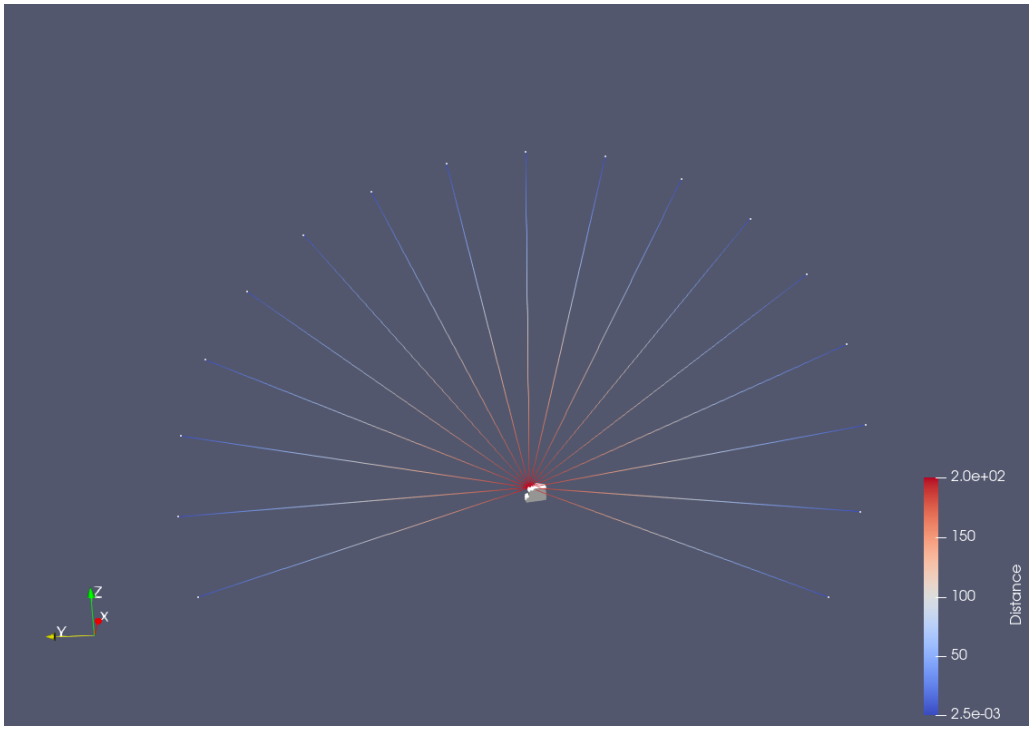


Figure B.3.: Visualisation of the sun rays throughout a day pointing to a grid point.

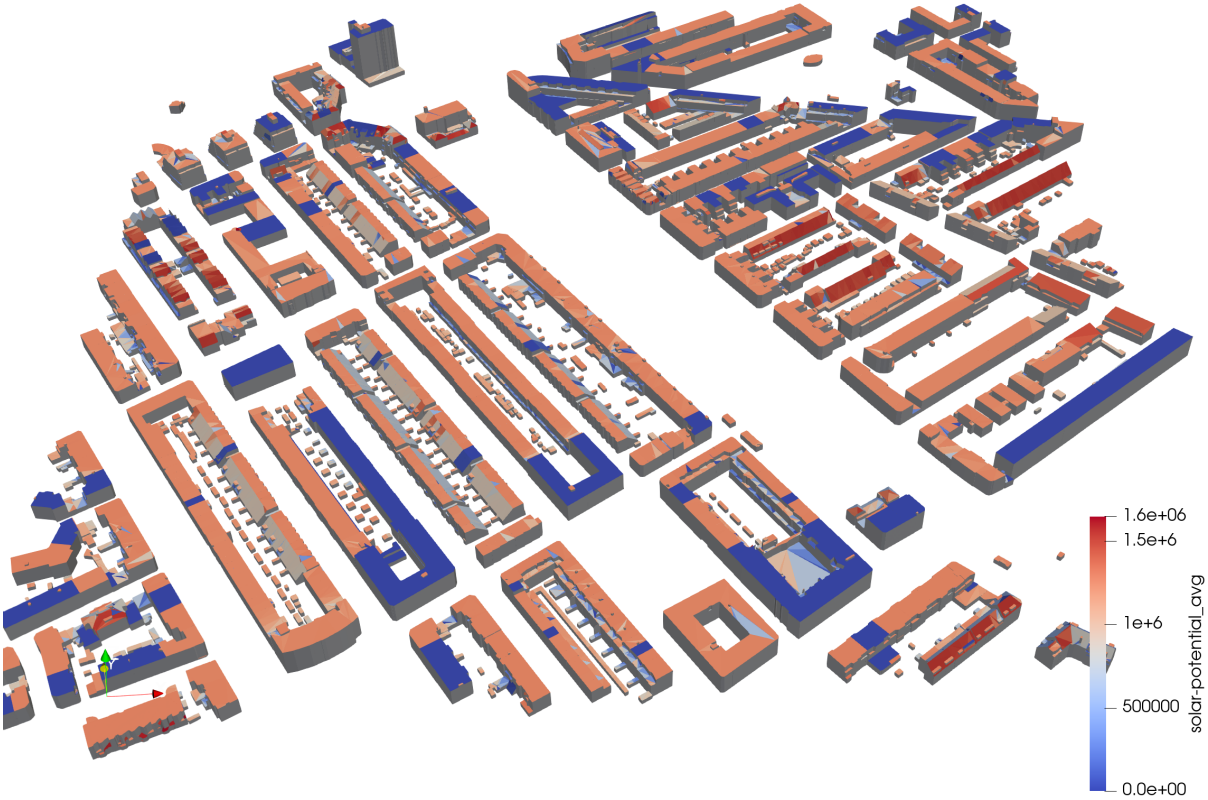


Figure B.4.: Visualisation of average solar potential values (in Wh/m^2) per triangle in ParaView - enlarged.

Bibliography

R. Alexandre. DEM Spatial Resolution - What does this mean for flood modellers? 2018. URL https://www.jbarisk.com/news-blogs/dem-spatial-resolution-what-does-this-mean-for-flood-modellers/.

ArcGIS. Area Solar Radiation (Spatial Analyst), 2016a. URL https://pro.arcgis.com/en/pro-app/2.8/tool-reference/spatial-analyst/area-solar-radiation.htm#:~:text=Typically%20observed%20values%20are%200.6,overhead)%20and%20for%20sea%20level.

ArcGIS. An overview of the Solar Radiation tools, 2016b. URL https://desktop.arcgis.com/en/arcmap/10.3/tools/spatial-analyst-toolbox/an-overview-of-the-solar-radiation-tools.htm.

F. Biljecki, G. B. M. Heuvelink, H. Ledoux, and J. Stoter. Propagation of positional error in 3D GIS: estimation of the solar irradiation of building roofs. *International Journal of Geographical Information Science*, 29(12):2269–2294, 2015. ISSN 1365-8816 1362-3087. doi: 10.1080/13658816.2015.1073292.

F. Biljecki, H. Ledoux, and J. Stoter. An improved lod specification for 3d building models. *Computers, Environment and Urban Systems*, 59:25–37, 2016. ISSN 01989715. doi: 10.1016/j.compenvurbsys.2016.04.005.

M. C. Brito, R. Amaro e Silva, and S. Freitas. Characteristic declination—a useful concept for accelerating 3D solar potential calculations. *Energy Technology*, 9(3), 2021. ISSN 2194-4288 2194-4296. doi: 10.1002/ente.202000943.

A. Calcabrini, H. Ziar, O. Isabella, and M. Zeman. A simplified skyline-based method for estimating the annual solar energy potential in urban environments. *Nature Energy*, 4(3):206–215, 2019. ISSN 2058-7546. doi: 10.1038/s41560-018-0318-6.

N. Charles. solpy, 2015. URL https://pypi.org/project/solpy/.

G. Desthieux, C. Carneiro, R. Camponovo, P. Ineichen, E. Morello, A. Boulmier, N. Abdennadher, S. Dervey, and C. Ellert. Solar Energy Potential Assessment on Rooftops and Facades in Large Built Environments Based on LiDAR Data, Image Processing, and Cloud Computing. Methodological Background, Application, and Validation in Geneva (Solar Cadaster), journal = Frontiers in Built Environment. 4, 2018. ISSN 2297-3362. doi: 10.3389/fbuil.2018.00014.

J. A. Duffie and W. A. Beckman. *Solar Engineering of Thermal Processes*. John Wiley Sons, Incorporated, Somerset, United States, 2013. ISBN 9781118418123. URL http://ebookcentral.proquest.com/lib/delft/detail.action?docID=1162079.

R. Erdélyi, Y. Wang, W. Guo, E. Hanna, and G. Colantuono. Three-dimensional solar radiation model (SORAM) and its application to 3-D urban planning. *Solar Energy*, 101:63–73, 2014. ISSN 0038092X. doi: 10.1016/j.solener.2013.12.023.

S. Freitas, C. Catita, P. Redweik, and M. C. Brito. Modelling solar potential in the urban environment: State-of-the-art review. *Renewable and Sustainable Energy Reviews*, 41:915–931, 2015. ISSN 13640321. doi: 10.1016/j.rser.2014.08.060.

P. Fu and P. Rich. A geometric solar radiation model with applications in agriculture and forestry. 2002.

D. Giannelli, C. León-Sánchez, and G. Agugiaro. Comparison and evaluation of different gis software tools to estimate solar irradiation. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, V-4-2022:275–282, 2022. ISSN 2194-9050. doi: 10.5194/isprs-annals-V-4-2022-275-2022.

GRASS Development Team. *Geographic Resources Analysis Support System (GRASS GIS) Software*. Open Source Geospatial Foundation, USA, 2020. URL https://grass.osgeo.org.

R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, 2005. ISBN 9780120887996.

W. Hetrick, P. Rich, F. Barnes, and S. Weiss. GIS-based solar radiation flux models. *American Society for Photogrammetry and Remote Sensing Technical papers*, 3:132–143, 1993.

J. Hofierka and M. Ri. The solar radiation model for open source GIS: Implementation and applications. 2002.

J. Hofierka and M. Suri. *r.sun - Solar irradiance and irradiation model*, 2007. URL https://grass.osgeo.org/grass78/manuals/r.sun.html.

J. Hofierka and M. Zlocha. A new 3-D solar radiation model for 3-D city models. *Transactions in GIS*, 16(5):681–690, 2012. ISSN 13611682. doi: 10.1111/j.1467-9671.2012.01337.x.

F. Jaugsch and M. O. Löwner. Estimation of solar energy on vertical 3D building walls on city quarter scale. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W2:135–143, 2016. ISSN 2194-9034. doi: 10.5194/isprs-archives-XLII-2-W2-135-2016.

E. Kabir, P. Kumar, S. Kumar, A. A. Adelodun, and K.-H. Kim. Solar energy: Potential and future prospects. *Renewable and Sustainable Energy Reviews*, 82:894–900, 2018. ISSN 13640321. doi: 10.1016/j.rser.2017.09.094.

J. Kerkhof. Bijna 1 op de 5 woningen heeft zonnepanelen, 2022. URL https://www.independer.nl/energie/info/onderzoek/zonnepanelen-2022.

KNMI. Automatische weerstations. URL https://www.knmi.nl/kennis-en-datacentrum/uitleg/automatische-weerstations.

J. B. Kodysh, O. A. Omitaomu, B. L. Bhaduri, and B. S. Neish. Methodology for estimating solar potential on multiple building rooftops for photovoltaic systems. *Sustainable Cities and Society*, 8:31–41, 2013. ISSN 22106707. doi: 10.1016/j.scs.2013.01.002.

H. Ledoux, K. Arroyo Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis. CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards*, 4(1), 2019. ISSN 2363-7501. doi: 10.1186/s40965-019-0064-0.

J. Liang and J. Gong. A sparse voxel octree-based framework for computing solar radiation using 3D city models. *ISPRS International Journal of Geo-Information*, 6(4), 2017. ISSN 2220-9964. doi: 10.3390/ijgi6040106.

J. Liang, J. Gong, W. Li, and A. N. Ibrahim. A visualization-oriented 3D method for efficient computation of urban solar radiation based on 3D–2D surface mapping. *International Journal of Geographical Information Science*, 28(4):780–798, 2014. ISSN 1365-8816 1362-3087. doi: 10.1080/13658816.2014.880168.

J. Liang, J. Gong, J. Zhou, A. N. Ibrahim, and M. Li. An open-source 3D solar radiation model integrated with a 3d geographic information system. *Environmental Modelling Software*, 64: 94–101, 2015. ISSN 13648152. doi: 10.1016/j.envsoft.2014.11.019.

J. Liang, J. Gong, X. Xie, and J. Sun. Solar3D: An open-source tool for estimating solar radiation in urban environments. *ISPRS International Journal of Geo-Information*, 9(9), 2020. ISSN 2220-9964. doi: 10.3390/ijgi9090524.

A. Majercik, C. Crassin, P. Shirley, and M. McGuire. A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering. *Journal of Computer Graphics Techniques*, 7(3), 2018.

M. H. Mon and M. M. Than. Location-Based R-Tree and Grid-Index for Nearest Neighbor Query Processing. *Proceedings of 2015 International Conference on Future Computational Technologies*, pages 136 – 142, 2015. doi: 10.17758/ur.u0315241.

OGC. OGC City Geography Markup Language (CityGML) En-coding Standard. 2012.

R. Peters, B. Dukai, S. Vitalis, J. van Liempt, and J. Stoter. Automated 3d reconstruction of lod2 and lod1 models for all 10 million buildings of the netherlands. *Photogrammetric Engineering Remote Sensing*, 88(3):165–170, 2022. ISSN 0099-1112. doi: 10.14358/pers.21-00032r2.

M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: Primitives and Intersection Acceleration*. 2004 - 2021.

C. E. S. R. I. Redlands. ArcGIS desktop: Release 10, 2011.

P. M. Redweik, C. Catita, and M. C. Brito. 3D local scale solar radiation model based on urban lidar data. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XXXVIII-4/W19:265–269, 2012. ISSN 2194-9034. doi: 10.5194/isprsarchives-xxxviii-4-w19-265-2011.

B. Rudisill. The Solar Resource. 2010. URL http://mcensustainableenergy.pbworks.com/w/page/20638192/The%20Solar%20Resource.

X. Song, Y. Huang, C. Zhao, Y. Liu, Y. Lu, Y. Chang, and J. Yang. An approach for estimating solar photovoltaic potential based on rooftop retrieval from remote sensing images. *Energies*, 11(11), 2018. ISSN 1996-1073. doi: 10.3390/en11113172.

M. B. d. Souza, A. Tonolo, R. L. Yang, G. M. Tiepolo, and J. Urbanetz Junior. Determination of diffused irradiation from horizontal global irradiation - study for the city of Curitiba. *Brazilian Archives of Biology and Technology*, 62(spe), 2019. ISSN 1678-4324 1516-8913. doi: 10.1590/1678-4324-smart-2019190014.

C. Spitters, H. Toussaint, and J. Goudriaan. Separating the diffuse and direct component of global radiation and its implication for modeling canopy photosynthesis. *Agricultural and Forest Meteorology*, 38:217–229, 1986.

N. Stendardo, G. Desthieux, N. Abdennadher, and P. Gallinelli. GPU-enabled shadow casting for solar potential estimation in large urban areas. application to the solar cadaster of greater Geneva. *Applied Sciences*, 10(15), 2020. ISSN 2076-3417. doi: 10.3390/app10155361.

Stouch Lighting Staff. Electricity and Energy Terms In Lighting (J, kW, kWh, Lm/W). 2015. URL https://www.stouchlighting.com/blog/electricity-and-energy-terms-in-led-lighting-j-kw-kwh-lm/w.

U.S. Department of Energy. Solar Radiation Basics. 2022.

K. van Groesen. Bijna 1 op de 5 woningen heeft zonnepanelen. 2022. URL https://weblog.independer.nl/persbericht/bijna-1-op-de-5-woningen-heeft-zonnepanelen/?referrer=https%3A%2F%2Fwww.google.com%2F&referer=https%3A%2F%2Fwww.google.com%2F.

P. Van Oosterom. Spatial access methods. *Geographical Information System Principles*, 1:385 – 400, 1999.

J. D. Viana-Fons, J. Gonzálvez-Maciá, and J. Payá. Development and validation in a 2D-GIS environment of a 3D shadow cast vector-based model on arbitrarily orientated and tilted surfaces. *Energy and Buildings*, 224, 2020. ISSN 03787788. doi: 10.1016/j.enbuild.2020.110258.

I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):6, 2007. ISSN 0730-0301. doi: 10.1145/1189762.1206075.

M. Wieland, A. Nichersu, S. M. Murshed, and J. Wendel. Computing Solar Radiation on CityGML Building Data. 2015.

K. J. Wong. Multithreading and Multiprocessing in 10 Minutes. 2022.

Z. Yao, C. Nagel, F. Kunde, G. Hudra, P. Willkomm, A. Donaubauer, T. Adolphi, and T. H. Kolbe. 3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML. *Open Geospatial Data, Software and Standards*, 3(1), 2018. ISSN 2363-7501. doi: 10.1186/s40965-018-0046-7.

Y. Zhou, M. Verkou, M. Zeman, H. Ziar, and O. Isabella. A Comprehensive Workflow for High Resolution 3D Solar Photovoltaic Potential Mapping in Dense Urban Environment: A Case Study on Campus of Delft University of Technology. *Solar RRL*, 6(5), 2021. ISSN 2367-198X 2367-198X. doi: 10.1002/solr.202100478.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.